Informe de Trabajo Práctico

Gastón Martínez Castro

27 de Noviembre, 2024

Introducción

Se realizó una implementación de un verificador para el lenguaje de especificación CSP (Communicating Sequential Processes) enseñado en la materia de Ingeniería de Software de la carrera.

Un poco de la motivación para hacer este trabajo fue intentar detectar automáticamente errores en pequeños programas, como una escritura fuera de los límites, una recursión infinita o condiciones de carrera. Para hacer esto, se escribió un programa de ejemplo en C, luego se hizo una especificación en CSP y, por último, se compararon los resultados de la ejecución en C con el mecanismo de aceptación de trazas de CSP.

Ejemplo 1: Escritura fuera de límites

El siguiente programa en C genera eventos para ser verificados contra la especificación en CSP:

```c
#include <stdio.h>

int main() {
    char s[10] = "hello world";
    for (int i = 0; s[i]; i++)
        printf("put.%d.\"%c\"",
            i+1, s[i]);
    for (int i = 0; s[i]; i++)
        printf("get.%d.\"%c\"",
            i+1, s[i]);
}
```

Es necesario notificar cada acción que se realiza para tener una traza que pueda verificarse contra la especificación.

```
MEM.i = put.i?v -> MEM.i.v
MEM.i.v = get.i!v -> MEM.i.v
        | put.i?u -> MEM.i.u

MEMORIA = MEM.1 || MEM.2 || MEM.3
    || MEM.4 || MEM.5 || MEM.6
    || MEM.7 || MEM.8 || MEM.9
    || MEM.10
```

Cuando intentamos validar la traza, el sistema detecta el evento `put.11."d"` como no aceptado, indicando una escritura fuera de los márgenes del arreglo.

Ejemplo 2: Casos base de una recursión

Otra aplicación de las especificaciones es detectar si algún caso base de una recursión no esta definido. Pongamos por ejemplo un programa que calcula los números de Fibonacci.

```c
#include <stdio.h>

int fibo(int N) {
    printf("call.%d", N);
    if (N == 0) return 1;
    return fibo(N-1) + fibo(N-2);
}

int main() {
    fibo(5);
}
```

La especificación en CSP sería:

```
FIBO.0 = call.0 -> SKIP
FIBO.n = call.n -> (FIBO.(n-1);
    FIBO.(n-2))
SPEC = FIBO.5
```

La especificación nuevamente indicará un error. Esto sucede porque el caso general de Fibonacci en la especificación solo está definido para números mayores o iguales a 2. Por lo tanto, cuando se

invoque `FIBO.2`, intentará buscar `FIBO.1` para continuar, pero no encontrará su definición. Esto nos permite identificar qué caso base falta en el programa original.

El lenguaje CSP

La sintaxis del lenguaje CSP usada está basada en el libro *CSP Book* de *Richard Hoare*[1], y el llamado *Machine Readable CSP* descrito en [2]. Las definiciones clave incluyen procesos y operadores:

seleccion externa	::=	`[]`
seleccion interna	::=	`|~|`
interrupcion	::=	`/\`
comp secuencial	::=	`;`
alternativa	::=	`|`
comp paralela	::=	`||`
prefijacion	::=	`->`

proceso	::=	*proceso* `[]` *proceso*
	|	*proceso* `|~|` *proceso*
	|	*proceso* `/\` *proceso*
	|	*proceso* `;` *proceso*
	|	*proceso* `|` *proceso*
	|	*proceso* `||` *proceso*
	|	*evento* `->` *proceso*
	|	*referencia*
	|	*STOP*
	|	*SKIP*

Los eventos se escriben en minúsculas y los procesos en mayúsculas. Los eventos o procesos pueden llevar índices o parámetros, separados por puntos. Cada índice puede ser un valor literal, una variable o una expresión.

Un tipo particular de indexación son los canales (?, !). Estos representan la comunicación mediante mensajes, como en get?v y get!5, donde se asigna un valor a una variable. Según Hoare, son solamente azúcar sintáctico y se deben tratar como índices comúnes, equivalentes a get.v y get.5.

evento	::=	*word* `.` *indices*
	|	*word* `!` *indice*
	|	*word* `?` *indice*
	|	*word*

indices	::=	*indice* `.` *indices*
	|	*indice* `!` *indice*
	|	*indice* `?` *indice*
	|	*indice*

referencia	::=	*WORD* `.` *parametros*
	|	*WORD*

parametros	::=	*parametro* `.` *parametros*
	|	*parametro*

Un archivo completo de CSP incluye sentencias y, opcionalmente, una traza para validación, separadas por "`==== O ====`".

Las sentencias son asignaciones o restricciones para ciertas recursiones infinitas. Cuando se busca un proceso cuyo identificador coincide con una cláusula de restricción, se utiliza el proceso definido en dicha cláusula para resolverlo.

programa	::=	*sentencias* `=O=` *traza*
sentencias	::=	*sentencia sentencias*
	|	*sentencia*
sentencia	::=	*asignacion*
	|	*restriccion*
asignacion	::=	*referencia* = *proceso*
restriccion	::=	*STOP referencia* = *proceso*
traza	::=	$evento_1$ *traza*
	|	$evento_1$

Los eventos de la traza son distintos porque en una secuencia de eventos reales nunca puede haber algún evento indeterminado, como sería put.i.v.

Osea que todos los eventos de la traza tienen indices literales.

Elaboración

Los procesos que incluyen restas se transforman en términos que solo involucran sumas o variables.

De esta forma, la definición de Fibonacci queda claramente establecida solo para los números mayores o iguales a 2.

```
1 FIBO.0 = call.0 -> SKIP
2 FIBO.(n+2) = call.(n+2) ->
3              (FIBO.(n+1); FIBO.n)
4 SPEC = FIBO.5
```

Aceptación de trazas

Los conceptos fundamentales para entender la aceptación de trazas son el paralelismo y el alfabeto de proceso. Una especificación generalmente está compuesta por la paralelización de varios procesos, y para que un evento sea aceptado, debe ser aceptado por la paralelización.

Para que una paralelización P || Q acepte un evento e, deben aceptarlo P y Q simultáneamente o no aceptarlo ninguno. Si solo lo acepta P (o solo Q), entonces el evento e no puede existir en el alfabeto de Q (o de P). Para comprender la razón de esto, es necesario entender el significado del operador de prefijación. Si el evento e no es un posible primer evento de Q (no es aceptado) pero existe en el alfabeto de Q, entonces existe una secuencia de prefijaciones $f_1 \to f_2 \to \cdots \to f_m \to e$, que establece que e no puede preceder a $f_1, f_2, \ldots, f_m$.

Para ver el alfabeto completo de un proceso P, debemos incluir todos los eventos en el cuerpo de P así como los eventos de cualquier proceso alcanzable desde P. El proceso *RP*.1 tras aceptar el evento *deteccion*?*b*, sigue teniéndolo en su alfabeto,, ya que se llama recursivamente. Esto significa que, si *RP*.1 está paralelizado con otros procesos que utilizan este evento, primero se ejecutarán *start*, *disponible*.1 (o *disponible*.2), *stop*, etc., y luego *deteccion*?*b*.

La implicancia de este simple principio es que las especificaciones pueden transmitir mucha más información y permitir muchos mas comportamientos de los que aparentan.

```
1  RP.1 = deteccion?b -> start ->
2       ( disponible.1 -> stop ->
3       tracking.b -> objetivo.1!b ->
4           RP.1
5       | disponible.2 -> stop ->
6       tracking.b -> objetivo.2!b ->
7           RP.1)
8
9  <RP.1 tras deteccion.1> = start ->
10      ( disponible.1 -> stop ->
11      tracking.b -> objetivo.1!b ->
12          RP.1
13      | disponible.2 -> stop ->
14      tracking.b -> objetivo.2!b ->
15          RP.1)
```

La aceptación de P [] Q, P |~| Q y P | Q:

- Si P y Q aceptan el evento, entonces P [] Q y P |~| Q lo aceptan y pasan a "P/e [] Q/e" y "P/e |~| Q/e", respectivamente. Esto es no determinismo angélico. La alternativa P | Q no acepta no determinismo.
- Si solo P acepta el evento (o solo Q), el evento se acepta en los tres casos y pasa a ser P/e (resp. Q/e).
- Si ninguno de P o Q lo aceptan, en los tres casos se rechaza.

La prefijación f -> P acepta el evento e (sin variables ni expresiones) si tiene el mismo nombre que f, la misma cantidad de indices y los indices coinciden literal con literal, literal con variable o literal con expresión*. *La única excepción es que un termino como n+2 no puede asignarse con literales que hagan a n menor a 0, como 0:=n+2 o 1:=n+2.

Los procesos STOP y SKIP no aceptan ningun otro evento. Estos son procesos base para denotar terminación errónea y terminación exitosa, respectivamente.

La composición secuencial P ; Q acepta los eventos de P hasta que este termine exitosamente. Cuando P sea *SKIP* entonces se pasa a comportar como Q: $SKIP; Q = Q$

La interrupción P /\ Q se comporta como P hasta que ocurra el primer evento de Q en cuyo caso pasa a comportarse como él. Igual que la alternativa, este operador no soporta no determinismo y el verificador nos avisará cuando ambos acepten un evento al mismo tiempo.

Recursión infinita

Es posible definir un proceso que recursione infinitamente utilizando esta sintaxis. Si deseamos conocer el alfabeto de un proceso como este, el alfabeto podría no ser finito. Por lo tanto, en el procedimiento de aceptación de trazas, que busca los alfabetos de los procesos, esta situación llevaría a una recursión infinita y la evaluación no se completaría nunca. Para evitar esto, es necesario mantener los alfabetos de los procesos acotados. Un ejemplo de cómo hacerlo se puede encontrar en los archivos de ejemplo 'examples/Horno.csp', donde el proceso TEMPORIZADOR debe ser limitado con

una cláusula STOP, ya que la posibilidad de incrementar el tiempo podría ejecutarse indefinidamente. Esta es una limitación razonable, dado que cualquier microondas tiene un límite de tiempo que se puede establecer.

Ejemplos

Se proporcionaron algunos ejemplos de validación de programas en C, junto con ejercicios específicos de la materia de ingeniería para probar el verificador. Si se desean probar algunas de las versiones alternativas de estas especificaciones (identificadas con un 1) se podrán observar los mensajes de error generados. Para obtener una vista más detallada del proceso de aceptación, se puede agregar la bandera "-d" para habilitar el modo de debugging.

Escribiendo especificaciones

Al momento de escribir una especificación y asegurarse que los comportamientos esperados son aceptados, el proceso de debuggeado es inevitable. Uno de los principales aportes de este proyecto es que permite identificar cuando una especificación aparentemente correcta presenta problemas, proporcionando mensajes de error que facilitan su corrección y permitiendo la rápida iteración entre varias versiones.

Referencias

[1] Charles Antony Richard Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985. Disponible en `http://www.usingcsp.com/cspbook.pdf`.

[2] Alexandre Boulgakov A.W. Roscoe Thomas Gibson-Robinson, Philip Armstrong. *Failures Divergences Refinement (FDR) Version 3*, 2013.