# Resumen Final ISW2 para rendir con JP Galeotti

*Disclaimer: Este es un resumen hecho en base a los videos que se publicaron en la cursada del 1C – 2021 dictada de manera virtual. La utilidad del mismo es brindar un apoyo complementario y bajo ningún concepto es material oficial ni un documento autocontenido de la materia.*

# Introducción al análisis de Programas

**El análisis de programas es el proceso de descubrir automáticamente hechos útiles sobre los programas,** por ejemplo, un error de programación.
Este análisis se puede clasificar en tres:

- Dinámico. Se produce en tiempo de ejecución, observando el comportamiento del programa mientras corre. **Infiere hechos sobre el programa monitoreando sus ejecuciones**. Valgrind es un ejemplo.
- Estático. Se analiza el código fuente luego de compilado. Lint es un ejemplo.
- Hibrido. Combina ambos modelos.

## Consumidores de Análisis de Programas

Fundamentalmente, hay tres tipos de consumidores, y para ellos y para cada caso se deberá hacer un trade off entre terminación del análisis, completeness y soundness:

- Compiladores. Lo hacen para poder simplificar código. Como ejemplo podemos detectar que una variable para ambas ramas de un if siempre tiene el mismo valor, entonces nos podríamos ahorrar el cálculo de esa variable dentro de ese if.
- Herramientas de calidad de software. Lo utilizan para encontrar invariantes, generar casos de test, encontrar errores de programación, etc
- IDEs. Usan estos análisis para ayudar al programador a entender mejor el programa, sugerir refactors, etc

Un análisis de programa es complete cuando, si reporta un hecho, el hecho siempre es verdadero. (Notar que podría no reportar todo). Es decir, da una sub aproximación de los hechos.
Un análisis de programa es sound cuando, si un hecho es verdadero, siempre lo reporta. (Notar que, si siempre reporta todo, en particular reporta los hechos reales). Es decir, da una sobre aproximación de los hechos.

## Análisis Dinámico

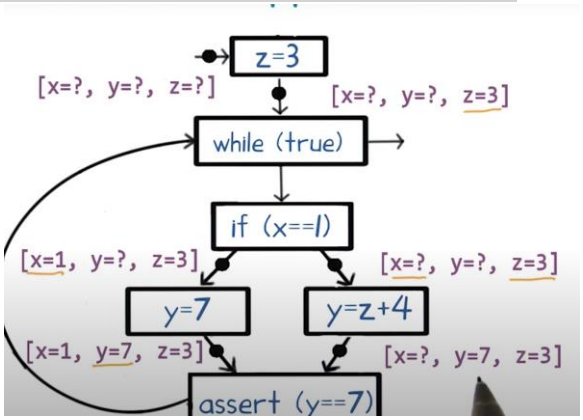
Los analizadores dinámicos pueden intentar detectar invariantes dentro de los programas. Ahora bien, como éstos solo corren un número finito de veces, **solo pueden llegar detectar posibles invariantes** (aunque en todas las corridas el resultado haya sido el mismo), pero no afirmarlos con certeza. De todas maneras, sí pueden descartar invariantes que no pueden darse en el programa, al existir distintos valores para una variable ya puede descartar que no va a haber un invariante.
El costo de ejecución de este tipo de análisis es proporcional al de ejecución del programa.
El análisis estático puede complementar la información y llegar a concluir invariantes con certeza.
**Como el análisis dinámico solo prueba con un conjunto finito de valores, podría perderse errores, es decir que no es sound, pero es complete (cuando reporta un error, efectivamente lo es).**

## Análisis estático

Como herramienta típica para poder operar mediante el análisis de programas tenemos el Flow Control Graph, que muestra todas las ejecuciones posibles de un programa y, siguiendo esos caminos y analizando sus nodos (estado único del programa) predecesores y sucesores se pueden sacar conclusiones generales del programa en cada momento dado.
El costo de ejecución de este tipo de análisis es proporcional al del tamaño del programa.

Para cada punto del programa se tiene un **estado abstracto**, en contraste con estados concretos, que son los valores reales que se obtienen en una ejecución en particular (análisis dinámico). Es por eso que, para ciertos estados, no puede precisar qué valor tienen algunas variables ya que este análisis no opera con valores concretos.

Programas que tienen ciclos o caminos ilimitados, pueden hacer que el analizador pierda valores constantes. **Es por eso que este tipo de análisis sacrifica Completitud, pero siempre termina. Además, cada vez que concluye el valor de una variable, ese valor valdrá para todas las posibles ejecuciones de un programa** (soundness).




**NO SE PUEDE TENER SOUND Y COMPLETITUD SI QUEREMOS GARANTIZAR QUE EL PROGRAMA TERMINE**

Undecidability => program analysis cannot ensure termination + soundness + completeness

## Dataflow Analysis

Consiste en utilizar el lenguaje **WHILE** combinado con **Control-Flow Graphs,** y es una herramienta para realizar análisis **estáticos** de programas. Existen diversos análisis que se pueden realizar que varían en lo que reportan y en cómo se van construyendo los conjuntos IN and OUT de información que va recolectando. La particularidad que mantienen es que utilizan el algoritmo de iteración caótica.

Una forma de ver que es sound e incomplete es que siempre en el caso de un branch, va a analizar las dos ramas, asumiendo una elección no determinística.

Esto quiere decir que todo lo real lo reportará (ya que cubre todos los caminos), pero que, si había caminos inalcanzables también los reportará (incomplete).

Una forma simple de entender un dataflow analysis es que, para cada punto del programa (siguiendo su flujo) va a reportar hechos, que dependerán del tipo de análisis que se esté realizando.

### Algoritmo de Iteración Caótica

Utiliza las definiciones de construcción de los conjuntos de acuerdo al método que esté utilizando y va iterando para cada nodo, actualizando los conjuntos IN (conjunto de hechos a la entrada de un cierto punto) y OUT (conjunto de hechos a la salida de un cierto punto del programa). **Constantemente recorre todos los nodos y actualiza ambos conjuntos hasta alcanzar un punto fijo. El orden en que se visitan los nodos no importa (de ahí, caótico).**

Este algoritmo **siempre termina**, y lo hace cuando la tabla de definiciones deja de cambiar.

Tiene garantizada la terminación, ya que los conjuntos IN y OUT son monótonos no decrecientes y finitos, entonces no pueden crecer infinitamente para los algoritmos MAY, y al revés para los MUST son monótonos no crecientes, entonces no pueden achicarse más del conjunto vacío.

Dataflow Analysis computa IN y OUT para nodo del flujo, definiendo como calcular ambos conjuntos en cada tipo de análisis y también utilizando conjuntos auxiliares como KILL y GEN.

## Tipos de Análisis

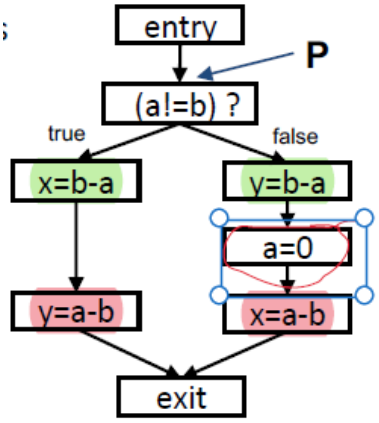
- Reaching Definitions: Provee información sobre variables potencialmente no inicializadas en un programa.
- Very Busy Expresions: Puede ayudar a reducir el tamaño del código.
- Available Expressions: Produce información que puede ser usada para evitar volver a computar una expresión.
- Live Variables: Produce información para asignar registros de manera eficiente a las variables del programa.

### Reaching Definitions

Busca determinar para cada punto del programa, cuales son las asignaciones previamente hechas y de qué punto vienen. Si a una variable se le asigna un valor en el punto P0 y luego no se modifica, cuando nos paremos en el punto P1, tendremos la información que esa variable está definida y su valor no fue sobreescrito. Por el contrario, cuando se produce una sobre escritura en un punto P2, deberemos afirmar que no vale la definición de P0 en ese punto del programa.

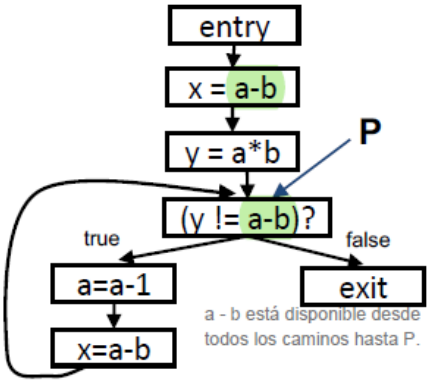
### Very Bussy Expressions

Busca expresiones que estén "ocupadas" a la salida de cada punto del programa. Una expresión está muy ocupada si, sin importar que camino se ejecute, la expresión siempre se usa antes que cualquiera de las variables.
Otra forma de definirlo puede ser como que, no importa por qué camino continúe la ejecución desde el punto P, la expresión E va a volver a ser usada antes que cambie el valor de las variables que la componen.
En el ejemplo la expresión E = *b-a.* Esa expresión se va a usar antes de que b o a se redefinan, por ende, podría calcularla una única vez antes del IF y se reduciría el código.




### Available Expressions

Busca para cada punto del programa qué expresiones ya fueron computadas y no fueron modificadas en todos los caminos hasta ese punto.
En el ejemplo la expresión *a-b* está disponible en P, ya que ni a ni b se modifican en ninguno de los dos caminos (ya que se calcula en el paso 2 y se recalcula en el paso 6). De esta forma, se puede reutilizar el valor pre calculado en esos pasos y ahorrar ese cómputo.




### Live Variables

Busca determinar para cada punto de programa que variables podrían estar vivas a la salida de cada punto. Una variable está viva si hay un camino hacia un uso de la variable que **no** la redefine.

## Patrón común de los 4 algoritmos

En resumen, los algoritmos son forward (predecesores a sucesores) o backward. Además, si los conjuntos toman uniones son "may" (*algún camino*) y si toman intersecciones son "must" (*todos los caminos*).

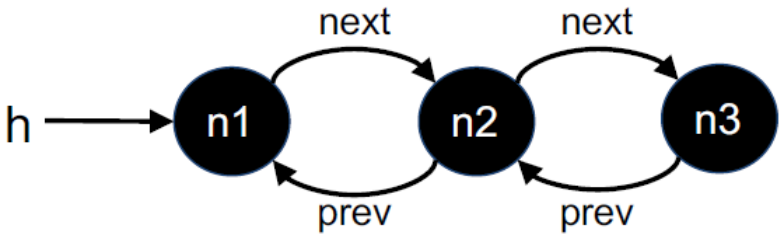
| | May | Must |
|---|---|---|
| Forward | Reaching Definitions | Available Expressions |
| Backward | Live Variables | Very Busy Expressions |

Notar que "Expressions" son MUST. Es decir que se parte de conjuntos con todas las expresiones y se van reduciendo.

## Pointer Analysis (May-Alias)

**El objetivo es razonar sobre flujos de datos no primitivos, es decir, punteros, objetos y referencias.**

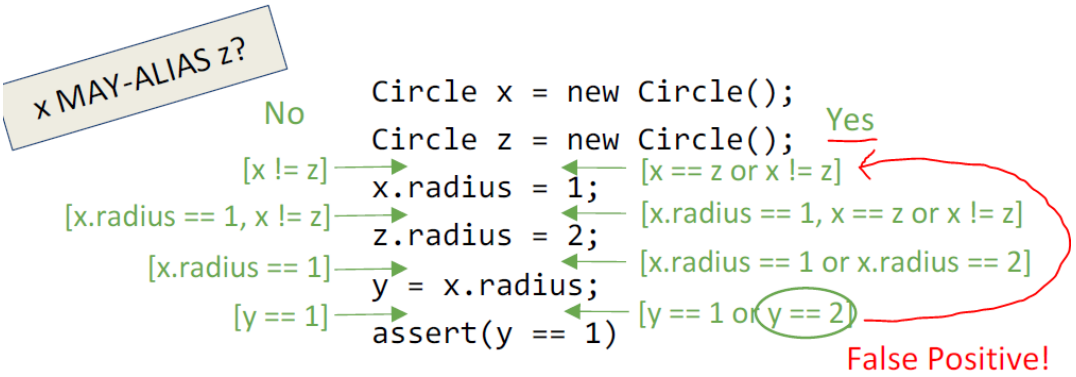
Este tipo de análisis es más desafiante que el dataflow analysis sin punteros, ya que gracias al pointer aliasing, un mismo espacio de memoria podría ser accedido por potencialmente infinitas expresiones distintas. En el ejemplo vemos que se puede acceder a la variable data, literalmente de infinitas formas por ser una lista doblemente enlazada:




h.data
h.next.prev.data
h.next.next.prev.prev.data
h.next.prev.next.prev.data

Hacer un seguimiento de todas estas posibles expresiones es poco performante como mínimo, e inviable como máximo. Pero podemos notar que todas esas expresiones son aliases entre sí, ya que acceden al mismo puntero al final.

**El problema de decidir si un puntero es alias de otro es en general, indecidible.** Para tener un algoritmo que termine, se debe sacrificar completeness, es decir que podremos obtener falsos positivos (más casos de los que realmente puedan ser, es decir, una sobre aproximación), pero no falsos negativos (es decir, que no hay hechos reales que no reporte). Claramente, estaremos en presencia de un analizador estático.
Como el algoritmo es **MAY**, que indique que dos punteros son alias, quiere decir en realidad que no puede decidir que no lo sean, entonces podrían serlo o no, eso duplica los posibles valores que después van tomando los punteros y claramente va a estar agregando un falso positivo.

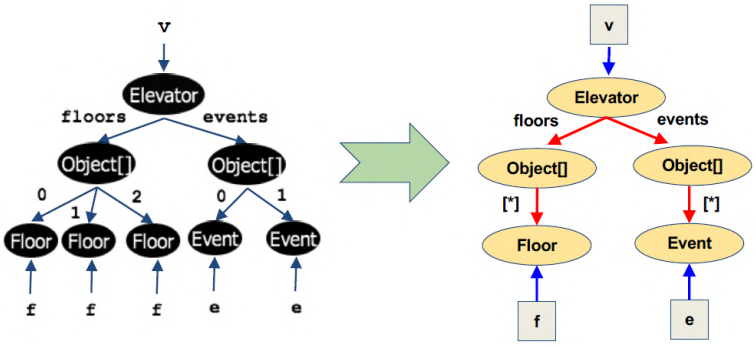



La tasa de falsos positivos puede variar según el tipo de análisis usado.
Basicamente, existen 4 aspectos a tener en cuenta para clasificar los distintos análisis de punteros:

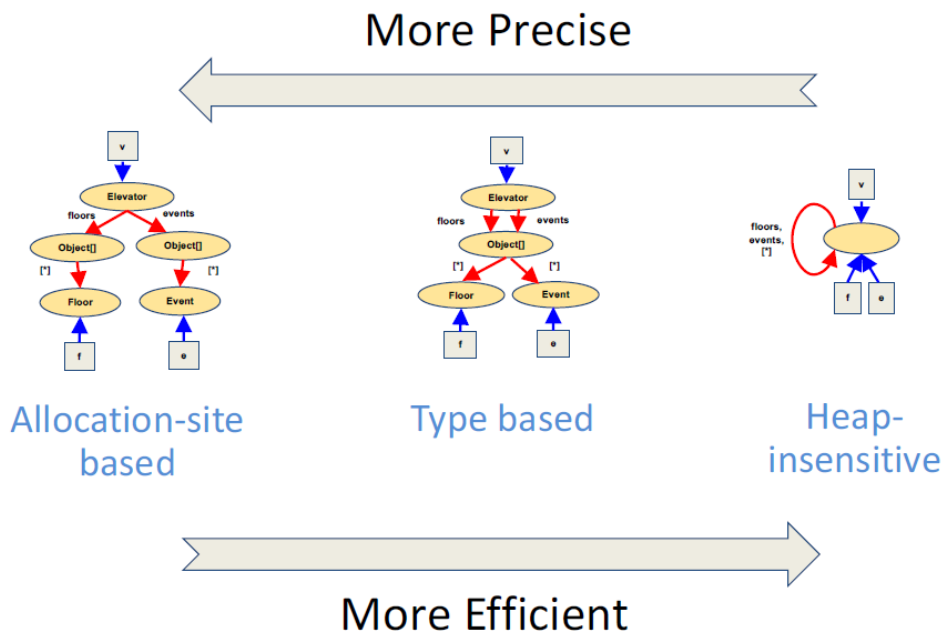
1. Es sensible a flujo
2. Es sensible a contexto
3. Qué tipo de Heap Abstraction usa
4. Como se modela el tipo de datos agregados.

## Heap Abstraction

Una técnica consiste en crear un grafo que abstraiga el heap y que cree un grafo que tengo nodos representando allocation sites, es decir, eventos donde una operación de reserva de memoria tiene lugar. También tienen nodos de variables, que apuntan al nodo de alocación de donde se originaron.
De esta manera, un árbol potencialmente infinito pasa a ser uno mucho mas compacto y finito, asegurando la terminación. La eficiencia está dada según cuantos objetos abstractos el algoritmo pueda crear.




Notar el colapso de cada asignación de floor y event.




Este tipo de análisis sufren de **insensibilidad de flujo**. Es decir que se eliminan del programa todas las nociones de orden del mismo. En cuanto al esquema de heap abstraction, podemos ver en la imagen que cuanto más abstrae, más eficiente será, pero también más impreciso.

### Allocation Site Abstraction

La forma de crear el grafo de allocation sites también es mediante iteración caótica donde se van agregando nodos visitando cada expresión hasta que el mismo deja de cambiar.
Dada la gramática para este análisis, se siguen una serie de reglas:

- Allocation Site Rule: para cada variable se crea una flecha que apunte a su allocation site. Si la misma variable apunta a dos distintos, se tendrán dos flechas.
- Copia: Si se produce una copia de una variable a otra, v1 = v2, entonces v1 **agregará** el allocation site de v2 como una nueva arista, mientras que v2 se mantendrá sin cambios.
- Escritura: Casos del tipo v1.f = v2. V2 permanecerá sin cambios, v1 tambien, pero se agregará al allocation site de f, el allocation site de v2. Es decir que se agrega una relación a nivel allocation site, y no variables.
- Lectura: Se agrega una arista desde la variable donde se almacena al allocation site del objeto que se está leyendo.

## Análisis basado en restricciones

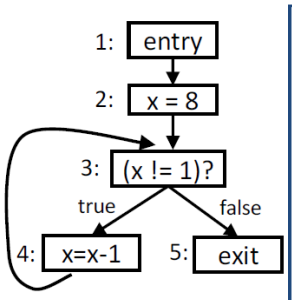
Este tipo de análisis se refiere a la **especificación**, más que a la implementación. Es decir, que vamos a crear restricciones sobre los hechos del programa. La idea es construir un programa para especificar tipos de análisis.
Diseñar un programa eficiente de análisis debe enfocarse más en la especificación que en la implementación.
Nosotros vimos Datalog, que permite especificar dos tipos de análisis estáticos:

- Intraprocedural: es decir, un análisis que está limitado a un solo procedimiento.
- Interprocedural: análisis que involucra múltiples procedimientos.

Como ejemplo intraprocedural esta Reaching Definitions que, partiendo de las definiciones originales de los conjuntos IN, OUT, KILL y GEN, se busca generar esas relaciones (restricciones, constraints) en el lenguaje y establecer las reglas que las fórmulas determinan.

**Input Relations:**
```
kill(n:N, d:D)
gen (n:N, d:D)
next(n:N, m:N)
```

**Output Relations:**
```
in (n:N, d:D)
out(n:N, d:D)
```



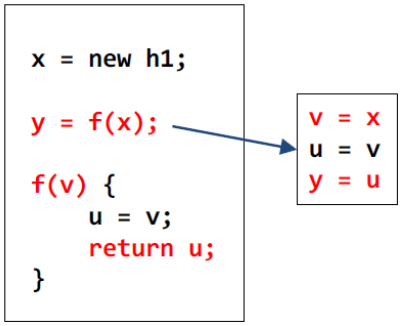

**Rules:**
```
out(n, d) :- gen(n, d).
out(n, d) :- in(n, d), !kill(n, d).
in (m, d) :- out(n, d), next(n, m).
```

**Input Tuples:**
```
kill(4, 2),
gen (2, 2), gen (4, 4),
next(1, 2), next(2, 3),
next(3, 4), next(3, 5),
next(4, 3)
```

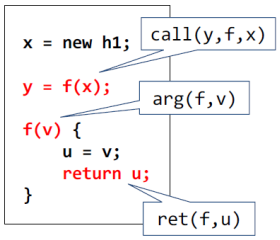
**Output Tuples:**
```
in (3, 2), in (3, 4), in (4, 2),
in (4, 4), in (5, 2), in (5, 4),
out(2, 2), out(3, 2), out(3, 4),
out(4, 2), out(4, 4), out(5, 2),
out(5, 4)
```

## Análisis Inter-procedural

Este análisis sirve para las llamadas a funciones y como el contexto interactúa entre ellas.
En una función simple que solo asigna, podemos generar un nuevo programa reemplazando la llamada de una función, por el contenido de la misma. Entonces, el pasaje de parámetros y el valor de retorno se pueden tratar como simples asignaciones.




Al conjunto de relaciones de entradas que teníamos le agregamos nuevas para declarar una función y sus parámetros (arg), lo que retorna (ret) y realizar la llamada con los valores de entrada y de retorno.




**Input Relations:**
```
new(v:V, h:H) arg(f:F, v:V) ret(f:F, u:V)
assign(v:V, u:V) call(y:V, f:F, x:V)
```

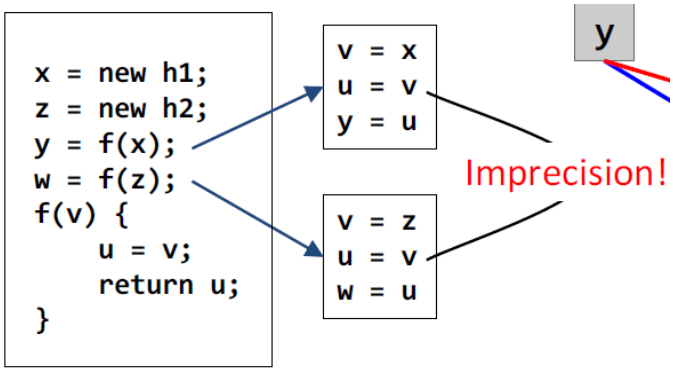
**Output Relations:**
```
points(v:V, h:H)
```

**Rules:**
```
points(v, h) :- new(v, h).
points(v, h) :- assign(v, u), points(u, h).
points(v, h) :- call(_, f, x), arg(f, v),
                points(x, h).
points(y, h) :- call(y, f, _), ret(f, u),
                points(u, h).
```

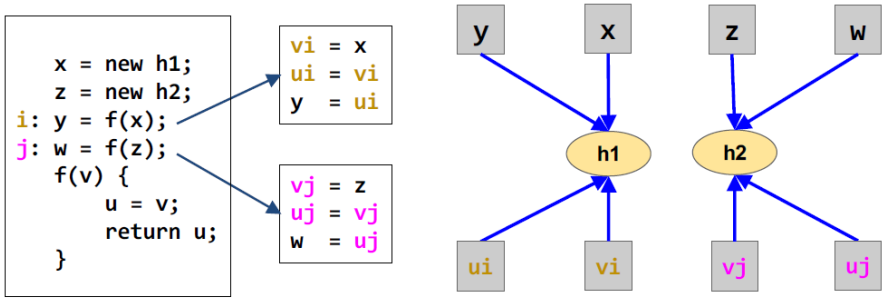
Este análisis es insensible al contexto, ya que todas las llamadas a f se tratan como una sola. Esto, puede ser más performante, pero pierde precisión.

## Sensibilidad al contexto

Consideremos esto




Para agregar sensibilidad, se utiliza la "clonación", es decir que se crean copias de las variables de parámetros y de retorno de las llamadas a f, para cada llamada.




Achieves context sensitivity by **inlining** procedure calls

Cloning depth ⬆: precision ⬆ vs. scalability ⬇

Tener en cuenta que, a mayor profundidad, si seguimos la clonación, se puede dar un crecimiento exponencial en las variables (por ejemplo, si cada función llama a otra función), lo que afecta gravemente la escalabilidad y performance del algoritmo. De hecho, si hay llamadas recursivas, se necesita una profundidad infinita de clonación, lo que es impracticable.

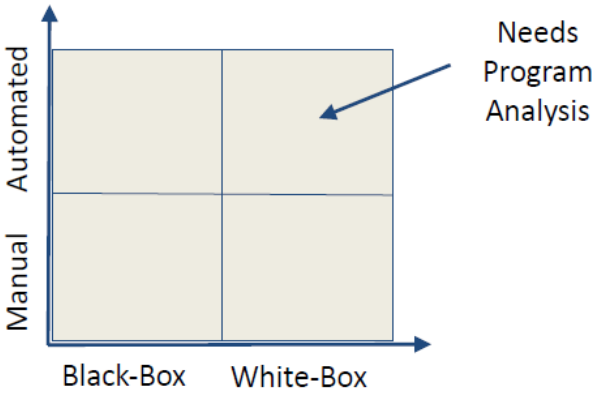
# Testing Automatizado

Proceso para verificar la corrección de un programa. Para poder determinar la correctitud**, las especificaciones deben ser explícitas** para que no haya incoherencias entre los desarrolladores y los testers; sin una especificación, no hay nada para testear. Además, **los tests y el desarrollo deben ser independientes**, para que no haya sesgos y se puedan detectar efectivamente errores de desarrollo. El desarrollador escribe la implementación; el tester escribe la especificación.
También es importante saber que las especificaciones no son estáticas, cambian y evolucionan con el tiempo, y es necesario poder adaptarse.

**Testing puede verse como una forma de chequear consistencia entre una implementación y una especificación**

**Dado que los recursos son finitos, típicamente no se crean tests para cubrir el 100% de los casos posibles. Además, la especificación evoluciona a través del tiempo. De ahí surge la idea de automatizar su creación.**

Las distintas técnicas de testing pueden ser clasificadas de la siguiente manera:




El eje Y hace referencia a cuanta interacción humana se necesita, mientras que el X, a cuanto acceso tiene el testing al código del programa. De todas formas, en la práctica se usan técnicas hibridas, por lo que no hay que mirar el gráfico como categorías discretas.
Un tipo de testing automatizado que actúa en consecuencia de las respuestas obtenidas por la aplicación o, monitorear el código mientras se está ejecutando para descubrir futuros tests, es un ejemplo de arriba a la derecha. Pero se necesita hacer análisis de programa como vimos al principio.

### Terminología

- Defecto: Error estático en el programa cometido por el programador. Está en el programa sin importar si es o no ejecutado.
- Infección: Se produce por la ejecución del defecto.
- Propagación: Ejecución o traza incorrecta transitando por estados que no debería pasar debido a haberse infectado.
- Falla: Eventualmente se produce un error externamente visible del programa, ya sea por un output incorrecto o por detenerse la ejecución abruptamente.

El desafío del testing es, entonces disparar el error. Esto es, lograr que el defecto se ejecute, se propague y resulte en una falla. Más aún, debemos poder reconocer en el testing que eso es efectivamente un error, es decir, que el test también falle.

### El problema del testing automatizado

Una ventaja obvia de testing automatizado es que tiene la capacidad de encontrar bugs más rápidamente, porque tiene la capacidad de emitir inputs y chequear outputs mucho más rápido. Además, no hay necesidad de escribir ni mantener tests ya que se hace automáticamente.
De todas formas, éstos pueden no ser tan eficientes y tener tan buena cobertura de código como los manuales realizados por el análisis de un ser humano.
Los humanos tienen la capacidad de escribir test suites mejores, menos triviales y tener potencialmente mejor cobertura. Es por eso que para tomar lo mejor de los dos mundos se usa un método hibrido, por ejemplo, especificando la gramática valida de los inputs manualmente y permitir que corran todos automáticamente.
Si bien, nos vemos tentados a querer automatizar todos los tests posibles, es una tarea difícil de lograr, aún para programas pequeños, el número de caminos de un programa crece mucho por cada branch del programa o loop, lo que hace difícil testear todas las instancias posibles.

## Especificaciones de un programa

Está claro que sin especificaciones no hay nada que testear, mucho menos poder automatizar.

Un mecanismo clásico de especificación es el de las Pre y Post condiciones. Una pre condición es un predicado que se asume válido antes que una función se ejecute. Una post condición es también un predicado que se asume válido luego de ejecutada la función y siempre y cuando se haya cumplido la pre.
Muchas veces no se busca cubrir todos los aspectos posibles por una post condición, ya que podría volverse más compleja que el código mismo que está intentando cubrir. Por eso, sacrifica eficiencia y agrega generalidad, aumentando la practicidad.

### Test Suite Quality

¿Cuán buena es nuestra suite de tests? ¿Cómo sabemos si escribimos demasiados, o demasiado pocos?
Con muy pocos tests, se pueden pasar algunos bugs que no estemos chequeando. Con demasiados, puede ser costoso para correr, más difícil de mantener, y puede haber redundancia.
Existen métricas para chequear nuestro test suite:

- Medidas de Cobertura de código: Por ejemplo, verificando si cada sentencia del código se ejecutó al menos una vez en la corrida de los tests. Se mide en porcentaje de algún aspecto del programa ejecutado en el test.
  Se pueden medir, por ejemplo, la cobertura de funciones (que y cuantas funciones se llamaron), cobertura de sentencias, cobertura de caminos tomados, etc.
- Análisis de mutaciones: Se hace mutar de manera random el programa y luego se ejecutan los tests en el código mutante. Si ningún test falla en el código mutante, puede indicar que el test suite no es lo suficientemente fuerte para distinguir código correcto de incorrecto.
  Esta estrategia se basa en que el programador es competente y el programa es casi correcto de entrada, es decir que no debería haber errores en la lógica central del mismo. Ejemplos de mutaciones pueden ser cambiar un símbolo “<” por “>”, una suma por una resta, etc. Un test suite robusto debería fallar por cada mutación agregada.
  Los cambios que se hacen son leves y sintácticamente válidos.

Podemos observar que aun teniendo un 100% de cobertura de sentencias, de todas maneras, podríamos tener un test suite no lo suficientemente bueno. Ya que, que una sentencia se haya ejecutado, pero después no hayamos chequeado el resultado de esa sentencia, podría hacernos perder potenciales bugs. Mutation Analysis busca resolver esto, en al menos alguna medida, ya que hace pequeñas variaciones del programa y espera que sean descubiertas por el test suite.

## Mutation Analysis

Como acabamos de ver, juzgaremos la efectividad de un test suite midiendo cuan bien puede encontrar defectos artificialmente introducidos en el programa.
Los mutantes son nuevos programas, muy parecidos al original salvo por algún cambio sintácticamente valido de alguna sentencia. Es decir, se introduce un typo. Si algún test del suite falla al ser corrido con el mutante, entonces decimos que fue detectado. Si al menos uno sobrevive, entonces quiere decir que se necesitan más o mejores tests para lograr también matarlo.
**Mutation Score = Mutantes muertos / Mutantes Totales.**

### Generación

Usamos operadores de mutación, que son reglas para derivar mutantes a partir de un programa.
También se pueden usar mutaciones basadas en fallas reales o errores típicos según el tipo de proyecto.
Otra manera un poco menos específica, es usar los operadores genéricos dado el lenguaje del programa original.
Un ejemplo de operador de mutación puede ser el de relaciones, que buscara modificar signos de comparación por otros (“>” por “<”, etc).

Existen mutaciones orientadas a objetos, por ejemplo, cambiar modificadores de acceso, sobreescribir métodos, eliminar constructor de clase base, etc. **(Tener en cuenta siempre, que el mutante debe compilar).**

## Mutantes Equivalentes

¿Qué pasa si un mutante es equivalente al programa original? Entonces ningún test va a fallar. ¿Cómo distinguimos si ningún test falló porque nuestro test suite es malo o porque era equivalente?
Una mutación es un cambio sintáctico, pero no siempre semántico. Es difícil detectar un mutante equivalente (indecidible). Puede pasar que se alcance el defecto y no producirse una infección, o producirse, pero no llegar a propagarse.

```
if(x > 0) {
  if(y > abs(x)) {
    // ...
  }
}
```

¿Es equivalente? SI

## Performance

Es fácil ver que, dado un programa y un cjto de operadores de mutación, la cantidad de mutantes que se pueden llegar a generar es altísima. Ademas, cada mutante es un programa independiente que se debe compilar y ejecutar para todo el suite, por lo que se puede volver fácilmente una tarea impracticable.
Buscaremos estrategias como hacer menos mutantes, hacerlos más inteligentes, o introducir cambios en código ya compilado.

### Uso de cobertura

Si sabemos de antemano qué parte del programa cubre cada test, entonces podemos hacer correr para un mutante solo los tests que involucran la mutación.

### Mutación Débil vs. Fuerte

Otra herramienta para ganar escalabilidad, es solo contemplar las mutaciones débiles. Es decir, solo chequear que se haya producido una infección (un cambio en el estado interno del programa esperado) sin necesidad de continuar ejecutando hasta que se propague y falle (mutación fuerte). Esto ahorra mucho tiempo de ejecución.
El estado interno se compara contra el estado interno del programa original. Es decir que si por ejemplo, debido a un mutante, una variable auxiliar cambia su valor, independientemente de si se propague o no la falla, declaro mutante muerto.

### Meta-Mutantes

Genera una única versión del programa con todos los mutantes embebidos y, mediante un keyword de configuración (tipo switch), elijo que mutante quiero activar en el programa y listo. Se ahorra tiempo de compilación.

# Random Testing (Fuzzing)

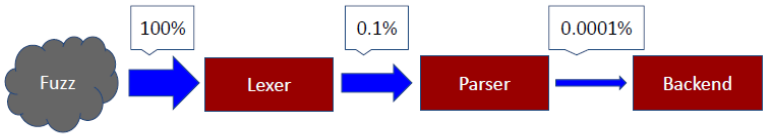
La idea es dar a un programa un cjto de inputs aleatorios y se observa el comportamiento del programa.
Random testing es un caso especial de mutation analysis, ya que puede pensarse como que se hacen modificaciones a la entrada (en cambio en mutation analysis se hacen cambios arbitrarios a cualquier punto del programa).
Random testing es un paradigma, no es una técnica out of the box que funciona en cualquier programa. Para que funcione bien, los inputs deben generarse con una distribución razonable.

Una aproximación aparentemente ingenua de esto es, generar inputs totalmente random y bombardear una aplicación. Sin embargo, puede darnos una medida de robustez de nuestro programa y puede ser útil en algunos dominios, como el de seguridad.
Existen otras herramientas (monkey tool para Android) que genera toques random en la pantalla y también tiene la capacidad de generar una llamada entrante, un cambio en el GPS, etc.
Otro dominio en el cuál es muy útil ese en el de programas concurrentes. Como el orden de la ejecución de los threads no es determinístico, un programa para un mismo input podría crashear en una ejecución y no en otra. Entonces, se busca generar un fuzzing del scheduler, agregando delays aleatorios, ya que eso reduce la prioridad de un thread para ejecutarse.

- ~~Previously:~~ Random testing (Fuzzing)
  - Security, mobile apps, concurrency

## Desventajas

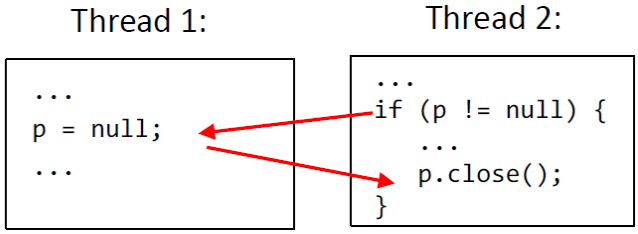
Una gran contra de Fuzzing es que puede ser muy útil para testear primeras etapas de un programa, pero puede ser muy ineficiente para testear etapas más tardías, lo que genera una cobertura de código baja.
También puede encontrar bugs que son poco importantes.
**Random Testing debe ser usado como complemento de testing** y no como reemplazo de otras técnicas, ya que puede cubrir muchos casos muy rápidamente, pero podría en la práctica no llegar nunca a cubrir casos que pueden ser importantes.




- The lexer is very heavily tested by random inputs
- But testing of later stages is much less efficient

## Profundidad del bug

Tiene sentido en programas concurrentes y es un número que se calcula de acuerdo a la cantidad de sentencias ordenadas **entre distintos threads** que deben suceder para desencadenar un bug.

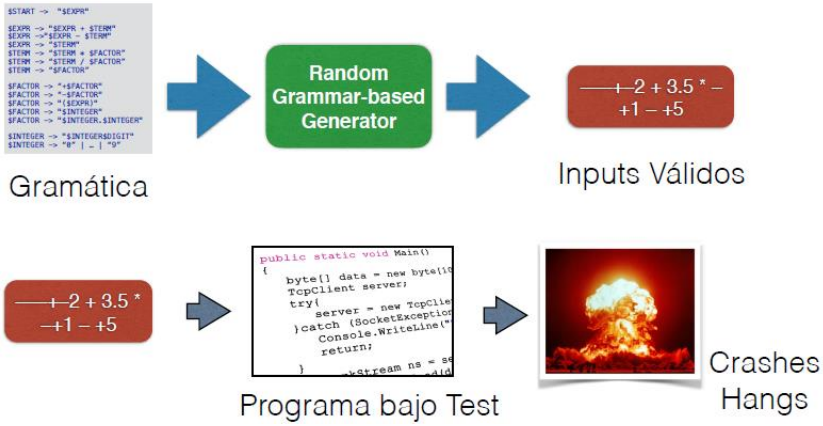



Este ejemplo tiene profundidad 2, ya que primero debe ejecutarse la condición del IF del Thread 2, **luego** la asignación a null de p, **y luego**, la llamada a p.close.

Cuanto más profundo el bug, más ordenadas se deben dar las secuencias de threads para poder explotarlo, es decir, más requisitos se deberán cumplir previamente. Sin embargo, empíricamente se ha mostrado que los bugs típicos tienen una profundidad baja.

## Grámaticas

A partir de derivar reglas de una grámatica (L->R, donde L es no terminal y R es o no terminal) se pueden generar aleatoriamente inputs válidos para el random testing. Es más útil que una generación totalmente aleatoria cuando tenemos cierta estructura sobre los inputs, como por ejemplo, links, mails, números con decimales, etc.




Un algoritmo que las genere es bastante simple. Tiene un loop que mientras tenga la gramatica símbolos no terminales, elige una gramatica al azar para ese símbolo no terminal y lo reemplaza (chequeando una cota de símbolos máximos predefinida).
Un ejemplo de herramienta es LangFuzz para Mozzila que tiene un Fuzzer que genera programas JS validos con el objetivo de detectar vulnerabilidades de seguridades. Ademas, tiene la capacidad de extender la gramatica que posee con nuevos terminales de fragmentos de código que en el pasado ocasionaron vulnerabilidades. De esta manera, se asegura atacar fixes parciales. Su gramática entonces contiene un lenguaje generador de gramáticas, y gramáticas con porciones completas de código (en forma de símbolos terminales) que, junto a un test suite generan un nuevo test que ponen a prueba.

## Random Testing de Caja Gris

A modo de repaso, el **fuzzing de caja negra** parte de una semilla de generación de inputs, se elige cada input y se le aplica una cantidad aleatoria de mutaciones entre 0 y k, según la técnica de mutaciones ya vista.
Entonces, mientras tengamos Budget, tomamos un nuevo input de la semilla, ejecutamos el test y reportamos resultado (notar que solo modificamos inputs y vemos outputs). Una vez ejecutados todos los tests de las semillas, empezamos a tomar inputs aleatorios de las semillas y le aplicamos k mutaciones.

En el test de caja gris, **se accede a información de ejecución del Sistema**, más allá de ver solo su salida.
La idea es **si un input aportó cobertura** al testing (branches cubiertos, líneas o bloques de código, etc), entonces ese input es agregado a un corpus de inputs de la semilla con los que luego querremos aplicar mutaciones.
La probabilidad de elegir un input en el futuro, se llama **energía del input**. A mayor su energía, mayores sus chances de ser elegido para ser ejecutado primero y mutado luego.
Notar la diferencia con el black en el else donde agrega el concepto de aumentar cobertura.

```
Def GreyBoxFuzz(SEED):
  numberOfExecutedTests := 0
  init_len := len(SEED)
  While Budget is not empty:
    If numberOfExecutedTests < init_len
       Input := SEED[numberOfExecutedTests]
       Execute Input (Report if crashes/hangs/etc)
    Else:
       Choose randomly input from SEED
       NewInput := mutate(input, K)   Se toma un input aleatorio y se calcula mutacion
       Execute NewInput (Report if crashes/hangs/etc)
       If NewInput adds new Coverage:   Si el input aporta cobertura, pasa al conjunto
         SEED += NewInput               de seeds y es posible que en el futuro
       Endif                            se le apliquen nuevas mutaciones.
    Endif
   numberOfExecutedTests+=1
  EndWhile
```

Al agregar nuevos inputs a la semilla mutados de inputs originales, la idea es que luego puedan volverse a elegirse y generar nuevas mutaciones que sigan aportando valor o encontrando problemas en el código.
Se hace evidente ver que cuantos más inputs agreguemos a la semilla, menor será la probabilidad de cada input de volver a ser elegido.
La versión mejorada con el uso de la energía de cada input, no elige aleatoriamente los inputs de la semilla para aplicar mutaciones, sino que elige tomando en cuenta la cantidad de energía de cada input sobre la energía total, dándole más probabilidad a aquellos inputs con mayor energía (Boosted Greybox Fuzzing). Los inputs con mayor energía serán los que hayan descubierto nuevos caminos en la corrida del test, más allá de si aportó o no cobertura al test suite.

## Generación Automática de Tests

Las pruebas automáticas no escalan bien para proyectos muy grandes. Por eso, esta técnica es más efectiva cuando se enfoca en unidades, partes más pequeñas de una aplicación (unit testing). Además, se basa en resultados de tests anteriores para guiar la creación de nuevos tests, entonces no solo se encuentran errores eficientemente, sino que ayuda a crear un test suite más robusto.


### Test Sistemático (Korat)

Existe una herramienta llamada Korat que es un generador determinístico de tests, y es eficiente para linked data structures, como listas, arboles, grafos, etc.
La idea es aprovechar las pre y post condiciones para generar tests automáticamente.
El problema radica en tomar un subconjunto finito de los infinitos tests posibles. Más aún, debe ser conciso (sin tests ilegales ni redundantes) y diverso (dar una buena cobertura de código para alguna medida).

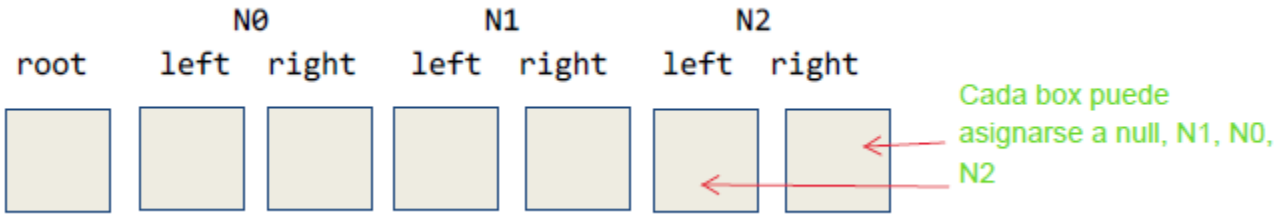
## Hipotesis del Input Pequeño

Al igual que cuando asumimos en Mutation Analysis que el programador es competente, y que asumimos en Fuzzy para programas concurrentes que los bugs tienen profundidad baja, acá asumiremos que, **si hay un test que hace que el programa falle, entonces hay un test que lo hace con un input de tamaño pequeño**. Por ejemplo, si una lista tiene un bug, es probable encontrarlo con tamaños de lista pequeños (-1,0,1,2) y poco probable que el bug se deba al tamaño grande de una lista (salvo que sea un problema de memoria, pero ahí el bug no es de la lista).
Es por esto, que Korat va a generar inputs aleatorios, pero de un tamaño pequeño k, lo que reduce drásticamente el mundo de los infinitos inputs posibles.

## Generacion de Tests

**Korat es una herramienta de caja blanca**, es decir que va a utilizar **los tipos** de los datos para generar tests. Entonces, basado en el tipo de datos, enumera los posibles inputs de tamaño pequeño con que se puede alimentar ese dato.

```
class BinaryTree {
  Node root;
  class Node {
    Node left;
    Node right;
  }
}
```




Visto este ejemplo de árboles de hasta 3 nodos, podemos enumerar una forma simple para el algoritmo de generación de tests:

- Se selecciona previamente mediante una configuración, el tamaño máximo de input (en el ejemplo, 3)
- Se generan todos los inputs posibles
- Mediante el uso de la precondición, se descartan aquellos que no la cumplan (en el ejemplo, arboles con nodos pero sin raíz)
- Se corren los tests para los sobrevivientes.

**De todas formas, con un k relativamente chico, la cantidad de inputs posibles puede crecer dramáticamente.** También es esperable que una gran cantidad de esos inputs posibles sean inválidos y descartados por la pre, así como también generar inputs distintos sintácticamente pero equivalentes.

**Cualquier técnica automatizada de tests de calidad, deberá lidiar eficientemente con este desafío.**

## Generación de tests más eficiente

Guiándonos en lo que acabamos de problematizar, la idea es, en vez de generar inputs y chequear contra la precondición, usar la precondición como guía para generar los inputs.
Para llevar eso a cabo, Korat agrega código a las precondiciones, esencialmente para monitorear como actúan y sobre que campos. Por ejemplo, si una precondición no accede a un campo en particular, entonces es porque no depende de él y serán inputs equivalentes.
Una vez que conoce las precondiciones, inspecciona en qué orden se ejecuta ese chequeo de precondicion y más aún, para un input dado, verifica cual fue el último campo que accedió e intentará "expandirlo" para generar un nuevo input. Cuando no haya más que expandir, hace backtracking. Nunca expande nada que no esté en la pre.

### Desventajas de Korat

Ya vimos que es altamente eficiente en generar tests donde se pueden enumerar los inputs, es decir estructuras enlazadas como listas, arboles, etc. Si la enumeración es débil y difícil de enumerar, como el conjunto de todos los enteros, entonces es menos efectivo ya que un input de tamaño 1, de por sí tiene infinitos candidatos para el caso de los números (o todo el código ASCII si es string), y no puede generar un mecanismo para detectar isomorfismos o inputs equivalentes.

**Otra clara debilidad es que es tan bueno como buenas sean la pre y post condiciones.**
**DSE también funciona bien para este tipo de estructuras y es guiado por el árbol de computo.**

## Feedback-directed Random testing (Randoop – Random Tester For OOP)

Es una herramienta complementaria a Korat que puede generar mejor una secuencia de inputs cuando tenemos estructuras de datos más complejas. **Genera tests unitarios orientados a objetos**, que consisten en una secuencia de llamadas a métodos para configurar un estado particular y luego, hacer una aserción sobre el estado final.
**Es utilizado para testear librerías mediante la interfaz que la misma librería provee en sistemas orientados a objetos.**
La idea es que se creen aleatoriamente nuevos tests guiados por el feedback de los tests creados previamente.
Podemos enumerar una forma simple sobre cómo trabaja Randoop:

- Genera una nueva secuencia de métodos (un nuevo test), extendiendo secuencias pasadas.
- Apenas la crea, la ejecuta y evalúa el resultado.
- Usa ese resultado para guiar los tests que vienen y determinar si la secuencia recién generada era ilegal o redundante (y en ese caso la descarta), o no, y la utilizará para seguir extendiéndola.

Como input previo, Randoop toma las clases objetivo a testear, un tiempo límite (Budget) para seguir generando tests y un conjunto de contratos (generalmente, están en la documentación de la misma librería a testear) que se desean satisfacer (o chequear si se violan).

```
e.g. "o.hashCode() throws no exception"
e.g. "o.equals(o) == true"
```

Como output, randoop dará los tests cases que violaron algún contrato (el contrato será el assert final).

### Algoritmo

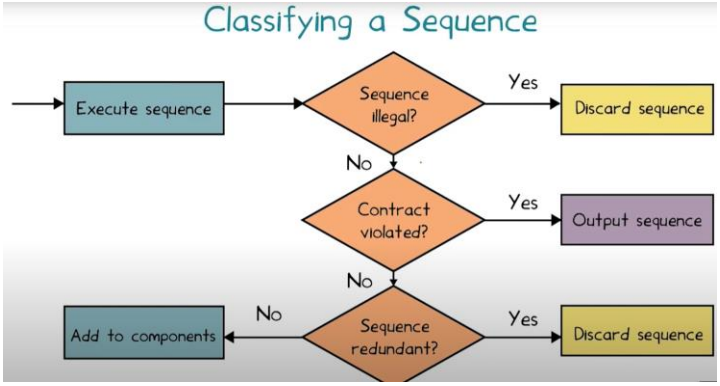
Para empezar, mantiene una colección de componentes singleton semilla, por ejemplo, un entero, un booleano, una fecha, etc. Si luego en la llamada a un método necesita un booleano, utiliza el que tiene en su colección. La lista de componentes se va agrandando y complejizando con el tiempo (puede tener listas enlazadas, arboles, etc)

```
components = { int i = 0; boolean b = false; . . . }      // seed components
```

Luego básicamente repite mientras le quede Budget:

- Crea una nueva secuencia
  - Aleatoriamente elige un método a llamar "m" de las clases objetivo a testear.
  - Si el método "m" elegido tiene argumentos, antes de realizar la llamada debe poder crear esos argumentos o tomarlos del conjunto de componentes semilla, así que lo hace y escribe el código arriba del método "m".
  - Ordena la secuencia creando esta nueva donde puede llamar al método m y guardar el resultado.
- Clasifica la nueva secuencia:
  - O bien la descarta por ilegal (falla en la ejecución previa a "m", o no cumple la pre) o redundante (un estado que ya se había capturado antes).
  - La muestra como output pues violo algún contrato
  - La agrega a los componentes existentes. Este último paso es el que le permite construir secuencias cada vez más complejas.

Las secuencias son clasificadas de acuerdo a este diagrama de secuencia:




## Ejecución Simbólica Dinámica (DSE)

Es una técnica híbrida de análisis estático y dinámico, o sea que mantendrá en paralelo tanto una ejecución estática como una dinámica. Es de caja blanca. **Busca maximizar la cobertura de caminos del programa**. Tiene falsos negativos (puede no encontrar todos los bugs), pero no falsos positivos. No está limitada a ningún lenguaje de programación.
La aproximación que toma es, para un programa, guardar tanto el estado **concreto** como el **simbólico.** Además, utiliza **constraint solvers** para poder guiar la ejecución en branchs y desviar caminos, lo que le permite explorar todos los caminos a testear.
En este tipo de ejecuciones ya no consideramos control flow graphs, sino que tomamos un árbol binario de computo donde cada nodo es un branch del programa.
Cada camino de este árbol representa una clase de equivalencia de inputs, es decir que si dos inputs distintos realizan el mismo camino, son equivalentes. El objetivo de la técnica entonces, será generar inputs no equivalentes.

```
void test_me(int x) {
    if (x == 94389) {
        ERROR;
    }
}
```

Probability of ERROR:

$1/2^{32} \approx 0.000000023\%$

Si miramos ese ejemplo, podemos entender que las técnicas vistas hasta ahora no serán eficientes para encontrar el error (random testing, ni korat, ni randoop, ni graybox fuzz). De alguna manera, queremos poder detectar ese branch y si, en una ejecución aleatoria salió por false, entonces en la ejecución siguiente, querremos forzar a true.
Ahora, si tomáramos un enfoque puramente estático deberíamos explorar cada condición del programa con un Theorem Prover para ver si la condición puede ser satisfecha, al igual que su negación, y determinar si hay branches que pueden llegar a no tomarse. Para programas muy chicos puede funcionar, pero en la práctica ya vimos que los arboles de computo pueden ser infinitamente grandes, por lo que se vuelve impracticable.
**Los theorem provers no son infinitamente poderosos y si no logra resolver la ecuación determina que ambas ramas son alcanzables, pudiendo generar un falso positivo (En ejecución simbólica pura, no dinámica).**

## Algoritmo de DSE

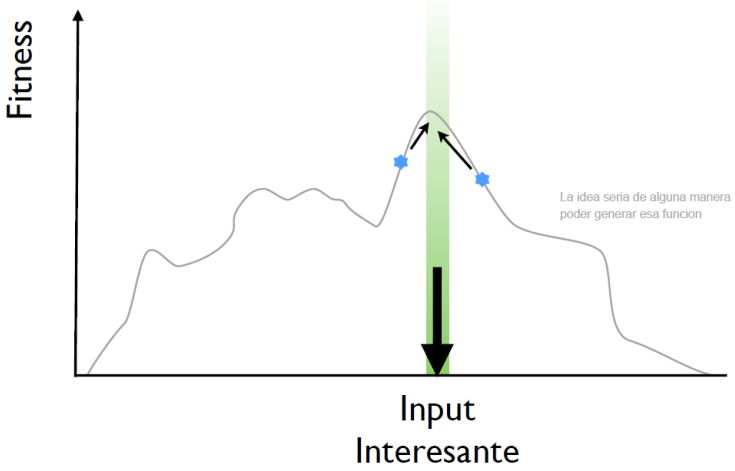
Funciona generando un input random y se observan las ramas que se toman (dinámico). También se hace un seguimiento simbólico (estático) del programa a la vez. Luego, hace **backtracking** hasta la última bifurcación tomada y decide si pueden haber valores distintos que haga que se tome la otra bifurcación. De ser así, se hace esa asignación y se vuelve a correr el programa con los nuevos valores que harán tomar el otro camino.
A medida que el programa corre, todas las ramas tomadas se van almacenando, y cuando se retrocede hacia una para intentar satisfacerla, el theorem prover buscará satisfacer todas las ecuaciones almacenadas hasta ese momento, para asegurarse que efectivamente se pueda tomar el camino deseado. (Ver ejemplo en video, muy claro). **Notar que para cambiar el camino de la última rama, niega la condición del ultimo path y concatena junto con el resto de las condiciones originalmente tomadas hasta llegar a ese punto.**

Cuando las restricciones son muy complejas para ser manejadas por el demostrador de teoremas, DSE "simplifica" las restricciones enviadas al demostrador reemplazando un valor simbólico con un valor concreto.
**Esta técnica se llama Incomplete Theorem Prover: No marcará como satisfactoria ninguna condición insatisfactoria, aunque sí podría decir que es insatisfactoria una condición que no lo es (falso negativo).**

**En resumen, DSE tiene terminación garantizada solo si el árbol de computo no es infinito, es complete (ya que, si alcanza un error, exhibe los valores concretos que lo llevaron a ese error), pero no es sound, ya que, que no haya encontrado errores no significa que no los haya.**

## Search Based Testing

La idea es **transformar problemas de Ingeniería de Software en problemas de optimización combinatoria**. Se genera un gran espacio de búsqueda (por ejemplo, todos los tests posibles para un programa que recibe dos enteros), y se aplican algoritmos heurísticos para poder resolver el problema. No siempre se encuentra la mejor solución, pero empíricamente se obtienen buenos resultados en tiempos razonables. Es decir, **sacrifican completitud, pero ganan eficiencia.**
Para lograr esto **es necesario definir alguna medida que nos pueda indicar donde se encuentran los inputs interesantes** para nuestro programa.
**Evo suite** utiliza técnicas genéticas para generar tests.



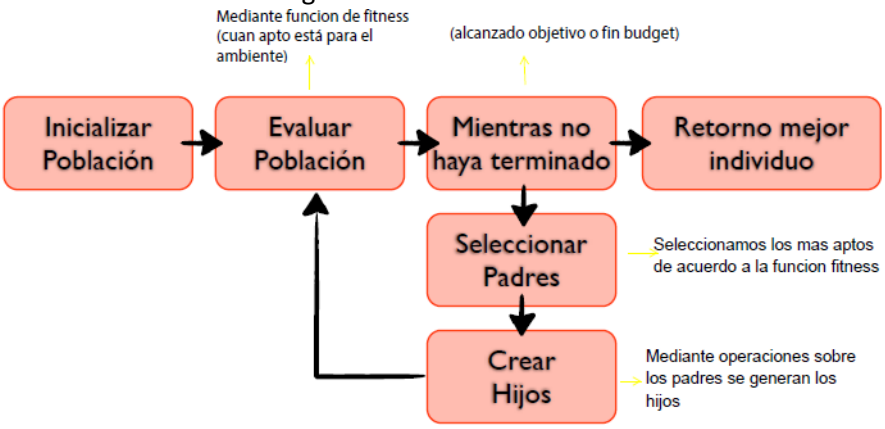

Notar que ofrece una alternativa tanto para random testing, como para DSE al **no necesitar del poder de un theorem prover** para guiar los tests.
La idea es tener una función de *fitness* tal que, para cada input nos dé un valor, y la función tome un valor máximo en inputs interesantes.

## Idea para Fitness

Primero, debemos **definir** una noción de cercanía del input que queremos alcanzar para un determinado objetivo.
Luego, por cada solución analizada, explorar el vecindario (debemos **definir** quiénes son los vecinos también) de otras soluciones y tomar la mejor, es decir la que esté aún más cerca de las de mi alrededor, y repetir esto hasta no poder mejorar más. **Esto se llama Hill Climbing.** Su implementación básica tiene el problema de máximos locales. Como alternativa, se usa Taboo Search. Otro problema, cuando el vecindario es muy grande (por ejemplo, tenemos un conjunto de 10 inputs reales, el vecindario está en $R^{10}$ ) Hill Climbing se torna muy lento en decidir y avanzar.
**La función de fitness es específica a cada problema.**

## Algoritmos Evolutivos

Pasar la información de una generación a otra.




Para la inicialización de la población, se puede hacer aleotariamente, es decir, algo parecido a random testing. También se puede utilizar un test suite con un test de cobertura de menor potencia, o incluso se lo puede hacer manualmente.
Luego, necesitamos una función de fitness para evaluarla, o sea, saber cuán bueno es un individuo como solución. Esta función siempre es específica a cada problema.
**¿Cómo medimos cuán cerca está un input de ser verdadero o falso?**

## Branch Distance

El objetivo es un branch a cubrir, entonces, para cada posible input va a medir la distancia a poder cubrir ese predicado.

| | |
|---|---|
| a == b | abs(a-b) = 0 ? 0 : abs(a-b) |
| Funcion de branch distance para distintos branches → | |
| a < b | a < b ? 0 : (a - b)+K |
| a <= b | a <= b ? 0 : (a - b) |
| a > b | a > b ? 0 : (b - a)+K |
| a >= b | a >=b ? 0 : (b - a) |

Si tomamos el primer caso, vemos que un input a=1, b=2 está más cerca que un input a=1, b=10.
Para las inecuaciones, se toma un valor K para evitar que la igualdad (a==b) sea tomada como objetivo alcanzado.

| | |
|---|---|
| a != b | a != b ? 0 else K |
| A && B | distance(A) + distance(B) |
| A \|\| B | min(distance(A), distance(B)) |
| !a | Aplicar De Morgan |

En estos casos se aplica distancia recursivamente, tal que para el and, se habrá alcanzado el objetivo si ambos predicados lo alcanzaron, es decir si ambas distancias son 0.
Notar que falta el caso de llamada a una función booleana que evalúe. En esta primera versión es muy ingenuo ya que no aporta insights para mejorar, pero más abajo en este resumen debería estar la respuesta.

## Limitaciones

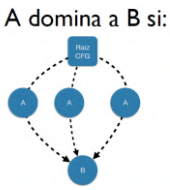
Así como está planteado, si tenemos branches anidados que agregan precondiciones sobre el branch final a cubrir, no se van a estar contemplando.

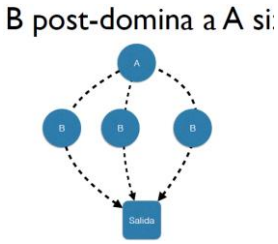
```
def testMe(x, y, z):
    if x==10:
      if y==20:
        if x * z == 2 * (y + 1):
          return True //branch a cubrir

    return False
```

## Dominadores

- El Nodo A **domina** al Nodo B si todo camino hacia B debe pasar por A
- Dominador Inmediato: El dominador mas cercano en todo camino desde la raíz
- La raíz no tiene dominador inmediato
- El Nodo B "post-domina" al Nodo A si todos los caminos desde A a la salida deben pasar por B
- Post dominador Inmediato: El dominador más cercano en todo camino hacia el nodo de salida
- Son Dominadores vistos en reversa (caminos desde el nodo de salida)




**Un nodo se domina a si mismo**. Notar que en en un programa secuencial, todas las sentencias anteriores a una en particular son dominadoras de ésta.

## Control de dependencia

- B es control-dependiente de A si:
  - A domina a B
  - B no "post-domina" a A
  - A tiene al menos dos sucesores
  - B post-domina a un sucesor de A

**A es una decisión, es decir, tiene dos sucesores. B entonces es control dependiente si, todos los caminos para llegar a B pasan por A y, además, hay al menos un camino desde A para salir de la ejecución que no pasa por B.**
Usando esa información se puede armar un control dependency graph. Analizando una ejecución en ese árbol, podemos ver a que distancia estuvimos de ejecutar el branch que queríamos. **Heurísticamente, es preferible un test a menor distancia de un branch a cubrir que otro.**

### Otra idea para Fitness

Podemos definir una función de fitness utilizando la cantidad de nodos dependientes al nodo que tenemos **menos** la cantidad de nodos que ya se ejecutaron.
Eso nos da una medida de control distance que es **el número de ejes entre el objetivo y el camino elegido.**
**Combinando esta aproximación junto con Branch Distance para ver cuan cerca estamos de ejecutar un determinado branch, podemos armarnos nuestra FF.** (Control dependency graph + Branch Distance en Control Flow Graph).
Además, se debe normalizar la Branch Distance para que no termine dominando el valor de FF por sobre el control distance.

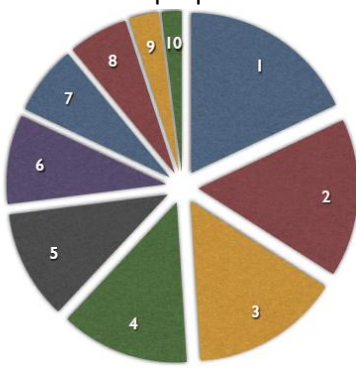
## Seleccionar Padres

Teniendo funciones de fitness definidas, debemos poder seleccionar padres.

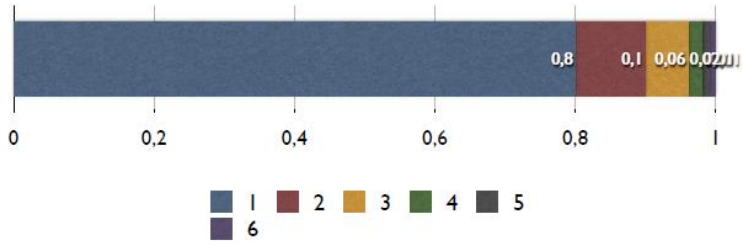
### Ruleta

La probabilidad de elegir un individuo es proporcional al valor de su función de fitness.

| Individual | Fitness |
|---|---|
| 1 | 2 |
| 2 | 1,8 |
| 3 | 1,6 |
| 4 | 1,4 |
| 5 | 1,2 |
| 6 | 1 |
| 7 | 0,8 |
| 8 | 0,6 |
| 9 | 0,4 |
| 10 | 0,2 |
| 11 | 0 |




Para elegir un individuo tomo un valor aleatorio entre 0 y 1, y me fijo a quien le corresponde en el siguiente diagrama.




Se puede observar que, si el individuo domina la selección, entonces prácticamente se pierde el concepto de ruleta.
**¡NO ES UN BUEN CRITERIO!**

### Selección por Ranking

En lugar de asignar un valor proporcional de acuerdo a su fitness, se los rankea. Es decir, se los ordena según su valor de fitness, y se reparte la probabilidad de acuerdo a su valor en el ranking. Entonces, no importa si el fitness entre uno y otro es abismal o muy pequeño, siempre van a estar proporcionalmente distribuidos de acuerdo al ranking.
**En la práctica funciona bien este criterio.**

### Selección por torneo

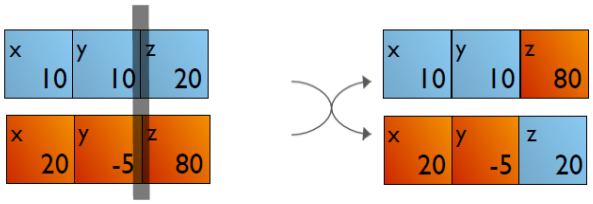
Se tiene un valor N de tamaño del torneo. Se eligen aleatoriamente N individuos, y se selecciona el mejor de esos N.
Notar que no siempre gana el de mayor fitness general ya que depende de la selección aleatoria, por lo que, cuanto más chico el N, menor la presión selectiva y mayor la probabilidad que tiene un individuo “chico” de ser elegido.

## Creación de Hijos

De acuerdo al criterio de selección de individuos, se eligen padres. Ahora, queremos generar nuevos inputs a partir de esos inputs padres.

### Crossover

Consiste en hacer un cruzamiento entre inputs de los padres, para dar inputs hijos. Para eso, se puede elegir un punto de cruzamiento, es decir, qué lugar de los inputs voy a alternar.

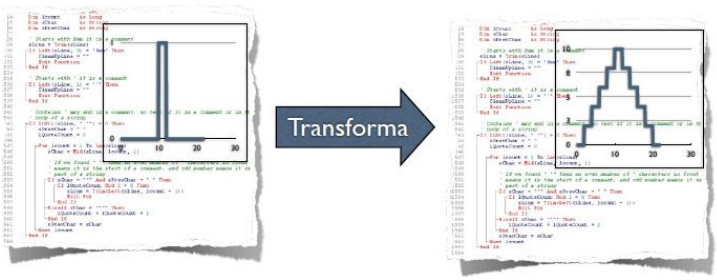



También se pueden tener dos puntos de cruzamiento, o incluso un punto de cruzamiento de tamaño variable (donde se toma un tamaño para un padre y otro para el otro, generando nuevos inputs de distinto tamaño).

**También se pueden hacer mutaciones sobre los padres de la forma que vimos antes.**

## Transformación de Testeo

Muchos programas tienen funciones de fitness arduas o complicadas para guiar generación de tests. Entonces la idea es, transformar el programa en uno equivalente con una mejor función de fitness. La transformación no siempre debe preservar el comportamiento o la firma original. El programa nuevo solo se usa para generar casos de tests. Luego es descartado.




1. Generar tests
2. Descartar programa transformado

**Generando estas transformaciones de acuerdo al caso en el que esté, es posible luego trabajar con las definiciones de Branch Distance vistas anteriormente.**

Para comparación de igualdad entre strings se reemplaza por distancia de Hamming o Levenshtein.

### Flag Level 1

```
boolean flag = (x == 10);

if(flag)

   ...
```

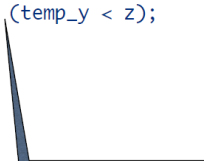
Desde que se ejecuta flag hasta que se evalua, ni flag ni x deben ser modificados. Simplemente reemplazamos flag con su definición. (Flag lvl 1 -> Flag lvl 0)

### Flag Level 2

Entre la definición de la variable flag que se ejecuta y su evaluación se pueden modificar los valores.

Lo que se hace es reemplazar las variables del flag original por variables temporales que tengan los valores en ese punto. Luego, estamos en el nivel 1. (Flag lvl 2 -> Flag lvl 1 -> Flag lvl 0).

```
int temp_x = x; int temp_y = y;
boolean flag = (temp_x >= 10) &&
               (temp_y < z);
// ..
x = 5;
y = x;
// ..
if(flag) {
```



### Flag Level 3

No solo se modifican los valores de la bandera, sino que también la bandera es modificada recursivamente. Se deben hacer todos los reemplazos para intentar llevarla a un nivel 2. No siempre es posible.

### Flag Level 4

La secuencia de flags contiene condicionales

### Flag Level 5

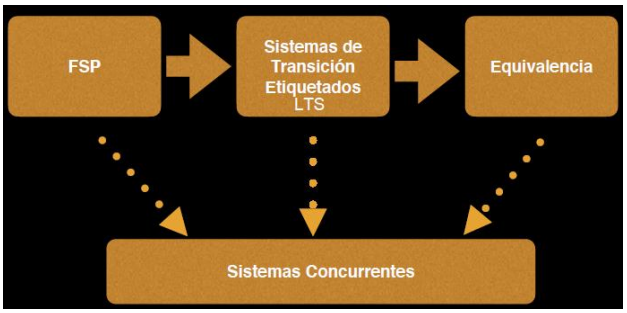
Definición de flag por ejemplo dentro de un loop, y se usa luego de eso.

# Modelado de Sistemas Concurrentes

Hasta ahora, solo habíamos estudiado procesos secuenciales y transformacionales (toman un input, procesan, retornan un valor).
Ahora, el cómputo se realizará en múltiples threads y nos enfocaremos en lo que hacen **durante** la ejecución. Los sistemas concurrentes son eficientes para aprovechar la performance del hardware, mejorar el throughput, tiempo de respuesta, etc. Además, es más apropiado para programan que interactúan con el ambiente y reaccionan a múltiples eventos (ya que la idea es poder recibir estímulos sin bloquearse).
**La idea de buscar métodos automáticos para este tipo de sistemas es porque son más difíciles de construir y además los errores son más difíciles de replicar.**

## Modelos de Concurrencia

Buscamos las primitivas para poder describir programas concurrentes (como el WHILE antes). Uno de esos lenguajes es CSP, que es un algebra de procesos. Otro de ellos es Finite State Process (FSP), que a diferencia de CSP trata de que toda expresión de como resultado un modelo con un número finito de estados, lo que hace más fácil de analizar los comportamientos de los programas.




## LTS

Son autómatas que tienen estados, un conjunto de acciones como etiquetas sobre transiciones entre estados, una relación para esos estados y esas transiciones y un estado inicial.
El alfabeto de un LTS es todo el conjunto de **acciones** de A excepto tau, que representa todas las formas en las que un LTS puede interactuar con su entorno.
La cantidad de estados es finita.
El grafico de un LTS no necesariamente representa todo el LTS. Por ejemplo, podría haber elementos en el alfabeto del mismo, pero no hay transiciones definidas sobre ese elemento. Es decir que el sistema tiene la **capacidad** de tomar esa acción, pero no lo hace.

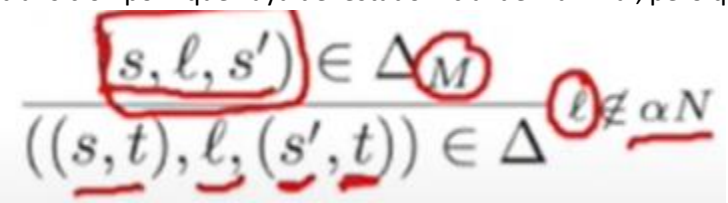
### Ejecuciones y Trazas

La ejecución de un LTS es una secuencia que representa cómo evoluciona el estado del LTS y que acciones va sucediendo.
**Las trazas son solo secuencias de etiquetas**, sin contemplar el estado o los que pasaron, sino que solo importan las acciones que sucedieron y cuales pueden suceder en el futuro a partir de eso.

## Composición en Paralelo

(Nos salteamos toda la parte de introducción a FSP y MTSA que son meros conceptos prácticos).
P y Q son procesos, entonces (P||Q) es la ejecución concurrente de ambos.
Es un nuevo LTS (es decir que nos describe como un proceso secuencial, la composición conjunta de otros dos procesos) que tiene como espacio de estados el prod cartesiano de los procesos originales. El alfabeto de la composición es la unión de ambos, el estado incial es el estado $P_0xQ_0$ y las relaciones de transición se dan asi:

- Dada una acción *l,* si esa acción pertenece a P, pero no a Q, entonces en la composición paralela habrá una transición por *l* que vaya del estado incial de P al final, pero que deje fijo al de Q.



Aquí está con M y N en vez de P y Q

- Análogo para Q y no P.
- Si la transición se encontraba en ambos procesos, entonces estamos ante un evento de sincronización. La composición **solo puede avanzar si ambos pueden avanzar por esa etiqueta**. En otras palabras, si Q pudiera avanzar por l en el estado actual que está, y P pudiera avanzar también por l, pero no en el estado actual en que está, entonces l no estará habilitado **hasta que P alcance el estado de sincronización con Q.**

La composición paralela también funciona para múltiples componentes (ya que es asociativa).

### Tau

Admitimos que el alfabeto de LTS pueda tener un elemento Tau, pero no pertenece a su alfabeto de comunicación, es decir, que no forma parte de la interfaz publica del LTS. Se usa para representar computo interno y podría estar encapsulando más de una secuencia de acciones.

**Se logra una acción Tau solo mediante ocultamiento de acciones (\ o @).**

En composición paralela, tau no sincroniza. Es decir, que siempre ocurre independientemente en cada LTS de la composición y, más aun, siempre va a poder transicionar por tau (justamente porque no sincroniza).

## Bisimulación

El objetivo es poder hablar sobre la semántica LTS. Dos términos FSP son semánticamente equivalentes si y solo si sus LTS lo son.

¿Podemos plantear un ismorfismo entre LTSs? Es decir, un mapeo de estados entre dos lts sintácticamente distintos para obtener dos lts sintácticamente iguales.

**Necesitamos relaciones reflexivas, simétricas y transitivas.** Abstractas con respecto a la estructura, que sea más detallista que las trazas (para distinguir determinismo de no determinismo), que se adapte al lenguaje de especificación.

Si P y Q son expresiones FSP tal que P≡Q, entonces queremos que toda expresión construida a partir de P sea equivalente a otra construida a partir de Q.

### Bisimulación Fuerte

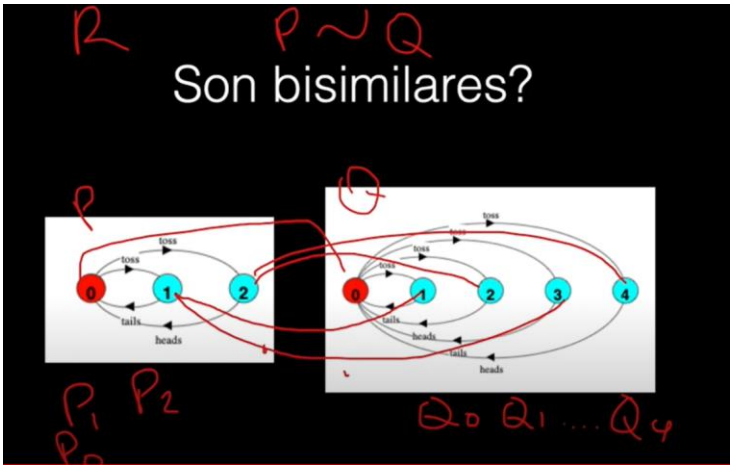
Es una relación binaria R entre dos LTSs.

Dos LTS P y Q están relacionados mediante una bisimulación fuerte (P,Q) en R si para cada acción o evento *a*:

- $(P \xrightarrow{a} P') \Rightarrow (\exists Q' \cdot Q \xrightarrow{a} Q' \ \wedge \ (P', Q') \in R)$
- $(Q \xrightarrow{a} Q') \Rightarrow (\exists P' \cdot P \xrightarrow{a} P' \ \wedge \ (P', Q') \in R)$

"Si P puede hacer *a*, Q tiene que poder imitar a P transitando por *a* y seguir siendo una bisimulación fuerte".

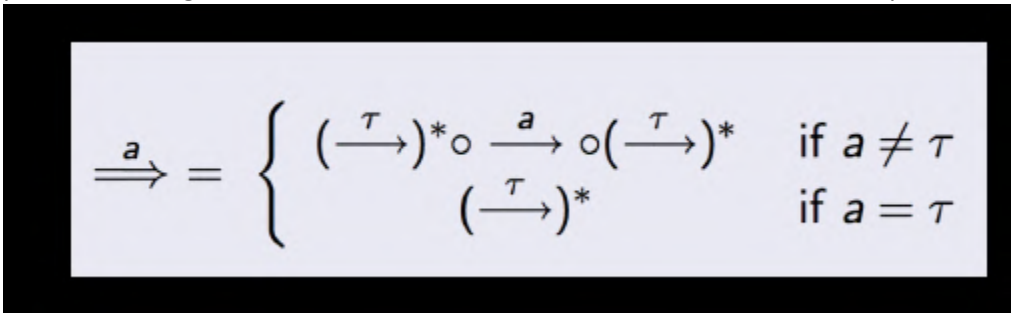
**Notar que la bisimulación es una relación entre LTSs, y que la función transitar (->) lo que hace es pasar de un LTS en un estado inicial a otro en otro estado inicial. Sabemos que es el mismo LTS en el fondo que solo transicionó, pero formalmente lo tomamos como un LTS distinto. Por eso parte de los lts originales perteneciendo a R y luego transicionando va agregando todos los lts generados también a R.**




Algorítmicamente, se usa la técnica de máximo punto fijo. Arranca con todos los pares posibles que pertenecen a una relación de bisimulación (~0) y a partir de ahí construye ~1 eliminando pares y asi hasta que se alcanza punto fijo.
P ~n+1 Q si, si P puede transicionar por *a* a P' entonces Q puede transicionar por *a* a Q' y P' ~n Q'. Análogamente empezando desde Q. ~n representa todos los pares que pueden simularse en n pasos.
Para simplificar, ~0 es el prod cartesiano de los LTSs.

## Bisimulación Débil

Consiste en realizar una clausura transitiva por tau mediante una nueva función de transitar. Es decir que P puede transitar por a hasta P' no solo si está en su conjunto de transiciones, sino también si tiene 0 o más taus por delante y por detrás. Luego, la definición de relación de bisimilitud es idéntica, solo que con este nuevo concepto de transitar.




Entonces ahora, mientras un lts **avanza de a un paso "real"** de transición, el otro tiene permitido avanzar por las clausuras transitivas.

**Definicion 3** (Weak Bisimulation) *Sea $\mathcal{P}$ el universo de todos los LTS. Una relacion binaria $R \subseteq \mathcal{P} \times \mathcal{P}$ es una bisimulacion debil si y solo si cuando $(P,Q) \in R$ entonces para cada accion $a \in Act \cup \{\tau\}$:*

- $(P \stackrel{a}{\longrightarrow} P') \Rightarrow (\exists Q' \cdot Q \stackrel{a}{\Longrightarrow} Q' \wedge (P',Q') \in R)$

Notar que aca es el transitar original

- $(Q \stackrel{a}{\longrightarrow} Q') \Rightarrow (\exists P' \cdot P \stackrel{a}{\Longrightarrow} P' \wedge (P',Q') \in R)$

# Análisis de Sistemas Concurrentes

Una vez que tenemos un modelo construido, es importante poder **validarlo**. Siempre la validación es contra un modelo mental creado por un humano, con la intención de elevar la confianza sobre el modelo creado.
Una estrategia de validación es la de manipular el modelo construido de manera tal que genera varias "vistas" distintas y que, esas vistas, sean inspeccionadas por humanos para poder encontrar hipótesis o sacar conclusiones.
Por ejemplo, dado un FSP, tenemos vistas como los LTS, los LTS minimizados, generar animaciones, etc.

Por otro lado, la **verificación,** es un proceso distinto. La idea es **garantizar matemáticamente** que dos modelos formales están vinculados entre sí mediante alguna relación formal. Dos modelos formales como el de una especificación y un sistema en sí. La validación puede ser manual, semi automática o automática.
Las estrategias más conocidas son:

- Bisimulación y especificación abstracta mediante ocultamiento.
- Chequear que sea libre de deadlocks.
- Libre de estados de error
- Dar observadores para garantizar comportamiento
- Verificar que se cumplen propiedades deseadas (LTL).

Alcanzabililidad

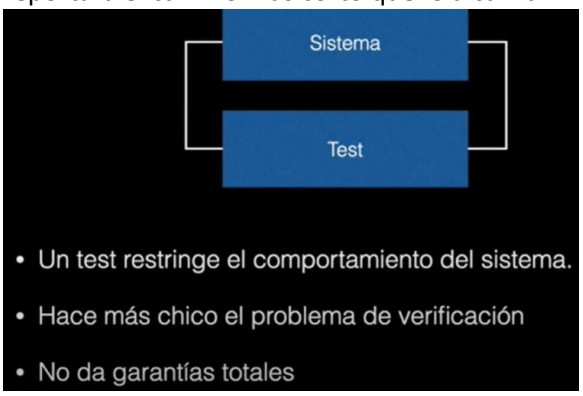
Satisfacibilidad

## Deadlock

Un estado de deadlock es aquel que no tiene transiciones salientes y que no es el fin del programa. Una estrategia para detectarlos es construir el LTS, hacer un breadth first search por los estados hasta encontrar el estado de deadlock. Usa esta técnica ya que, al encontrar un estado sin transiciones salientes, queda registrado el camino más corto hasta ese estado y sirve como feedback. La complejidad es lineal respecto al tamaño del LTS, lo cual es mucho ya que la composición paralela de LTS genera una explosión de estados (exponencial respecto al número de componentes sobre el número de estados).

## Estados de Error

Se extiende la definición de LTS para agregar un estado distinguido de error. Si un estado de error se alcanza, se produce un deadlock. **Si cualquier componente alcanza un estado de error, entonces en la composición paralela también se alcanza el de error.** Es decir que el estado de error general es la unión de todos los estados de error de todos los componentes.
Esto se puede aprovechar para escribir un nuevo proceso que interactúe sincronizando solo con la parte más externa de nuestra aplicación para indicar cuál es el comportamiento esperado, y cuál es el que llevaría a algún estado de error.
Algorítmicamente se procede igual que deadlock buscando el estado distinguido de error. Si es alcanzable, entonces reportará el camino más corto que lo alcanza.




La desventaja del test es que restringe el comportamiento del sistema, haciendo más chico el problema de verificación, y además bloquea el sistema indicando que, para el caso buscado todo anduvo bien.

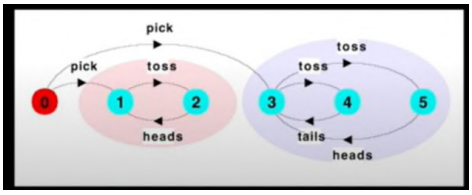
## Observadores

Cambiamos el enfoque y ahora queremos usar observadores que describan el comportamiento esperado del sistema de manera global. De tal forma que, mientras la propiedad se cumpla el sistema no se bloquee y que solo vaya a error cuando no se cumpla.
**Si la propiedad se cumple, entonces (S||Obs) ~ S.**

La idea entonces es definir el comportamiento aceptable. Entonces, desde cualquier estado saldrán todas las acciones posibles: Aquellas que pertenecen al comportamiento aceptable siguen el flujo del programa, las que no están en el comportamiento aceptable automáticamente van a un estado de error.

## Progreso

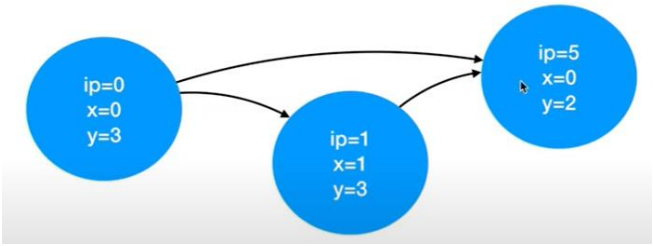
Definimos progreso como un conjunto de acciones **útiles** donde por lo menos una de esas acciones aparece infinitas veces, asumiendo fairness. Es decir que las cosas que quiero que pasen, pasan infinitas veces en una ejecución infinita, el sistema siempre progresa.
La definición de esas acciones depende del dominio del sistema.
**Una ejecución de LTS es válida bajo Fair Choice si, cuando un estado es visitado infinitas veces, todas sus transiciones salientes también son visitadas infinitas veces.**
Algorítmicamente, la idea es detectar componentes fuertemente conexas del LTS terminales (de las cuales no se puede salir). Luego se mira que cada conjunto cumpla con las condiciones de progreso establecidas. Si alguno no la cumple, entonces no se cumple Progress, y con breadth first search reproduce los pasos para alcanzar los estados de esa componente.




Si tenemos un sistema S que satisface progreso mediante una propiedad L, **no necesariamente** S||T sigue satisfaciendo L. Si por ejemplo componemos el problema de la moneda con un FSP que hace STOP pero que tiene el estado heads, entonces, nunca se alcanzará heads (directamente el programa se congelará y dejará de haber progreso en absoluto).

## LTL

Es una lógica modal que sirve para escribir propiedades que queremos que se cumplan sobre nuestros modelos concurrentes. **Es temporal lineal ya que el tiempo es uno solo y es discreto (a diferencia de CTL, que el tiempo se ramifica).** Agrega los operadores de box y diamante a la lógica proposicional conocida. **Se interpretan sobre estructuras de kripke.** La semántica está dada por una relación de satisfacción (|=) entre formulas.
Podemos pensar las estructuras de kripke como estados de un programa concurrente, y sus transiciones, sus cambios de estado.




**Model Checking:** Si queremos verificar que se cumple una propiedad P en un programa y tenemos la representación en kripke de ese programa y P escrita en LTL, básicamente querremos que w0 |= P, donde w0 es el estado inicial del programa. Para nosotros, los programas son los concurrentes que se representan con FSP y LTS.

## LTL para LTS

Un estado de LTL es una posición en una traza, y las estructuras de kripke entonces se reinterpretan como un conjunto de trazas y cada elemento *i* de la estructura tendrá el elemento *i* de la traza.
Para una traza dada T, decimos que satisface una propiedad P, si T[1] |= P, donde T[1] es el primer elemento de la traza.
**Luego, un modelo M satisface P, si para toda traza T del modelo, T|=P.**

## Verificación de Problemas de Programas Concurrentes

Dado un LTS M, programa concurrente, vamos a escribir fórmulas LTL que queremos que cumpla M.
Vamos a definir un algoritmo que devuelva verdadero si **todas** las trazas de M satisfacen P.
El algoritmo consiste en **convertir a un autómata la negación de P**. Luego generar todas las trazas de ese autómata y comparar contra las trazas de M.
Si ambos conjuntos de trazas son disjuntos, quiere decir que en M ninguna traza es traza de no P, o sea que ninguna traza satisface no P, o sea que todas satisfacen P, o sea M satisface P.
Si hay alguna traza en la intersección, ese es un contraejemplo que podemos mostrar que hace que M no cumpla P.

# Algoritmo (+Detalle)

- Entrada: LTL P, LTS M

1. Convertir la fórmula LTL $\neg P$ a un autómata de Büchi $A_{\neg P}$ que caracteriza todas las trazas que satisfacen $\neg P$

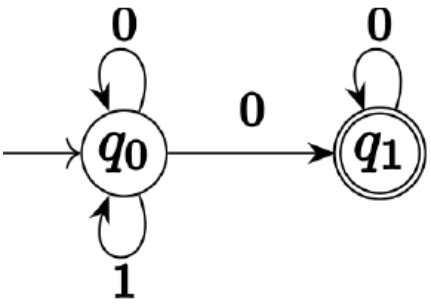
2. Convertir el LTS M a un autómata de Büchi $A_M$ que caracteriza todas las trazas que contiene M

3. Hacer el producto de los autómatas de Büchi $A_{\neg P}$ y $A_M$

4. Si $L(A_{\neg P} \times A_M) \neq \emptyset$, entonces existe una traza de M que no cumple (i.e. la propiedad P no se cumple en el sistema M)

## Autómatas de Buchi

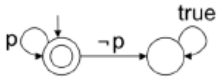
Como las propiedades LTL tienen la capacidad de versar sobre trazas infinitas, entonces necesitamos un autómata que pueda reconocer cadenas infinitas de su lenguaje, de ahí sale la motivación para usar este autómata.
Tiene estados finitos, pero reconocen lenguajes de palabras infinitas, es decir, lenguajes ω-regulares. **Este autómata acepta una cadena cuando visita un estado de aceptación infinitas veces**. Es decir, cuando inf(ejecución) intersección los estados finales es no vacía. El autómata puede ser no determinístico.



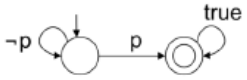

- ¿Qué cadenas infinitas acepta este autómata?
  - ¿Acepta 0...? Sí
  - ¿Acepta 0101..? No
- En general, acepta el lenguaje $(0 \mid 1)^*0^\omega$

# Ejemplos LTL2Buchi

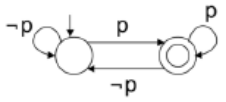
□p




<>p




□(<>p)




El tamaño del autómata crece exponencialmente con respecto al tamaño de la fórmula.

## Autómatas de Buchi generalizados

A diferencia de la versión que veníamos viendo, este autómata tiene requisitos más estrictos para poder aceptar una cadena. Los estados finales de aceptación ahora son un conjunto de conjuntos. **Ahora, se aceptará la cadena si ésta visita una infinita cantidad de veces al menos un estado de cada conjunto de estados de aceptación**. O sea, para cada $F_i$ en F, debe haber un estado que sea visitado infinitas veces. Esto permite generar expresiones más complejas de aceptación y, por ende, lenguajes con más requisitos.

Notar que pasar de autómata común a generalizado es directo, ya que se toma el conjunto F del común y se lo propone como el único conjunto en el conjunto de estados de aceptación del generalizado.

## LTL a Buchi

Se usa una idea parecida a la de Tableau, en la que una formula proposicional se desarma en un árbol top down y se chequean que los caminos resultantes no se cierren para que la misma pueda ser satisfecha.

Entonces la idea es tener un autómata tal que, al avanzar de s a s' por *α,* la fórmula de s' sea como la de s luego de haber procesado *α*. Primero se debe convertir LTL a forma normal positiva donde solo las proposiciones pueden estar negadas.

Luego, usa una serie de conjuntos auxiliares New, Old, Next, Incoming y va consumiendo los elementos de New. A partir de ese autómata, se construye un Buchi generalizado con todos los subconjuntos de proposiciones de la fórmula LTL, y luego se convierte a Buchi simple.

## LTS a Buchi

Es también necesario convertir el LTS con el modelo M que queremos chequear a un autómata de Buchi. Luego, con ambos autómatas construidos, se hace el producto de ambos (M y ¬P). Si no hay ninguna traza que viva en ese producto significa que M satisface P.

- Sea un LTS P=$\langle S, A, \Delta, s_0 \rangle$, podemos construir un autómata de Büchi $A_P = \langle \Sigma, Q, \Delta', Q_0, F \rangle$ donde:
  - $\Sigma$ = Act (conjunto de acciones observables)
  - Q = S Mismos estados
  - $\Delta' = \Delta$ es la relación de transición Misma funcion de transicion
  - $Q_0 = \{s_0\}$ es el único estado inicial Mismo estado inicial
  - $F = S$ son todos estados de aceptación

Notar que todos son estados de aceptación aquí, ya que la aceptación en si misma vendrá del lado de la formula LTL. Hay un lema que dice que el lenguaje de aceptación del producto de dos autómatas es la intersección de ambos lenguajes de aceptación. Sabemos que, del lado de nuestro modelo, tenemos todas las trazas posibles aceptadas, y del lado de ¬P tenemos como lenguaje solo las trazas que acepta ¬P. Si la intersección es no vacía, significa que M no satisface P. Buscar si el lenguaje es vacío o no implica buscar un ciclo en el autómata y luego componentes fuertemente conexas. Un ejemplo de todo.

## CTL

Es una lógica de branching. **La motivación del uso de esta lógica es que tiene la capacidad de distinguir no determinismo, que no tiene LTL, ya que LTL solo mira una traza.** Se consideran entonces, los árboles de cómputo que genera un LTS, estos árboles como ya vimos antes son potencialmente infinitos ya que desenrolla los loops. **Dos LTS son bisimilares también si sus árboles de cómputo son isomorfos.** Cada camino posible en el árbol representa una posible ejecución del sistema. Para la semántica de CTL, ahora debemos poder versar sobre caminos además de estados. Por eso agrega operadores A y E, que hablan de todos los caminos del árbol, o de que exista al menos uno respectivamente.

# Conclusión: LTL y CTL son incomparables