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

Tesis de Licenciatura en
Ciencias de la Computación

Verificación de uso de protocolos a través de modelos de habilitación

Tesistas
Romina Bernatene - rbernate@dc.uba.ar - LU 212/98
Florencia Fotorello - ffotorel@dc.uba.ar - LU 407/98

Directores
Lic. Guido de Caso
Lic. Hernán Czemerinski

Diciembre 2011

Resumen

A medida que los sistemas de software van ocupando lugares cada vez más claves dentro de los negocios, un error en los mismos agudiza el impacto de sus consecuencias. Por este motivo, la confiabilidad del software pasó a ser uno de los aspectos más importantes dentro de la Ingeniería del Software.

Una de las piezas principales para el correcto funcionamiento de un sistema de software es el establecimiento de una buena comunicación entre sus componentes, para que cooperen adecuadamente generando el resultado esperado. A través de un *protocolo* se enmarca la comunicación y se brindan las reglas a respetar.

El correcto uso de un protocolo garantiza el éxito en la interacción entre una API (*Application Programming Interface*) y su cliente, por lo tanto verificar que esto suceda es de sumo interés. Sin embargo, realizarlo tiene una complejidad muy alta ya que un protocolo define un conjunto potencialmente infinito de ejecuciones.

En este trabajo se presenta una técnica de verificación de la utilización de protocolos en base a un modelo de habilitación EPA (*Enableness-Preserving Abstraction*). Este modelo se construye utilizando una herramienta llamada *Contractor*, que genera el mismo a través del contrato de la API. Para aplicar la técnica de verificación propuesta se desarrolló un prototipo de herramienta encargado de dar soporte al análisis.

Agradecimientos

Es difícil decidir por quién empezar a agradecer, mucha gente querida nos acompañó y apoyó en todo este proceso. Considerando las dificultades y las diferentes etapas por las que pasamos como personas, como estudiantes y como profesionales nuestro primer agradecimiento es para nuestras familias, que estuvieron ayudándonos siempre en esta etapa final. Principalmente queremos agradecer a nuestros esposos, *Juan y Leonardo*, que tomaron el cuidado de nuestras respectivas niñas para que tengamos el tiempo de dedicarnos a nuestra tesis. Numerosas veces nos alcanzaron a nuestras reuniones, nos apoyaron incondicionalmente y hasta nos dieron sus consejos en algunos temas.

Agradecemos a nuestras niñas, *Sofía* y *Zoe*, que esperaban ansiosas a que sus madres terminen de estudiar y ayudaban a contar "*uno, dos y tres"* antes de ejecutar algún proceso.
Y a nuestras familias originarias y amigos, que no se cansaban de preguntar cuándo terminábamos la tesis y nos acompañaron también a lo largo de toda la carrera.

Queremos agradecer especialmente a nuestros directores, *Guido* y *Hernán*, quienes supieron asesorarnos, guiarnos y mostrarnos el modo de lograr nuestro objetivo.
También a la *Facultad de Ciencias Exactas y Naturales de la Universidad de Buenos Aires*, a sus directivos, docentes, compañeros de cursada, amigos y todas aquellas personas que hicieron y hacen que sea posible contar con una formación académica única.

Tabla de Contenido

1 Introducción

Los sistemas de software están incorporados en las tareas diarias que se desarrollan, tanto en operaciones simples como en operaciones críticas y complejas. La variedad de ámbitos en los cuales los sistemas de software tienen un rol clave incluyen áreas críticas tales como financieras, bancarias, médicas, telecomunicaciones, militares y aeroespaciales, entre otras. Muchas veces un simple error en el sistema de software es el causante de pérdidas multimillonarias o incluso de vidas [HWSB].

La necesidad de contar con software confiable hace que la verificación de la correctitud del mismo tenga un rol cada vez más importante. Garantizar la *calidad* del software permite asegurar la eficiencia, confiabilidad, usabilidad e integridad del sistema. Es decir, permite establecer el grado en el que un conjunto de características inherentes cumple con los requisitos. Para poder brindar garantías acerca de la calidad del software pueden utilizarse dos técnicas: *verificación formal o testing*.

La verificación formal requiere de un modelo formal (como por ejemplo, *model-checking*), mientras que la técnica de testing consiste en la ejecución de casos de test y su posterior chequeo de correctitud. Estas dos técnicas, una más formal y la otra más experimental (si bien testing tiene resultados formales) pueden ayudarse mutuamente. Por ejemplo, con especificaciones formales se pueden construir *oráculos* para evaluar el resultado de los casos de test.
A continuación brindaremos más detalles sobre ambas técnicas.

En cuanto a la verificación formal existe por un lado la *verificación estática*, donde no se ejecuta el programa y se consideran todas las trazas posibles. Por otro lado, existe la *verificación dinámica*, que se basa en ejecutar y observar el comportamiento del software.
Una de las características deseables en la verificación es que sea automatizable.

Con respecto al testing como alternativa para el chequeo de la calidad del software, el mismo se lleva a cabo experimentalmente basado en un listado de requerimientos que el sistema de software debe cumplir. A través de la especificación de *casos de pruebas* por parte de un grupo de analistas, se procede a la ejecución y chequeo de la correctitud de los mismos en un ambiente controlado, especialmente construido para tal fin.

Gran variedad de software involucra la utilización de *protocolos,* los cuales establecen conjuntos de reglas que gobiernan la forma de interacción entre una clase o API y su cliente (por ejemplo, una API podría ser la red bancaria y su cliente podría ser un cajero automático). Por lo tanto, verificar esta correcta comunicación es uno de los aspectos relevantes a analizar. En algunos casos, esta interacción es flexible pero en otros es estricta y no respetarla ocasiona que fácilmente se incurra en errores que afectan el comportamiento esperado del software.
Reglas, como por ejemplo, "es necesario abrir un archivo antes de leerlo" o "no se pueden realizar más operaciones *pop* que *push* en una pila", permiten comprender que existe una secuencia lógica a respetar por parte del cliente.

El principal problema que se presenta al querer garantizar el correcto uso de un protocolo es que el conjunto de interacciones permitido por el protocolo en la mayoría de los casos es infinito, por lo cual resulta impracticable realizar un testeo exhaustivo.

En este trabajo se propone una técnica para la verificación del uso correcto de un protocolo por parte del cliente basado en los modelos EPA (*Enabledness-preserving Abstraction*) que se explicarán en detalle más adelante. Adicionalmente, se presentará un prototipo de herramienta que permite realizar esta verificación. El prototipo está compuesto por dos aplicaciones. La primera aplicación se encarga de instrumentar al cliente para poder obtener información valiosa con el fin de realizar luego la verificación. La segunda aplicación es la encargada de realizar la verificación en dos fases diferenciadas con distinto nivel de precisión.
Para la implementación del prototipo se han utilizado diferentes herramientas de soporte: *Contractor*, *AT&T FSM Library*, *CVC3* y conceptos de *Programación Orientada a Aspectos*.

2 Motivación

Como se hace referencia en la introducción, el protocolo determinado entre un cliente y un proveedor establece reglas que deben ser cumplidas por ambas partes. Sin embargo, durante el proceso de interacción entre el cliente y el proveedor, pueden presentarse incumplimientos o violaciones al protocolo. Estas violaciones pueden ser originadas por cualquiera de las partes. El impacto que acarrea una violación en el protocolo influye directamente en las funcionalidades y en la calidad del software.

Generalmente los proveedores son desarrollados por gente experta, son estables y altamente reutilizables. Por estos motivos, se asume que el proveedor es robusto, es decir, respeta el protocolo con lo cual la problemática queda acotada a las violaciones por parte del cliente.
La detección de estas violaciones en el cliente es de gran interés, pero no resulta fácil poder lograrla ya que realizarla de manera manual involucra un costo muy alto. Realizar este análisis en producción, tampoco resulta en una opción atractiva dado que deriva en un impacto muy alto en la performance del software.

Con el fin de modelar las diferentes interacciones correctas entre el cliente y el proveedor, podemos ver el protocolo representado mediante un modelo infinito como ser LTS (*Labelled Transition System*) el cual es un sistema de transición de estados, cuyas etiquetas son acciones con parámetros concretos, y sirve para indicar el orden en que deben realizarse las mismas. Sin embargo, cuando se especifica una clase de un proveedor, su protocolo difícilmente se encuentre explicitado. Como alternativa para poder derivar cómo sería esta interacción, se puede utilizar el *contrato* descripto para la clase [Mey92], lo cual está siendo una práctica cada vez más usual en el desarrollo de software [RGMA03].

Las dificultades mencionadas previamente nos motivaron a buscar una manera de detectar las violaciones al protocolo por parte de un cliente de una clase de manera automática y sin interferir en la performance de la aplicación. Para esto se utiliza un modelo EPA que se genera a partir del contrato de una clase. El modelo EPA es una FSM (*Finite State Machine*) que permite representar qué acciones están habilitadas en cada uno de los estados.

Se presenta a continuación un ejemplo simple de cómo se visualizan diferentes interacciones entre el cliente y el proveedor en un modelo EPA de la clase *Circular Buffer* [CBGU09].

El contrato entre el cliente y el proveedor se presenta en la Figura 1.

CircularBuffer

variable a array of integers
variable wp, rp integer

inv $0 \leq rp < |a| \wedge 0 \leq wp < |a| \wedge |a| > 3 \wedge wp \neq rp$
start $|a| > 3 \wedge rp = |a| - 1 \wedge wp = 0$

action write(integer n)

```
pre     (wp < rp − 1) ∨ (wp = |a| − 1 ∧ rp > 0) ∨ (wp < |a| − 1 ∧ rp < wp)

post    rp′ = rp ∧ (wp < |a| − 1 ∨ wp′ ⇒ wp + 1) ∧ (wp = |a| − 1 ∨ wp′⇒ 0)
        ∧ (a′ = updateArray(a, wp, n))

action  integer read()
 pre    (rp < wp − 1) ∨ (rp = |a| − 1 ∧ wp > 0) ∨ (rp < |a| − 1 ∧ wp < rp)

post    rv = a[rp ] ∧ wp′ = wp ∧ a′ = a ∧ (rp < |a| − 1 ⇒ rp′ = rp + 1)
        ∧ (rp = |a| − 1 ⇒ rp′ = 0)
```

Figura 1. Contrato de la Clase Circular Buffer

La especificación incluye tres variables de estado:

- **a** representa un array de entero con *slots* utilizados por el buffer para almacenar datos,
- **wp** es un puntero al primer slot disponible para almacenar nuevos datos,
- **rp** es un puntero al último slot desde el cual los datos fueron leídos.

La especificación incluye pre y postcondiciones para las dos acciones que se pueden aplicar al *Circular Buffer*: *read* y *write*.

La acción *write* requiere que el buffer cuente con slots vacíos y da como resultado un *Circular Buffer* que ha incrementado en uno el puntero de escritura *wp* a menos que haya alcanzado el tamaño del *array a*, en este caso el puntero de escritura toma el valor 0.
La acción *read* requiere que el buffer cuente con slots con datos sin haber sido leídos aún y actualiza el puntero *rp* utilizando la misma estrategia que la acción *write* utiliza con el puntero *wp*.

Finalmente, la especificación incluye un invariante que requiere que el *Circular Buffer* tenga más de tres slots para almacenar datos y requiere que ambos punteros sean distintos y se encuentren dentro de los límites del *Circular Buffer*, es decir, entre 0 y el tamaño del array. Existe además una condición sobre los estados aceptables de inicio para los *Circular Buffers*.

A partir del contrato se puede construir un modelo EPA como el de la Figura 2.

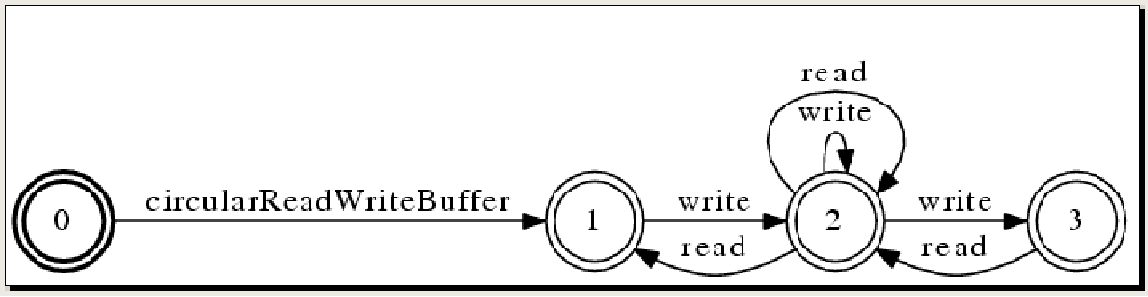

Figura 2. EPA para Circular Buffer

En este ejemplo, el modelo EPA indica que en el estado 1 el *Circular Buffer* se encuentra vacío y sólo es posible realizar una operación de escritura. El siguiente estado indica que el *Circular Buffer* tiene datos almacenados y que se pueden realizar tanto operaciones de lectura como de escritura.

El estado 3 indica que el *Circular Buffer* está lleno y que no es posible escribir más datos, sólo leerlos.

Supongamos que una de las trazas capturadas durante la ejecución de un cliente fuese la siguiente:
write ⇨ ***write*** ⇨ ***read*** ⇨ ***read***
Dado que esta secuencia de acciones está contemplada en el modelo EPA, no se puede concluir que se haya realizado un uso incorrecto del protocolo. En el modelo EPA están representadas todas las trazas válidas, pero además están incluidas otras trazas que no lo son.

En cambio, si la traza capturada fuese:
read ⇨ ***write*** ⇨ ***read*** ⇨ ***read***
podemos apreciar que la secuencia de acciones no está contemplada en el modelo EPA, con lo cual se concluye que el cliente no ha respetado el protocolo.

Supongamos ahora que tenemos otra traza generada con la siguiente secuencia de acciones:
write ⇨ ***read*** ⇨ ***read***
Si bien la traza está contemplada dentro del modelo EPA, la segunda operación de *read* está intentando leer un slot sin que el mismo haya sido previamente escrito en el buffer, lo cual no es correcto de acuerdo al protocolo impuesto por el contrato dado en la Figura 1. Este caso no es detectado debido a que el modelo EPA es no determinístico. Sin embargo, si se consideran los valores de las variables *rp, wp* y *a* en cada punto de la traza alcanzaría para desambiguar el no determinismo del modelo EPA y establecer que efectivamente el cliente no ha respetado el protocolo.

La propuesta presentada en este trabajo consiste en detectar este tipo de trazas inválidas utilizando un modelo EPA generado a partir del contrato de la clase. En la siguiente sección se formalizarán los distintos conceptos utilizados para su desarrollo.

3 Definición del problema

En este capítulo se planteará de manera formal qué tipo de problema es el que se contempla junto a los fundamentos teóricos utilizados para llegar a la solución propuesta.

Como se ha comentado en el capítulo anterior, podemos modelar el *protocolo* entre un cliente y una API mediante una máquina de estados potencialmente infinita LTS.
Formalizamos estos conceptos de la siguiente manera:

Definición 1 *(LTS, Labelled Transition System)*
Una estructura de la forma $I = < V, D, A, S, S_0, \Delta >$ *es llamada* Labelled Transition System *cuando:*

- $\mathcal{V}$ *es un conjunto finito de nombres de variables.*
- $\mathcal{D}$ *es un valor de dominio para las variables en* $\mathcal{V}$.
- $\mathcal{A}$ *es un conjunto de etiquetas de acción.*
- $\mathcal{S}$ *es un conjunto de funciones desde* $\mathcal{V}$ *hacia* $\mathcal{D}$ *(es decir,* $\mathcal{S} \subseteq \mathcal{V} \rightarrow \mathcal{D}$).
- $S_0 \subseteq \mathcal{S}$ *es el conjunto de estados iniciales.*
- Δ: $\mathcal{S}$ x $\mathcal{A}$ x $\mathcal{D} \rightarrow \varrho(\mathcal{S})$ *es una función de transición.*

Por ejemplo, para Circular Buffer sería:

$\mathcal{V} = \{rp, wp\}$
$\mathcal{D} = \{\mathbb{N}\}$
$\mathcal{A} = \{read, write\}$

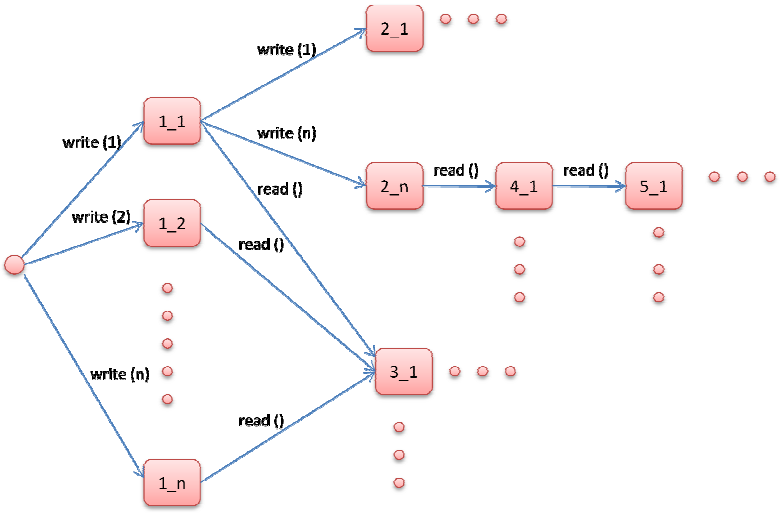

Figura 3. LTS para Circular Buffer.

Definición 2 *(Protocolo)*
Llamamos protocolo *a un LTS potencialmente infinito que representa todas las ejecuciones legales entre un cliente y una API.*

El problema que se presenta al definir un protocolo de esta manera es que el modelo es infinito y por lo tanto, no es posible trabajar adecuadamente sobre él. En base a esta limitación es que surge nuestra propuesta de utilizar los modelos EPA como una abstracción finita con la cual trabajar. Asimismo, debido a que un protocolo no suele estar especificado, se derivará a partir de un *contrato*. Por lo tanto, se definirá un modelo EPA directamente a partir del contrato de la API.

Se presentarán a continuación de manera formal los conceptos mencionados. Comenzaremos por definir qué es un *contrato* y finalmente llegaremos a la descripción de una *abstracción finita* de un contrato, la cual nos permitirá realizar las verificaciones deseadas.

Se denota $\mathbb{P}$ *(X)* al conjunto de predicados de primer orden cuyas variables libres están incluidas en *X*. Utilizaremos el operador X' para referirnos al conjunto de variables $\{x' \mid x \in X\}$.

Definición 3 *(Contrato)*
Una estructura de la forma $C = \langle V, inv, init, A, P, Q \rangle$*, es llamada un* contrato *cuando:*

• *V es un conjunto finito de variables.*
• $inv \in \mathbb{P}(V)$ *es el invariante del sistema.*
• $init \in \mathbb{P}(V)$ *es el predicado que se debe cumplir inicialmente.*
• $A = \{a_1, \ldots, a_n\}$ *es un conjunto finito de etiquetas de acción.*
• $P : A \to \mathbb{P}(V \cup \{p\})$ *es un mapeo total que asigna una precondición para cada una de las etiquetas de acción. Notar que la variable distinguida p representa el nombre de cualquier parámetro de una acción.*
• $Q : A \to \mathbb{P}(V \cup V' \cup \{p\})$ *es un mapeo total que asigna una postcondición para cada una de las etiquetas de acción, donde v' representa el nuevo valor de la variable v luego de la ejecución de la acción.*

Con el fin de simplificar y sin perder generalidad, se establece el número de parámetros en 1. Más parámetros pueden ser establecidos si se piensa a *p* como el nombre de una *n*-upla.

A modo de ejemplo, se presenta a continuación el contrato correspondiente a Circular Buffer:

$V = \{wp, rp, a\}$
$inv = 0 \leq rp < |a| \wedge 0 \leq wp < |a| \ \wedge |a| > 3 \ \wedge wp \neq rp$
$init = |a| > 3 \wedge rp = |a| - 1 \wedge wp = 0$
$A = \{write, read\}$
$P_{write} = \{(wp \ < rp - 1) \vee (wp = |a| - 1 \ \wedge rp > 0) \vee (wp < |a| - 1) \wedge rp < wp\}$
$Q_{write} = \begin{Bmatrix} rp' = rp \ \wedge (wp < |a| - 1 \wedge wp' \Rightarrow wp + 1) \wedge (wp = |a| - 1 \ \vee wp' \Rightarrow 0) \\ \wedge \left(a' = updateArray(a, wp, n)\right) \end{Bmatrix}$
$P_{read} = \{(rp < wp - 1) \vee (rp = |a| - 1 \ \wedge wp > 0) \vee (rp < |a| - 1) \wedge wp < rp\}$
$Q_{read} = \begin{Bmatrix} \exists\, rv\, (rv = a[rp'] \wedge wp' = wp \ \wedge a' = a \ \wedge (rp < |a| - 1 \ \Rightarrow rp' = rp + 1) \\ \wedge (rp = |a| - 1 \Rightarrow rp' = 0)) \end{Bmatrix}$

A continuación se presentará cómo se deriva el protocolo a partir del contrato de la API.

Definición 4 *(Implementación de un contrato)*
Dado un contrato $C = \langle V, inv, init, A, P, Q \rangle$*, un dominio de valores* $\mathbb{D}$ *y una interpretación* $\mathbb{D}^{op}$ *para los símbolos que aparecen en predicados. Decimos que un Labelled Transition System de la forma* $I =$

$\langle \mathcal{V}, \mathcal{D}, \mathcal{A}, \mathcal{S}, \mathcal{S}_0, \Delta \rangle$ *es una* implementación para el contrato C *bajo la interpretación* $\langle \mathbb{D}, \mathbb{D}^{op} \rangle$ *si y sólo si se cumple lo siguiente:*

1. $\mathcal{V} \supseteq V, \mathcal{D} = \mathbb{D}, \mathcal{A} = A$
2. *init(s) es true para cada* $s \in \mathcal{S}_0$
3. *Existe un conjunto de estados* $\mathcal{S}_v \subseteq \mathcal{S}$ *tales que el inv(s) es true para cada* $s \in \mathcal{S}_v$, $\mathcal{S}_0 \subseteq \mathcal{S}_v$ *y para cada* $a_i \in A$ *y* $d \in \mathcal{D}$ *tal que la precondición de la acción* a_i $P_{a_i}(s \cup \{p \mapsto d\})$ *es true entonces* $\Delta(s, a_i, d)$ *es no vacío y sus elementos s' están todos incluidos en* $\mathcal{S}_v$. *Adicionalmente, se mantiene* $Q_{a_i}(s \cup s' \cup \{p \mapsto d\})$.

Cabe aclarar que dado un contrato siempre se lo puede implementar de la forma propuesta.

Una posible implementación del contrato de Circular Buffer para un tamaño de buffer 4, es el LTS $I = \langle \mathcal{V}, \mathcal{D}, \mathcal{A}, \mathcal{S}, \mathcal{S}_0, \Delta \rangle$, donde:

$\mathcal{V} = \{a, wp, rp\}$
$\mathcal{D} = \mathbb{Z} \cup (\{0,1,2,3\} \to \mathbb{Z})$
$\mathcal{A} = \{read, write\}$
$\mathcal{S} = \{s \mid |s(a)| = 4 \wedge 0 \leq s(rp) < 4 \wedge 0 \leq s(wp) < 4 \wedge s(rp) \neq s(wp)\}$
$\mathcal{S}_0 = \{(a \mapsto [0 \mapsto 0, 1 \mapsto 0, 2 \mapsto 0, 3 \mapsto 0], rp \mapsto 3, wp \mapsto 0)\}$

Se utilizan *Finite States Machines* para proveer una representación abstracta de un contrato, o más precisamente, de las implementaciones permitidas por un contrato. Una FSM es definida como una estructura $M = \langle S, S_0, \Sigma, \delta \rangle$, donde S es un conjunto finito de estados, $S_0 \subset S$ es el conjunto de estados iniciales, Σ es un alfabeto finito y $\delta: S \times \Sigma \longrightarrow \wp(\mathrm{s})$ es una función de transición.

Se continuará con la definición formal de la abstracción finita EPA, denominada FSCA, como una FSM que es capaz de simular cualquier posible implementación de un contrato.

Definición 5 *(FSCA, Finite State Contract Abstraction)*
Dado un contrato $C = \langle V, inv, init, A, P, Q \rangle$, *una interpretación* $\langle \mathbb{D}, \mathbb{D}^{op} \rangle$ *y una Finite State Machine (FSM)* $M = \langle S, S_0, \Sigma, \delta \rangle$ *decimos que M es una* Finite State Contract Abstraction (FSCA) *de C bajo la interpretación* $\langle \mathbb{D}, \mathbb{D}^{op} \rangle$ *si y sólo si para cada implementación* $I = \langle \mathcal{V}, \mathcal{D}, \mathcal{A}, \mathcal{S}, \mathcal{S}_0, \Delta \rangle$ *de C existe una función total* $abs_I : \mathcal{S}_v \to S$ *tal que:*

1. $abs_I(\mathcal{S}_0) \subseteq S_0$
2. Para cada $s \in \mathcal{S}_v$, *y cada etiqueta de acción* a_i *y parámetro d tal que se cumple* P_{a_i}, *entonces* $abs_I(\Delta(s, a_i, d)) \subseteq \delta(abs_I(s), a_i)$.

La idea principal al configurar el nivel de abstracción para soportar la validación de contratos es capturar diferentes estados del contrato que son importantes en términos de las operaciones que están habilitadas en un momento dado. Esto significa que se agrupan estados de implementaciones de contratos basados en las precondiciones que son satisfechas en esos estados.

Luego mostraremos cómo construir una FSCA a partir de un contrato. El nivel particular de abstracción para las FSCAs que serán construidas está basado en la noción de *enabledness* que se detallará a continuación.

Definición 6 *(Enabledness Equivalence States)*
Dado un contrato $C = \langle V, inv, init, A, P, Q \rangle$, *una implementación de la forma* $I = \langle \mathcal{V}, \mathcal{D}, \mathcal{A}, \mathcal{S}, \mathcal{S}_0, \Delta \rangle$ *de C bajo* $\langle \mathbb{D}, \mathbb{D}^{op} \rangle$ *y dos estados* $s, t \in \mathcal{S}$ *decimos que s y t son estados enabledness equivalent (escrito como* $s \equiv_e t$*) si y sólo si para cada* $a \in \mathcal{A}$*:*

• $\exists\, d\,.\, P_a\,(s \cup \{p \mapsto d\}) \iff \exists\, d'.\, P_a\,(t \cup \{p \mapsto d'\})$

Una abstracción *enabledness-preserving* es una abstracción de estado finito de un contrato en la cual los estados son particionados por *enabledness equivalence*.

Definición 7 *(Enabledness-preserving FSCA)*
Una FSCA $M = \langle S, S_0, \Sigma, \delta \rangle$ *de un contrato* $C = \langle V, inv, init, A, P, Q \rangle$ *bajo la interpretación* $\langle \mathbb{D}, \mathbb{D}^{op} \rangle$ *es enabledness-preserving si y sólo si para cada implementación I de C existe* $abs_I : \mathcal{S}_v \rightarrow S$ *(una función de abstracción) tal que dado un par de estados s, t en* $\mathcal{S}_v$*, entonces se cumple que*
$s \equiv_e t \iff abs_I(s) = abs_I(t)$.

En el resto del documento se hará referencia a EPA, enabledness-preserving FSCA y FSCA de manera indistinta.

Con el fin de construir una enabledness-preserving FSCA, primero necesitamos definir la noción de *invariante* de un conjunto de acciones. Dado un subconjunto de acciones *as* de un contrato *C*, queremos caracterizar todos los estados *s* de implementaciones de *C* que satisfacen el invariante del contrato *inv* en el que cada acción *a* en *as* es posible desde *s* (existe un parámetro *p* para cada acción *a* en *as* tal que la precondición P_a de la acción *a* se cumple) y también que cada acción *a* que no está en *as* no es posible desde *s*.

Definición 8 *(Invariante de un Conjunto de Acciones)*
Dado un contrato $C = \langle V, inv, init, A, P, Q \rangle$, *el invariante de un conjunto de acciones* $as \subseteq \wp(A)$ *es el predicado* $inv_{\text{as}} \in \mathbb{P}\,(V\,)$ *definido como:*

$$inv_{as} \stackrel{\text{def}}{=} inv \wedge \bigwedge_{a \in as} \exists p.\, P_a \wedge \bigwedge_{a \notin as} \nexists p.\, P_a$$

Utilizando los invariantes de un conjunto de acciones, la construcción de una abstracción enabledness-preserving es directa.

Definición 9 *(Algoritmo de Construcción de una FSCA utilizando la propiedad Enabledness-preserving)*
Dado un contrato $C = \langle V, inv, init, A, P, Q \rangle$ *y una interpretación* $\langle \mathbb{D}, \mathbb{D}^{op} \rangle$*, procedemos a construir una FSM* $M = \langle S, S_0, \Sigma, \delta \rangle$ *donde:*

1. $S = \wp(A)$
2. $as \in S_0$ *si y sólo si* $init \Rightarrow inv_{\text{as}}$

3. $\Sigma = A$

4. Para todos los as $\in S$ *y* $a \in \Sigma$, *si* $a \notin as$ *entonces* $\delta(as, a) = \emptyset$, *en caso contrario:*

$$\delta(as, a) \supseteq \{bs | inv_{as} \wedge Q_a \wedge inv'_{bs} \; es \; satisfacible\}$$

Se puede observar que la FSCA enabledness-preserving construida nunca tiene dos estados con las mismas acciones habilitadas.

Es sencillo probar que la FSCA construida de acuerdo a la Definición 9 es una abstracción enabledness-preserving.

Teorema 1. *Dado un contrato C y una interpretación* $\langle \mathbb{D}, \mathbb{D}^{op} \rangle$, *entonces si M es construida según la Definición 9 es una enabledness-preserving FSCA de C bajo* $\langle \mathbb{D}, \mathbb{D}^{op} \rangle$.

La prueba del teorema puede realizarse simplemente mostrando que dada una implementación I,

$$abs_I(s) \stackrel{\text{def}}{=} \{a | \exists \, d \in \mathbb{D}. P_a(s \cup \{p \rightarrow d\})\}$$

es una función de abstracción tal que cada par de estados *s*, *t* satisface que $s \equiv_e t \Leftrightarrow \text{abs}_I(s) = \text{abs}_I(t)$.

La utilización de modelos FSCA será la base para determinar cuándo un cliente cumple o no el protocolo según las trazas de ejecución que se analicen. Se verá a continuación la definición de traza de ejecución y se determinará cuándo la misma es válida o inválida de acuerdo al modelo FSCA.

Definición 10 *(Traza de ejecución)*
Llamaremos traza de ejecución o traza a la secuencia de acciones invocadas desde el cliente. Las mismas se encuentran posicionadas de acuerdo al orden de invocación.
Notaremos a la traza de la siguiente manera:

$T_i = [a_1, \ldots, a_j]$

donde T_i *es la traza y* a_k *son las acciones invocadas siendo* $1 \leq k \leq j$

Definición 11 *(Conformidad de clientes con el modelo FSCA)*
Diremos que una traza de ejecución es válida *si la secuencia de acciones de la misma es aceptada en el modelo FSCA de la API. Contrariamente, diremos que una traza de ejecución es* inválida *si no está contenida en el modelo.*

La propuesta planteada tiene como ventaja poder trabajar, a través del modelo FSCA, con una abstracción finita del problema.

Dado que se realiza una reducción del modelo, hay ciertas limitaciones de la propuesta que es necesario tener en cuenta. Ya no se trabaja con el protocolo en sí, sino con una aproximación del mismo dado que no es posible representar con un lenguaje regular un problema que es intrínsecamente más complejo representado por un LTS. Otra limitación es que no se tienen en cuenta los parámetros que pueden existir en una acción. Adicionalmente, el modelo obtenido es *no determinístico*, es decir, desde al menos un estado es posible transicionar a estados diferentes con una misma acción, lo cual requiere un análisis complementario para tomar una correcta decisión sobre cuál es el siguiente estado. Daremos más detalles acerca de cómo se resuelve el no determinismo en el siguiente capítulo.

Hemos elegido trabajar con el modelo FSCA debido a las ventajas y facilidades que brinda. El nivel de abstracción resulta adecuado para poder realizar la verificación del cumplimiento del protocolo y nos permite tener la certeza de que si una traza de ejecución no puede ser simulada, entonces se trata necesariamente de una traza inválida, dado que la abstracción se construye de manera tal que es posible simular toda posible implementación de un contrato. De esta manera, al tener la capacidad de poder detectar claramente cuándo una traza es inválida, podemos confirmar que el cliente de la API realiza un inadecuado uso del protocolo.

Dado que ya se ha descripto por qué el modelo FSCA servirá como marco teórico para llevar a cabo la propuesta, se procederá a explicar de qué manera se generará este modelo para una API dada.

Para poder generar el modelo FSCA de una API utilizamos una herramienta llamada *Contractor*. Contractor requiere como entrada cierta información de la API sobre la cual realizará la abstracción. Esta información puede provenir directamente del *código fuente* de la API [CBGU11] o bien, puede provenir del *contrato* de la clase [CBGU09].

En resumen, hasta ahora se ha planteado la necesidad de contar un modelo finito para poder abstraer la representación infinita LTS de un protocolo. Vimos que los modelos EPA permiten realizar esta abstracción y se decide utilizar el contrato de la clase para generar el modelo FSCA.

4 Verificación de clientes utilizando modelos de habilitación

En primer lugar, para poder realizar la verificación es necesario contar con la información adecuada. Por este motivo, se presenta un instrumentador con el fin de proveer una solución para la recolección de dicha información.
Luego, se continúa detallando las características de la verificación la cual se encuentra constituida por en dos fases.

4.1 Instrumentación

El *Instrumentador* se encarga de generar código, creando un archivo que debe ser incluido en el proyecto del cliente. Para esto, se basa en la *Programación Orientada a Aspectos*, dado que el registro de la información es transversal al proyecto. Mediante aspectos se intercepta el código fuente en las circunstancias necesarias para la generación de las trazas de ejecución.

El código generado se encargará de crear por cada instancia de la clase un archivo de traza de ejecución con la información que se muestra en la Figura 4.

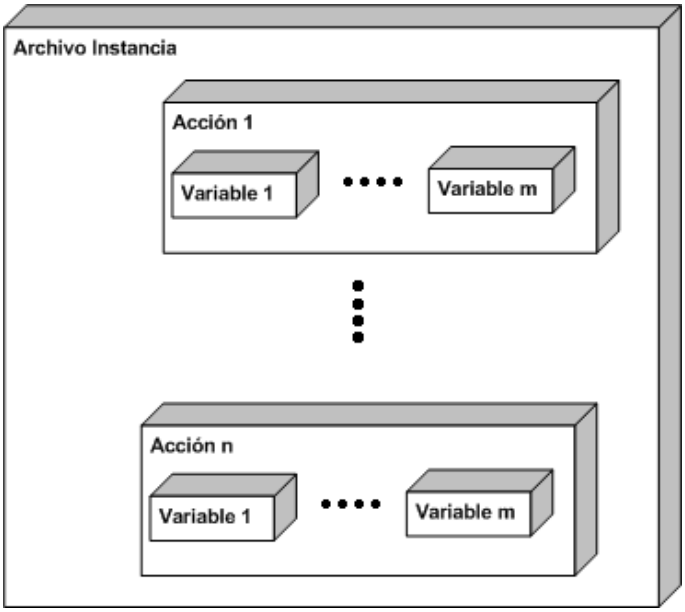

Figura 4. Esquema del archivo de traza de ejecución.

Cada acción corresponde a la secuencia de acciones ejecutadas sobre una instancia determinada de la clase. Dado que el nombre de la acción implementada no es necesariamente el mismo nombre de la acción que figura en el contrato, se provee la funcionalidad de mapeo entre ambas definiciones. El nombre de la acción registrada será la correspondiente al contrato.

Cada variable registrada en la traza de ejecución corresponde a una variable definida en el contrato. Debido a que no necesariamente la implementación contenga las mismas, se solicita al programador que provea la correspondiente función de abstracción para cada una de las

variables. De este modo, independientemente de la manera en que hayan sido implementadas las variables, se pueden observar los valores desde el punto de vista del contrato. Los valores de las variables registradas serán los obtenidos mediante las funciones de abstracción correspondientes. Se registran todas las variables del contrato por cada acción ejecutada.

4.2 Verificación

El código generado por el Instrumentador permite, al ser incluido en el proyecto, obtener las trazas durante la ejecución del programa cliente. Con esta información se procede a realizar la verificación.

La verificación está constituida por dos fases, ambas utilizan el modelo de habilitación FSCA generado a partir de contrato de la API para el análisis correspondiente. A continuación brindaremos más detalles sobre cada una de las fases.

4.2.1 Características de la Fase 1

En primer lugar, describiremos la información de entrada necesaria para realizar la verificación de la fase 1. Se requiere de un contrato de la API y de las trazas obtenidas durante la ejecución del cliente a analizar.

La FSCA que brinda *Contractor* tiene un nivel de abstracción que permite determinar los diferentes estados que son relevantes en términos de operaciones habilitadas en un momento dado de la ejecución. Con la FSCA podemos simular todas las posibles trazas de ejecución válidas de acuerdo a un contrato. Por lo tanto, aquellas trazas de ejecución que no puedan ser simuladas, son consideradas inválidas. El objetivo en esta fase es establecer en qué conjunto se localiza una traza de ejecución, si en el de la FSCA o en su complemento.

Para poder cumplir con este objetivo se efectúa el siguiente análisis entre la FSCA y la traza.
En primer lugar, se realiza una interpretación de la traza, analizándola como una FSM de la siguiente manera:

Definición 12 *(FSM de la traza)*

Dada una traza T_i procederemos a construir la FSM de la traza *FSM (T_i)= (Σ, $\mathbb{S}$, $\mathbb{S}_0$, δ,F) donde:*

- $\Sigma = \{a_k | \ a_k \in T_i and \ 1 \leq k \leq j\}$
- $\mathbb{S} = \{S_i | \ 0 \leq i \leq j\}$
- $\mathbb{S}_0 = S_0$
- $\delta = \{(S_{i-1}, a_i) \rightarrow S_i \mid 1 \leq i \leq j\}$
- $F = S_j$

Luego, para poder determinar si T_i pertenece a la FSCA se realiza la intersección entre la FSM(T_i) y la FSCA. Esta intersección determina si la traza, única palabra definida por FSM(T_i), pertenece al lenguaje establecido por la FSCA. En caso afirmativo, la intersección resultará igual a la FSM(T_i).

Dado que es de interés determinar en qué punto de la ejecución se realizó una invocación errónea, se redefine la FSM(T_i). En particular, se redefine únicamente el conjunto F de estados finales, siendo ahora F=S.
De este modo, ahora el lenguaje definido por FSM(T_i) será igual a todas las subtrazas de ejecución. Llamamos *subtraza de ejecución* a cualquier prefijo de la traza. Es decir,
siendo T_i= [a_1, ..., a_j] 1≤j la traza de ejecución, llamamos S_i *subtraza* de T_i si S_i= [a_1, ..., a_k] k≤j.

Al realizarse la intersección entre la FSCA y la FSM(T_i) modificada, el resultado de la intersección será una FSM que determina todas las subtrazas de T_i que son aceptadas por la FSCA. Si T_i pertenece al lenguaje de dicha FSM, entonces la traza se considera válida. En caso contrario, se brindará como resultado la mayor subtraza S_i perteneciente a la FSM resultante.

4.2.2 Características de la Fase 2

En la primera fase de verificación sólo se permite distinguir un primer subconjunto de trazas inválidas. Esto se debe a que pueden existir trazas de ejecución inválidas que pueden ser simuladas en la FSCA. Por ejemplo, el caso de Circular Buffer donde se realizaban más operaciones de lectura que de escritura que se mencionó en la sección Motivación. Por este motivo, se procede a la ejecución de una segunda fase cuando una traza fue aceptada por la fase 1 como válida.

Esta segunda fase se basa en la propiedad de la FSCA de ser *enabledness-preserving* y por lo tanto, la misma nunca tiene dos estados con las mismas acciones habilitadas. Esto es debido a que en cada estado se cumple el *invariante de un conjunto de acciones* mencionado en la Definición 8. Esta propiedad es la que permite establecer determinismo en una máquina de estados no determinística.

En caso de una transición no determinística, se evalúan los diferentes invariantes de los estados que se pueden alcanzar con la acción. Utilizando el invariante de los estados y los valores de las variables del contrato capturados luego de la ejecución de la acción, podemos determinar cuál es el próximo estado válido. Cabe aclarar que a lo sumo uno de los estados alcanzables por la acción va a ser válido. Si ninguno de los invariantes se satisface, entonces la traza se considera inválida.

Para poder realizar esta fase de verificación se requiere de un mayor nivel de información. Adicionalmente a la acción invocada, hay información de contexto que es relevante para el análisis. La información a la que nos referimos es el valor de las variables del contrato luego de ejecutarse la acción. Ampliaremos la definición de la traza para poder referirnos a la misma:

T_i=[[a_1, $o_1(t_1)$,..., $o_m(t_1)$], ..., [a_j, $o_1(t_j)$,..., $o_m(t_j)$]] 1≤j y m= *la cantidad de variables del contrato*

donde T_i es la traza, a_k las acciones invocadas y $o_h(t_k)$ es el valor obtenido por el observador de la variable *h* del contrato en el momento posterior a la ejecución de la acción a_k. Definiremos al *observador* de una variable como una función de abstracción que devuelve el valor correspondiente a la variable *h* a partir de las variables definidas en la API.

Para realizar la verificación se procede a recorrer la FSCA. Partiendo del estado inicial, se van utilizando las acciones de la traza para decidir el próximo movimiento. Dado que en ocasiones

la acción es insuficiente para tomar una determinación, se procede al análisis del invariante de los posibles siguientes estados.

Siendo a_k la próxima acción de la traza y siendo $\{S_1,..., S_n\}$ los siguientes estados posibles de acuerdo a la acción a_k, entonces se instanciará cada uno de los invariantes de los estados de este conjunto con los valores $o_1(t_k),...., o_m(t_k)$ asociado a a_k.
Cabe aclarar que una vez que se encuentra un estado válido, se lo toma como siguiente estado y no se continúan analizando los estados restantes. Esto se debe a la particularidad de la definición de los estados, solamente uno de los mismos puede ser satisfecho con la misma instanciación de valores.

5 Implementación

En esta sección describiremos los componentes que forman parte de nuestro sistema y cómo se relacionan. Por un lado, describiremos las dos aplicaciones principales desarrolladas: *Instrumentador* y *Analyzer*, y por otro lado, las herramientas complementarias utilizadas: *Contractor*, *AT&T FSM Library* y *CVC3*.

En la Figura 5 se puede visualizar la arquitectura general de la solución.

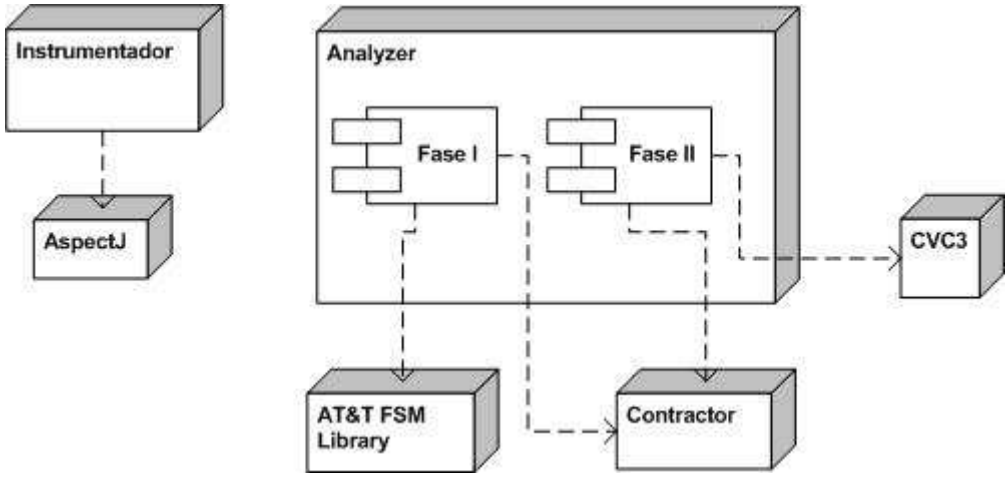

Figura 5. Arquitectura básica de los componentes del toolkit.

5.1 Instrumentador

Para poder contar con información acerca de los métodos de una clase particular invocados durante la ejecución de un cliente, utilizamos *Programación Orientada a Aspectos* (POA).

Un *aspecto* (aspect) es una funcionalidad transversal que se implementa de forma modular y separada del resto del sistema. Se define como un *join point* a un punto de ejecución dentro del sistema donde un aspecto puede ser conectado, como una llamada a un método, el lanzamiento de una excepción o la modificación de un campo. Un *advice* es la implementación del aspecto, es decir, contiene el código que implementa la nueva funcionalidad. Un *pointcut* define los advices que se aplicarán a cada join point.

Mediante aspectos, se ha podido capturar cada uno de los métodos que se ejecutaron a través de la definición de pointcuts y asociando un advice que registre la información del método junto al valor de las variables del contrato. De este modo se logra instrumentar el cliente. Llamamos *traza* al archivo XML que contiene esta información.

Para poder capturar las trazas de una aplicación para su posterior análisis, se ha implementado la herramienta llamada *Instrumentador*. El Instrumentador está escrito en Java y es el encargado de crear el soporte necesario para poder capturar las trazas de ejecución mediante la creación de un archivo de AspectJ donde se definen los poincuts y advices de manera automática para cada contrato. Se utiliza un poincut con un *after advice* por cada uno de los métodos de la clase de manera de poder capturar los valores de las variables del contrato posterior a la ejecución.

Dado que es necesario poder capturar los valores de las variables que figuran en el contrato para poder realizar luego la verificación, el archivo de AspectJ incluye código para poder tomar esta información a medida que se invoca cada uno de los métodos de la clase. Por este motivo, se tiene como requerimiento que el programador de la clase agregue *observadores* para cada una de las variables del contrato, llamándolos *get<nombreDeLaVariable>*.

El *Instrumentador* recibe como entrada los siguientes datos, que se pueden apreciar en la Figura 6:

- el path al archivo del contrato de la clase que se desea analizar,
- el archivo de mapeo entre los nombres de las acciones que figuran en el contrato y los métodos implementados. Este archivo es opcional, por default se asume que ambos nombres son iguales.
- el directorio de salida, donde se guardará el archivo generado.
- el nombre del paquete al que debe pertenecer el archivo generado. Este valor es opcional.

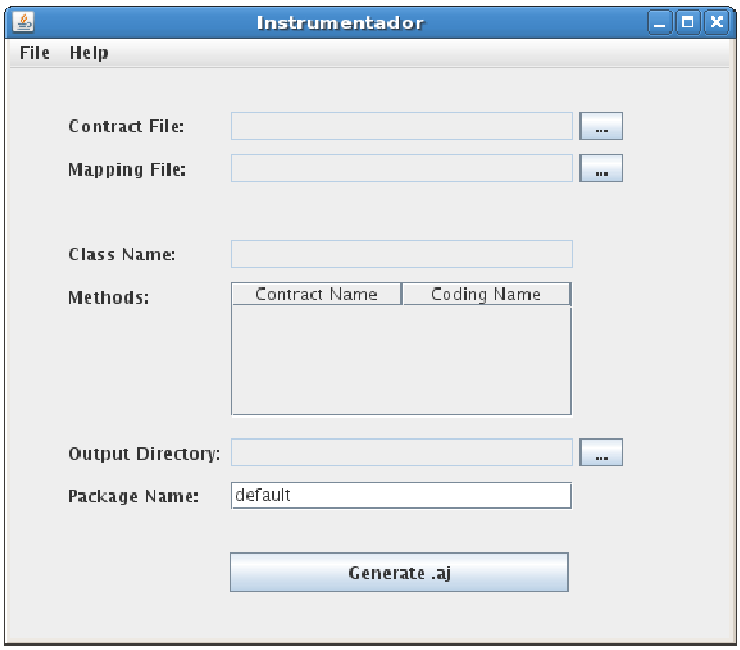

Figura 6. Interfaz del Instrumentador.

Como salida devuelve un archivo con extensión *.aj* que se deberá incluir en el proyecto donde se encuentra el código fuente del cliente. Esta aplicación deberá estar desarrollada en lenguaje Java. Luego, cuando se ejecute el cliente, automáticamente se creará un archivo de traza XML por cada uno de los objetos de la clase que se generen durante la ejecución.

5.2 Analyzer

Para realizar el análisis de las trazas de ejecución se realizan dos fases de verificación. La primera fase de verificación se ejecuta siempre mientras que la segunda fase es realizada solamente cuando la primera fase detectó como válida a la traza de ejecución. Esto se debe a que el costo de la fase 2 es mayor que el de la fase 1.

5.2.1 Fase 1

En esta primera fase se trabaja para determinar casos de uso incorrectos del protocolo analizando la traza de ejecución para verificar si la misma es válida en base a la FSCA generada por *Contractor*. En caso de detectarse un uso incorrecto se podrá visualizar la traza hasta el momento previo en que se produjo el mismo.
Para realizar este análisis se utilizaron las aplicaciones complementarias *Contractor* [CBGU] y *AT&T FSM Library* [MPR].
Para comenzar a ver en detalle la implementación de la fase 1, se presentará primero una versión simplificada a través del siguiente diagrama (Figura 7).

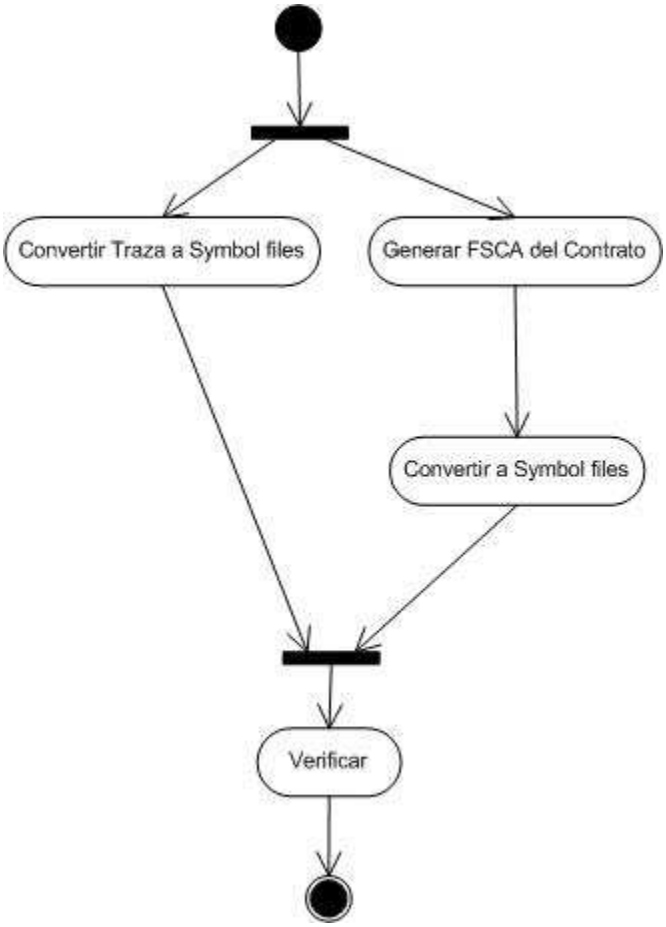

Figura 7. Diagrama fase 1.

La idea principal que se grafica es la conversión de los datos de entrada a un mismo formato para su posterior análisis. A continuación se detallan cada una de las actividades.

Convertir Traza a Symbol files

La traza se encuentra registrada en un archivo con formato XML. Con el objetivo de analizarla, la misma es interpretada como una FSM lineal, correspondiéndose cada una de las acciones a una transición de estado distinto dentro de la FSM. Todos los estados se consideran estados finales, dado que son de interés las subsecuencias de la traza para el análisis.

Dicha FSM es representada a través de *symbol files,* formato utilizado por *AT&T FSM Library,* compuestos por un archivo con extensión *.stxt* donde se almacenan las transiciones de estados y un archivo con extensión *.syms* donde se encuentran mapeadas las acciones a un nombre alfanumérico.

Generar FSCA del Contrato

Se ejecuta *Contractor* con el contrato provisto para generar la FSCA.

Convertir a symbol files

Será necesario interpretar la FSCA utilizando symbol files, formato utilizado por *AT&T FSM Library*.

Se generan los archivos *.stxt* y *.syms* correspondientes.

Verificar

Esta actividad se realiza principalmente mediante la utilización de comandos de *AT&T FSM Library*, por este motivo ambas FSMs se traducen a symbol files. Para explicar esta actividad acudimos nuevamente a la ayuda de un diagrama (Figura 8).

Figura 8. Diagrama para la actividad Verificar.

Convertir FSMs a Determinísticas

Se utiliza un comando de *AT&T FSM Library* (*fsmdeterminize*) con el propósito de obtener FSMs determinísticas.
En el caso de la FSM generada a partir de la traza, no es necesaria la ejecución de este comando ya que la misma es creada de manera determinística.

Convertir FSMs a e-free

Se utiliza un comando de *AT&T FSM Library* (*fsmrmepsilon*) con el propósito de obtener FSMs *e-free*. Las FSMs *e-free* son aquellas FSM que no tienen transiciones *e*, o sea, transiciones que permiten pasar de un estado a otro sin que ocurra una acción.
En el caso de la FSM generada a partir de la traza no es necesaria la ejecución de este comando ya que la misma es generada sin la utilización de transiciones *e*.

Intersecar FSMs

Ambas FSMs generadas se intersecan utilizando un comando de *AT&T FSM Library* (*fsmintersect*). Dado que se utiliza una herramienta externa cerrada no es posible calcular con precisión la complejidad de la operación, pero al ser una de las FSMs lineal se estima que la misma es lineal. Ambas máquinas son determinísticas y *e-free* en esta instancia, con lo cual se puede ejecutar este comando sin inconveniente.
El resultado de la intersección será la traza o una subsecuencia de la traza.

Evaluar equivalencia con FSM de la traza

La FSM resultante de la intersección es comparada con la FSM de la traza utilizando el comando de *AT&T FSM Library fsmequiv*. Si son equivalentes, significa que la traza es válida. Si no lo son, el resultado obtenido será la mayor subsecuencia de la traza admitida por la FSCA.

5.2.2 Fase 2

Puede suceder que una traza de ejecución sea válida en la verificación de la fase 1, pero esto no indica necesariamente que el cliente esté haciendo un correcto uso del protocolo. Por este motivo, se realiza una segunda fase de verificación en donde se chequea si las variables del contrato de la traza de ejecución respetan el invariante de estado derivado del contrato. Sirve para casos donde la FSCA de la clase es *no determinística,* debido a que si es *determinística* esta fase no permite detectar una traza inválida que no haya sido detectada en la fase 1.

Para poder realizar esta verificación se utilizan los valores de las variables que fueron registrados en la traza de ejecución luego de cada acción invocada por el cliente.
La aplicación complementaria utilizada para realizar la segunda fase de verificación fue el demostrador automático de teoremas llamado *CVC3* [BT07]. CVC3 requiere un formato especial de archivo de entrada, el cual es generado para cada uno de los invariantes de los estados sobre los cuales se realiza la verificación.

Presentaremos por medio de un diagrama (Figura 9) los pasos necesarios para que se lleve a cabo la segunda fase.

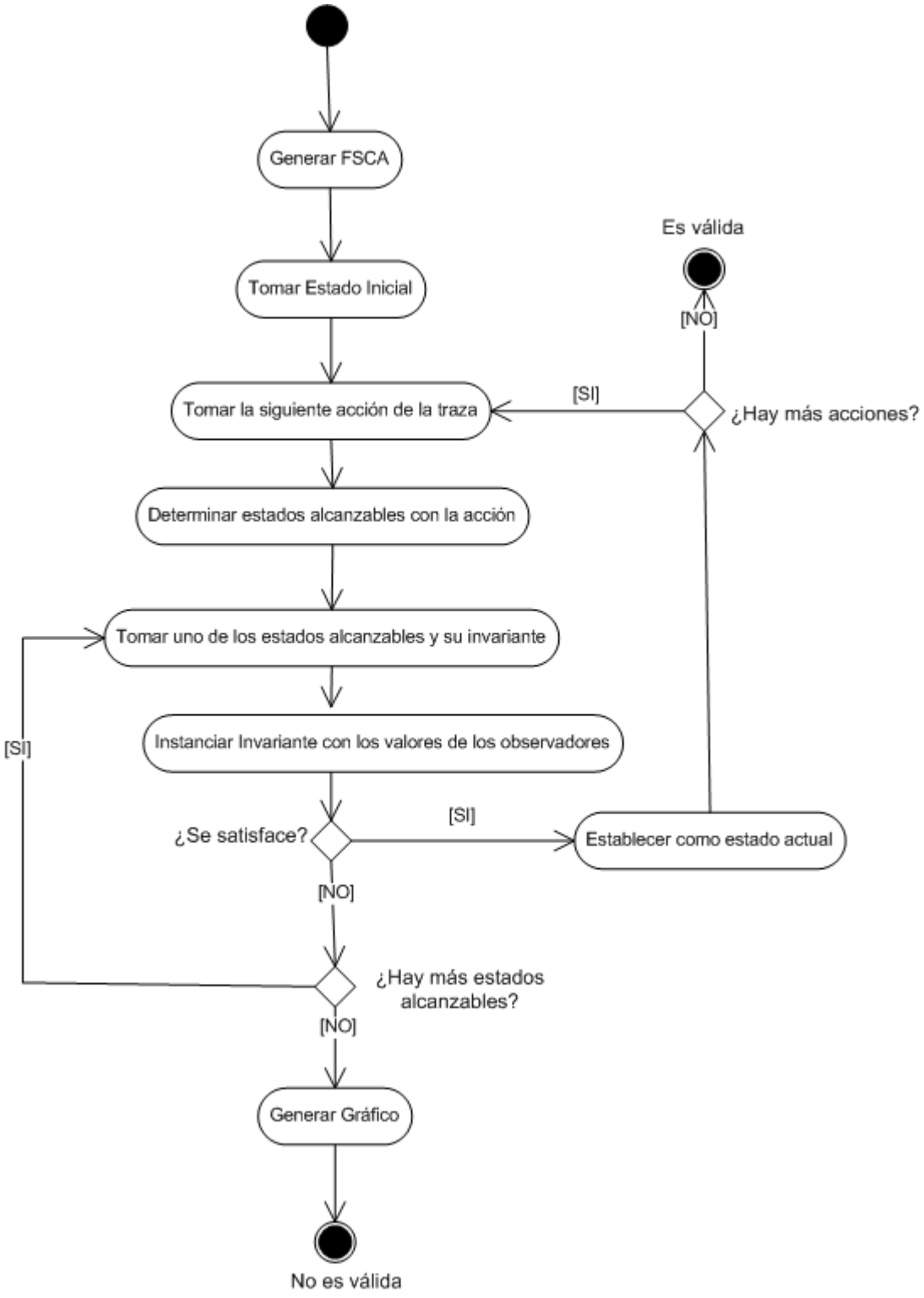

Figura 9. Diagrama fase 2.

Generar FSCA

Se ejecuta *Contractor* tomando como input el contrato provisto para generar la FSCA. En este caso se utiliza la opción especial **--print-details** de *Contractor* para poder obtener dentro del resultado los invariantes de los estados.

Tomar estado inicial

Se toma como *estado actual* el estado inicial indicado en la FSCA.

Tomar siguiente acción de la traza

Se toma la siguiente acción de la traza sin analizar. Se van tomando de a una por vez las acciones de la traza de ejecución en forma secuencial.

Determinar estados alcanzables con la acción

Se determina el conjunto de los estados para analizar a los que se podría llegar luego de ejecutar la acción a partir del estado actual.

Verificar invariante de uno de los estados alcanzables

Se toma uno de los estados alcanzables determinados sin analizar.

Para verificar si el invariante del estado se satisface con los valores de las variables obtenidos en la traza se utiliza la herramienta *CVC3*. Por este motivo, se procede a convertir el invariante y sus instancias al formato de archivo de entrada de *CVC3*.

En particular, se debe realizar una conversión especial debido a que *CVC3* no posee el tipo de datos *Boolean*, el cual es frecuentemente utilizado en los contratos. Por este motivo se utiliza e interpreta *Bitvector*. Luego se ejecuta *CVC3* con este archivo de entrada.

Establecer como estado actual

Cuando uno de los estados alcanzables satisface el invariante, el mismo es determinado como el estado siguiente al que se transicionará.

A lo sumo uno de los invariantes de los posibles estados alcanzables puede ser satisfecho. Por este motivo, cuando uno de los mismos se satisface no se continúa analizando los restantes estados.

¿Hay más estados alcanzables?

Aquí se verifica si hay estados del conjunto de estados alcanzables que se encuentran sin analizar.

¿Hay más acciones?

Se chequea si hay más acciones en la traza de ejecución para analizar.

Generar gráfico

Finalmente, cuando se llega a la conclusión de que la traza es inválida, se genera un gráfico indicando la subtraza alcanzada.

5.3 Procedimiento de verificación

En esta sección explicaremos los pasos necesarios para realizar la verificación utilizando las herramientas implementadas.

Situación
Se cuenta con una clase robusta, la cual fue implementada por expertos y ampliamente testeada. Se ha implementado un nuevo cliente que hace uso de dicha clase. El objetivo final es verificar si este nuevo cliente hace un uso correcto del protocolo. El nuevo cliente forma parte de un *proyecto* escrito en el lenguaje de programación Java.

Requirimientos iniciales
Es condición necesaria para realizar la verificación contar con el *contrato* de la clase y que la clase tenga implementados los observadores correspondientes a las variables del contrato.

Pasos
1- Agregar una copia de la clase al proyecto cliente.
2- Si los nombres de los métodos de la clase no coinciden con los nombres que figuran en el contrato, es posible ingresar un *archivo de mapeo* entre el nombre del método implementado y el nombre que figura en el contrato. Alternativamente, es posible realizar el mapeo desde la interfaz de la aplicación Instrumentador.
3- Ejecutar la aplicación Instrumentador, ingresando como entrada el contrato junto con el archivo de mapeo (opcional). El Instrumentador generará un archivo *Aspectj* con extensión *.aj*.
4- Agregar el archivo *.aj* al proyecto del cliente.
5- Ejecutar el cliente. Se generarán de manera automática las trazas de ejecución correspondientes.
6- Ejecutar la aplicación *Analyzer*, ingresando como entrada el contrato junto con las trazas de ejecución capturadas en el punto anterior.
7- Analizar el resultado obtenido para cada una de las trazas de ejecución. Analyzer mostrará si la traza es válida o inválida, es decir, si el nuevo cliente implementado cumple o no con el protocolo. En caso de que sea inválida, Analyzer mostrará la mayor subtraza válida.

5.4 Limitaciones

En esta sección mencionaremos algunos comentarios generales relacionados con las herramientas implementadas.

Describiremos algunas limitaciones de la herramienta *Instrumentador* que pueden ser funcionalidades a implementar en una próxima versión del toolkit.

El *Instrumentador* se basa en AspectJ para generar el código que será utilizado para realizar la captura de las trazas de ejecución. Hay un caso especial que no está contemplado que es cuando dentro de un método de la clase se invoca a otro método de la misma clase y ambos son métodos que se desean capturar. Por ejemplo, supongamos que el cliente de la clase *Stack* invoca al

método *push* y que el método *push*, dentro de su implementación, invoca al método *pop*. La traza deseada debería contener sólo la invocación al método *push* y no al método *pop*, dado que desde el punto de vista del cliente, ésta fue la invocación realizada. El cliente visualiza a cada método de la clase como una 'caja negra', por lo tanto las llamadas internas de cada método no deberían ser capturadas. En la versión actual, las llamadas internas se capturan. De todas maneras, este punto no resultó relevante dado que ninguna de las clases con las que experimentamos tenía esta característica.

Otra de las limitaciones de esta versión del toolkit es que no se considera la sobrecarga de operadores en los métodos de la clase del contrato. Es decir, se asume que no hay métodos de una clase con el mismo nombre, o al menos, si los hay, poseen las mismas precondiciones y postcondiciones.

En cuanto al contrato, sólo se acepta que las variables del contrato sean del tipo *integer* o *boolean*. Estos son los tipos de datos empleados en los contratos sobre los cuales se trabajó y se realizaron las pruebas de la herramienta.

6 Experimentos

En esta sección describiremos distintas clases y clientes con los que se ha trabajado con el fin de poder comprobar las distintas funcionalidades de la aplicación *Analyzer*.

Se divide esta sección en cuatro partes. En primer lugar, se describen las distintas problemáticas y situaciones que surgieron durante la etapa de experimentación. Posteriormente, se mencionan las características de las clases analizadas brindando una breve descripción de las mismas, el contrato utilizado y la FSCA generada por *Contractor* para cada una de las clases. Luego, se hace referencia a las experimentaciones realizadas con la fase 1 de la aplicación y finalmente, en las experimentaciones relacionadas con la fase 2.

6.1 Comentarios

Generalmente, antes de comenzar la ejecución de un método de la clase proveedora, se realizan las verificaciones apropiadas sobre las precondiciones del contrato. De esta manera, el método no se ejecuta si la precondición no es satisfecha y por lo tanto, la ejecución finaliza.
Debido a que el chequeo de las precondiciones es responsabilidad del cliente y no de la clase proveedora (desde el punto de vista del diseño por contratos) y que nuestro interés se basa en conocer si el cliente realizó o no una invocación correcta, se eliminaron dichos chequeos dentro de las clases proveedoras que utilizamos durante la experimentación.

En ciertas clases fue necesario modificar la implementación para evitar circunstancias en donde la ejecución finalice con excepciones como por ejemplo, *null pointer*. Esta situación se daba muchas veces como consecuencia de eliminar las verificaciones de la clase.

Con el fin de poder simular trazas de ejecución inválidas, se implementaron clientes que generasen trazas de distintas características para cada fase de verificación.

A continuación, se muestran a modo de ejemplo los casos más relevantes para cada una de las clases con la que se ha experimentado.

6.2 Clases utilizadas

6.2.1 Clase Signature

Descripción

Una de las clases analizadas fue *Signature*. La misma es provista por el JDK (*Java Development Kit*) 1.4. *Signature* es una clase utilizada para proveer a las aplicaciones la funcionalidad del algoritmo de firma digital. Las firmas digitales son utilizadas para autenticación y validación de integridad de datos digitales.

Signature provee las siguientes operaciones:

- *initVerify*: inicializa el objeto para su verificación utilizando la clave pública del certificado dado.
- *initSign*: inicializa el objeto para su firma.

- *sign*: finaliza la operación de firma y almacena los bytes resultantes de la firma en el buffer provisto.
- *verify*: verifica la firma indicada.
- *update*: actualiza los datos a ser firmados o verificados, utilizando el arreglo de bytes especificado.

Contrato

```
<contract name="Signature" invariant="s_state = 0 OR s_state = 2 OR s_state = 3" >
  <variable name="s_state" type="INT" />

  <constructor name="Signature" pre="TRUE" post="s_state' = 0" >
  </constructor>

  <action  name="initVerify" pre="TRUE" post="s_state' = 3" />
  <action  name="initSign" pre="TRUE" post="s_state' = 2" />
  <action  name="sign" pre="s_state = 2" post="TRUE" />
  <action  name="verify" pre="s_state = 3" post="TRUE" />
  <action  name="update" pre="s_state = 3 OR s_state = 2" post="TRUE" />
</contract>
```

FSCA

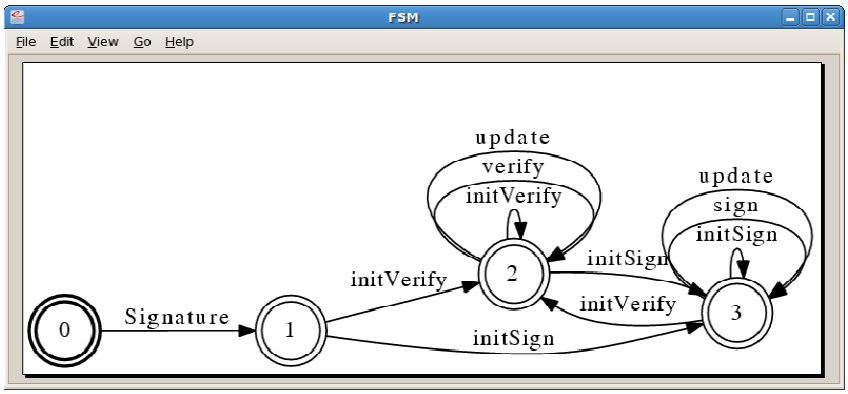

Figura 10. FSCA de la clase Signature.

6.2.2 Clase Circular Buffer

Descripción

La clase *Circular Buffer* es una estructura de datos de tamaño fijo que almacena los elementos en forma circular. Esta clase fue implementada especialmente para este experimento y es la misma que fue utilizada como ejemplo en el capítulo Motivación.

Contrato

Este contrato se corresponde con el de la Figura 1, pero utiliza el formato requerido por *Contractor*.

```
<contract name="CircularReadWriteBuffer" invariant="0  <= rp AND rp  < size AND 0  <= wp AND wp  < size
AND NOT (rp = wp) AND 3  < size" >

  <variable name="rp" type="INT" />
  <variable name="wp" type="INT" />
  <variable name="size" type="INT" />
```

```
  <constructor name="CircularReadWriteBuffer" pre="dim > 3" post="size' = dim AND size' - 1 = rp' AND 0 =
wp'" >
          <parameter name="dim" type="INT" />
  </constructor>

  <action name="write" pre="(wp  < rp - 1) OR (wp  < size - 1 AND rp  < wp) OR (wp = size - 1 AND rp > 0)"
post="(wp  < size - 1 => wp' = wp + 1) AND (wp = size - 1 => wp' = 0)" />

  <action name="read"  pre="(rp  < wp - 1) OR (rp  < size - 1 AND wp  < rp) OR (rp = size - 1 AND wp > 0)"
post="(rp  < size - 1 => rp' = rp + 1) AND (rp = size - 1 => rp' = 0)" />

</contract>
```

FSCA

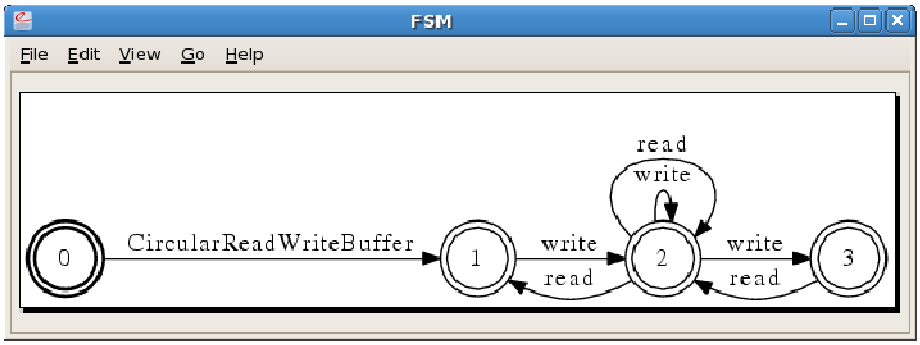

Figura 11. FSCA de la clase Circular Buffer.

6.2.3 Clase Simplified Socket

Descripción

Los *sockets* son un sistema de comunicación entre diferentes procesos. Más exactamente, un *socket* es un punto de comunicación por el cual un proceso puede emitir o recibir información. Utilizan una serie de métodos para establecer el punto de comunicación, conectarse a un determinado puerto, escuchar en él, leer o escribir información y finalmente para desconectarse. La clase *Simplified Socket* es una versión simplificada de la clase *Socket* provista por el JDK 1.4.

Contrato

```
<contract name="SimplifiedSocket" invariant="TRUE">

   <variable name="S_created" type="BOOLEAN" />
   <variable name="S_bound" type="BOOLEAN" />
   <variable name="S_connected" type="BOOLEAN" />
   <variable name="S_closed" type="BOOLEAN" />
   <variable name="S_shutIn" type="BOOLEAN" />
   <variable name="S_shutOut" type="BOOLEAN" />
   <variable name="S_oldImpl" type="BOOLEAN" />

<constructor name="SimplifiedSocket" pre="TRUE" post="NOT S_created' AND NOT S_bound' AND NOT
S_connected' AND NOT S_closed' AND NOT S_shutIn' AND NOT S_shutOut' " />

<action name="connect" pre="NOT S_closed AND NOT(NOT S_oldImpl AND (S_connected OR S_oldImpl))"
post="S_connected' AND S_bound' AND S_created'" />

<action name="bind" pre="NOT (S_closed) AND (S_oldImpl OR NOT(S_bound OR S_oldImpl))"
post="S_bound' AND S_created'" />

<action name="getInputStream" pre="NOT(S_closed) AND (S_connected OR S_oldImpl) AND NOT(S_shutIn)"
post="TRUE" />

<action name="getOutputStream" pre="NOT(S_closed) AND (S_connected OR S_oldImpl) AND
```

```
NOT(S_shutOut) " post="TRUE" />

<action name="close" pre="TRUE" post="S_closed' " />

<action name="shutdownInput" pre="NOT(S_closed) AND (S_connected OR S_oldImpl) AND NOT(S_shutIn)"
post=" S_shutIn' AND S_created' " />

<action name="shutdownOutput" pre="NOT(S_closed) AND (S_connected OR S_oldImpl) AND
NOT(S_shutOut)" post=" S_shutOut' AND S_created'" />

</contract>
```

FSCA

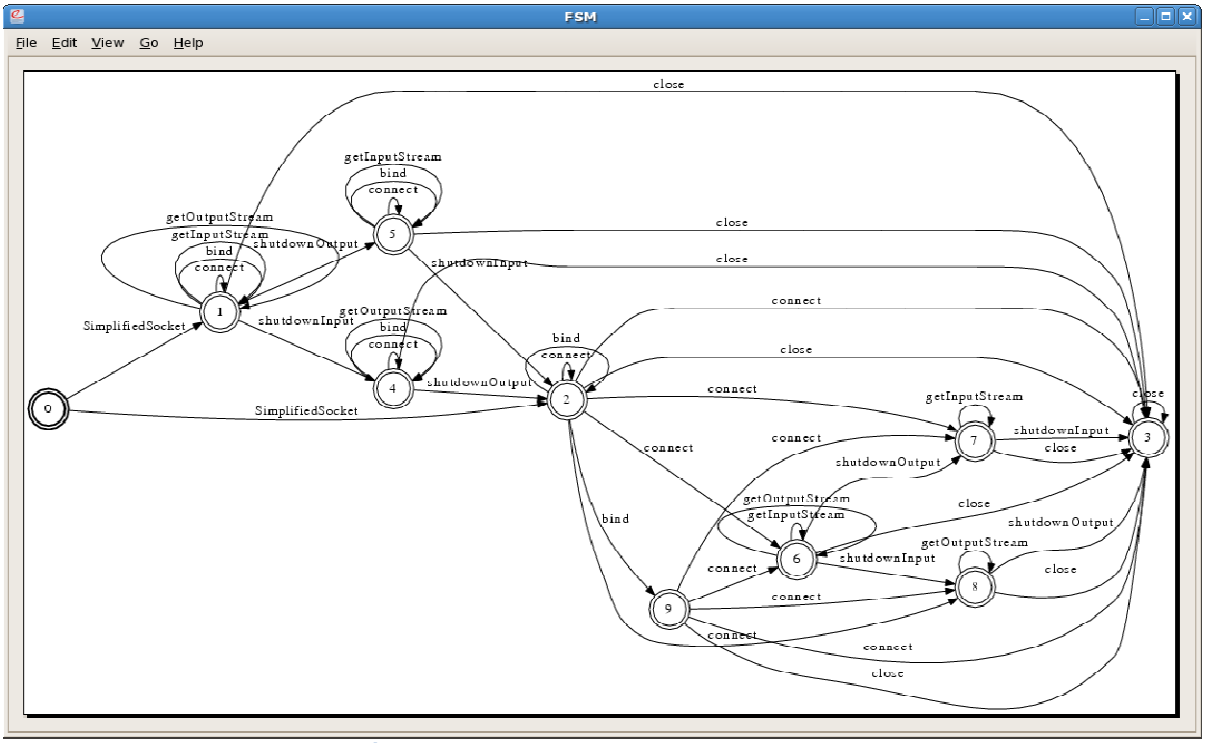

Figura 12. FSCA de la clase Simplified Socket..

6.2.4 Clase Stack

Descripción

La clase *Stack*, implementada especialmente para este experimento, representa una pila (*last-in-first-out*, LIFO) de objetos. Provee las operaciones *push* y *pop* para agregar y tomar un objeto de la pila, respectivamente. Los objetos de esta clase reciben como parámetro un valor que establece el tope o la capacidad máxima del *Stack*.

Contrato

```
<contract name="STACK" invariant=" last_write >= -1 AND  size  > last_write AND size>2">

   <variable name="size" type="INT"/>
   <variable name="last_write" type="INT"/>

   <constructor name="Stack" pre="s>2" post="size'=s AND last_write'= -1">
      <parameter name="s" type="INT"/>
   </constructor>

   <action name="push" pre="size - 1  > last_write" post="last_write' = last_write + 1 ">
     <parameter name="e" type="INT"/>
   </action>
```

```
   <action name="pop" pre="last_write > -1 " post="last_write' = last_write - 1">
   </action>

</contract>
```

FSCA

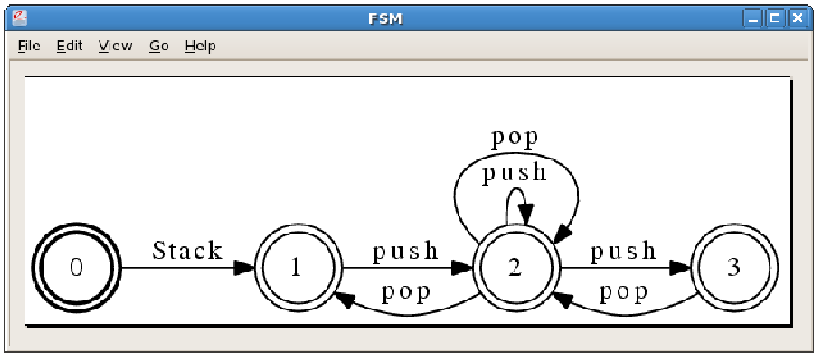

Figura 13. FSCA de la clase Stack.

6.3 Experimentaciones Fase 1

6.3.1 Clase Signature

El cliente elegido utiliza a la clase *Signature* para realizar la firma digital y su posterior verificación.

El objetivo de este experimento fue analizar si la implementación del cliente que utiliza la clase *Signature* respeta el protocolo establecido con la misma.

Para esto, ejecutamos el cliente de la clase *Signature* con el fin de obtener las trazas de ejecución. El cliente implementado se encarga de realizar una firma digital y luego realiza la verificación de la misma utilizando dos claves públicas diferentes.

Si el cliente se encuentra correctamente implementado, el resultado esperado de esta prueba es que puedan pasarse satisfactoriamente las dos fases de análisis que realiza la aplicación *Analyzer*. En caso de que haya un uso incorrecto del protocolo detectable en el cliente, se indicará en cuál de las fases se ha encontrado el problema y en qué traza de ejecución se detectó el mismo.

Dado que el cliente durante su ejecución crea tres objetos distintos de la clase *Signature*, se obtienen tres trazas de ejecución, una por cada objeto creado.

Las trazas de ejecución y los gráficos correspondientes son los que figuran a continuación. En los gráficos (Figuras 13,14 y 15) pueden verse claramente las secuencias de acciones invocadas, mientras que en las trazas de ejecución se puede observar además los valores de las variables del contrato en cada uno de los estados.

```
<root>

<action aname="Signature">
  <variable name="s_state" value="0"/>
</action>
```

```
<action aname="initSign">
  <variable name="s_state" value="2"/>
</action>

<action aname="update">
   <variable name="s_state" value="2"/>
</action>

<action aname="update">
   <variable name="s_state" value="2"/>
</action>

<action aname="sign">
   <variable name="s_state" value="2"/>
</action>

</root>
```

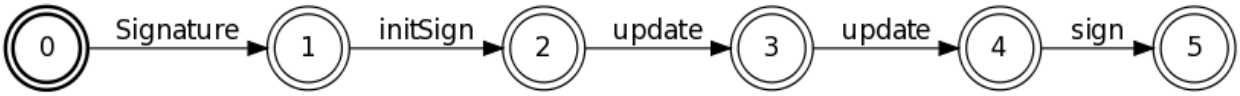

Figura 14. FSM de la traza de ejecución 1.

```
<root>

<action aname="Signature">
  <variable name="s_state" value="0"/>
</action>

<action aname="initVerify">
  <variable name="s_state" value="3"/>
</action>

<action aname="update">
  <variable name="s_state" value="3"/>
</action>

<action aname="update">
  <variable name="s_state" value="3"/>
</action>

<action aname="verify">
  <variable name="s_state" value="3"/>
</action>

</root>
```

Figura 15. FSM de la traza de ejecución 2.

```
<root>

<action aname="Signature">
  <variable name="s_state" value="0"/>
</action>

<action aname="initVerify">
  <variable name="s_state" value="3"/>
</action>

<action aname="update">
  <variable name="s_state" value="3"/>
</action>

<action aname="update">
  <variable name="s_state" value="3"/>
```

```
</action>

<action aname="verify">
  <variable name="s_state" value="3"/>
</action>

</root>
```

Figura 16. FSM de la traza de ejecución 3.

El resultado obtenido en este ejemplo, fue que *Analyzer* ha indicado que las dos fases han sido superadas satisfactoriamente para todas las trazas. Esto quiere decir que la secuencia de acciones de la clase *Signature* invocadas por el cliente son un camino válido en la FSCA de la Figura 10 (o sea, se ha superado la fase 1) y que las condiciones sobre el valor de las variables incluidas en el contrato se ha respetado (o sea, se ha superado la fase 2).

Supongamos ahora que el cliente en lugar de generar la traza de la Figura 14, genera la siguiente traza:

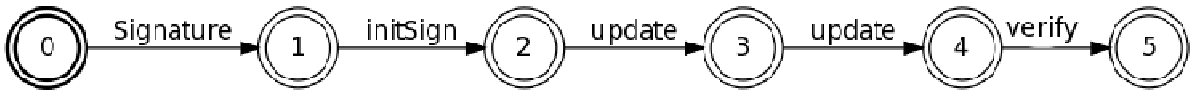

Figura 17. FSM de la traza.

Con esta traza, *Analyzer* indicará que la traza es inválida por no haber superado la fase 1 y mostrará que sólo ha podido avanzar hasta la segunda invocación del método *update*, como puede verse en la Figura 17.

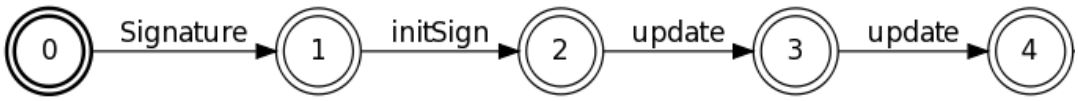

Figura 18. Subtraza válida.

Luego de *update*, el siguiente método invocado fue *verify* el cual no forma un camino válido en la FSCA de la clase (Figura 10).

6.3.2 Clase Circular Buffer

Para realizar experimentaciones con la clase *CircularBuffer*, implementamos un cliente que invoca una operación *read* y posteriormente un *write*. La traza resultante fue la siguiente:

```
<root>

<action aname="CircularReadWriteBuffer">
   <variable name="rp" value="5"/>
   <variable name="wp" value="0"/>
   <variable name="size" value="6"/>
</action>
```

```
<action aname="read">
   <variable name="rp" value="0"/>
   <variable name="wp" value="0"/>
   <variable name="size" value="6"/>
</action>

<action aname="write">
   <variable name="rp" value="0"/>
   <variable name="wp" value="1"/>
   <variable name="size" value="6"/>
</action>

</root>
```

El gráfico correspondiente a la traza es el que se presenta en la Figura 19.

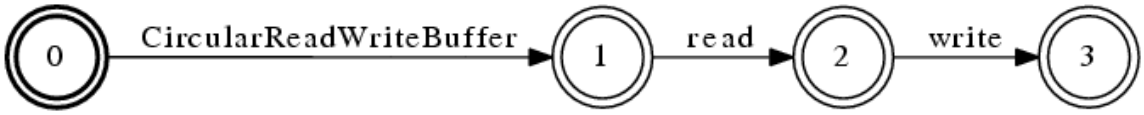

Figura 19. FSM de la traza de ejecución.

Según el protocolo de la clase, no es posible realizar esta secuencia de acciones, por lo tanto *Analyzer* indica que la secuencia es inválida dado que no satisface la fase 1.
En la Figura 20 se presenta la subtraza válida.

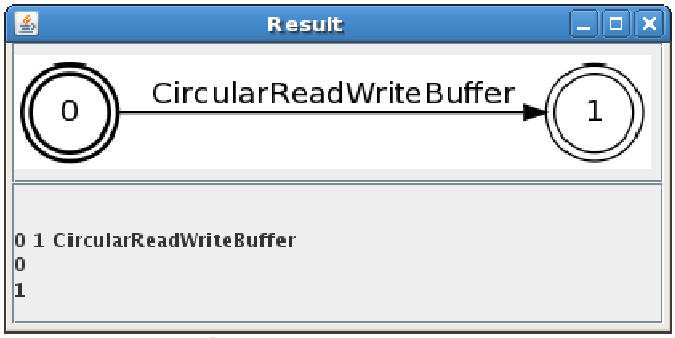

Figura 20. Subtraza válida.

6.3.3 Clase Simplified Socket

El cliente utilizado para las experimentaciones, realiza una conexión al socket de la máquina local en un puerto fijo. Se han realizado modificaciones en el cliente para poder simular usos incorrectos del protocolo y verificar luego si los mismos eran correctamente detectados por *Analyzer*.

Una de las trazas obtenidas fue la siguiente:

```
<root>

<action aname="SimplifiedSocket">
   <variable name="S_created" value="false"/>
   <variable name="S_bound" value="false"/>
   <variable name="S_connected" value="false"/>
   <variable name="S_closed" value="false"/>
```

```
   <variable name="S_shutIn" value="false"/>
   <variable name="S_shutOut" value="false"/>
   <variable name="S_oldImpl" value="false"/>
</action>

<action aname="bind">
   <variable name="S_created" value="true"/>
   <variable name="S_bound" value="true"/>
   <variable name="S_connected" value="false"/>
   <variable name="S_closed" value="false"/>
   <variable name="S_shutIn" value="false"/>
   <variable name="S_shutOut" value="false"/>
   <variable name="S_oldImpl" value="false"/>
</action>

<action aname="connect">
   <variable name="S_created" value="true"/>
   <variable name="S_bound" value="true"/>
   <variable name="S_connected" value="true"/>
   <variable name="S_closed" value="false"/>
   <variable name="S_shutIn" value="false"/>
   <variable name="S_shutOut" value="false"/>
   <variable name="S_oldImpl" value="false"/>
</action>

<action aname="close">
   <variable name="S_created" value="true"/>
   <variable name="S_bound" value="true"/>
   <variable name="S_connected" value="true"/>
   <variable name="S_closed" value="true"/>
   <variable name="S_shutIn" value="false"/>
   <variable name="S_shutOut" value="false"/>
   <variable name="S_oldImpl" value="false"/>
</action>

<action aname="getInputStream">
   <variable name="S_created" value="true"/>
   <variable name="S_bound" value="true"/>
   <variable name="S_connected" value="true"/>
   <variable name="S_closed" value="false"/>
   <variable name="S_shutIn" value="false"/>
   <variable name="S_shutOut" value="false"/>
   <variable name="S_oldImpl" value="false"/>
</action>

<action aname="getOutputStream">
   <variable name="S_created" value="true"/>
   <variable name="S_bound" value="true"/>
   <variable name="S_connected" value="true"/>
   <variable name="S_closed" value="false"/>
   <variable name="S_shutIn" value="false"/>
   <variable name="S_shutOut" value="false"/>
   <variable name="S_oldImpl" value="false"/>
</action>

</root>
```

El gráfico correspondiente a esta traza de ejecución es el que se presenta en la Figura 21.

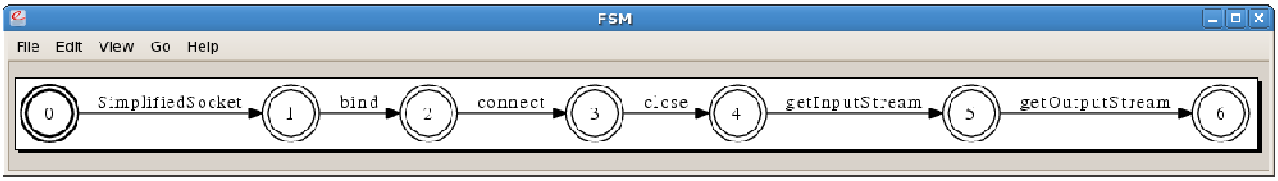

Figura 21. FSM de la traza de ejecución.

Esta traza de ejecución es considerada inválida dado que luego de ejecutar el método *close* se ejecutó el método *getInputStream* y esta secuencia no es considerada válida según la FSCA de la clase.

Por lo tanto, la traza que pudo ser consumida fue hasta el método *close* (Figura 22).

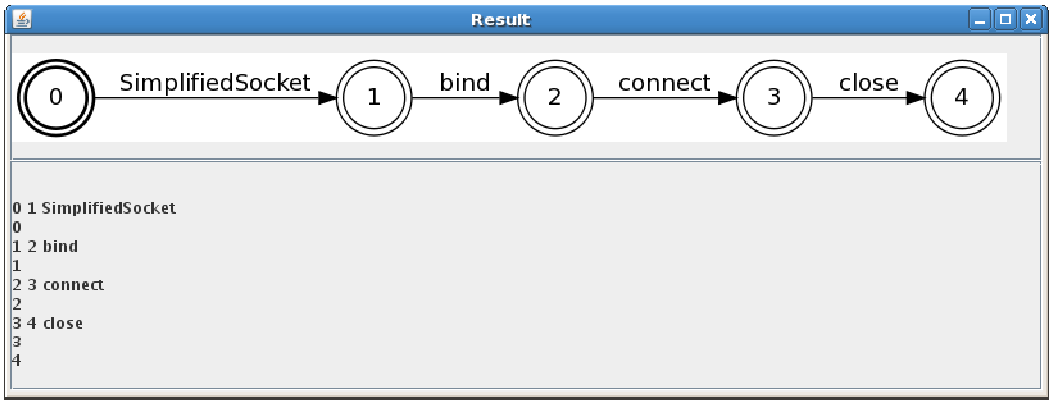

Figura 22. Subtraza válida.

Otra de las trazas analizadas fue la siguiente:

```
<root>

<action aname="SimplifiedSocket">
   <variable name="S_created" value="false"/>
   <variable name="S_bound" value="false"/>
   <variable name="S_connected" value="false"/>
   <variable name="S_closed" value="false"/>
   <variable name="S_shutIn" value="false"/>
   <variable name="S_shutOut" value="false"/>
   <variable name="S_oldImpl" value="false"/>
</action>

<action aname="bind">
   <variable name="S_created" value="true"/>
   <variable name="S_bound" value="true"/>
   <variable name="S_connected" value="false"/>
   <variable name="S_closed" value="false"/>
   <variable name="S_shutIn" value="false"/>
   <variable name="S_shutOut" value="false"/>
   <variable name="S_oldImpl" value="false"/>
</action>

<action aname="connect">
   <variable name="S_created" value="true"/>
   <variable name="S_bound" value="true"/>
   <variable name="S_connected" value="true"/>
   <variable name="S_closed" value="false"/>
```

```
   <variable name="S_shutIn" value="false"/>
   <variable name="S_shutOut" value="false"/>
   <variable name="S_oldImpl" value="false"/>
</action>

<action aname="shutdownInput">
   <variable name="S_created" value="true"/>
   <variable name="S_bound" value="true"/>
   <variable name="S_connected" value="true"/>
   <variable name="S_closed" value="false"/>
   <variable name="S_shutIn" value="true"/>
   <variable name="S_shutOut" value="false"/>
   <variable name="S_oldImpl" value="false"/>
</action>

<action aname="shutdownInput">
   <variable name="S_created" value="true"/>
   <variable name="S_bound" value="true"/>
   <variable name="S_connected" value="true"/>
   <variable name="S_closed" value="false"/>
   <variable name="S_shutIn" value="true"/>
   <variable name="S_shutOut" value="false"/>
   <variable name="S_oldImpl" value="false"/>
</action>

<action aname="close">
   <variable name="S_created" value="true"/>
   <variable name="S_bound" value="true"/>
   <variable name="S_connected" value="true"/>
   <variable name="S_closed" value="true"/>
   <variable name="S_shutIn" value="true"/>
   <variable name="S_shutOut" value="false"/>
   <variable name="S_oldImpl" value="false"/>
</action>

</root>
```

El gráfico correspondiente a la traza es el de la Figura 23.

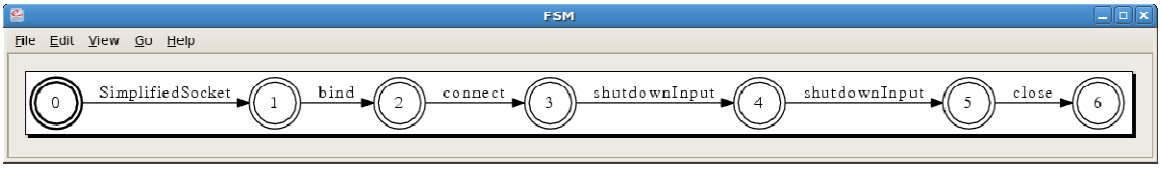

Figura 23. FSM de la traza de ejecución.

Esta traza es inválida dado que no es posible realizar dos operaciones *shutdownInput* en forma consecutiva.
La traza que pudo ser consumida fue hasta la primera invocación del método *shutdownInput* (Figura 24).

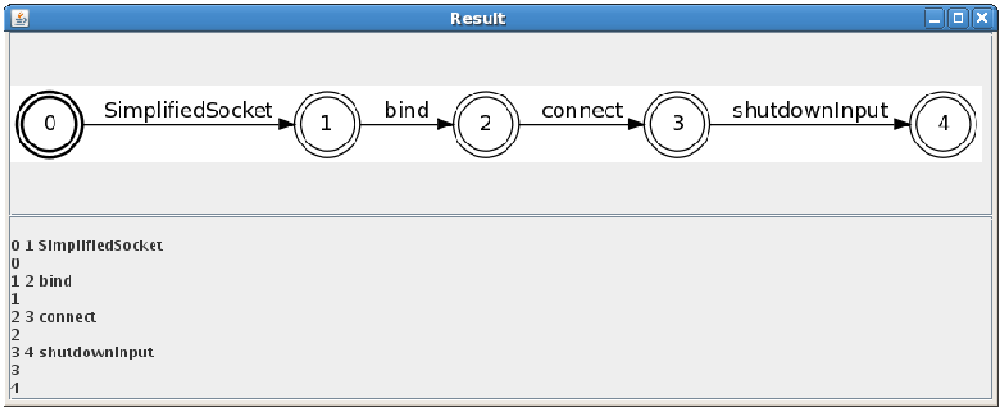

Figura 24. Subtraza válida.

Otra traza que también es considerada inválida en la fase 1 es la siguiente:

```
<root>

<action aname="SimplifiedSocket">
   <variable name="S_created" value="false"/>
   <variable name="S_bound" value="false"/>
   <variable name="S_connected" value="false"/>
   <variable name="S_closed" value="false"/>
   <variable name="S_shutIn" value="false"/>
   <variable name="S_shutOut" value="false"/>
   <variable name="S_oldImpl" value="false"/>
</action>

<action aname="bind">
   <variable name="S_created" value="true"/>
   <variable name="S_bound" value="true"/>
   <variable name="S_connected" value="false"/>
   <variable name="S_closed" value="false"/>
   <variable name="S_shutIn" value="false"/>
   <variable name="S_shutOut" value="false"/>
   <variable name="S_oldImpl" value="false"/>
</action>

<action aname="connect">
   <variable name="S_created" value="true"/>
   <variable name="S_bound" value="true"/>
   <variable name="S_connected" value="true"/>
   <variable name="S_closed" value="false"/>
   <variable name="S_shutIn" value="false"/>
   <variable name="S_shutOut" value="false"/>
   <variable name="S_oldImpl" value="false"/>
</action>

<action aname="shutdownInput">
   <variable name="S_created" value="true"/>
   <variable name="S_bound" value="true"/>
   <variable name="S_connected" value="true"/>
   <variable name="S_closed" value="false"/>
   <variable name="S_shutIn" value="true"/>
   <variable name="S_shutOut" value="false"/>
   <variable name="S_oldImpl" value="false"/>
</action>

<action aname="shutdownOutput">
   <variable name="S_created" value="true"/>
   <variable name="S_bound" value="true"/>
   <variable name="S_connected" value="true"/>
```

```
   <variable name="S_closed" value="false"/>
   <variable name="S_shutIn" value="true"/>
   <variable name="S_shutOut" value="true"/>
   <variable name="S_oldImpl" value="false"/>
</action>

<action aname="getInputStream">
   <variable name="S_created" value="true"/>
   <variable name="S_bound" value="true"/>
   <variable name="S_connected" value="true"/>
   <variable name="S_closed" value="false"/>
   <variable name="S_shutIn" value="false"/>
   <variable name="S_shutOut" value="true"/>
   <variable name="S_oldImpl" value="true"/>
</action>

<action aname="close">
   <variable name="S_created" value="true"/>
   <variable name="S_bound" value="true"/>
   <variable name="S_connected" value="true"/>
   <variable name="S_closed" value="true"/>
   <variable name="S_shutIn" value="false"/>
   <variable name="S_shutOut" value="false"/>
   <variable name="S_oldImpl" value="false"/>
</action>

</root>
```

El gráfico correspondiente a la traza es el de la Figura 25.

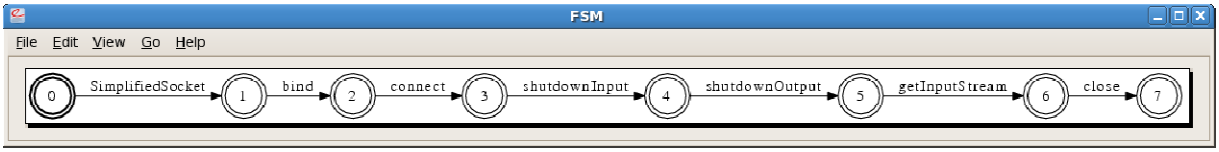

Figura 25. FSM de la traza de ejecución.

No es posible, según la FSCA de la clase, invocar al método *getInputStream* luego de haber invocado a *shutdownInput*. Por lo tanto, la cadena consumida ha sido hasta la acción previa de la invocación de *getInputStream* (Figura 26):

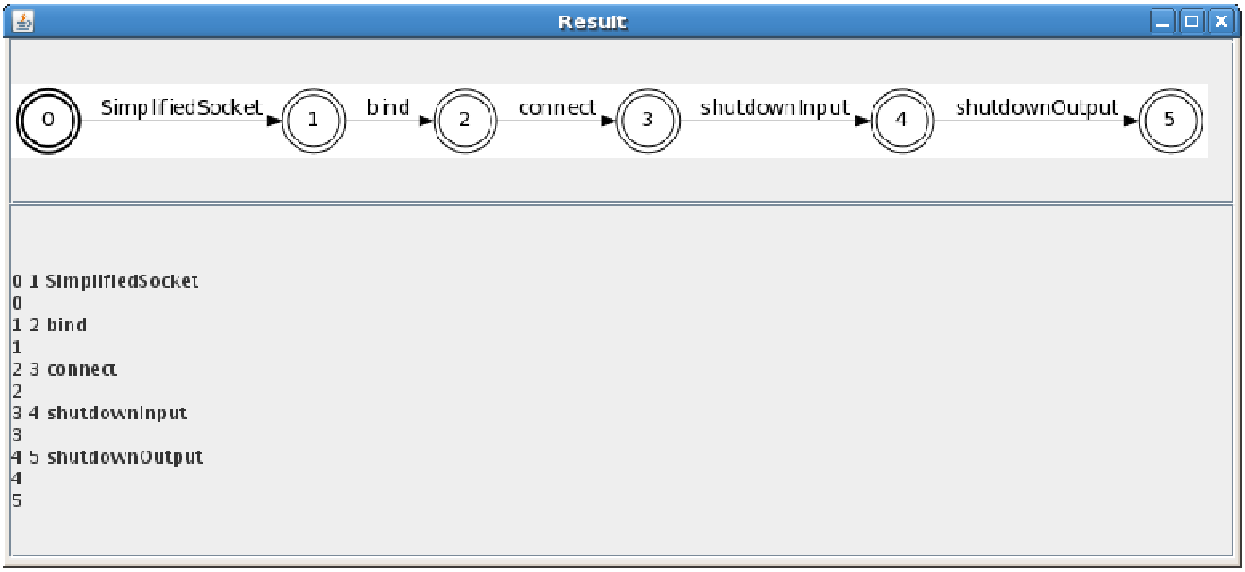

Figura 26. Subtraza válida.

6.4 Experimentaciones Fase 2

6.4.1 Clase Signature

Con respecto a la clase *Signature*, la FSCA relacionada es una máquina de estado determinística. Esto quiere decir que hay un sólo movimiento posible para cada acción ejecutada. Debido a esta característica no se podrán detectar en la fase 2 más usos incorrectos por parte del cliente que los detectados en la fase 1.

6.4.2 Clase Circular Buffer

La primera prueba consistió en implementar un cliente que invoque tres operaciones de lectura y luego cuatro operaciones de lectura. Dado que la FSCA asociada a la clase es no determinística (Figura 11), durante la fase 2 de la verificación se analizará cada uno de los caminos posibles teniendo en cuenta cuál es el adecuado verificando el valor de las variables en cada typestate.

La traza capturada para este ejemplo fue la siguiente:

```
<root>

<action aname="CircularReadWriteBuffer">
   <variable name="rp" value="5"/>
   <variable name="wp" value="0"/>
   <variable name="size" value="6"/>
</action>

<action aname="write">
   <variable name="rp" value="5"/>
   <variable name="wp" value="1"/>
   <variable name="size" value="6"/>
</action>

<action aname="write">
   <variable name="rp" value="5"/>
   <variable name="wp" value="2"/>
   <variable name="size" value="6"/>
</action>

<action aname="write">
   <variable name="rp" value="5"/>
   <variable name="wp" value="3"/>
   <variable name="size" value="6"/>
</action>

<action aname="read">
   <variable name="rp" value="0"/>
   <variable name="wp" value="3"/>
   <variable name="size" value="6"/>
</action>

<action aname="read">
   <variable name="rp" value="1"/>
   <variable name="wp" value="3"/>
   <variable name="size" value="6"/>
</action>
```

```
<action aname="read">
   <variable name="rp" value="2"/>
   <variable name="wp" value="3"/>
   <variable name="size" value="6"/>
</action>

<action aname="read">
   <variable name="rp" value="3"/>
   <variable name="wp" value="3"/>
   <variable name="size" value="6"/>
</action>

</root>
```

Cuando la aplicación *Analyzer* realiza la verificación de la fase 2, detecta que no se satisface el invariante del typestate en el último *read* (Figura 27), e indica que la secuencia es inválida

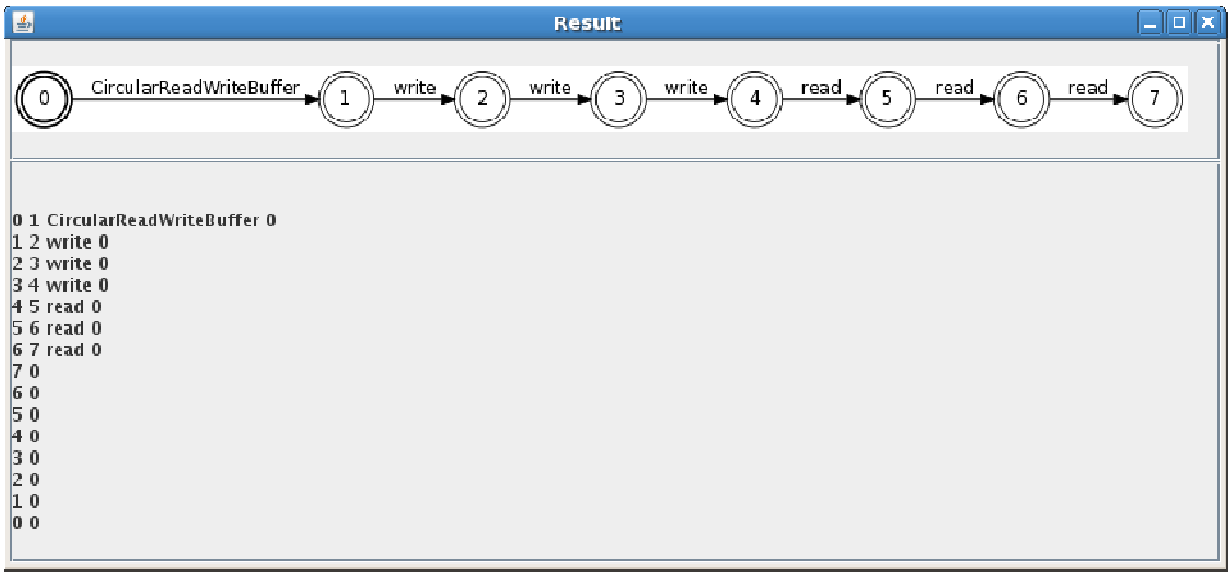

Figura 27. Subtraza válida.

6.4.3 Clase Simplified Socket

Al modificar el cliente de esta clase, se capturó la siguiente traza:

```
<root>

<action aname="SimplifiedSocket">
   <variable name="S_created" value="false"/>
   <variable name="S_bound" value="false"/>
   <variable name="S_connected" value="false"/>
   <variable name="S_closed" value="false"/>
   <variable name="S_shutIn" value="false"/>
   <variable name="S_shutOut" value="false"/>
   <variable name="S_oldImpl" value="false"/>
</action>

<action aname="bind">
   <variable name="S_created" value="true"/>
   <variable name="S_bound" value="true"/>
   <variable name="S_connected" value="false"/>
   <variable name="S_closed" value="false"/>
```

```
   <variable name="S_shutIn" value="false"/>
   <variable name="S_shutOut" value="false"/>
   <variable name="S_oldImpl" value="false"/>
</action>

<action aname="getInputStream">
   <variable name="S_created" value="true"/>
   <variable name="S_bound" value="true"/>
   <variable name="S_connected" value="false"/>
   <variable name="S_closed" value="false"/>
   <variable name="S_shutIn" value="false"/>
   <variable name="S_shutOut" value="false"/>
   <variable name="S_oldImpl" value="false"/>
</action>

<action aname="close">
   <variable name="S_created" value="true"/>
   <variable name="S_bound" value="true"/>
   <variable name="S_connected" value="false"/>
   <variable name="S_closed" value="true"/>
   <variable name="S_shutIn" value="false"/>
   <variable name="S_shutOut" value="false"/>
   <variable name="S_oldImpl" value="false"/>
</action>

</root>
```

El gráfico correspondiente a la traza es el de la Figura 28.

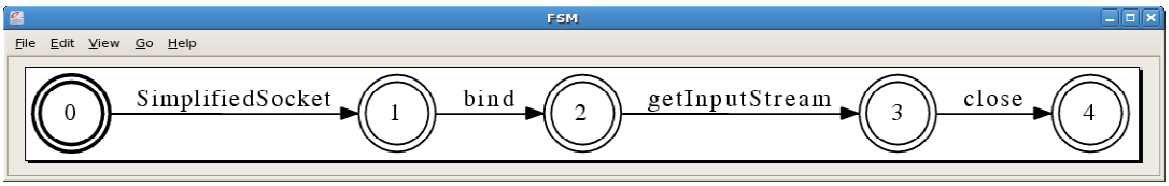

Figura 28. FSM de la traza de ejecución.

El cliente ha invocado al método *getInputStream* sin haber realizado previamente la invocación a *connect*. Si bien esta es una cadena válida según la FSCA, es inválida en el análisis de la fase 2 dado que es necesario que el valor de la variable *S_connected* se encuentre en *true* antes de invocar a *getInputStream.* La cadena consumida fue la de la Figura 29.

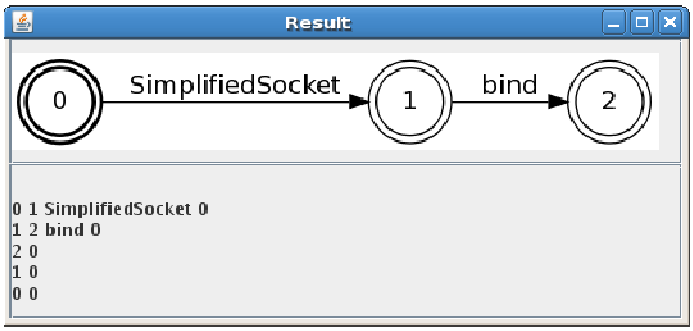

Figura 29. Subtraza válida.

6.4.4 Clase Stack

El objetivo de esta prueba fue analizar el comportamiento del cliente cuando se supera la capacidad máxima del *Stack*.

Al ejecutar el cliente, la traza capturada fue:

```
<root>

<action aname="Stack">
   <variable name="size" value="5"/>
   <variable name="last_write" value="-1"/>
</action>

<action aname="push">
   <variable name="size" value="5"/>
   <variable name="last_write" value="0"/>
</action>

<action aname="push">
   <variable name="size" value="5"/>
   <variable name="last_write" value="1"/>
</action>

<action aname="push">
   <variable name="size" value="5"/>
   <variable name="last_write" value="2"/>
</action>

<action aname="push">
   <variable name="size" value="5"/>
   <variable name="last_write" value="3"/>
</action>

<action aname="push">
   <variable name="size" value="5"/>
   <variable name="last_write" value="4"/>
</action>

<action aname="push">
   <variable name="size" value="5"/>
   <variable name="last_write" value="5"/>
</action>

<action aname="pop">
   <variable name="size" value="5"/>
   <variable name="last_write" value="4"/>
</action>

<action aname="pop">
   <variable name="size" value="5"/>
   <variable name="last_write" value="3"/>
</action>

</root>
```

El resultado obtenido fue que *Analyzer* indicó en la fase 2 que el cliente no respetó el protocolo establecido con la clase *Stack* dado que está tratando de apilar 6 elementos cuando se había establecido una capacidad máxima de 5 elementos. De esta manera la máxima traza válida es la de la Figura 30.

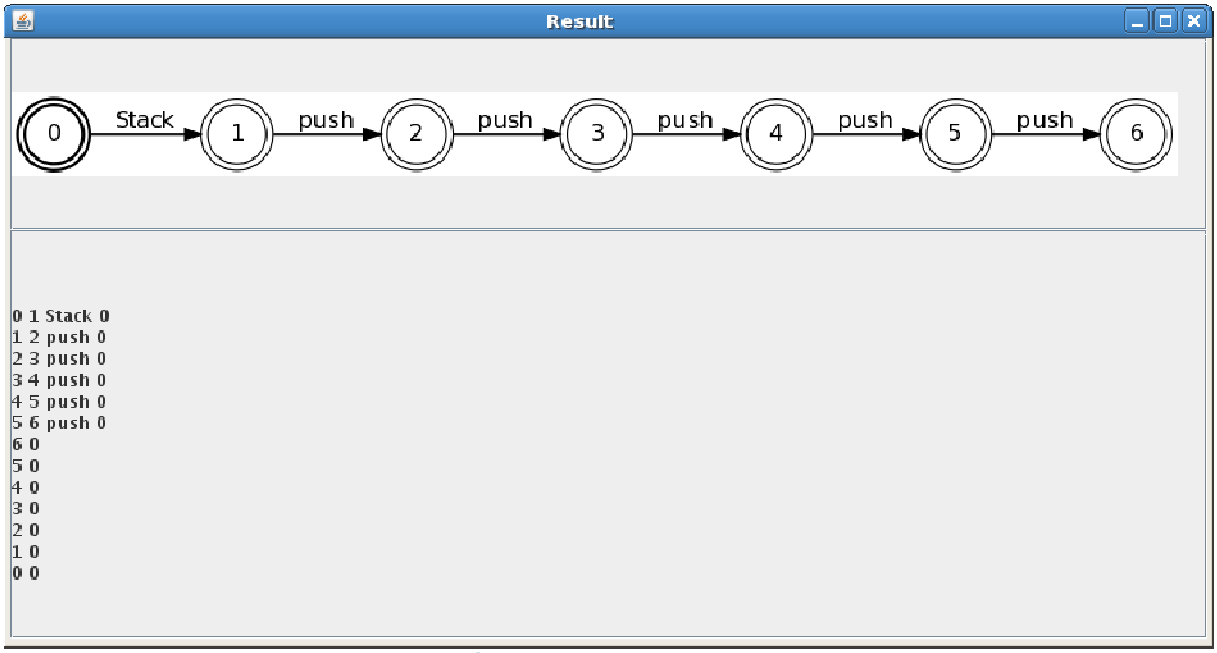

Figura 30. Subtraza válida.

6.5 Análisis de performance

Describiremos a continuación cómo impacta en la performance de la herramienta *Analyzer* el hecho de procesar trazas cada vez de mayor longitud.
Para esto, definimos primero el concepto de longitud de traza. Sea *T* una traza de ejecución, llamaremos *longitud L de la traza de ejecución T* a la cantidad de acciones de *T*.

En el siguiente gráfico se muestra el tiempo alcanzado para cada longitud de traza analizada. Los datos corresponden a trazas válidas simuladas para la clase *Signature* utilizando una máquina virtual con 3GB de memoria RAM sobre un procesador Intel COREi5.

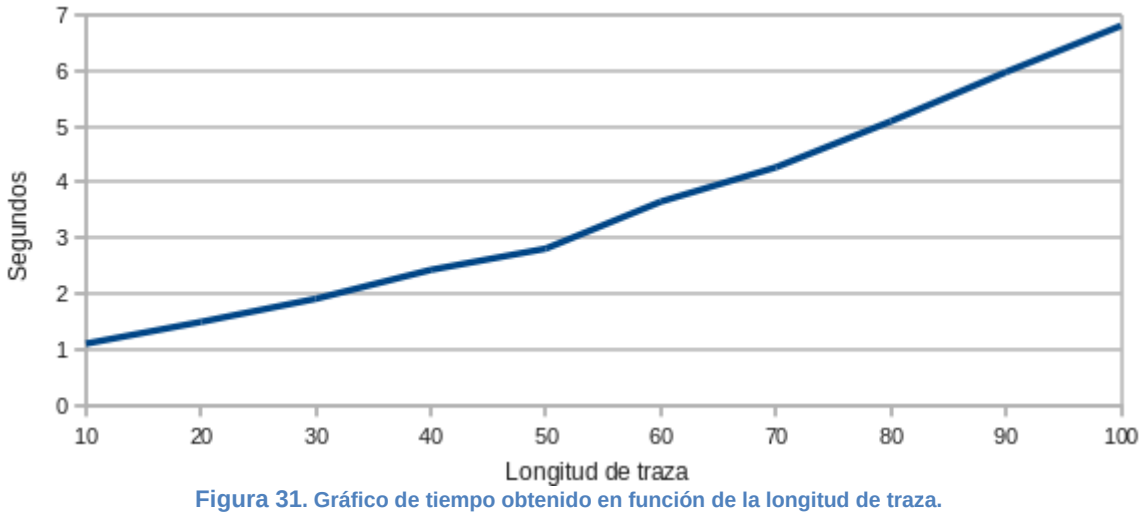

Figura 31. Gráfico de tiempo obtenido en función de la longitud de traza.

Longitud de traza (L)	*Tiempo en segundos*
10	1,099
20	1,489
30	1,905
40	2,424
50	2,806
60	3.649
70	4,265
80	5,092
90	5,975
100	6,807

Figura 32. Tabla de tiempo obtenido en función de la longitud de traza.

A continuación se presentan los valores de los tiempos obtenidos utilizando la clase *Circular Buffer*.

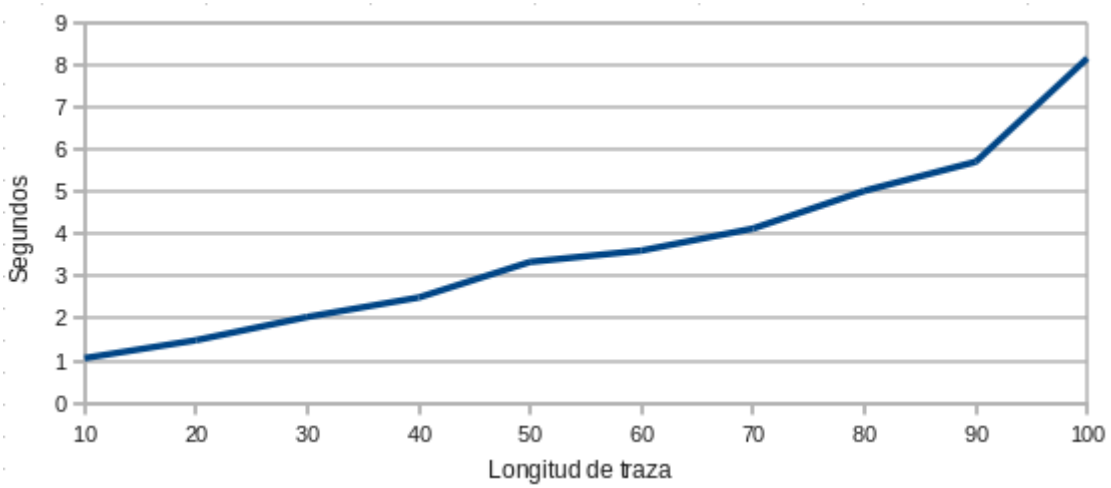

Figura 33. Gráfico de tiempo obtenido en función de la longitud de traza.

Longitud de traza (L)	*Tiempo en segundos*
10	1,061
20	1,485
30	2,036
40	2,498
50	3,340
60	3,609

70	4,130
80	5,022
90	5,711
100	8,162

Figura 34. Tabla de tiempo obtenido en función de la longitud de traza.

Como puede verse, a medida que se incrementa la cantidad de acciones de la traza de ejecución, se incrementa el tiempo de análisis. De todas maneras, el incremento del tiempo de ejecución es de forma lineal y en tiempos considerables.

7 Trabajos Relacionados

Los factores de calidad de software en los que generalmente se pone foco suelen ser la *reutilización*, *escalabilidad* y *compatibilidad*, pero no debe descuidarse la *confiabilidad* (correctitud y robustez) en base a ellos.

Las primeras técnicas para analizar la correctitud de un programa se remontan a finales de 1960 con los trabajos de Floyd [Flo67] y Hoare [Hoa69], donde se proponen metodologías para especificar las condiciones que deben cumplir las distintas porciones de software.

Una de las herramientas básicas para colaborar en la confiabilidad de un producto de software es la incorporación de *aserciones*. Una aserción es una expresión que involucra ciertas entidades del software y establece propiedades que las mismas deben satisfacer en ciertos puntos de la ejecución del software [Mey97].

Se han desarrollado lenguajes de programación que contienen especificaciones embebidas que son utilizadas como propiedades que garantizan la correcta ejecución de una porción de software. Un ejemplo de este tipo de lenguajes es *Eiffel* [Mey97]. Eiffel es un lenguaje de programación orientado a objetos que permite que los programas sean ampliados con anotaciones para precondiciones y postcondiciones. Las mismas son comprobadas en tiempo de ejecución, verificando dinámicamente la correctitud de un programa mientras se está ejecutando. Este modo de funcionamiento tiene como desventaja que no es posible garantizar previamente que una ejecución no vaya a violar un contrato.

Otro lenguaje que contempla la verificación es el lenguaje multiparadigma *D* de programación [Bri02]. Este lenguaje permite integrar los contratos directamente en el código del programa.

JML (*Java Modeling Language*) es un lenguaje de especificación de interfaces para Java. JML combina la practicidad de los lenguajes DBC (*Desing by Contract*) como Eiffel con la expresividad y formalidad de los lenguajes de especificación orientados a modelos [LC06]. A diferencia de Eiffel, JML es agregado sobre un lenguaje de programación preexistente, por este motivo se utilizan anotaciones en forma de comentarios especiales para indicar las precondiciones y postcondiciones. Alrededor de JML surgieron varias herramientas de análisis estático, como por ejemplo *ESC/Java2*, y además herramientas de análisis dinámico como por ejemplo *JML RAC*.

Otro lenguaje destacado es Spec#. Spec# consiste no sólo en un lenguaje y un compilador sino también en un verificador automático de programas llamado Boogie, el cual verifica la especificación de manera estática. Para verificar si un programa es correcto, se utiliza un lenguaje intermedio llamado Boogie-PL. [DL05]

Otro trabajo relacionado es el toolkit llamado *SLAM*. SLAM tiene como objetivo verificar si una aplicación de software escrita en lenguaje C satisface propiedades críticas en las interfaces que utiliza y colabora con los ingenieros de software en el diseño para asegurar la confiabilidad y

correctitud en su funcionamiento [BTRS02]. Este análisis permite que se detecten errores en etapas tempranas del proceso de codificación.
El proceso de verificación requiere que el usuario sólo declare la propiedad que desea verificar. Luego, se crea una abstracción de manera automática basada en dicha propiedad y se refina de manera interactiva. Para esto se utiliza un lenguaje de especificación llamado SLIC (*Specification Language for Interface Checking*).
SLAM está compuesto por un verificador de modelos (*model checker*), una herramienta de abstracción de predicados y una herramienta de descubrimiento de predicados.

En resumen, existe una gran variedad de herramientas para realizar tanto el análisis estático como el análisis dinámico del cumplimiento de un contrato. Nuestra propuesta se ubica dentro del conjunto de herramientas dinámicas. Tenemos como objetivo verificar dinámicamente la preservación de un *typestate* de los objetos de una clase determinada por parte de un cliente.

Los typestates permiten especificar propiedades de los objetos adicionales a los *types* de un lenguaje de programación [DF04]. Los typestates capturan los aspectos del estado de un objeto.

Cuando el estado de un objeto cambia, su typestate también lo hace. Mientras que el *type* de un objeto determina el conjunto de operaciones permitidas, el *typestate* determina el subconjunto de operaciones permitidas en un contexto en particular [STR86].
Los typestates sirven como técnica para realizar el análisis de un programa, aumentando la confiabilidad al detectar secuencias de ejecución inválidas semánticamente, pero válidas sintácticamente.

Una de las herramientas para la construcción de un modelo de validación basado en typestates es *Contractor*. El typestate inferido por *Contractor* es *enabledness preserving*, brindando un nivel de abstracción adecuado para realizar de manera exitosa validaciones sobre artefactos de software.

Para poder analizar los distintos typestates es necesario agregar monitores en el software para que se realice la captura. Lo ideal es que la instrumentación impacte lo menos posible en la performance del sistema a analizar. Hay herramientas que se encargan mediante un análisis estático, de reducir el *overhead* causado por la instrumentación a través de varias fases de análisis [BHL07]. Otra propuesta abarca un análisis híbrido de los typestates, aprovechando la información calculada en el análisis estático para reformular el problema original en un análisis dinámico *residual* de typestates [DP07]. Este análisis dinámico residual permite reducir significativamente los costos del monitoreo en runtime.

La forma de realizar el monitoreo de los typestates en nuestra propuesta no causa un impacto perceptible en las clases con las que experimentamos, por lo que no fue necesario poner foco en el overhead adicionado. Esto se debe a que en tiempo de runtime solamente se focaliza en la recolección de datos y el análisis propiamente se realiza de manera offline.

8 Conclusiones

Iniciamos este trabajo partiendo de la necesidad de contar con una herramienta que facilitase la verificación del software de manera automática.

Para esto, nos basamos en *Contractor* y en su teoría de *modelos de habilitación* subyacente para realizar el toolkit que se utiliza para verificar si los clientes de una clase determinada realizan un correcto uso del protocolo establecido con la misma.

La utilización de diferentes aplicaciones (*Contractor*, *AT&T FSM Library*, *CVC3*) permitieron que en forma colaborativa se pueda alcanzar el objetivo inicial. Asimismo, la Programación Orientada a Aspectos nos brindó una herramienta importante para poder ingresar una actividad de logueo sin invadir el código ya existente del cliente, dado que el código generado queda aislado en una nueva clase.

Gracias a sus dos fases de verificación, se detectan incumplimientos en el cliente relacionados con la habilitación de las acciones en un momento determinado de la ejecución.
La experimentación con diferentes clases y clientes permite concluir que la herramienta *Analyzer* detecta usos incorrectos del protocolo por parte del cliente de una clase en base a un modelo EPA. Dado que un modelo EPA es una abstracción del LTS, hay usos incorrectos del protocolo que quedan fuera del alcance de este método de verificación.

Debido a que los tiempos de verificación no son elevados, se puede considerar un número alto de trazas a analizar. A mayor cantidad de trazas analizadas, mayor la posibilidad de detectar usos incorrectos del protocolo.

La principal ventaja de poder detectar este tipo de errores es permitir al programador conocer dónde se produce esta falla, ya que puede visualizar mediante un grafo la secuencia de acciones que respetan el protocolo previamente al incumplimiento. Adicionalmente, al realizarse la verificación de manera *offline* se tiene un bajo impacto en la performance del cliente.

8.1 Trabajo a futuro

A partir de *Analyzer*, hay varias funcionalidades adicionales que pueden ser agregadas en una versión posterior. Una de ellas consiste en brindar mayor nivel de detalle acerca de dónde y por qué se detectó la violación del protocolo por parte del cliente. Cuanto más detallada sea esta información, más fácil será para el programador localizar en el código fuente la falla y modificar el mismo.

La herramienta *Instrumentador* permite instrumentar clientes escritos en lenguaje *Java*. Ampliar el conjunto de lenguajes de programación que puedan ser instrumentados es otra de las funcionalidades a incorporar.

9 Apéndice

Código fuente de la clase Circular Buffer en lenguaje Java con los observadores agregados

```
public class CircularReadWriteBuffer {

	private String buffer[];
	private int w;
	private int r;
	private int size = 5;

	public CircularReadWriteBuffer() {
		buffer = new String[size];
		w = 0;
		r = 4;
	}

	public String getrp() {
		return String.valueOf(r);
	}

	public String getwp() {
		return String.valueOf(w);
	}

	public String getsize() {
		return String.valueOf(size);
	}

	public void write(String e) {
		buffer[w] = e;
		w = (w + 1) % buffer.length;
	}

	public String read() {
		r = (r + 1) % buffer.length;
		return buffer[r];
	}

}
```

10 Bibliografía

[BBKL10] Ball, Thoma; Bounimova, Ella; Levin, Vladimir; Kumar, Rahul y Lichtenberg, Jakob. *The Static Driver Verifier Research Platform*, 2010.

[BHL07] Bodden, E.; Hendren, L.J.; Lhoták, O. *A Staged Static Program Analysis to Improve the Performance of Runtime Monitoring*. In Proceedings of ECOOP. 2007, 525-549.

[BT07] Barrett, Clark; Tinelli, Cesare. *CVC3*. In Werner Damm and Holger Hermanns, editors, Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07), volume 4590 of Lecture Notes in Computer Science, pages 298-302. Springer, July 2007. Berlin, Germany. http://cs.nyu.edu/acsys/cvc3/index.html

[BTRS02] Ball, Thomas; Rajamani, Sriram K. *The Slam Project: Debugging System Software via Static Analysis*, 2002

[CBGU] de Caso, Guido; Braberman, Víctor; Garbervetsky, Diego; Uchitel, Sebastián. *Contractor*, http://lafhis.dc.uba.ar/contractor/

[CBGU09] de Caso, Guido; Braberman, Víctor; Garbervetsky, Diego; Uchitel, Sebastián. *Validation of Contracts using Enabledness Preserving Finite State Abstractions.* 31st IEEE/ACM International Conference on Software Engineering (ICSE), páginas 452-462, mayo 2009.

[CBGU11] de Caso, Guido; Braberman, Víctor; Garbervetsky, Diego; Uchitel, Sebastián. *Program abstractions for behavior validation*. International Conference on Software Engineering (ICSE), páginas 381-390, mayo 2011.

[CoCo77] Cousot, P.; Cousot, R. *Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints*. In POPL 77: Principles of Programming Languages, pages 238 252. ACM, 1977.

[DF04] R. DeLine and M. Fahndrich. *Typestates for Objects*. Ecoop 2004-Object-Oriented Programming: 18th European Conference, Oslo, Norway, June, 2004: Proceedings, 2004.

[DP07] Dwyer, M.B.; Purandare, R. *Residual dynamic typestate analysis: Exploiting static analysis results to reformulate and reduce the cost of dynamic analysis*. In: International Conference on Automated Software Engineering (ASE). pp. 124–133. ACM Press (May 2007)

[Flo67] Floyd, R. W. *Assigning meanings to programs*. In J. T. Schwartz, editor, Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19, pages 1932, Providence, 1967. American Mathematical Society.

[FPB75] Jr. Fred P. Brooks. *The mythical man-month*. In Proceedings of the international conference on Reliable software, page 193, New York, NY, USA, 1975. ACM Press.

[GLB75] Good, Donald I.; London, Ralph L.; Bledsoe , W. W. *An interactive program verication system.* In Proceedings of the international conference on Reliable software, pages 482492, New York, NY, USA, 1975. ACM Press.

[Hoa69] Hoare, C. A. R. *An axiomatic basis for computer programming*. Commun. ACM, 12(10):576580, 1969.

[HWSB] History's Worst Software Bugs:
http://www.wired.com/software/coolapps/news/2005/11/69355?currentPage=all

[IEEE-83] *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Standard 729, 1983.

[LC06] Leavens, Gary T.; Cheon, Yoonsik . *Design by Contract with JML*, agosto 2006.

[Mey92] Meyer, Bertrand. *Applying 'Design by Contract'*. IEEE Computer, Vol. 25, No. 10, pp 40-51, 1992.

[MPR] Mohri , Mehryar; Pereira, Fernando C. N.; Riley, Michael D. *AT&T FSM LibraryTM, Finite-State Machine Library*. http://www2.research.att.com/~fsmtools/fsm/

[RGMA03] Rossel, Gerardo; Manna, Andrea. *Diseño por contratos: construyendo software confiable*. Revista Digital Universitaria, 2003.

[STR86] E. Strom, S. Yemini. *Typestate: A programming language concept for enhancing software reliability*. IEEE Transactions on Software Engineering, Volume 12 Issue 1, Jan. 1986
IEEE Press Piscataway, NJ, USA.