Abstracción para Especificaciones DynAlloy

Raúl A. Fervari

Grupo de Procesamiento de Lenguaje Natural
Facultad de Matemática, Astronomía y Física
Universidad Nacional de Córdoba
fervari@famaf.unc.edu.ar

4 de octubre de 2010

Métodos Formales

Qué son?

Una familia de metodologías de desarrollo basadas en el uso de formalismos lógico-matemáticos, que apuntan a garantizar la corrección del software.

Para qué se utilizan?

- Especificar, diseñar y construir software, de manera rigurosa.
- Verificar que un programa es correcto respecto de su especificación.
- Descubrir errores e inconsistencias en las especificaciones de software.

Métodos Formales

Qué son?

Una familia de metodologías de desarrollo basadas en el uso de formalismos lógico-matemáticos, que apuntan a garantizar la corrección del software.

Para qué se utilizan?

- Especificar, diseñar y construir software, de manera rigurosa.
- Verificar que un programa es correcto respecto de su especificación.
- Descubrir errores e inconsistencias en las especificaciones de software.

Métodos Formales Livianos

Qué son?

Una variedad de metodologías formales que apuntan a la simplicidad de aplicación y a la automatización del proceso de verificación.

Ventajas

- Análisis/verificación con poca intervención humana (en algunos casos automática).
- Intentan alivianar el uso de métodos formales, evitando herramientas "pesadas" (que requieren mucha experiencia, como los demostradores de teoremas).
- En algunos casos las garantías de corrección son inferiores a las de los métodos formales pesados.

Métodos Formales Livianos

Qué son?

Una variedad de metodologías formales que apuntan a la simplicidad de aplicación y a la automatización del proceso de verificación.

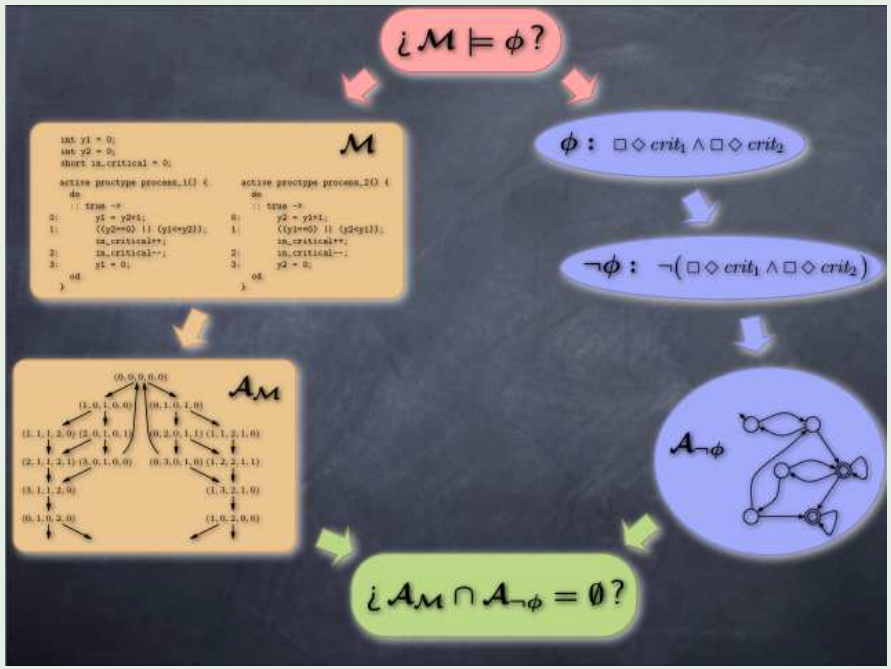
Ventajas

- Análisis/verificación con poca intervención humana (en algunos casos automática).
- Intentan alivianar el uso de métodos formales, evitando herramientas "pesadas" (que requieren mucha experiencia, como los demostradores de teoremas).
- En algunos casos las garantías de corrección son inferiores a las de los métodos formales pesados.

Métodos Formales Livianos - (Ejemplos)

Model Checking

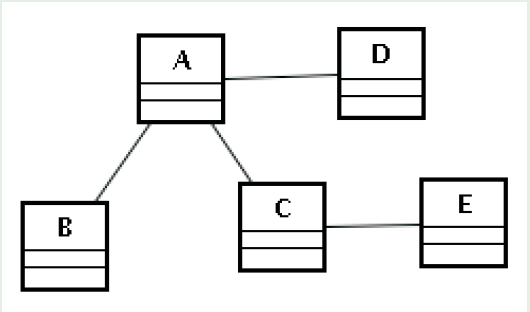
Técnica automática que dado un modelo M de un sistema y una propiedad P, permite verificar la validez de P en M.



Métodos Formales Livianos - (Ejemplos)

Análisis basado en SAT-Solving

Es un mecanismo que consiste en buscar modelos que satisfagan fórmulas proposicionales.
Un SAT-Solver es una herramienta que resuelve este problema.



$$((p \wedge q) \vee \neg r) \wedge (\neg(r \wedge q) \vee \neg(r \wedge p))$$

El Lenguaje Alloy

Características

- Lenguaje de especificación basado en relaciones.
- Sintaxis simple, con semántica intuitiva.
- Es declarativo, por lo tanto las abstracciones que brinda el lenguaje son similares a las que los programadores están acostumbrados.
- Las especificaciones pueden analizarse automáticamente mediante Alloy Analyzer.
  - Alloy Analyzer está basado en SAT-Solving.
  - Es necesario acotar el espacio de búsqueda.
  - La herramienta muestra un contraejemplo en caso de violación.

Sintaxis de Alloy

Componentes

La sintaxis de Alloy es muy simple, convirtiéndose en un lenguaje fácil de aprender a manipular. Sólo cuenta con 5 construcciones:

- *Signaturas:* permite describir dominios de datos.
- *Predicados:* describen operaciones asociadas con las signaturas.
- *Axiomas o Hechos:* describen propiedades que se consideran verdaderas en el modelo.
- *Aserciones:* propiedades cuya validez sobre el modelo se desea chequear.
- *Comandos:* (run,check) instrucciones que el Alloy Analyzer ejecuta.

Modelo de Listas

Signaturas

```
sig Data { }
one sig Null { }
sig Node {
  val: Data,
  next: Node+Null
}
sig List {
  head: Node+Null
}
```

Predicados

```
pred Tail[l,l':List]{
  l.head!=Null and l'.head=(l.head).next
}
```

Modelo de Listas

Signaturas

```
sig Data { }
one sig Null { }
sig Node {
  val: Data,
  next: Node+Null
}
sig List {
  head: Node+Null
}
```

Predicados

```
pred Tail[l,l':List]{
  l.head!=Null and l'.head=(l.head).next
}
```

Modelo de Listas (Cont.)

Axiomas

```
fact AcyclicLists {
  all l: List, n: Node | n in l.head.(*next) => n !in n.(^next)
}
```

Propiedad a verificar

```
assert TailAcyclic{
  all l,l':List,  |
    l.head != Null and noCycles[l.head] and Tail[l,l'] implies
    noCycles[l'.head]
}
```

Comando

```
check TailAcyclic
```

Modelo de Listas (Cont.)

Axiomas

```
fact AcyclicLists {
  all l: List, n: Node | n in l.head.(*next) => n !in n.(^next)
}
```

Propiedad a verificar

```
assert TailAcyclic{
  all l,l':List,  |
    l.head != Null and noCycles[l.head] and Tail[l,l'] implies
    noCycles[l'.head]
}
```

Comando

```
check TailAcyclic
```

Modelo de Listas (Cont.)

Axiomas

```
fact AcyclicLists {
  all l: List, n: Node | n in l.head.(*next) => n !in n.(^next)
}
```

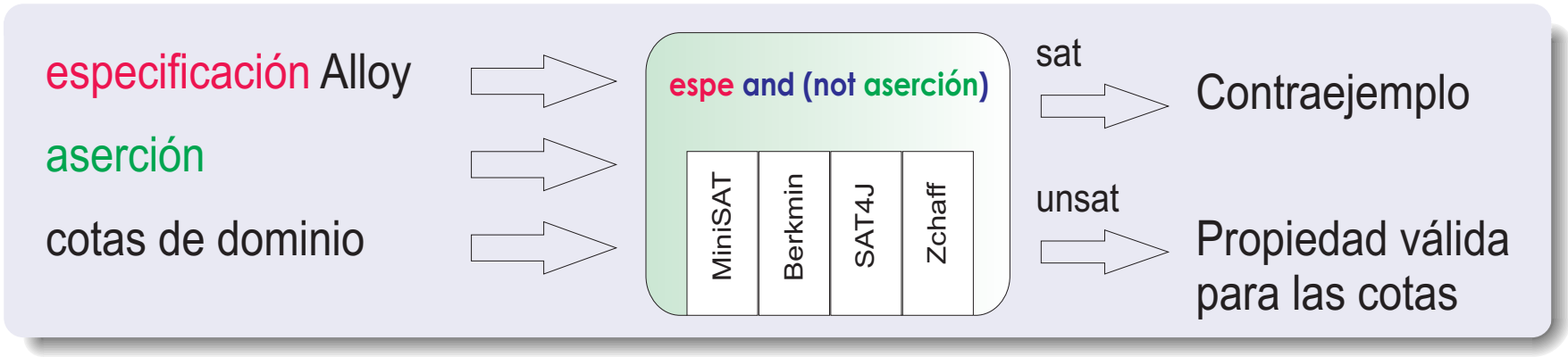
Propiedad a verificar

```
assert TailAcyclic{
  all l,l':List,  |
    l.head != Null and noCycles[l.head] and Tail[l,l'] implies
    noCycles[l'.head]
}
```

Comando

```
check TailAcyclic
```

El Alloy Analyzer



El lenguaje DynAlloy

Extensión del lenguaje de especificación Alloy para modelar propiedades de ejecuciones de sistemas de software de manera más adecuada.

- Permite especificar programas de manera operacional, a través de acciones definidas mediante pre y post-condición.
- Las acciones pueden combinarse para formar programas más complejos utilizando:
  - Composición secuencial (;).
  - Elección no determinística ($+$).
  - Iteración acotada ($*$).

El lenguaje DynAlloy (Cont.)

- Se traduce automáticamente la especificación DynAlloy a Alloy por medio de la herramienta *DynAlloy Translator*, para poder ser analizada por *Alloy Analyzer*.
- Provee un soporte automático para realizar verificación de propiedades de programas, por medio de una búsqueda exhaustiva de contraejemplos (con dominios e iteraciones acotados).

El lenguaje DynAlloy - Ejemplos

Especificación

```
act Head[l: List, d: Data]
   pre = { l.head != Null}
   post = { d' = (l.head).val }
act Tail[l: List]
   pre = { l.head != Null}
   post = { l'.head = (l.head).next }
```

Propiedad a verificar

```
assertCorrectness check[l:List, d:Data] {
   pre = { l.head != Null and noCycles[l.head] }
   program = { Tail[l] ; (Head[l,d] + Tail[l])* }
   post = { noCycles[l'.head] }
}
```

El lenguaje DynAlloy - Ejemplos

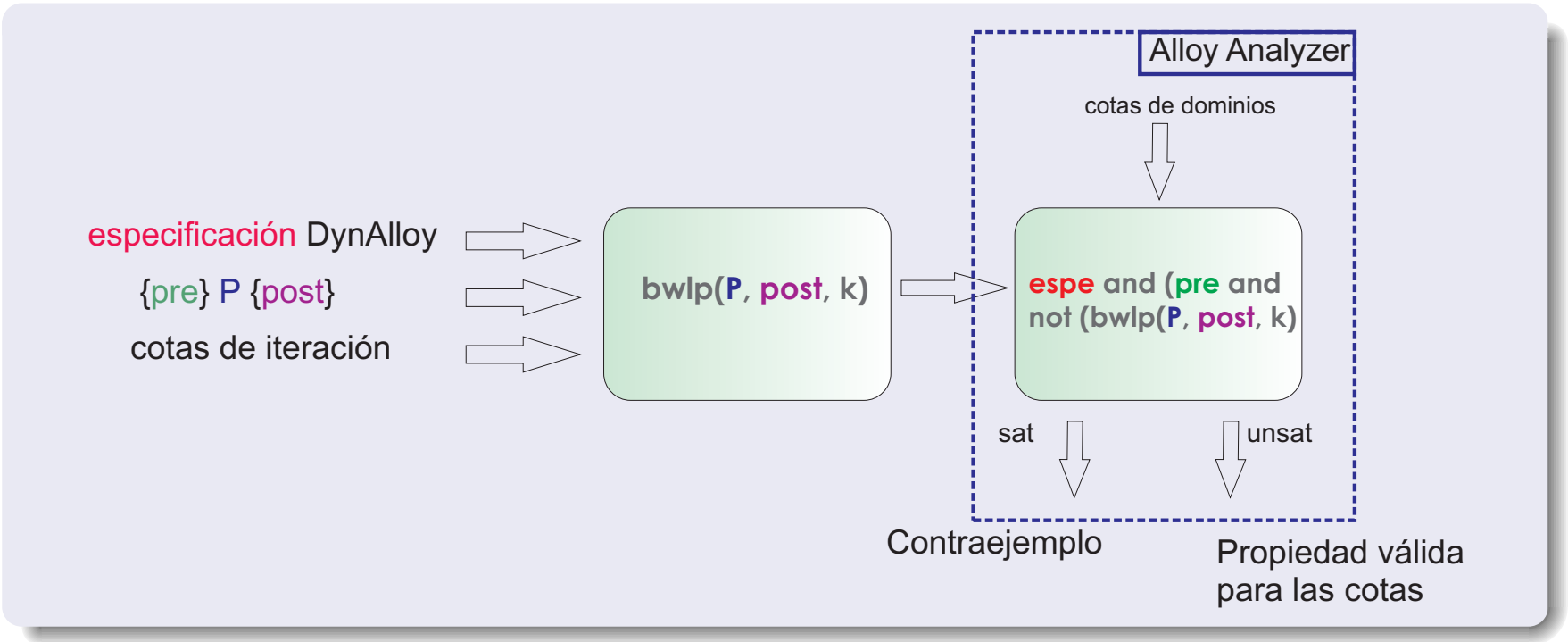
Especificación

```
act Head[l: List, d: Data]
   pre = { l.head != Null}
   post = { d' = (l.head).val }
act Tail[l: List]
   pre = { l.head != Null}
   post = { l'.head = (l.head).next }
```

Propiedad a verificar

```
assertCorrectness check[l:List, d:Data] {
   pre = { l.head != Null and noCycles[l.head] }
   program = { Tail[l] ; (Head[l,d] + Tail[l])* }
   post = { noCycles[l'.head] }
}
```

Análisis de Especificaciones DynAlloy



Escalabilidad

La principal limitación de los mecanismos algorítmicos de verificación es el **problema de la explosión combinatoria de estados**: el espacio de estados de un programa crece de manera combinatoria con el número de componentes del sistema y su estado.
Para SAT-Solving ocurre algo similar: el tamaño de las fórmulas crece de manera exponencial.
Existen varios enfoques para lidiar con ese problema, entre ellos:

- Algoritmos simbólicos.
- Model Checking on the fly.
- Reducción de orden parcial.
- **Abstracción**.

Abstracción

- La abstracción consiste en hacer el estado del sistema menos detallado, sin perder los elementos esenciales para la descripción de la propiedad de interés ni la posibilidad de ser analizada.
- Permite que los mecanismos de verificación algorítmicos sean aplicables a sistemas más grandes y complejos.

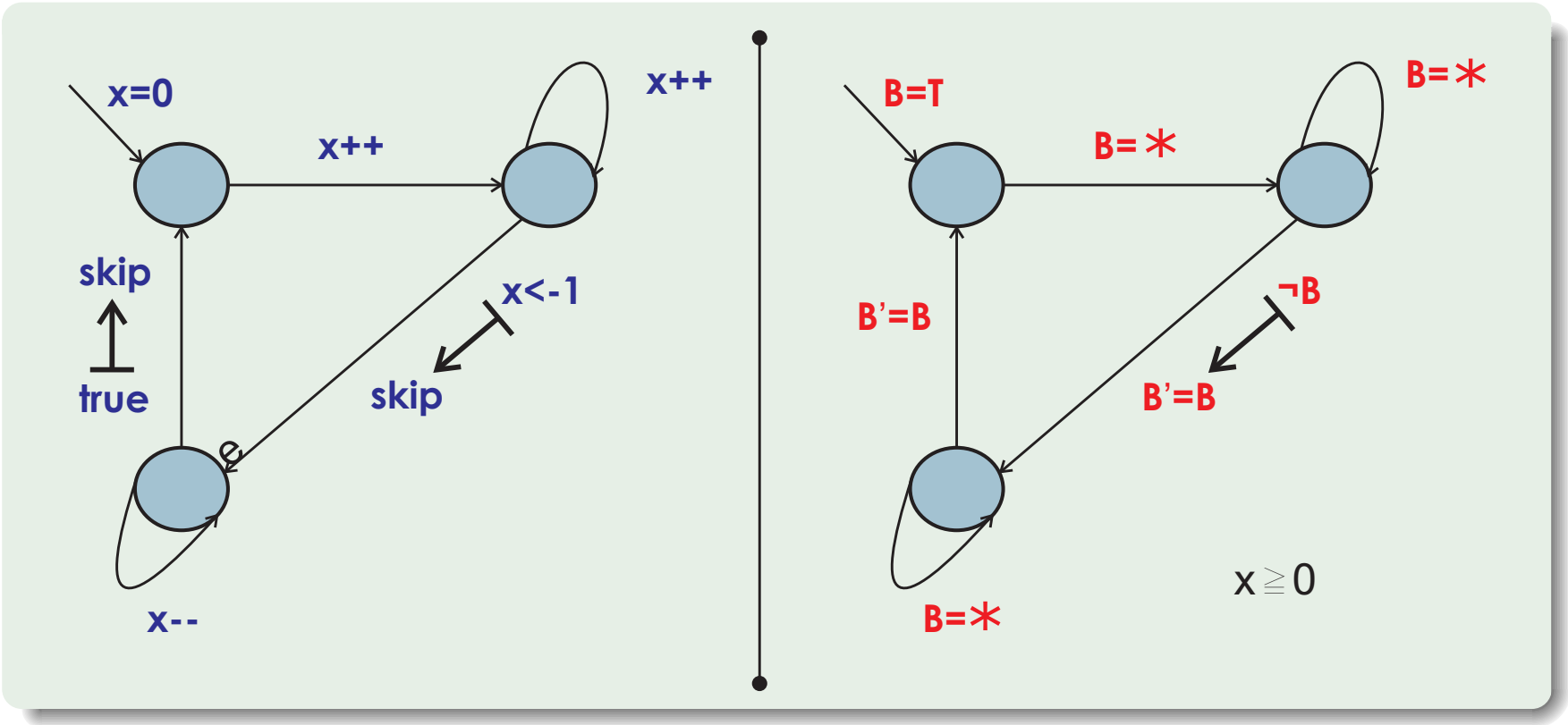
Abstracción por Predicados

- La abstracción por predicados es un mecanismo que consiste en construir un modelo aproximado del sistema, con menos detalle.
- Permite construir automáticamente una abstracción del sistema a partir de un modelo (en nuestro caso una especificación DynAlloy) y de una familia de predicados de estado sobre el sistema, los cuales son provistos por el usuario, y verificar una propiedad sobre el modelo abstracto.

Abstracción por Predicados - (Cont.)

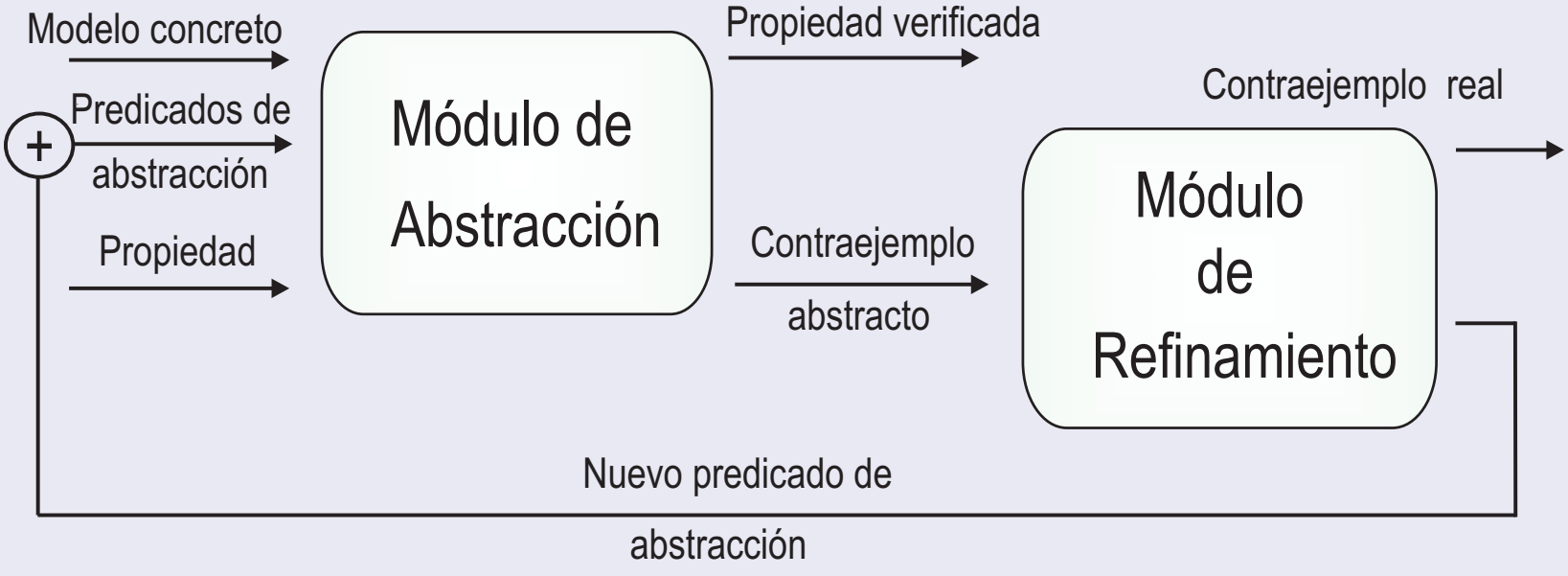
- Es posible asegurar que si la propiedad vale en el modelo abstracto, también vale en el modelo concreto (la abstracción es conservativa).
- Si se encuentra una violación de la propiedad en el modelo abstracto, no necesariamente la propiedad en inválida en el modelo concreto (la abstracción es débil): puede tratarse de un contraejemplo *espúreo*, caso en el cual es necesario refinar la abstracción.

Abstracción por Predicados - Ejemplo



Abstracción por Predicados - La Herramienta

Abstracción por Predicados "in a nutshell"



Abstracción por Predicados - La Herramienta

- La implementación fue realizada sobre el lenguaje de programación Java.
- Interactúa con las herramientas DynAlloy Translator y Alloy Analyzer.
- La herramienta está diseñada para incorporar fácilmente nuevos algoritmos de abstracción y formas de descubrir nuevos predicados.

Abstracción por Predicados - Algoritmos

- La herramienta cuenta con dos algoritmos de abstracción: OnDemand, WithGraph.
- Ambos algoritmos permiten el refinamiento de la abstracción en casos de encontrar algún contraejemplo espúreo.
- Los algoritmos interactúan con un módulo que almacena calculos previos, evitando invocaciones a Alloy.
- WithGraph utiliza un grafo como estructura de control, lo que hace posible la detección de ciclos.

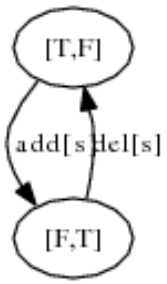
Detección de Ciclos

Consideramos el siguiente ejemplo de un modelo DynAlloy de conjuntos:

- Acciones: add[s] y del[s].
- Programa: {empty[s]} (add[s];del[s])* {empty[s']}

Ejecución Abstracta:

- Predicados de abstracción: empty[s] y one[s].
- Grafo de la ejecución:



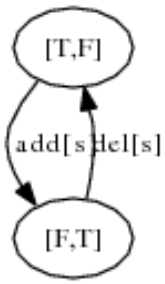
Detección de Ciclos

Consideramos el siguiente ejemplo de un modelo DynAlloy de conjuntos:

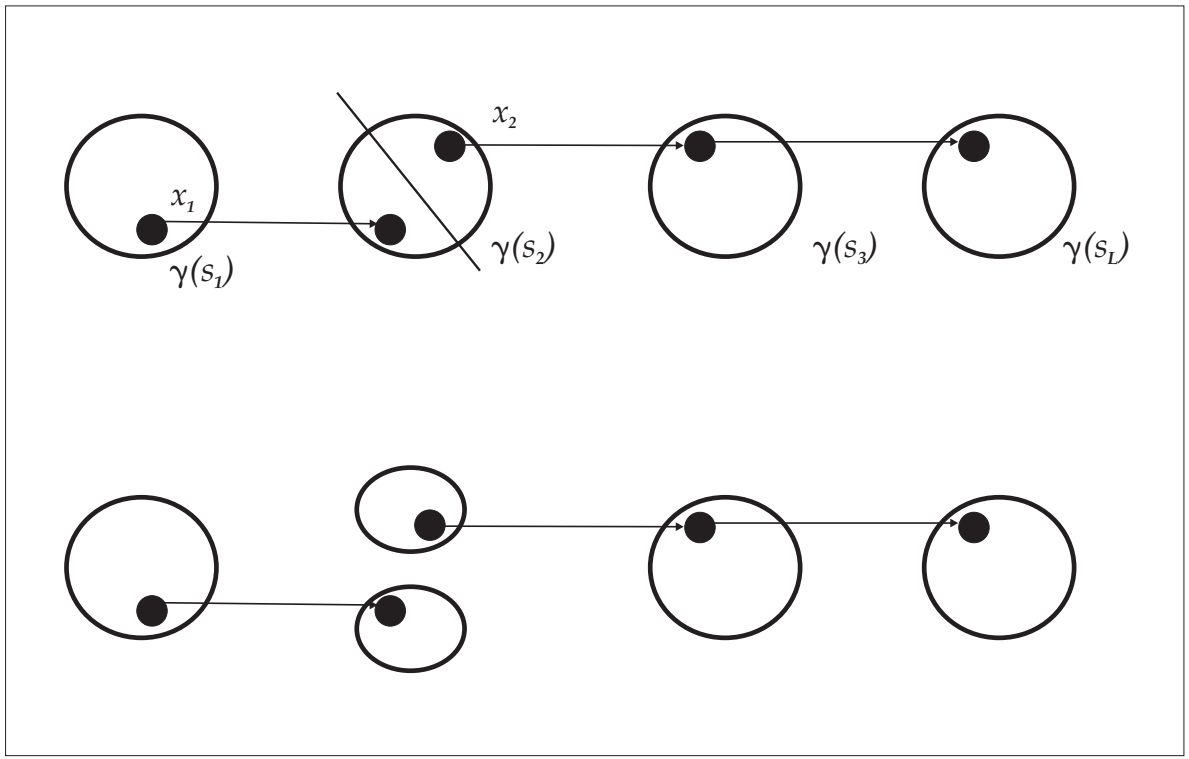
- Acciones: add[s] y del[s].
- Programa: {empty[s]} (add[s];del[s])* {empty[s']}

Ejecución Abstracta:

- Predicados de abstracción: empty[s] y one[s].
- Grafo de la ejecución:



Refinamiento de Abstracciones



Refinamiento de Abstracciones (Cont.)

- La técnica de refinamiento de abstracciones está basada en descubrimiento de predicados.
- En casos de que la herramienta encuentre un contraejemplo espúreo, el módulo de refinamiento se encarga de encontrar un nuevo predicado de abstracción utilizando el cálculo de la *weakest precondition* (wp).
- La técnica, tal como la dicta la teoría, experimentalmente fue ineficaz. Por lo tanto, le agregamos cierta heurística lo que nos otorgó mejores resultados.
- La técnica amerita mayor investigación sobre el tema, lo cual escapa a los objetivos de este trabajo final.

Refinamiento de Abstracciones - Pérdida de detalle

Consideremos otra vez el modelo de conjuntos, con el siguiente programa:

```
{empty[s]} add[s];add[s];del[s];del[s] {empty[s']}
```

La traza de ejecución abstracta sería la siguiente:

```
[T, F]
add[s]
[F, T]
add[s]
[F, F]
del[s]
[F, X]
del[s]
[X, X]
```

Como se puede apreciar, no es posible expresar en abstracto que el conjunto tiene dos elementos. En este caso es necesario un predicado que exprese esta propiedad.

Refinamiento de Abstracciones - Pérdida de detalle

Consideremos otra vez el modelo de conjuntos, con el siguiente programa:

```
{empty[s]} add[s];add[s];del[s];del[s] {empty[s']}
```

La traza de ejecución abstracta sería la siguiente:

```
[T, F]
add[s]
[F, T]
add[s]
[F, F]
del[s]
[F, X]
del[s]
[X, X]
```

Como se puede apreciar, no es posible expresar en abstracto que el conjunto tiene dos elementos. En este caso es necesario un predicado que exprese esta propiedad.

Refinamiento de Abstracciones - Pérdida de detalle

Consideremos otra vez el modelo de conjuntos, con el siguiente programa:

```
{empty[s]} add[s];add[s];del[s];del[s] {empty[s']}
```

La traza de ejecución abstracta sería la siguiente:

```
[T, F]
add[s]
[F, T]
add[s]
[F, F]
del[s]
[F, X]
del[s]
[X, X]
```

Como se puede apreciar, no es posible expresar en abstracto que el conjunto tiene dos elementos. En este caso es necesario un predicado que exprese esta propiedad.

Casos de Estudio

Objetivo

- La eficiencia en el procedimiento de **abstracción** de la herramienta.
- La respuesta del módulo de **refinamiento**, en caso de existir contraejemplos espúreos.

Cuestiones

- Los tiempos se compararon con los de la herramienta Alloy Analyzer.
- Se utilizaron las implementaciones *OnDemandAbstractor* y *AbstractorWithGraph* para realizar los experimentos.
- Se observó con diferentes parámetros de entradas, para evaluar el rendimiento de la herramienta.

Casos de Estudio

Objetivo

- La eficiencia en el procedimiento de **abstracción** de la herramienta.
- La respuesta del módulo de **refinamiento**, en caso de existir contraejemplos espúreos.

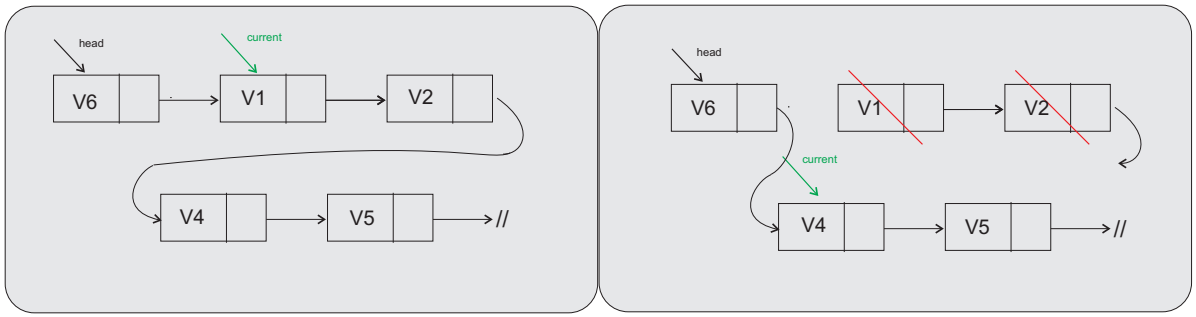
Cuestiones

- Los tiempos se compararon con los de la herramienta Alloy Analyzer.
- Se utilizaron las implementaciones *OnDemandAbstractor* y *AbstractorWithGraph* para realizar los experimentos.
- Se observó con diferentes parámetros de entradas, para evaluar el rendimiento de la herramienta.

Caso de Estudio I: Abstracción

Lista

- **Modelo:** Representación de una lista.
- **Programa:** Dado una lista acíclica, eliminar ciertos valores de la misma.
- **Propiedad a verificar:** La lista resultante del programa, preserva aciclidad.
- **Predicados de entrada necesarios:** 8



Caso de Estudio I: Abstracción (Cont.)

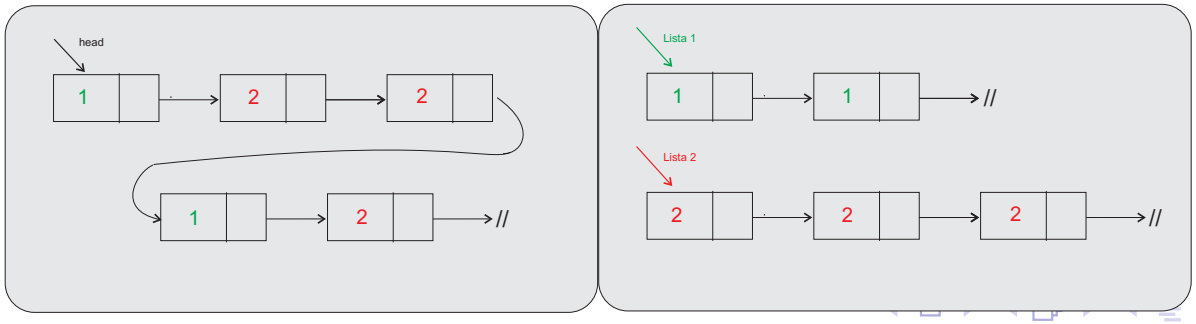
Loop unrolls	Alloy	OnDemand	WithGraph
15	17m42s	1h 46m 42s	15s
16	28m 31s	TIMEOUT	15s
17	MEM error		15s
...			

Figura: Tiempos de test: eliminación preserva aciclicidad.

Caso de Estudio II: Abstracción

División de lista en dos

- **Modelo:** Representación de una lista.
- **Programa:** Dado una lista, se divide en dos nuevas listas, según el valor de sus elementos: si el elemento es un 1 se lo asigna a una lista, y si es un 2 a la otra.
- **Propiedad a verificar:** indica que las listas resultantes contienen, una todos 1 y la otra todos 2.
- **Predicados de entradas necesarios:** 17



Caso de Estudio II: Abstracción (Cont.)

Loop unrolls	Alloy	OnDemand	WithGraph
4	1m	1m 46s	1m 5s
10	2m 10s	2m 44s	1m 9s
12	45m 20s	21m 45s	1m 9s
13	TIME OUT	70m 30s	1m 12s
14		TIMEOUT	1m 12s
...			
100			1m 12s
...			

Figura: Resultados de test: división de lista en dos.

Caso de Estudio III: : Abstracción + Refinamiento

Caso de Estudio I + Refinamiento

- **Modelo:** Representación de una lista.
- **Programa:** Dado una lista acíclica, eliminar ciertos valores de la misma.
- **Propiedad a verificar:** La lista resultante del programa, preserva aciclidad.
- **Predicados de entrada necesarios:** 6

Caso de Estudio III: : Abstracción + Refinamiento (Cont.)

Loop unrolls	Alloy	WithGraph+Refinamiento
15	17m 42s	31s
16	28m 31s	31s
17	MEM error	31s
18	MEM error	31s
...		
100	MEM error	31s
...		

Figura: Alloy Analyzer vs. WithGraph+Refinamiento.

Conclusiones

+ En el caso de que se detecten ciclos, es posible incrementar las cotas de iteraciones tanto como se lo desee.
+ A veces se lograron mejoras significativas en los tiempos de análisis.
+ Para algunos casos fue posible eliminar los contraejemplos espúreos.
- Depurar contraejemplos espúreos es muy costoso para los SAT-Solvers.
- Es necesario mejorar la técnica de refinamiento, para no recurrir a heurísticas para descubrir nuevos predicados.

Experiencias

+ El presente trabajo nos permitió realizar dos publicaciones.
+ Nos vimos obligados a profundizar nuestros conocimientos en algunos aspectos que no están cubiertos en la carrera.
- El tiempo demandado para finalizar este trabajo final, fue bastante superior a lo esperado (2 años).

Trabajos Futuros

- Estudiar más a fondo la teoría de refinamiento de abstracciones.
- Integrar la herramienta con un traductor de código Java anotado a DynAlloy, para poder verificar propiedades sobre programas Java.