Multiejecución Segura para Programación Interactiva

Dante Zanarini[1,2]

[1] CIFASIS, Blvd. 27 de Febrero 201 bis, Rosario, Argentina
[2] Universidad Nacional de Rosario, Argentina
zanarini@cifasis-conicet.gov.ar

Abstract. La confidencialidad de datos es cada día más importante. Debido a esto se han desarrollado políticas de seguridad como no interferencia que buscan evitar la filtración de datos privados en canales públicos. Recientemente, se ha probado que la multiejecución segura es una mecanismo efectivo para garantizar una forma de no interferencia. En este trabajo, probamos que la multiejecución segura es también efectiva para una política de seguridad aplicable a programas interactivos que a) pueden no terminar b) producen efectos cuyo orden es observable. Además, probamos que, sorprendentemente, para programas no interferentes la multiejecución segura puede preservar la semántica, a pesar del orden observable de los efectos.

1 Introducción

La cantidad de información personal sensible que se pone a disposición de los sistemas informáticos va en aumento. A pesar de esto, las aplicaciones disponen de pocas herramientas para garantizar la confidencialidad de los datos y son susceptibles a ser atacadas con el fin de robar datos confidenciales o incluso robar una identidad. En este trabajo estamos interesados en la política de seguridad conocida como no interferencia [6].

Un programa interactivo permite a los usuarios enviar y recibir mensajes durante su ejecución [8]. La omnipresencia de tales programas (web-browsers, programas que interactúan con un sistema de archivos, terminales, teclados, etc) resulta evidente y pone de manifiesto la necesidad de desarrollar técnicas que permitan analizar formalmente la seguridad en el flujo de información.

Tradicionalmente, los análisis de seguridad en el flujo de información son realizados en forma estática (mediante un sistema de tipos) o dinámicamente (mediante monitores de ejecución), o una combinación de ambos. Recientemente, se descubrió un nuevo enfoque, llamado multiejecución segura (SME) [4]. Este enfoque, a diferencia de los anteriores, no necesita del diseño de sistemas de tipos o monitores de programas, sino que funciona ejecutando un programa una vez por cada nivel de seguridad y en cada ejecución modificando la semántica de las operaciones de entrada y salida. Devriese y Piessens aplicaron este enfoque a un lenguaje de programación imperativo estándar aumentado con eventos de

entrada-salida, y probaron para un scheduler en particular que cualquier programa P es no interferente cuando se interpreta bajo SME. Adicionalmente, demostraron la preservación semántica bajo multiejecución en el caso que P sea un programa que termine.

En este trabajo nos proponemos estudiar SME en el contexto de programación interactiva. Una programa interactivo puede estar diseñado para no terminar (por ejemplo, un programa que imprima en una terminal los datos que recibe por teclado). Adicionalmente, al observar el comportamiento de esta clase de programas, es posible que uno distinga el orden en que ocurren las acciones, a diferencia de los programas imperativos en los cuales uno sólo observa el estado final de las variables almacenadas en memoria.

Concretamente, presentamos una semántica SME para programas interactivos, y demostramos que, para cualquier programa p, su semántica bajo multiejecución es segura, independientemente del scheduler que resuelva el no determinismo entre los hilos de ejecución generados. Finalmente, probamos que para programas no interferentes la multiejecución segura puede preservar la semántica incluso en programas que no terminan, a pesar del orden observable de los efectos.

El trabajo se organiza como sigue. En la sección 2 se presenta el modelo de computación elegido para abstraer la semántica de programas interactivos, y se define una noción de no interferencia para programación interactiva. La sección 3 presenta una semántica basada en SME para tal modelo. En la sección 4 se prueba la corrección de SME para programación interactiva. Por último, la sección 5 presenta algunas conclusiones, trabajos relacionados y trabajos en curso y a futuro.

2 Modelo de computación interactiva

Asumimos la presencia de un conjunto $\mathcal{L}$ de *niveles de seguridad*, equipado con una relación binaria $\sqsubseteq$, que determina un orden parcial sobre $\mathcal{L}$. Para los ejemplos presentados en este trabajo, consideramos $\mathcal{L} = \{L, H\}$, con $L \sqsubseteq H$; H representa confidencialidad alta, mientras que L representa confidencialidad baja. Decimos que $l \sqsubset l'$ si $l \sqsubseteq l'$ y $l \neq l'$.

Un *usuario* u *observador* es cualquier agente externo que interactúa con un programa en ejecución. Cada observador tiene asociado un nivel de seguridad, que determina el nivel más alto de confidencialidad de la información a la cual le es permitido acceder.

Consideremos un programa interactivo que recibe entradas de un conjunto de canales de entrada (por ejemplo, un sistema de archivos, un teclado o mouse); y produce salidas enviando mensajes a través un conjunto de canales de salida (por ejemplo, una terminal). Asociaremos a cada canal (de entrada o de salida) ch un nivel de seguridad $lvl(ch)$, que denota el nivel de confidencialidad de un mensaje que se transmite por ch. Asumimos que el envío de estos mensajes son la única forma de comunicarse con el entorno y que los usuarios sólo observan (algunos de) estos eventos.

Cada operación observable genera un evento, y la sucesión de eventos generados por un programa es el *comportamiento observable* de este. En el resto del trabajo, utilizaremos los términos *secuencia* o *stream* indistintamente para referirnos a una lista posiblemente infinita de eventos. Formalmente, el conjunto de streams de elementos de un tipo A se introduce coinductivamente a través la siguiente gramática

$$S ::= [] \mid s :: S \tag{1}$$

donde $s : A$. Si s, s' son streams y s es finito, entonces $s{+\!+}s'$ es el stream que se obtiene concatenando s y s'. En caso que s sea infinito, $s{+\!+}s' = s$. Finalmente, si A es un tipo con igualdad $=_A$, definimos una relación coinductiva de igualdad entre streams a través de las siguientes reglas:

$$\frac{}{[] = []} \qquad\qquad \frac{s =_A s' \quad S = S'}{s :: S = s' :: S'}$$

Además de generar eventos observables, un programa puede realizar *operaciones internas* o *silenciosas*. Por ejemplo, asignación de variables locales, evaluación de expresiones que no involucran eventos observables, etc.

2.1 Acciones observables

Los eventos del sistema vienen dados por la siguiente gramática

$$Ev = ch?v \mid ch!v \mid \bullet \tag{2}$$

donde v es un elemento de un tipo de valores *Value*. El evento $ch!v$ representa la generación de un evento de salida, el cual se transmite a través del canal ch; mientras que el evento $ch?v$ denota la recepción de un valor v del canal de entrada ch. La acción $\bullet$ representa una acción silenciosa o invisible.

No todos los eventos generados por un programa resultan observables para todos los usuarios. Por ejemplo, si ch es un canal tal que $l \sqsubset lvl(ch)$, entonces un usuario cuyo nivel de seguridad es l no podrá distinguir la generación de un mensaje sobre ch de una acción silenciosa. La función $obs : \mathcal{L} \to Ev \to Ev$ recibe un nivel l y un evento $e : Ev$, y nos devuelve un evento que representa qué observa un usuario cuyo nivel de seguridad es l cuando se genera e.

$$obs_l(\bullet) = \bullet \; ; \; obs_l(c?v) = \begin{cases} \bullet & \text{si } l \sqsubset lvl(c) \\ c?v & \text{en otro caso} \end{cases} \; ; \; obs_l(c!v) = \begin{cases} \bullet & \text{si } l \sqsubset lvl(c) \\ c!v & \text{en otro caso} \end{cases}$$

Dado un stream S de eventos, estamos interesados en obtener cuál es el primer evento observable por un usuario de nivel l, si existe.

Definición 1. *La relación $S \rhd_l e :: S'$ (S l-revela e seguido de S') se define inductivamente con las siguientes reglas*

$$\frac{obs_l(e) = e' \quad e' \neq \bullet}{e :: S \rhd_l e' :: S} \qquad\qquad \frac{obs_l(e) = \bullet \quad S \rhd_l e' :: S'}{e :: S \rhd_l e' :: S'}$$

La definición inductiva de la relación nos asegura que siempre se descartará a lo sumo una cantidad finita de $\bullet$. Observemos que no siempre existen e y S' tales que $S \triangleright_l e :: S'$, tal como se ilustra en el siguiente ejemplo.

Ejemplo 1. *Sean* ch_L, ch_H *tales que* $lvl(ch_L) = L$ *y* $lvl(ch_H) = H$. *Si* $S = [\bullet, \bullet, ch_H!3, \bullet, ch_H?6, \bullet]$, *tenemos que* $(S \triangleright_H [ch_H!3, \bullet, ch_H?6, \bullet])$. *Sin embargo, no existen* e, S' *tales que* $S \triangleright_L e :: S'$.

Utilizaremos el predicado $silent_l$ para indicar que un stream S no genera ningún evento visible a nivel l, definido coinductivamente a partir de las siguientes reglas:

$$\frac{}{silent_l([])} \qquad \frac{obs_l(e) = \bullet \quad silent_l(S)}{silent_l(e :: S)}$$

2.2 Programas

Definimos una abstracción que capture el comportamiento observable de un programa reactivo en ejecución, como la interpretación coinductiva de la siguiente gramática:

$$\begin{aligned} IO\ a := & \ \mathbf{return} : a \to IO\ a \\ & \mid \mathbf{read} : Ch \to (Value \to IO\ a) \to IO\ a \\ & \mid \mathbf{write} \ : (Ch \times V) \to IO\ a \to IO\ a \\ & \mid \mathbf{silent} \ : IO\ a \to IO\ a \end{aligned}$$

donde Ch es un conjunto de nombres de canales.

El programa **return** x concluye sus cálculos devolviendo el valor x al entorno. Un programa de la forma (**read** $ch\ f$) representa la lectura del valor v a través del canal ch, el cual es manejado mediante el programa $(f\ v) : IO\ a$. El programa (**write** $(ch, v)\ p$) genera el evento visible $ch!v$, para luego comportarse como el programa $p : IO\ a$. Finalmente, un programa de la forma **silent** p realiza una acción interna $\bullet$, para luego comportarse como el programa p.

Dado que nuestro modelo de interacción con el entorno es sólo a través de mensajes de acuerdo a (2), de aquí en más trabajaremos sobre programas de tipo $IO_1 := IO\ \mathbf{1}$, donde $\mathbf{1}$ es un tipo de datos con un único elemento u. Escribiremos **return** en lugar de **return** u. Cuando resulte conveniente utilizaremos la notación $v \leftarrow \mathbf{read}\ ch; f(v)$ para referirnos al programa (**read** $ch\ \lambda v.f(v)$); y **write** $(ch, v); p$ para **write** $(ch, v)\ p$.

El tipo IO nos permite abstrernos de los detalles particulares de un lenguaje de programación que no son relevantes a la hora de analizar la interacción con el entorno, separando las acciones que describen interacción de aquellas que describen computaciones sin efectos.

2.3 Semántica de programas interactivos

Los programas en IO_1 son interpretados sobre un sistema de canales de entrada I y un sistema de canales de salida O. Un sistema de canales es un función que asocia nombres de canales a streams de elementos de tipo *Value*. Para los ejemplos, utilizaremos como valores números enteros.

$$\frac{}{\langle \mathbf{return}, I, O\rangle \xrightarrow{\bullet} \top} \qquad \frac{I(ch) = v :: q}{\langle \mathbf{read}\ ch\ f, I, O\rangle \xrightarrow{ch?v} \langle f(v), I[ch \leftarrow q], O\rangle}$$

$$\frac{}{\langle \mathbf{silent}\ p, I, O\rangle \xrightarrow{\bullet} \langle p, I, O\rangle} \qquad \frac{O(ch) = q}{\langle \mathbf{write}\ (ch, v)\ p, I, O\rangle \xrightarrow{ch!v} \langle p, I, O[ch \leftarrow q{+\!\!+}[v]]\rangle}$$

Fig. 1. Semántica small step para IO_1

Asumiremos, al igual que O'Neill et al. [8], que siempre que un programa interactivo intente una lectura de un canal ch, esta siempre podrá realizarse luego de un tiempo finito. Es decir, si un programa de la forma (**read** ch f) se interpreta sobre una entrada I, entonces $I(ch)$ es no vacío. Para representar canales posiblemente vacíos y conservar esta hipótesis, podríamos extender el conjuntos de valores *Value* agregando un nuevo elemento *Timeout*. Si un canal se encuentra vacío, entonces al realizar una lectura el valor recibido sería *Timeout*.

Definición 2 (Configuración). *Una configuración es, o bien una tupla de la forma $\langle p, I, O\rangle$, donde $p : IO_1$, e I, O son sistemas de canales, o bien $\top$. La configuración $\top$ es alcanzada por un programa interactivo que concluye sus cálculos a través de la instrucción* **return**. *Para un programa p y una entrada I, la configuración inicial de p respecto de I asigna a cada canal de salida el stream vacío. Es decir, $C_{p,I} = \langle p, I, \lambda ch.[]\rangle$.*

La figura 1 presenta una semántica small step para IO_1 como una relación tal que $\langle p, I, O\rangle \xrightarrow{e} C$ si la primer acción observable de p con entrada I genera el evento e, y C es la configuración resultante de dicha ejecución. Si f es una función, entonces $f[x \leftarrow y]$ se comporta como f en todos los valores de su dominio salvo quizás en x, donde vale y. A partir de la relación $\longrightarrow$, podemos definir el comportamiento de un programa en IO_1.

Definición 3 (Comportamiento de programas en IO_1). *La relación $\Longrightarrow$ se define coinductivamente a través de las siguientes reglas, donde C es una configuración, e un evento y B un stream de eventos:*

$$\frac{C \xrightarrow{e} C' \qquad C' \Longrightarrow B}{C \Longrightarrow e :: B} \qquad \frac{}{\top \Longrightarrow []}$$

Decimos que el comportamiento del programa p al recibir como entrada I es B sii $C_{p,I} \Longrightarrow B$.

Ejemplo 2. *Consideremos el siguiente programa:*

$$eco_2 = v \leftarrow \mathbf{read}\ i_L; \mathbf{write}\ (o_H, v); \mathbf{write}\ (o_L, v); eco_2$$

Donde i_L es una canal de entrada tal que $lvl(i_L) = L$; y o_L, o_H son canales de salida tales que $lvl(o_L) = L$, $lvl(o_H) = H$. Si $I(i_L) = [0, 1, 2, 3, \dots]$, entonces $C_{eco_2,I} \Longrightarrow [i_L?0, o_H!0, o_L!0, i_L?1, o_H!1, o_L!1, \dots]$.

2.4 No interferencia

Informalmente, un programa p es no interferente si el comportamiento observable por un usuario cuyo nivel de seguridad es l no depende de los datos de entrada de un nivel superior. Es decir, si un usuario no puede distingir dos entradas para p, entonces tampoco será capaz de distinguir el comportamiento de p al ejecutarse con tales entradas. De esta idea surge la necesidad de definir relaciones de similaridad a un determinado nivel de seguridad, que formalicen la noción de *indistinguibilidad*.

Definición 4 (Similaridad de entradas y salidas). *Sea X, X' sistemas de canales. Decimos que X e X' son similares a nivel l, y lo notamos $X =_l X'$ si y solo si, para cada canal ch tal que $lvl(ch) \sqsubseteq l$ se cumple que $X(ch) = X'(ch)$.*

En otras palabras, $X =_l X'$ si coinciden en todos los canales cuyos valores pueden ser observados por un usuario de nivel l.

¿Qué significa que dos comportamientos de un programa sean indistinguibles a un determinado nivel de seguridad? En el contexto de programación reactiva [2] y programación interactiva [8] pueden definirse varias nociones de similaridad. En [2] se definen diferentes nociones de seguridad para sistemas cuya semántica se basa en comportamientos. Según los autores, dos de estas nociones de seguridad son de interés práctico: ID-*security* y CP-*security*. Ambas nociones son similares, pero difieren en el tratamiento de programas que divergen silenciosamente. En este trabajo utilizaremos CP-*security* (seguridad coproductiva, o seguridad CP), que es la noción de seguridad más fuerte entre ambas (CP-*security* implica ID-*security* [2]).

Definición 5 (CP-similaridad). *Sean S, S' dos streams. Decimos que S y S' son CP-similares a nivel l, sii $S \simeq_l S'$, donde la relación $\simeq_l$ se define coinductivamente mediante las siguientes reglas*

$$\frac{silent(S) \qquad silent(S')}{S \simeq_l S'} \qquad\qquad \frac{S \rhd_l e :: S_1 \qquad S' \rhd_l e :: S_1' \qquad S_1 \simeq_l S_1'}{S \simeq_l S'}$$

Intuitivamente, dos streams S, S' son CP-similares a nivel l si todos sus eventos l-observables coinciden.

Una vez que definimos relaciones de similaridad para entradas y comportamientos, podemos precisar la noción de no interferencia para programas interactivos. Nuestra definición estará parametrizada por un funtor de tipo que captura la noción de programa (por ejemplo, IO), así como también por una relación semántica $\hookrightarrow$, que relaciona programas y sistemas de canales de entrada y salida con comportamientos. Las instancias que usaremos en este trabajo son la relación $\Longrightarrow$ presentada en la def. 3 y la relación $\Longrightarrow_{me}$ (def. 10).

Definición 6 (No Interferencia). *Sea p un programa interactivo, I, I' sistemas de canales y B, B' streams de eventos. Decimos que p es seguro si siempre que $\langle p, I, O\rangle \hookrightarrow B$ y $\langle p, I', O\rangle \hookrightarrow B'$, para cualquier nivel l se cumple que $I =_l I'$ implica $B \simeq_l B'$; donde $O = \lambda ch.[]$ es un sistema de canales de salida inicialmente vacío.*

3 Multiejecución segura

En la interpretación de multiejecución segura, un mismo programa se ejecuta varias veces, una por cada nivel de seguridad, modificando en cada ejecución la semántica de las operaciones observables. Los eventos de salida de nivel l se realizan únicamente en la ejecución de dicho nivel. Una instrucción de entrada sobre un canal ch cuyo nivel de seguridad es l se realiza sólo en la ejecución de nivel l; para niveles de seguridad inferiores, se continua la ejecución proveyendo como entrada un valor por defecto; mientas que para las ejecuciones de nivel superior a l se provee un mecanismo de reutilización del valor leído en el programa de nivel l.

Dado que los eventos visibles para un observador cuyo nivel de seguridad es l se realizan únicamente en las ejecuciones de nivel l o inferior, y que las entradas de niveles superiores no son accesibles a estos programas (dado que las lecturas de niveles superiores a l son remplazadas por valores por defecto), intuitivamente podemos concluir que las salidas observables a nivel l no dependen de los valores de entrada de niveles superiores.

3.1 Un lenguaje para multiejecución

El procedimiento de multiejecución se realiza a través del tipo ME, definido coinductivamente como:

$$\begin{aligned} ME\ a := & \ \mathbf{return} : a \to ME\ a \\ & \mid \mathbf{read} : Ch \to (Value \to ME\ a) \to ME\ a \\ & \mid \mathbf{write} : (Ch \times Value) \to ME\ a \to ME\ a \\ & \mid \mathbf{reuse} : Ch \to (Value \to ME\ a) \to ME\ a \\ & \mid \mathbf{broadcast} : (Ch \times Value) \to ME\ a \to ME\ a \\ & \mid \mathbf{silent} : ME\ a \to ME\ a \end{aligned}$$

El tipo ME puede verse como una extensión de IO. Cuando la ejecución de nivel l realiza una operación de lectura sobre un canal ch, el valor leído es almacenado (utilizando el constructor **broadcast**) para ser consumido (utilizando el constructor **reuse**) por las ejecuciones de nivel superior a l.

La figura 2 define una transformación que, dado un programa en IO y un nivel l, nos devuelve un programa en ME de acuerdo a la semántica de multiejecución. El valor $\cdot$ es un elemento arbitrario de $Value$ al que llamaremos *valor por defecto*.

Definición 7. *Sea $l \in \mathcal{L}$. Un programa $p : ME_1$ es l-estricto si se cumple el predicado coinductivo $strict_l$, definido por las siguientes reglas:*

$$\frac{lvl(ch) = l \quad \forall v, strict_l(f(v))}{strict_l(\mathbf{read}\ ch\ f)} \qquad \frac{lvl(ch) = l \quad strict_l(p)}{strict_l(\mathbf{write}\ (ch, v)\ p)}$$

$$\frac{}{strict_l(\mathbf{return})} \qquad \frac{strict_l(p)}{strict_l(\mathbf{broadcast}\ (ch, v)\ p)} \qquad \frac{\forall v, strict_l(f(v))}{strict_l(\mathbf{reuse}\ ch\ f)}$$

$$
\begin{array}{ll}
me_l(\textbf{return } x) & = \textbf{return } x \\
me_l(\textbf{read } ch\ f) & = \begin{cases} v \leftarrow \textbf{read } ch; \textbf{broadcast } (ch, v)\ me_l(f(v)) & \text{si } lvl(ch) = l \\ v \leftarrow \textbf{reuse } ch; me_l(f(v)) & \text{si } lvl(ch) \sqsubset l \\ me_l(f(\cdot)) & \text{en otro caso} \end{cases} \\
me_l(\textbf{write } (ch, v)\ p) & = \begin{cases} \textbf{write}(ch, v)\ me_l(p) & \text{si } lvl(ch) = l \\ me_l(p) & \text{otherwise} \end{cases} \\
me_l(\textbf{silent } p) & = \textbf{silent } me_l(p)
\end{array}
$$

Fig. 2. Transformación a multiejecución para un programa en IO y un nivel l

Intuitivamente, p es l-estricto si todas sus operaciones que generan eventos visibles (**read** y **write**) se realizan sobre canales de nivel l.

Lema 1. *Para cualquier programa $p : IO_1$, y nivel $l \in \mathcal{L}$, $me_l(p)$ es l-estricto.*

Proof. Por coinducción en p siguiendo la definición de me_l. □

Este resultado será de especial utilidad cuando probemos que la interpretación bajo multiejecución de un programa es no interferente.

Ejemplo 3. *Consideremos el programa eco_2 del ejemplo 2. Tenemos*

$$
\begin{array}{l}
me_L(eco_2) = v \leftarrow \textbf{read } i_L; \textbf{write } (o_L, v); me_L(eco_2) \\
me_H(eco_2) = v \leftarrow \textbf{reuse } i_L; \textbf{write } (o_H, v); me_H(eco_2)
\end{array}
$$

Ejemplo 4. *Sea el programa no seguro $(eco = v \leftarrow \textbf{read } i_H; \textbf{write } (o_L, v); eco)$, donde i_H es un canal de entrada tal que $lvl(i_H) = H$. Tenemos*

$$
\begin{array}{l}
me_H(eco) = v \leftarrow \textbf{read } i_H; me_H(eco) \\
me_L(eco) = \textbf{write } (o_L, \cdot); me_L(eco)
\end{array}
$$

Observamos que, independientemente del contenido en i_H, en el canal o_L no se filtrará información de confidencialidad alta.

Definición 8. *Sea $p : IO_1$, su intepretación bajo multiejecución es una función que asocia a cada nivel $l \in \mathcal{L}$ un programa en ME_1. Formalmente, definimos la función me como sigue: $me(p) = \lambda l.me_l(p)$*

3.2 Semántica

Para $p : IO_1$, estamos interesados en definir una semántica para (el conjunto de programas en) $me(p)$. Observemos que, dado que tales programas se ejecutan concurrentemente, se debe contar con un scheduler que determine el orden de ejecución de los procesos. En nuestro modelo, un scheduler es representado como un stream infinito de elementos de $\mathcal{L}$, donde el primer elemento de la lista determina a qué nivel de ejecución corresponde la siguiente instrucción.

Las operaciones **broadcast** y **reuse** serán interpretadas sobre un *contexto*, en el cual se almacenarán, para cada nivel de seguridad, todos los valores de entrada leídos por las ejecuciones de niveles inferiores. Concretamente, un contexto mantendrá una copia de todos los valores leídos, para cada nivel l y nombre de canal ch tal que $lvl(ch) \sqsubset l$, los cuales serán consumidos en la ejecución a nivel l por la instrucción **reuse**.

Definición 9 (Configuración para multiejecución). *Una configuración para multiejecución es, o bien* $\top$*, o bien una tupla de la forma* $\langle th, r, ctx, sch, I, O\rangle$*, donde*

- $th : \mathcal{L} \to ME_1$ *asocia a cada nivel de seguridad un programa en* ME_1
- $r \subseteq \mathcal{L}$ *es un conjunto de niveles en ejecución. Si un nivel de seguridad* l *no pertenece a este conjunto, entonces* $th(l) =$ **return***; es decir, el programa asociado a* l *no generará mas eventos observables;*
- ctx *es una función que asocia a cada nivel de seguridad* l *y a cada canal* ch *tal que* $lvl(ch) \sqsubset l$ *un canal* $ctx(l, ch)$*. Este canal será accedido por la ejecución de nivel* l *cuando requiere reutilizar una entrada generada en la ejecución de nivel* $lvl(ch)$*.*
- sch *es un scheduler,* I *es un sistema de canales de entrada, y* O *es un sistema de canales de salida.*

Si $p : IO_1$*, su configuración inicial respecto de una entrada* I *y un scheduler* s *se define como*

$$C_{p,s,I} = \langle me(p), \mathcal{L}, ctx_p, s, I, \lambda ch.[]\rangle\,,$$

donde para cada l *y* ch *tal que* $lvl(ch) \sqsubset l$*,* $ctx_p(l, ch) = []$*.*

La figura 3 define una relación parcial de transición entre configuraciones. Para la regla *broadcast* se utiliza el operador $\odot$, que toma un contexto ctx y un par (ch, v), y devuelve un nuevo contexto ctx' en el cual, para cada nivel l tal que $lvl(ch) \sqsubset l$, $ctx'(l, ch) = ctx(l, ch)_{++}[v]$.

Las reglas ret_1 y ret_2 contemplan el caso en que el hilo de ejecución elegido concluye sus cálculos. Si no quedan más hilos en ejecución, entonces la evaluación termina, mientras que en el caso que otros hilos de ejecución continuen activos, sólo se quita el nivel que ha terminado de la lista de procesos activos. Las reglas *read*, *write* y *silent* son similares a las introducidas para IO_1. En el caso que el thread elegido por el scheduler requiera reutilizar un dato a través de la instrucción (**reuse** ch f), el comportamiento varía de acuerdo a si el dato se encuentra o no el contexto. En el primer caso, se actualiza el hilo de ejecución a $f(v)$, donde v es el valor leído (el cual se consume del contexto). En el segundo caso, el hilo seleccionado no puede progresar, y la configuración cambia sólo en el scheduler. La regla *broadcast* se encarga de poner a disposición de los threads de nivel superior al seleccionado para ejecución el valor v en el canal ch. Finalmente, la regla *nop* contempla el caso en que el scheduler selecciona un hilo de ejecución que ha concluido sus cálculos. Si $C \xrightarrow{e}_{me} C'$ decimos que la configuración C transiciona a C' mediante el evento e.

$$ret_1 \frac{sch = l_i :: sch' \quad r = \{l_i\} \quad th(l_i) = \textbf{return}}{\langle th, r, ctx, sch, I, O\rangle \xrightarrow{\bullet}_{me} \top}$$

$$ret_2 \frac{|r| \geq 2 \quad sch = l_i :: sch' \quad th(l_i) = \textbf{return}}{\langle th, r, ctx, sch, I, O\rangle \xrightarrow{\bullet}_{me} \langle th, r \setminus \{l_i\}, ctx, sch', I, O\rangle}$$

$$read \frac{sch = l_i :: sch' \quad th(l_i) = \textbf{read}\ ch\ f \quad I(ch) = v :: q}{\langle th, r, ctx, sch, I, O\rangle \xrightarrow{ch?v}_{me} \langle th[l_i \leftarrow f(v)], r, ctx, sch', I[ch \leftarrow q], O\rangle}$$

$$write \frac{sch = l_i :: sch' \quad th(l_i) = \textbf{write}\ (ch, v)\ p \quad O(ch) = q}{\langle th, r, ctx, sch, I, O\rangle \xrightarrow{ch!v}_{me} \langle th[l_i \leftarrow p], r, ctx, sch', I, O[ch \leftarrow q +\!\!+ [v]]\rangle}$$

$$reuse_1 \frac{sch = l_i :: sch' \quad th(l_i) = \textbf{reuse}\ ch\ f \quad ctx(l_i, ch) = v :: q}{\langle th, r, ctx, sch, I, O\rangle \xrightarrow{\bullet}_{me} \langle th[l_i \leftarrow f(v)], r, ctx[(l, ch) \leftarrow q], sch', I, O\rangle}$$

$$reuse_2 \frac{sch = l_i :: sch' \quad th(l_i) = \textbf{reuse}\ ch\ f \quad ctx(l_i, ch) = []}{\langle th, r, ctx, sch, I, O\rangle \xrightarrow{\bullet}_{me} \langle th, r, ctx, sch', I, O\rangle}$$

$$broadcast \frac{sch = l_i :: sch' \quad th(l_i) = \textbf{broadcast}\ (ch, v)\ p}{\langle th, r, ctx, sch, I, O\rangle \xrightarrow{\bullet}_{me} \langle th[l_i \leftarrow p], r, ctx \odot (ch, v), sch', I, O\rangle}$$

$$silent \frac{sch = l_i :: sch' \quad th(l_i) = \textbf{silent}\ p}{\langle th, r, ctx, sch, I, O\rangle \xrightarrow{\bullet}_{me} \langle th[l_i \leftarrow p], r, ctx, sch', I, O\rangle}$$

$$nop \frac{sch = l_i :: sch' \quad l_i \notin r}{\langle th, r, ctx, sch, I, O\rangle \xrightarrow{\bullet}_{me} \langle th, r, ctx, sch, I, O\rangle}$$

Fig. 3. Semántica small step para multiejecución

Definición 10 (Comportamiento bajo multiejecución). *Sea C una configuración para multiejecución, decimos que el comportamiento bajo multiejecución de C es B sii $C \Longrightarrow_{me} B$, donde $\Longrightarrow_{me}$ se define coinductivamente a través de las siguientes reglas:*

$$\frac{C \xrightarrow{a}_{me} C' \quad C' \Longrightarrow_{me} B}{C \Longrightarrow_{me} a :: B} \qquad \frac{}{\top \Longrightarrow_{me} []}$$

Las definiciones introducidas en esta sección nos permiten definir la *semántica bajo multiejecución* de un programa en IO_1, la cual se encontrará parametrizada por un scheduler y, al igual que para la semántica presentada en la sección 2, un sistema de canales de entrada.

Definición 11 (Semántica bajo SME para IO_1). *Sean $p : IO_1$, s un scheduler e I un sistema de canales. Decimos que el comportamiento bajo multiejecución de p respecto del scheduler s al recibir como entrada I es B sii $C_{p,s,I} \Longrightarrow_{me} B$.*

Ejemplo 5. *Consideremos el programa eco del ejemplo 4. Sea $s = H :: L :: s$ un scheduler que alterna entre los niveles H y L. Si $I(i_H) = [v_1, v_2, \dots]$ tenemos*

$C_{eco,s,I} \Longrightarrow_{me} [i_H?v_1, o_L!\cdot, i_H?v_2, o_L!\cdot, \dots]$. *Si nos quedamos con los eventos que observa un usuario de nivel L, tenemos el siguiente stream:* $[o_L!\cdot, o_L!\cdot, o_L!\cdot, \dots]$.

4 No interferencia y Preservación Semántica

En la sección 3 definimos una semántica para programas interactivos siguiendo un procedimiento denominado multiejecución, con el objeto de proveer un mecanismo que garantice la seguridad de la información. Naturalmente, surgen dos interrogantes sobre esta nueva forma de interpretar programas: ¿Se garantiza no interferencia?, ¿Se preserva la semántica original de un programa?

Para responder la primer pregunta, demostraremos que, para un programa en IO_1, su semántica bajo multiejecución tal como se introdujo en la def. 11 garantiza no interferencia, independientemente del scheduler con que se resuelva el no determinismo entre los diferentes hilos de ejecución.

De la afirmación precedente surge necesariamente que si un programa no es seguro, entonces su comportamiento bajo multiejecución será diferente que su comportamiento *normal*, i.e, tal como se definió en la sección 2 ¿Qué sucede si un programa es seguro? En este caso, desearíamos que el comportamiento bajo multiejecución preserve la semántica original del programa. Sin embargo, esta propiedad no es sencilla de alcanzar en programas interactivos, dado que en el comportamiento puede observarse el orden en que se ejecutan las operaciones visibles, y por lo tanto la preservación semántica dependerá necesariamente del scheduler elegido, tal como se ilustra en el siguiente ejemplo.

Ejemplo 6. *Consideremos el siguiente programa seguro:*

$$p = v \leftarrow \mathbf{read}\ i_H; \mathbf{write}\ (o_L, 1); \mathbf{write}\ (o_H, v); p$$

Si I es tal que $I(i_H) = [v_1, v_2, \dots]$, *la semántica definida en la sección 2 asegura que* $C_{p,I} \Longrightarrow [i_H?v_1, o_L!1, o_H!v_1, i_H?v_2, o_L!1, \dots]$. *Sin embargo, si tomamos* $s = L :: L :: H :: s$, *obtenemos* $C_{p,s,I} \Longrightarrow [o_L!1, o_L!1, i_H?v_1, o_L!1, o_L!1, o_H!v_1, \dots]$.

4.1 No Interferencia en SME

Dado $p : IO_1$, ¿podemos afirmar que su comportamiento bajo multiejecución es seguro, independientemente del hecho que p lo sea?

Concretamente, si $l \in \mathcal{L}$ e I, I' son sistemas de canales de entradas tales que $I =_l I'$, $C_{p,s,I} \Longrightarrow_{me} B$ y $C_{p,s,I'} \Longrightarrow_{me} B'$, ¿vale que $B \simeq_l B'$, independientemente de la elección del scheduler s? Para demostrar este resultado, utilizaremos la técnica de *bisimulación* [10]. Encontraremos una bisimulación débil entre las configuraciones iniciales, lo cual no garantizará que los comportamientos que las configuraciones iniciales son CP-similares. Por cuestiones de espacio, omitiremos los detalles de las pruebas y sólo daremos las ideas principales que nos permiten alcanzar los resultados.

Definición 12. *Una configuración* $C = \langle th, r, ctx, s, I, O\rangle$ *es estricta sii, para cada nivel l,* $th(l)$ *es l-estricto.*

Lema 2. *Si* $C = \langle th, r, ctx, s, I, O\rangle$ *es estricta y* $C \xrightarrow{e} C'$*, entonces* C' *es estricta.*

Proof. Sea $s = l_i :: s'$*. Dado que* $th(l_i)$ *es* l_i*-estricto por hipótesis, separando en casos por cada uno de los posibles constructores de* $th(l_i)$ *obtenemos que el nuevo hilo de ejecución a nivel* l_i *será estricto, dado que la continuación de un programa* l_i*-estricto es* l_i*-estricto por definición. Como el resto de los hilos de ejecución no cambia, obtenemos el resultado buscado.* □

Definición 13 (Similaridad de configuraciones).
Sean $C_1 = \langle th_1, r_1, ctx_1, s_1, I_1, O_1\rangle$ *y* $C_2 = \langle th_2, r_2, ctx_2, s_2, I_2, O_2\rangle$ *dos configuraciones distintas de* $\top$*. Decimos que* $C_1 \approx_l C_2$ *si se cumplen las siguientes condiciones:*

- *Para cada* l' *tal que* $l' \sqsubseteq l$*,* $th(l) = th(l')$*, donde la igualdad sobre el tipo* ME_1 *se refiere a igualdad sintáctica;*
- *Si un hilo de ejecución cuyo nivel de seguridad es a lo sumo* l *se encuentra en ejecución en* C_1*, entonces también está activo en* C_2 *y viceversa. Es decir,* $\{l' \in r_1 \mid l' \sqsubseteq l\} = \{l' \in r_2 \mid l' \sqsubseteq l\}$*;*
- *Para cada* l' *tal que* $l' \sqsubseteq l$*,* $ctx_1(l') =_l ctx_2(l')$*. Es decir, los contextos de niveles no superiores a* l *coinciden en todos sus canales;*
- $s_1 = s_2$*,* $I_1 =_l I_2$ *y* $O_1 =_l O_2$*.*

Los siguientes lemas nos darán un punto de partida para la bisimulación.

Lema 3. *Sea* $p : IO_1$*,* s *un scheduler, e* I *un sistema de canales de entrada. La configuración* $C_{p,s,I}$ *es estricta.*

Proof. Por el lema 1 vale que para cada l*,* $me_l(p)$ *es* l*-estricto.* □

Lema 4. *Sea* $p : IO_1$*,* $l \in \mathcal{L}$ *e* I, I' *sistemas de canales de entrada tales que* $I =_l I'$*. Entonces* $C_{p,s,I} \approx_l C_{p,s,I'}$*.*

Proof. Inmediata comparando componente a componente ambas configuraciones. □

Probamos a continuación que las transiciones observables a nivel l de configuraciones estrictas y similares (a nivel l) necesariamente emiten el mismo evento, y resultan en configuraciones estrictas y similares a nivel l.

Lema 5. *Sean* $C_1 = \langle th_1, r_1, ctx_1, s_1, I_1, O_1\rangle$ *y* $C_2 = \langle th_2, r_2, ctx_2, s_2, I_2, O_2\rangle$ *dos configuraciones estrictas tales que* $C_1 \approx_l C_2$ *y* $l \in \mathcal{L}$*. Si* $C_1 \xrightarrow{e}_{me} C_1'$ *y* $obs_l(e) \neq \bullet$*, entonces existe una única* C_2 *tal que*

(1) $C_2 \xrightarrow{e}_{me} C_2'$*, (2)* $C_1' \approx_l C_2'$*, y (3)* C_1' *y* C_2' *son estrictas.*

Proof. La unicidad de C_2' *está garantizada pues la relación* $\longrightarrow_{me}$ *es determinística. Consideremos* $e = ch?v$ *(el caso* $e = ch!v$ *es análogo). Para probar (1), sea* $l' = lvl(ch)$*. Como* $obs_l(ch?v) \neq \bullet$*, necesariamente* $l' \sqsubseteq l$*. Teniendo en cuenta que* C_1 *es estricta, concluímos que* $s_1 = l' :: s'$*, y el thread que generó el*

evento es $th_1(l') =$ **read** *ch f, transicionando a través de la regla read. Como* $l' \sqsubseteq l$, *y* $C_1 \approx_l C_2$, *entonces* $th_2(l') =$ **read** *ch f y* $l' \in r_2$,*y considerando que* C_2 *tiene el mismo scheduler que* C_1, *la transición por* $\longrightarrow_{me}$ *de* C_2 *será a través de la regla read, generando un evento de la forma ch?x, para algún x. Sólo nos resta ver que* $x = v$, *lo cual es cierto dado que* $I_1(ch) = I_2(ch)$, *pues* $I_1 =_l I_2$ *y* $lvl(ch) \sqsubseteq l$. *Probamos ahora (2). Sabemos que ambas configuraciones transicionan mediante la regla read, entonces, si* $q = I_1(ch)$, $C_1' = \langle th_1[l' \leftarrow f(v)], r_1, ctx_1, s', I_1[ch \leftarrow q], O_1\rangle$ *y* $C_2' = \langle th_2[l' \leftarrow f(v)], r_2, ctx_2, s', I_2[ch \leftarrow q], O_2\rangle$. *Es fácil observar que, asumiendo* $C_1 \approx_l C_2$, *se cumple que* $C_1' \approx_l C_2'$. *La prueba de (3) es inmediata a partir del lema 2* □

Hemos probado que configuraciones similares generan eventos observables similares. Los siguientes lemas consideran los casos en que se transiciona con eventos no observables, o se finaliza la ejecución alcanzando la configuración $\top$. Por razones de espacio omitimos las pruebas.

Lema 6. *Sea* $C = \langle th, r, ctx, s, I, O\rangle$ *una configuración estricta tal que* $C \xrightarrow{e}_{me} C'$, $C' \neq \top$. *Si* $s = l :: s'$, *entonces para cualquier nivel* l' *tal que* $l' \sqsubseteq l$, $C \approx_l' C'$.

Lema 7. *Sean* $C_1 = \langle th_1, r_1, ctx_1, s_1, I_1, O_1\rangle$ *y* $C_2 = \langle th_2, r_2, ctx_2, s_2, I_2, O_2\rangle$ *dos configuraciones estrictas tales que* $C_1 \approx_l C_2$ *y* $l \in \mathcal{L}$. *Si* $C_1 \xrightarrow{\bullet}_{me} \top$, *y* $C_2 \Longrightarrow_{me} B$, *entonces* $silent_l(B)$.

El lema 7 asegura que si dos configuraciones son l similares, entonces si una termina, la otra sólo emitirá eventos no observables para un usuario de nivel l.

Teorema 8. *Sean* $l \in \mathcal{L}$, *e* I, I' *tales que* $I =_l I'$. *Entonces para todo scheduler s, si* $C_{p,s,I} \Longrightarrow_{me} B$ *y* $C_{p,s,I'} \Longrightarrow_{me} B'$, *entonces* $B \simeq_l B'$.

Proof. La demostración es por coinducción en la definición de $\Longrightarrow_{me}$. *Cada uno de los casos que se generan se prueba utilizando los lemas anteriores.* □

Corolario 9. *Para cualquier programa* $p : IO_1$, *su comportamiento bajo multiejecución es no interferente.*

Proof. Inmediata a partir del teorema 8. □

4.2 Preservación Semántica

Una vez demostrado que el procedimiento de multiejecución es seguro, nos preguntamos si es posible preservar la semántica de los programas no interferentes. A continuación probaremos que existe al menos un scheduler que preserva el comportamiento de programas no interferentes. Formalmente, queremos demostrar el siguiente resultado.

Teorema 10. *Sea* $p : IO_1$ *un programa no interferente bajo la semántica* $\Longrightarrow$, *I un sistema de canales de entrada, y* $C_{p,I}$ *su configuración inicial de acuerdo a la def. 2. Existe un scheduler s tal que si* $C_{p,I} \Longrightarrow B$, *entonces* $C_{p,s,I} \Longrightarrow_{me} B$.

Para la demostración de este teorema se sigue un procedimiento similar al de la prueba de no interferencia para multiejecución. La idea es construir una bisimulación (fuerte en este caso) entre $C_{p,I}$ y $C_{p,s,I}$.

Más concretamente, puede demostrarse que, si

$$C_{p,I} = C_0 \xrightarrow{e_1} C_1 \xrightarrow{e_2} C_2 \xrightarrow{e_3} \dots$$

entonces existen configuraciones de multiejecución tales que

$$C_{p,s,I} = C'_0 \xrightarrow{e_1}_{me} C'_1 \xrightarrow{e_2}_{me} C'_2 \xrightarrow{e_3}_{me} \dots$$

encontrando una relación $\approx'_l$ tal que $C_i \approx'_l C'_i$ para cada i. Observemos que esta relación es entre configuraciones de multiejecución y configuraciones de ejecución *normal*.

5 Conclusiones, trabajos relacionados y futuros

En este trabajo hemos presentado una semántica SME para programas interactivos. Nuestro modelo de computación define una abstracción sobre las formas de interacción de un programa con el entorno. Consideramos que la idea de separar la parte *pura* de un programa, i.e. aquellas computaciones que no generan efectos visibles de interacción a través de efectos de entrada-salida; de la parte *visible* nos permite atacar mejor el problema de la seguridad en modelos de computación interactiva, obteniendo resultados que son independientes de los detalles concretos de lenguajes de programación específicos.

Una ventaja que creemos tiene nuestro enfoque es la posibilidad de extender fácilmente el modelo para contemplar otras formas de interacción (por ejemplo, comunicación sincrónica, excepciones). Para esto, sólo es necesario extender el tipo *IO* y *ME* con los constructores apropiados, y capturar la semántica de las nuevas interacciones mediantes reglas apropiadas en las relaciones $\longrightarrow$ y $\longrightarrow_{me}$. Con una adecuada modularización, las pruebas presentadas pueden reutilizarse y extenderse para contemplar las nuevas reglas.

Actualmente nos encontramos trabajando en una generalización de la prueba de preservación semántica, que nos permitirá probar este resultado para una clase más amplia de schedulers. La idea es permitir schedulers en los cuales el comportamiento bajo multiejecución no necesariamente es el mismo que bajo ejecución normal, pero se preserva si uno observa sólo el orden en que se generan los eventos para cada uno de los niveles por separado.

En [7], se presenta una implementación en Haskell de multiejecución segura basada en mónadas. La interpretación de las acciones observables se realiza sobre la mónada IO de Haskell. Nuestro modelo está basado fuertemente en tal implementación.

Recientemente, Bielova et. al [1] presentan un modelo de multiejecución para programación reactiva en el contexto de navegadores, que incluye pruebas de no interferencia y preservación semántica. A diferencia de este trabajo, utilizan ID-security como noción de no interferencia. Creemos interesante incorporar a

nuestro trabajo el análisis de políticas de seguridad utilizadas en navegadores y lenguajes tales como JavaScript.

Existe una gran variedad de políticas de seguridad basadas en no interferencia para sistemas que presentan alguna clase de interacción [2, 5]. Consideramos de interés estudiar nuestro modelo teniendo en cuenta otras nociones de no interferencia además de la aquí presentada (CP-security).

Otro objetivo de nuestro modelo es poder tratar el problema de la declasificación [9, 3], en el cual bajo ciertas condiciones se permite que la información fluya de niveles de seguridad altos a niveles de seguridad menores. Particularmente, estamos interesados en estudiar la corrección de SME en sistemas que permite declasificación, el cual es un problema abierto hasta el momento.

References

1. Natalia Bielova, Dominique Devriese, Fabio Massacci, and Frank Piessens. Reactive non-interference for a browser model. In *Proceedings of the 5th International Conference on Network and System Security (NSS 2011),*, pages 97–104, September 2011.
2. Aaron Bohannon, Benjamin C. Pierce, Vilhelm Sjöberg, Stephanie Weirich, and Steve Zdancewic. Reactive noninterference. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 79–90, New York, NY, USA, 2009. ACM.
3. A. Bossi, C. Piazza, and S. Rossi. Modelling downgrading in information flow security. In *Proc. IEEE Computer Security Foundations Workshop*, pages 187–201, June 2004.
4. D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proc. of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 109–124, Washington, DC, USA, 2010. IEEE Computer Society.
5. Riccardo Focardi and Roberto Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3:5–33, 1994.
6. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.
7. Mauro Jaskelioff and Alejandro Russo. Secure multi-execution in Haskell. In *Proceedings of Andrei Ershov International Conference on Perspectives of System Informatics (PSI'11)*, Akademgorodok, Novosibirsk, Russia, 2011.
8. Kevin R. O'Neill, Michael R. Clarkson, and Stephen Chong. Information-Flow security for interactive programs. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 190–201. IEEE, 2006.
9. A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, volume 3233 of *LNCS*, pages 174–191. Springer-Verlag, October 2004.
10. D. Sangiorgi. *Introduction to Bisimulation and Coinduction.* Cambridge University Press, 2011.