Verificando diseños BON mediante Alloy

Pablo Castro Pablo Ponzio
Ramiro Demasi
Universidad Nacional de Río Cuarto
Departamento de Computación
e-mail:{pcastro,pponzio,ramiro}@dc.exa.unrc.edu.ar
Gabriel Baum
LIFIA - Universidad Nacional de La Plata
e-mail:gbaum@sol.info.unlp.edu.ar

Resumen

En este artículo presentamos una técnica para traducir diseños estructurales expresados en el lenguaje BON, al lenguaje formal Alloy. En donde, la principal ventaja de la traducción es que puede realizarse automáticamente mediante herramientas de software.

Adicionalmente, esta metodología puede ser usada para validar propiedades sobre los diseños utilizando el Alloy Analyzer. Para finalizar, mostramos la aplicación a un caso de estudio de Darwin Tool, una herramienta que implementa parte de esta traducción.

Palabras Clave: Métodos Formales, Ingeniería de Software, Orientación a Objetos, Diseños Arquitecturales, Alloy, BON.

CACIC 2005 - II Workshop de Ingeniería de Software y Bases de Datos

1. Introducción

Con la aparición del paradigma de orientación a objetos, surgieron diferentes lenguajes de modelado para soportar esta nueva tecnología. En los últimos tiempos apareció como una alternativa a UML (Unified Modeling Language [5]), uno de los más utilizados, el lenguaje BON (Bussiness Object Notation [3]). En [4] se muestra como BON permite una aplicación transparente de los métodos formales. Esto se debe principalmente a su notación simple y a que fue diseñado para soportar el método de Diseño por Contratos, incluyendo un lenguaje formal de aserciones. Otras características importantes de BON son *Seamlessness* (desarrollo sin transiciones), que intuitivamente significa que el paso de una etapa en el desarrollo a la siguiente es directo, y Reversibilidad, que implica que los cambios realizados en una etapa del desarrollo son reflejados en las anteriores. Estas favorecen el reuso y simplifican el mantenimiento del software.

El presente trabajo se basa en la integración de dos herramientas que pueden ser utilizadas en la etapa de diseño del desarrollo de un sistema de software, la notación BON, y el lenguaje formal Alloy. Mostramos aquí una forma de realizar la traducción de un diseño en BON a una especificación Alloy, basada en [4], la cual puede efectuarse por medio de herramientas de software, de manera automática.

Además mostramos un caso de estudio en el que se utilizó una herramienta implementada por los autores, llamada Darwin Tool, que permite traducir un subconjunto de BON a Alloy.

La estructura de este artículo es la siguiente: en la secciones 2 y 3 se presenta una breve introducción a los lenguajes BON y Alloy respectivamente. En la sección 4 se muestra la traducción de BON a Alloy. En 5 se presenta la herramienta Darwin Tool y en 6 una aplicación de la misma a través de un caso de estudio. Finalmente, presentamos las conclusiones y los trabajos futuros.

2. Introducción a BON

BON (Business Object Notation)[3, 7] es un lenguaje de modelado para especificar y describir sistemas bajo el paradigma de orientación a objetos. Una de sus características es que no sólo provee un lenguaje gráfico para la descripción de sistemas, sino que también posee un proceso recomendado para el desarrollo de software. Otra característica importante es que todo modelo gráfico BON tiene una forma textual equivalente. BON es extremadamente simple en comparación a otros lenguajes: solo posee 2 diagramas de modelado, y no permite diferentes vistas de un diseño, lo cual evita las inconsistencias que puedan ser introducidas al existir distintas descripciones de un mismo componente.

BON fue diseñado para soportar tres técnicas principales: *Seamlessness*, *Reversibilidad* y *Diseño por Contratos*.

- Seamlessness: se refiere al principio de utilizar un conjunto consistente de conceptos y notaciones a través del ciclo de vida del software, desde el análisis a la implementación y mantenimiento. Por lo que el paso siguiente en cada etapa del desarrollo es directo. Los beneficios de *seamlessness* son numerosos.

- Reversibilidad: los cambios producidos en una etapa del desarrollo pueden ser automáticamente reflejados en las etapas anteriores. Por ejemplo, un cambio en una clase de Eiffel es directamente reflejada en cambios sobre la clase de diseño.

- Diseño por Contratos: es una metodología de desarrollo de software orientado a objetos que se basa en el concepto de contrato. Un contrato es una especificación de las responsabilidades y características de una componente de software. Esta noción está basada en la idea de pre-postcondición, introducida por Hoare y Floyd en los años 70 - lo que permite producir software de alta calidad y fácilmente mantenible.

Al igual que otros lenguajes orientados a objetos, el principal mecanismo de especificaciones en un modelo BON es el concepto de clase. Una clase en BON introduce un nuevo tipo y se la considera un módulo. Las clases BON estan compuestas de: un nombre, un conjunto de *features*, y un contrato compuesto de precondición, postcondición e invariante de clase. El invariante es una aserción que todas las instancias de la clase deben satisfacer. El contrato especifica las funcionalidades de la clase, y es expresado utilizando el lenguaje de aserciones que posee BON, el cual está basado en la lógica de primer orden tipada.

Los *features* (o características) de una clase pueden ser *queries* o *commands*:

- Queries: Dentro de esta categoría entran las funciones y atributos de los lenguajes tradicionales. Es importante destacar que las *queries* no producen cambios en el estado del sistema.

- Commands: Corresponden a los procedimientos de los lenguajes Orientados a Objetos tradicionales. Se ejecutan para producir cambios de estado.

Cada *feature* tiene asociada una restricción de visibilidad, y puede tener diferentes modos de implementación, puede ser abstracta, efectiva o redifinida.

Es interesante destacar que BON sólo soporta dos tipos de relaciones dentro de un diagrama de clases:

- Relación Cliente-Proveedor: indica que una clase (el cliente), utiliza algún servicio brindado por otra clase (el proveedor). Hay tres tipos de relaciones cliente-proveedor en BON: asociación, agregación y asociación compartida (*shared association*).
- Relación de Herencia: una clase hereda comportamiento de una o más clases padres. La herencia introduce una relación de refinamiento sobre los contratos de las clases asociadas. Además, en BON, una clase hijo es considerada un subtipo de sus clases padres.

3. Introducción a Alloy

Alloy [1, 2] es un lenguaje formal, basado en la lógica relacional de primer orden, que permite modelar formalmente sistemas de software; especificando el espacio de estados, restricciones, y propiedades de los mismos. Un modelo Alloy está compuesto por signaturas, hechos, predicados, funciones y aserciones.

Una signatura introduce un conjunto de átomos, y a la vez un tipo. Además en la misma declaración pueden definirse relaciones con otros tipos, como por ejemplo:

```
sig Persona { DNI: Int }
```

que define el tipo `Persona`, con una relación binaria llamada `DNI`, que asocia cada persona con su número de documento. Las relaciones pueden ser de cualquier aridad y pueden imponerse restricciones de cardinalidad sobre las mismas como en:

```
sig Persona { DNI: Int, trabaja_en: set Empresa }
sig Empresa { empleados: set Persona, id: empleados -> one Int }
```

en donde la relación `trabaja_en`: $Empresa \times Empresa$ relaciona a una persona con las empresas en que trabaja, e `id`: $Empresa \times Empleados \times Int$ es una relación ternaria que a cada empleado de la empresa le asigna un único entero que es su identificador.

En Alloy, los conjuntos introducidos por diferentes signaturas son disjuntos a menos que se declare una signatura como extensión de otra. Esto se lograría de la siguiente forma: supongamos que, en el ejemplo anterior, queremos diferenciar a las personas que trabajan para alguna empresa de las que no:

```
sig Persona { DNI: Int }
sig Empleado extends Persona { trabaja_en: set Empresa }
sig Empresa { empleados: set Empleado, id: empleados -> one Int }
```

Aquí, la signatura `Empleado` es un subconjunto del asociado a `Persona` y además un nuevo tipo.

Los hechos (o *facts*) son fórmulas que restringen los valores posibles para los conjuntos y relaciones. Por ejemplo, si queremos asegurar que dos personas diferentes no tienen el mismo DNI:

```
fact unicoDNI { all p1, p2: Persona |
                p1 != p2 => p1.DNI != p2.DNI }
```

además, un empleado está relacionado a una empresa por medio de la relación `trabaja_en` si y sólo si pertenece al conjunto de empleados de la misma:

```
fact { all emp: Empleado | all e: Empresa |
       e in emp.trabaja_en <=> emp in e.empleados }
```

En las fórmulas anteriores el operador (.) representa la composición de relaciones y la palabra clave `in` significa pertenencia a un conjunto.

También se pueden definir funciones, fórmulas parametrizadas que pueden ser invocadas en diferentes contextos, reemplazando sus parámetros formales por parámetros actuales del mismo tipo, así como predicados, los cuales difieren de las funciones en que, en lugar de expresiones, son fórmulas parametrizadas que no retornan valores. Un ejemplo de predicado sería el siguiente:

```
pred Despedir (e, e': Empresa, emp: e.empleados, emp': Empleado){
  e'.empleados = e.empleados - emp
  e'.id = e.id - emp -> (emp.(e.id))
  emp'.trabaja_en = emp.trabaja_en - e
  emp'.DNI = emp.DNI
}
```

que quita al empleado `emp` del conjunto de empleados de la empresa `e`. Aquí las variables primadas `e'` y `emp'` indican el estado de la empresa `e` y del empleado `emp` posterior a la ejecución de la operación.

Por último, en un modelo Alloy podemos definir aserciones, es decir, fórmulas que el diseñador espera que sean válidas, con respecto a las restricciones impuestas por medio de los hechos. Por ejemplo, podemos esperar que después de despedir a un empleado, este ya no tenga ninguna relación con la empresa:

```
assert despedido {
  all e1,e2: Empresa | all emp1: e.empleados |
  all emp2: Empleado | Despedir(e1, e2, emp1, emp2) =>
  not (e2 in emp2.trabaja_en) && not (emp2 in e2.empleados)
}
```

Alloy posee dos tipos de comandos que se pueden ejecutar sobre modelos:

- Check *assert* for *n*: Busca un contraejemplo de la aserción *assert*, instanciando cada signatura con a lo sumo *n* átomos.
- Run *f* for *n*: Busca un modelo para la función o predicado *f*, *n* tiene el mismo significado que el comando anterior.

Una particularidad de Alloy es que fue diseñado conjuntamente con una herramienta, el Alloy Analyzer, que permite simular modelos y ejecutar los comandos mencionados anteriormente, con el fin de chequear que se cumplan ciertas propiedades de los modelos. Cabe aclarar que el hecho de que el Alloy Analyzer no encuentre un modelo para una aserción, función o predicado, no significa que la aserción se verifique, ni que la función o predicado sea insatisfacible, ya que para poder realizar las verificaciones es necesario imponer una cota en la cantidad de átomos de las signaturas y relaciones.

A pesar de esto, una gran cantidad de aplicaciones prácticas pueden beneficiarse de la utilización de este lenguaje.

4. La traducción BON a Alloy

Como mencionamos anteriormente, se desarrolló una forma de traducir BON a Alloy. Podemos presentar esta traducción mediante los siguientes items.

4.1. Clases y estados

El primer paso en la traducción es expresar los tipos de un modelo BON en la especificación Alloy. Para esto, se introduce en la especificación una signatura por cada clase en el modelo. Por ejemplo, si tenemos la clase `Persona` debe agregarse a la especificación:

```
sig Persona { }
```

La signaturas no tienen estructura interna debido a que esta se expresará en una signatura más general llamada `State`. Intuitivamente, `State` representa los posibles estados del sistema. Cabe destacar que `State` simula el concepto de estado en Alloy. Por ejemplo, si tenemos las clases `A`, `B` y `C`, la signatura *State* debería ser la siguiente:

```
sig State {
  as: set A,
  bs: set B,
  cs: set C
}
```

intuitivamente `as`, `bs` y `cs` son las posibles instancias de `A`, `B` y `C` respectivamente.

4.2. Términos y fórmulas

La traducción de las fórmulas es muy sencilla. Los operadores se traducen según la siguiente tabla: Los términos como `t.query` que representan una navegación se traducen indicando un estado

BON	Alloy
and	&&
or	||
$implies$	=>
iff	<=>
not	!
$x : T$	$x : T$
$x.r$	$x.r$

determinado. De esta manera, lo anterior se traduce en: `t.(s.query)` siendo `s` una variable de la signatura `State`.

4.3. Queries

La traducción de las *queries* depende principalmente de su contrato y del tipo de retorno. Una *query* sin contrato, y cuyo tipo de retorno es no Booleano, se traduce en una relación de la signatura `State`, como se muestra en el siguiente ejemplo:

```
class C
  feature
    q: T   -- T /= Bool
  end
```

el modelo anterior se traduce en:

```
one sig undef { }
sig C { }
sig State {
  cs: set C,
  ts: set T,
  q: cs -> one (ts + undef)
}
```

La relación `q` de `State` asocia cada instancia de `C` con un valor de tipo `T`, el cual representa el valor del atributo `q` en un estado particular. Notemos que se agregó una signatura llamada `undef`, cuyo único átomo representa el valor `void` de BON. Que una instancia de la clase `C` este asociada al valor `undef` por medio de `q` en un cierto estado `s`, indica que dicha instancia no posee un valor para el atributo `q` en `s`.

El caso de una *query* con contrato es similar al anterior, sólo que en este caso debemos restringir los conjuntos de instancias de las clases a aquellos que respeten el contrato de la *query*. Esto se logra introduciendo hechos a la especificación Alloy. Por ejemplo, equipemos la *query* `q` anterior con pre y postcondición:

```
class C
  feature
    q: T
      require pre
      ensure post
    end
end
```

ahora su traducción es:

```
sig C { }
sig State {
  ts: set T,
  cs: set C,
  q: cs -> one (ts + undef)
}

pred Pre_q(s: State, self: s.cs) {
  τ(pre)
}
pred Post_q(s: State, self: s.cs, result: s.ts) {
  τ(post)
}
fact { all s: State | all self: s.cs |
       Pre_q(s, self) && Post_q(s, self, self.(s.q)) }
```

En donde τ es una función de traducción de fórmulas Alloy a aserciones BON, que implementa las reglas introducidas en la tabla 1.

El caso de las *queries* Booleanas es diferente a los anteriores ya que Alloy no provee el tipo *Bool*, por lo que para utilizarlo deberíamos introducirlo primero, junto con todas sus operaciones. El problema que surge es que si agregamos los Booleanos a cada especificación, la verificación sería más ineficiente. La solución propuesta es traducir los atributos Booleanos como predicados, para poder utilizar las operaciones Booleanas de Alloy sobre ellos. Esto se llevaría a cabo de la siguiente forma:

```
class C
  feature
    q: Bool
      require pre
      ensure post
    end
end
```

se traduce en:

```
sig C { }
sig State {
  cs: set C
}

pred Pre_q(s: State, self: s.cs) {
  τ(pre)
}
pred Post_q(s: State, self: s.cs) {
  τ(post)
}
pred q(s: State, self: s.cs) {
  Pre_q(s, self)
  Post_q(s, self)
}
```

4.4. Commandos

Un comando se traduce en un predicado, que tiene como parámetros adicionales dos estados, `s` y `s'`, donde intuitivamente `s` representa el estado del sistema anterior a la ejecución del comando, y `s'` representa al estado posterior de la ejecución del mismo. Veamos un ejemplo:

```
class C
  feature
    q1: T1   -- T1 /= Bool
    q2: T2   -- T2 /= Bool
    cmd
      require pre
      ensure post
    end
end
```

suponiendo que `cmd` modifica sólo el atributo *q1*, el modelo anterior se traduce en:

```
sig C { }
sig State {
  cs: set C,
  q1: cs -> one (t1s + undef),
  q2: cs -> one (t2s + undef)
}

pred Pre_cmd(s: State, self: s.cs) {
```

```
  τ(pre)
}
pred Post_cmd(s, s': State, self: s.cs) {
  τ(post)
  self.(s'.q2) = self.(s.q2)
  Delta_C(s, s', self)
}
pred cmd(s, s': State, self: s.cs) {
  Pre_cmd(s, self)
  Post_cmd(s, s', self)
}
pred Delta_C(s, s': State, self: s.cs) {
  s'.cs = s.cs
  all c: s.cs - self |
  c.(s'.q1) = c.(s.q1) && c.(s'.q2) = c.(s.q2)
}
```

Notar que, en la postcondición del método, se mencionan todos los atributos de `self` que no cambian. Esto se debe a una incompatibilidad semántica entre BON y Alloy. Mientras que en el primero los atributos que no se nombran en una postcondición se supone que no cambian, en Alloy los atributos que no se mencionan pueden tomar cualquier valor. Por esta razón se introduce el predicado `Delta_C`, que asegura que todos los objetos que no se nombran en la postcondición no sufren cambios.

4.5. Los invariantes

Una de las ventajas más relevantes de la traducción de BON hacia Alloy es que nos permite verificar automáticamente que:

- Los comandos mantienen a los invariantes.
- Los invariantes pueden ser satisfacibles, es decir, no son inconsistentes.

Para traducir un invariante se utilizan los párrafos *pred* y también los *assert*, los primeros sirven para expresar la fórmula lógica del invariante, y los segundos nos permiten verificar que los comandos mantienen a los invariantes. Un punto a destacar es que los predicados que expresan a los invariantes son dependientes de los estados, es decir, estos toman como parámetro a un estado. Ilustremos estas nociones con un ejemplo:

```
class C
  feature
    com
      require pre
      ensure post
    end
  invariant inv
end
```

las signaturas de la clase y `State` se generan como se mostró anteriormente, y el invariante se traduce:

```
pred C_Inv(s: State) {
  all o: s.cs | τ(inv)
}
```

Aquí es importante destacar que la cuantificación sobre `s.cs` se debe a que el invariante predica sobre los elementos de la clase `C`. Falta decir que el comando `com` debe mantener el invariante, esto puede expresarse por medio de un *assert*, de la siguiente forma:

```
assert {
  all s, s': State | all c: s.cs | C_Inv(s)
                    && C_com(s, s', c) => C_Inv(s')
}
```

En el caso de que una especificación tenga diversos invariantes, supongamos:

```
Inv1,..., InvN
```

debe verificarse que exista un modelo que satisfaga a todos (es decir, que la especificación es consistente), esto puede hacerse por medio de un fórmula que combine a todos, es decir:

```
pred Consistent(s:State) {
  Inv1(s) && Inv2(s) && ... && InvN(s)
}
```

Y luego con el comando `run Consistent` se puede verificar automáticamente el modelo con la herramienta Alloy Analyzer.

4.6. Verificando Herencia

La herencia es una de las herramientas más importantes de los lenguajes orientados a objetos, y como tal es soportada por BON. Es interesante analizar como la herencia influye en los contratos, a saber:

- Los invariantes de las superclases deben ser satisfechas por las subclases.
- Las precondiciones de los métodos pueden ser solo debilitadas por las subclases.
- Las postcondiciones de los métodos pueden ser solo fortalecidas por las subclases.
- Los items redefinidos pueden solo tener el mismo tipo, o tipos descendientes de ellos.

Teniendo en cuenta los anteriores items, puede decirse que la herencia es concebida como un refinamiento. Para tratar automáticamente con la herencia, BON agrega en los métodos de las subclases - mediante una disyunción - la precondición definida en la superclase. De igual forma, en las postcondiciones se agregan - por medio de conjunciones - las postcondiciones de las superclases, cumpliendo de esta forma la idea de refinamiento. Similarmente, en las subclases se agregan por medio de conjunciones los invariantes de las superclases. Sin embargo, esta metodología puede traer algunos problemas:

- El invariante de una subclase puede ser contradictorio con el de su superclase, con lo cual la subclase nunca podría satisfacer su invariante.

- La postcondición de una rutina de una subclase puede ser incompatible con la postcondición de la superclase, lo cual significa que el proveedor de un servicio nunca podrá cumplir un contrato, disparándose una excepción siempre que este servicio sea invocado.
- La precondición de una rutina puede sufrir el mismo problema que las postcondiciones, provocando que ningún cliente pueda cumplir el contrato del servicio afectado.

Para detectar estos problemas podemos utilizar Alloy, para lo cual se deben agregar las siguientes obligaciones de prueba (*asserts*) en la traducción:

- La conjunción de la precondición de una rutina heredada con la precondición original debe ser satisfacible.
- La conjunción de los invariantes debe ser satisfacible.
- La conjunción de la postcondición de una rutina heredada con la postcondición original debe ser satisfacible.

Por otra parte, estructuralmente la relación de subtipo puede ser expresada por medio del modificador *extends* en Alloy 3.0. Como ejemplo veamos el siguiente modelo:

```
class A
  feature
    q: A
      require pre
      ensure post
    end
  invariant inv
end

class B
  inherit A redefine q
  feature
    q: B
      require pre'
      ensure post'
    end
  invariant inv'
end
```

la traducción de los tipos del modelo es:

```
sig A {}
sig B extends A {}
sig State {
  as: set A,
  bs: set B,
  q: as -> one (as + undef)
} { bs in as }                                          (1)
fact { all s: State | b: s.bs | b.(s.q) in s.bs }       (2)
```

En donde, la línea (1) garantiza que en los estados se va a preservar la relación de inclusión entre el conjunto de instancias de `B` y el conjunto de instancias de `A`. Mientras que en (2) se expresa la restricción de que `q` está redefinida en `B`. El *feature* `q` de `A` se traduce de la manera usual. A continuación mostramos como se realiza la traducción de la *query* `q` de `B`.

```
pred B_Pre_q(s: State, self: s.bs) {
  τ(pre')
}
pred B_Post_q(s: State, self: s.bs) {
  τ(post')
}
fact { all s: State | all self: s.bs |
       (B_Pre_q(s, self) || A_Pre_q(s, self))        (3)
       (B_Post_q(s, self) && A_Post_q(s, self)) }    (4)
```

Observemos que en la línea (3) se debilita la precondición original de la *query* `q`, y en la línea (4) se fortalece la postcondición original. Que exista un modelo que satisfaga dicha especificación, equivale a que `pre'` y `post'` son consistentes con la pre y postcondición original de `q` (`pre` y `post`), esto puede chequearse automáticamente con el Alloy Analyzer.

Para finalizar veamos como probar la consistencia del invariante de `B` con respecto al invariante de `A`. Para esto introducimos un predicado que realiza la conjunción de los invariantes individuales de `A` y `B`:

```
pred A_Inv(s: State) {
  τ(inv)
}
pred New_B_Inv(s: State) {
  τ(inv')
}
pred B_Inv(s: State) {
  A_Inv(s) && New_B_Inv(s)
}
```

Ejecutando el comando `run B_Inv n` en el Alloy Analyzer logramos nuestro objetivo.

En la sección siguiente describimos la herramienta Darwin Tool que implementa parte de la traducción expuesta anteriormente.

5. Darwin Tool

Darwin Tool es una herramienta que permite editar diagramas en notación BON, y realizar automáticamente la traducción de los mismos a Alloy.

Por el momento, la herramienta implementa una parte de la traducción presentada, que incluye: clases, relaciones cliente-proveedor y el subconjunto proposicional del lenguaje de aserciones provisto por BON.

5.1. Un Caso de Estudio utilizando Darwin Tool

Hemos testeado el funcionamiento de la herramienta con varios casos de estudio, en este artículo mostramos sólo uno de ellos por cuestiones de espacio.

El siguiente gráfico presenta en detalle un diseño BON. El cual fue traducido automáticamente por Darwin y verificado por la herramienta Alloy Analyzer, obteniendo un modelo del diseño que asegura su consistencia.

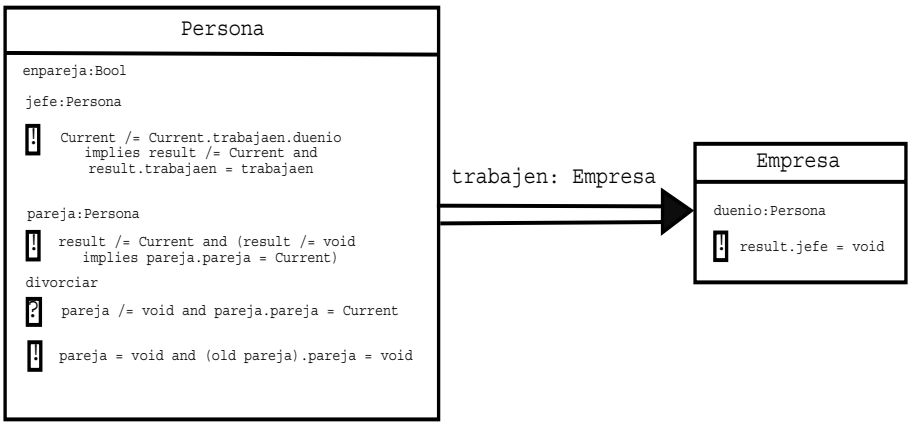

Figura 1: Caso de estudio

Uno de los modelos encontrados por el Alloy Analyzer para este caso de estudio es el siguiente:

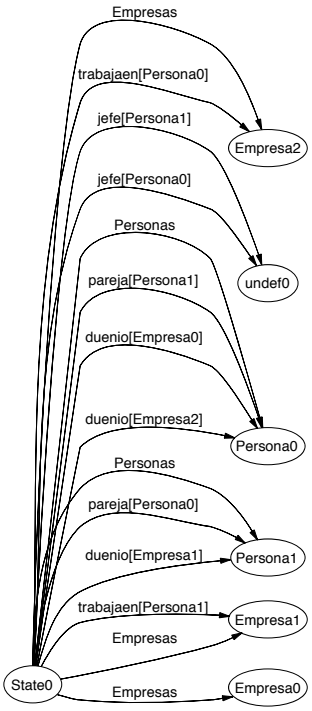

Figura 2: Modelo encontrado por Alloy Analyzer

6. Conclusiones

Consideramos que este proyecto hace un aporte hacia la integración de métodos formales con las técnicas de diseño informales, las cuales son usadas ampliamente en la actualidad.

Por una parte, la poca utilización de los métodos formales en la industria se debe a que requiere cierta clase de conocimientos matemáticos por parte del usuario, y este tipo de entrenamiento es costoso en tiempo y dinero. Por otra parte, también sabemos que, en aplicaciones críticas, los métodos formales son necesarios si se quiere asegurar el correcto funcionamiento de las mismas. Aquí es

donde Darwin Tool entra en escena, ya que, permite utilizar Alloy para validar ciertas propiedades de un diseño realizado en la notación BON, esto implica que la utilización de formalismos es intuitiva para el usuario, es decir, no necesita tener conocimientos matemáticos para aplicarlos. Este enfoque es completamente diferente al de las herramientas existentes en la actualidad, como por ejemplo *z-eves*, que permite modelar en el lenguaje *Z* y realizar verificaciones con la asistencia del desarrollador; Isabelle/HOL, el demostrador semi-automático de teoremas; e inclusive el mismo Alloy, ya que fueron pensadas para usuarios con una preparación adecuada.

7. Trabajos Futuros

Actualmente estamos extendiendo y testeando las herramientas y técnicas explicadas, lo cual implica un conjunto de tareas que consideramos como trabajos futuros:

- Formalizar la traducción y demostrar su correctitud.
- Extender el lenguaje BON para soportar operadores de la lógica relacional.
- Extender Darwin Tool para que soporte la edición y traducción de todas las construcciones de BON.

Referencias

[1] D. Jackson. Alloy 3.0 Reference Manual. May 2004.

[2] D. Jackson. Micromodels of Software: Lightweight Modeling and Analisis with Alloy. February 2002.

[3] Kim Waldén and Jean-Marc Nerson. Seamless Object-Oriented Software Architecture. Addison-Wesley 1994.

[4] Castro Pablo, Baum Gabriel. Integrando BON con Alloy.

[5] J. Rumbaugh G. Booch and I. Jacobson. The Unified Modeling Language User Guide. Addison-Wesley 1999.

[6] Richard F. Paige and Jonathan S. Ostroff. A Comparation of Business Object Language and the Unified Modeling Language. May 1999.

[7] Richard F. Paige. An Introduction to BON. August 1999.