Utilizando Contratos de Reuso con Alloy

Pablo Castro[1] **Gabriel Baum**[2]

[1]Departamento de Computación
Universidad Nacional de Río Cuarto
email: pcastro@dc.exa.unrc.edu.ar
[2]Laboratorio de Investigación y Formación en Informática Avanzada
Universidad Nacional de La Plata
Calle 50 y 115 –1er. Piso – (1900) La Plata
TE/FAX: (0221) 4228252
e-mail: gbaum@info.unlp.edu.ar

Palabras Clave : Ingeniería de Software, Métodos Formales, Teoría de la Computación.

Resumen

El desarrollo de sistemas de software confiables exige la utilización de herramientas que posibiliten razonar rigurosamente acerca de su corrección y consistencia. Este tipo de análisis revela su importancia crítica cuando los sistemas evolucionan en el tiempo, sufriendo modificaciones que pueden alterar seriamente su eficacia o aún volverlos totalmente inútiles.

El establecimiento del Proceso Unificado y el lenguaje UML como un estándar representan un avance, aunque insuficiente para alcanzar dichos objetivos. Más aún, el problema de la evolución ha recibido poco o ningún tratamiento en dicho contexto. En este trabajo se presenta una alternativa para avanzar en esa dirección, a través de la utilización de contratos de reuso[4] -expresados en UML- para describir evoluciones y su traducción a un lenguaje formal de primer orden, llamado Alloy[2], que provee herramientas para verificar algunas propiedades fundamentales de estos contratos. La traducción propuesta es sencilla, eficiente, modular, y provee bases ciertas para un proceso de evolución incremental de los diseños.

1.Introducción

Durante la producción de un sistema de software frecuentemente se definen diseños que serán reutilizados en un futuro, como de la misma forma se utilizan librerías de clases que facilitan a los desarrolladores su trabajo y que proponen soluciones a ciertos problemas paradigmáticos. Esta metodología de trabajo de reutilización de diseños provoca graves problemas a la hora de construir sistemas, muchas veces algunos cambios en los diseños originales pueden provocar inconsistencias y efectos no deseados en el producto a construir, y es por esto que se debe ser extremadamente [illegible] hora de documentar los diseños reusables. La aparición del lenguaje UML(*Unified Modeling Language*) y su aceptación como un estándar, si bien representa una sensible mejora no ha resuelto el problema. El punto es que este lenguaje no tiene asociada una semántica bien definida y, aunque existen diversos trabajos en la actualidad para definirla ([8]a[19]), ninguna de ellas ha sido reconocida unánimemente en los ambientes académicos e industriales.

En este artículo se propone la utilización del lenguaje Alloy [2] para chequear la consistencia de las partes reutilizables de un diseño orientado a objetos, que serán documentadas mediante los contratos de reuso con semántica de comportamiento [4,13]. Sobre esta base, la idea es construir un método que permita verificar de manera rigurosa los diversos artefactos (clases, módulos, relaciones, etc.) que componen un diseño. Alloy es un lenguaje formal, de una notación fácil de entender y que provee herramientas para chequear la consistencia de modelos automáticamente.

2.Evolución de diseños

La tarea de desarrollar sistemas de software se ha convertido en un proceso incremental, en el que las diversas partes y el todo cambian sucesivamente sus características, propiedades e interrelaciones en mayor o menor grado a lo largo del tiempo. En un diseño orientado a objetos es habitual que se modifiquen atributos de clases, relaciones entre ellas o aún que se agreguen o eliminen algunos de dichos artefactos; en dichas situaciones, es posible que un diseño originalmente libre de conflictos se convierta en inconsistente y, mas aún, que dichos conflictos se propaguen a través de las sucesivas iteraciones del proceso de desarrollo, hasta convertir al sistema en un producto de muy baja calidad. El mismo problema aparece cuando se pretenden reutilizar *frameworks* o librerías de clases, las cuales son conjuntos de clases con una determinada funcionalidad que permiten resolver ciertos problemas.

Los contratos de reuso fueron creados para documentar sistemáticamente los componentes reusables de un sistema. Básicamente, un contrato de reuso es un conjunto de participantes que interactúan y pueden ser modificados mediante operadores de reuso. En este trabajo se consideran los contratos de reuso con semántica de comportamiento definidos en [4], debido a que estos integran tanto la parte estructural como de interacción entre sus participantes. Como se señala en el mencionado trabajo, "Un contrato de reúso con semántica de comportamiento es un conjunto de participantes que interactuan y además un invariante"; el contrato de reuso puede expresarse en UML y su invariante en *OCL (Object Contraints Language*), en la figura 1 se da un pequeño ejemplo para ilustrar la noción.

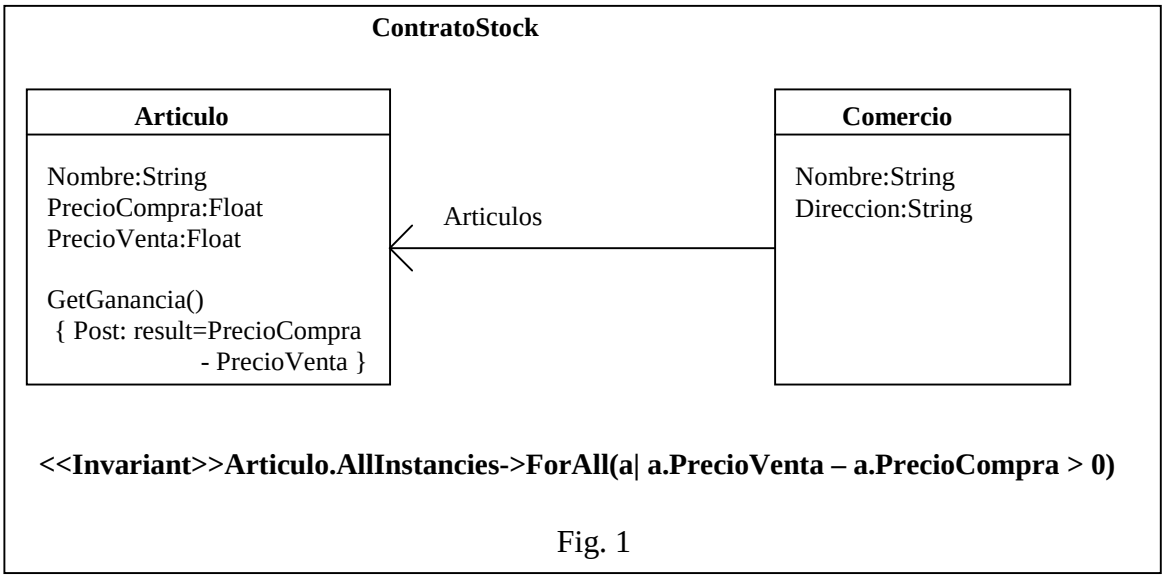


Fig. 1

Más precisamente, un contrato de reuso es un cuádruplo proveniente del dominio semántico CONTRATO donde:

CONTRATO = $\mathbf{P}$ (PART) $\times$ $\mathbf{P}$(CON) $\times$ $\mathbf{P}$ (INTERF) $\times$ INV donde:

PART = es el dominio para los nombre de los participantes,

CON = Es el dominio para las relaciones de conocimientos,

INTERF = Dominio para las interfaces de operación,

INV = Dominio para los invariantes (expresiones OCL bien formadas),

Siendo $\mathbf{P}$ (X) el conjunto de partes de X.

El modelo matemático completo puede ser consultado en [4]. Los contratos pueden ser modificados utilizando operadores de reuso, estos operadores toman un contrato de reuso y un modificador de reuso y devuelven un contrato modificado.

Los contratos de reuso con semántica son una buena herramienta para documentar diseños, en particular los operadores de reuso registran los cambios sobre un contrato, sin embargo nada de esto asegura la consistencia lógica del diseño. Este aspecto puede tratarse de manera análoga al caso de los programas, haciendo algún tipo de experimentos o verificando matemáticamente ciertas propiedades, y probando si el diseño presenta o no inconsistencias. Sin embargo, esta tarea puede llegar a tornarse tediosa en grandes proyectos, por lo cual es esencial contar con una herramienta que automáticamente controle la consistencia de el modelo en cuestión. En la próxima sección presentamos el lenguaje ALLOY que posibilita verificar automáticamente la consistencia de modelos.

3. El lenguaje ALLOY.

Alloy es un pequeño lenguaje que permite formalizar aspectos estructurales de un sistema de *software*, así como también describir ciertos cambios dinámicos producidos por la ejecución de ciertas operaciones. “Alloy es cómodo para un análisis semántico completo que puede proveer chequeo de consecuencias y consistencia, y ejecución simulada “[1]. Aunque el lenguaje esta inspirado en Z, un reconocido lenguaje algebraico, posee algunas características que lo hacen mas simple de entender por personas que no poseen conocimientos matemáticos.

Una descripción en Alloy se realiza por partes llamadas párrafos (*paragraphs*) que pueden clasificarse en cuatro tipos: signaturas (*signatures*), funciones (*fun*), restricciones (*facts*), y aserciones (*assertions*).

Las **signaturas** permiten definir tipos de manera modular y pueden poseer campos (*fields*) que definen relaciones con otras signaturas. Mediante las signaturas es posible definir incrementalmente nuevos tipos, por ejemplo:

```
Sig Persona{
        }
```

```
Sig Equipo{
        Integrantes:Set Persona
        Lider:Persona }
```

Define la noción de equipo como un conjunto de personas (del tipo básico Persona) que posee un líder. La expresión “e.lider” (con e un átomo de la signatura Equipo) debe interpretarse como la “navegación” a través de la relación líder que permite alcanzar un único átomo de tipo Persona, intuitivamente el líder del equipo.

Los párrafos **Fun** son formulas párametrizadas que pueden producir un cambio de estado, su nombre deriva de la palabra inglesa *function* debido a que mediante estas se pueden definir funciones matemáticas, aunque no necesariamente debe de ser así. Un pequeño ejemplo de una fórmula *fun* puede ser el siguiente:

```
Fun Destitucion(e:Equipo,e':Equipo,p:Persona){
    e'.lider = p
    e'.integrantes=e.integrantes
}
```

Donde el segundo argumento de la función puede verse –siguiendo la tradición de Z- como el resultado de la función; así mismo, el resultado de la función puede declararse implícitamente, por ejemplo:

```
Fun Destitucion(e:Equipo,e1:Equipo): Equipo{
  Result.lider = p
  Result.integrantes = e.integrantes
}
```

donde "result" es el resultado de la operación. Las formulas **Fact** involucran a las signaturas y sus campos, y son utilizadas para definir ciertas restricciones sobre la especificación realizada; en cambio, las formulas **Assert** sirven para expresar formulas que son tomadas como válidas (hipótesis), o bien como fórmulas que expresan propiedades sobre la especificación, que pueden usarse para chequear si un modelo dado cumple con las propiedades en ellas expresadas. Por ejemplo:

```
Fact {
      All e1,e2:equipo | e1 != e2 => e1.lider != e2.lider
     }
```

Establece la restricción que, en dos equipos diferentes, los lideres deben ser diferentes(dicho de otro modo, líder será una función inyectiva).

```
Assert {
 All e,e':Equipo | All p1,p2 | Destituir(e,e',p1) => e'.lider = p1
       }
```

Expresa que después de aplicar la función Destituir con parámetros "e:Equipo" y "p1:Persona", el líder del equipo resultante será p1.

Un aspecto relevante de Alloy es que provee herramientas para controlar automáticamente si las aserciones que conforman una determinada descripción se cumplen, es decir, es factible controlar automáticamente si un dado diseño cumple ciertas propiedades deseables. Mas aún, es posible chequear la consistencia de una especificación completa, esto es, verificar si existen modelos del sistema que se pretende desarrollar. Dado que Alloy es básicamente una extensión de la lógica de primer orden y por lo tanto no es decidible, es necesario imponer un alcance o *scope* en ingles, dado por la cantidad de átomos en el universo, para poder realizar el chequeo automático. Aunque esta restricción restringe la verificación a conjuntos finitos, en la práctica la clase de problemas tratables es de gran interés y amplitud . De esta manera es posible controlar que un diseño es consistente con *n* átomos y asegurar que en ese *scope* se cumplen ciertas propiedades. Alloy a sido testeado en casos prácticos con un *scope* del orden de 2^100 átomos.

4. La aplicabilidad de Alloy con contratos de reúso

En esta sección se presenta la traducción de los contratos de reuso al lenguaje Alloy y de esta manera se permite realizar un control automático sobre los diseños. A fin de no abundar en tecnicismos, la definición formal de la traducción se presenta en el anexo, en esta sección se dan dos ejemplos de traducción para ilustrar al lector como se traducen los contratos.

El primer ejemplo (Fig 2.) describe una cuenta bancaria muy simplificada. La cuenta esta compuesta por tres participantes: la cuenta en si misma, el titular y el banco, cada uno de ellos con los atributos obvios, y tres "relaciones de conocimiento" que representan las relaciones

fundamentales entre ellos, desde el punto de vista del contrato. Finalmente, un invariante describe una propiedad relevante de la situación, a saber, que el saldo de una cuenta nunca debe ser negativo.

Contrato CtaBancaria

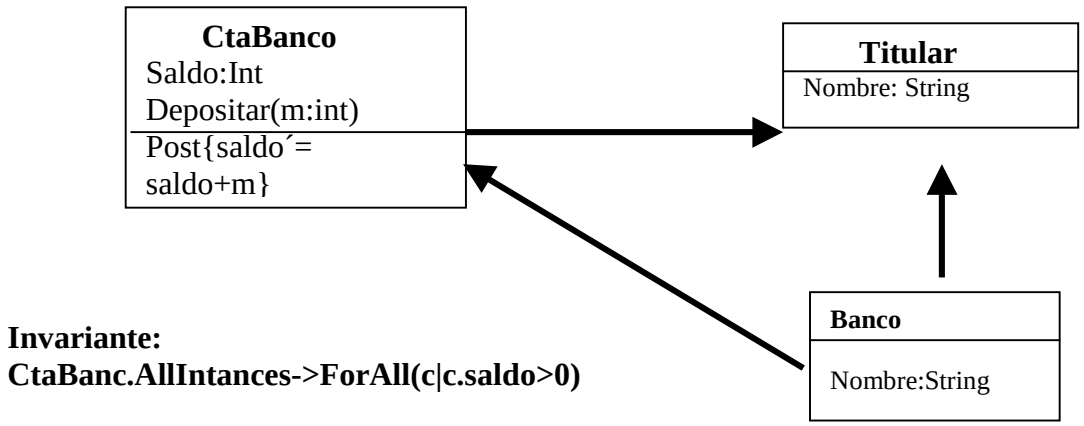


Fig. 2

Traducción a Alloy:

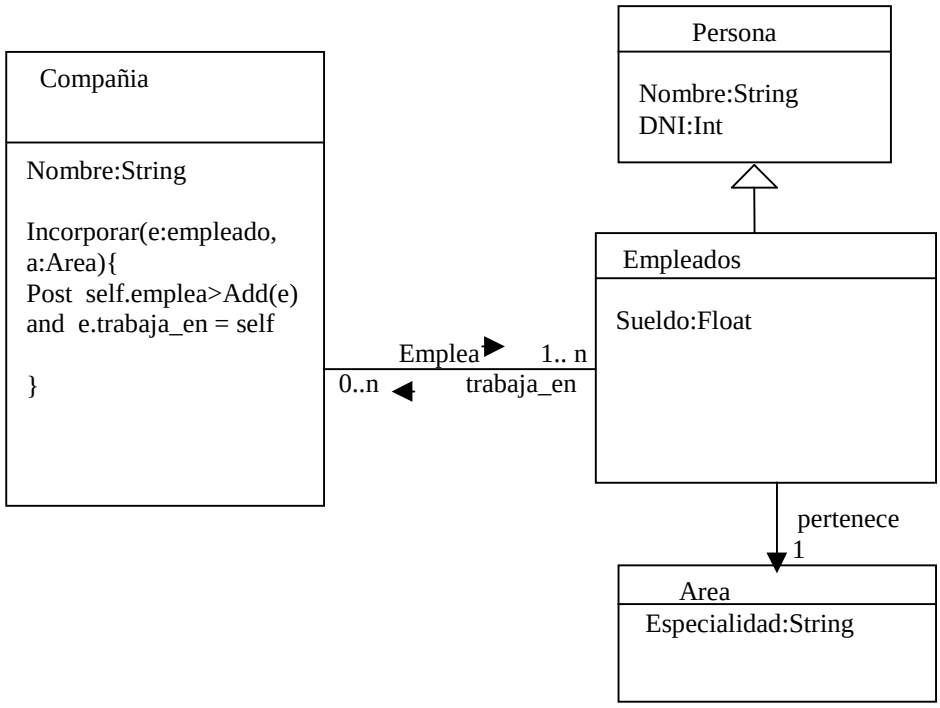
```
Sig Banco {
    Nombre: String
    Cuentas: Set CtaBanco
}

Sig Titular {
    Nombre: String
}

Sig CtaBanc {
    Saldo:Int
    Titul : Titular
}

fun Depositar (c:CtaBanc,c':CtaBanc,m:Int){
   c'.saldo = Suma(c.saldo,m)
}

Sig CtaBancaria{
  Participante1: Set CtaBanc
  Participante2: Set Titular
  Participante3: Set Banco
}

fun Inv(c:CtaBancaria.Participante1){ 0 in c.saldo.lte }

Assert DepositarPreservInv{
      All c,c' : CtaBancaria.Participante1 | all m:Int |
                        Inv(c) && Depositar(c,c',m)  => Inv(c') }
```

Observando el ejemplo anterior se puede realizar una interpretación intuitiva de la traducción: las relaciones entre los participantes son expresadas por campos en las signaturas, los métodos de las clases son traducidos como formulas *fun*, donde el primer parámetro es un átomo perteneciente a la signatura correspondiente a la clase, y el segundo es el estado posterior del mismo (o el resultado del método, si este lo tuviese). El contrato es expresado como una nueva signatura y el invariante por medio de una formula *fun* y una aserción(*assert*) que será utilizado al momento de la verificación.

El siguiente ejemplo (Fig.3) es una descripción de las relaciones que existen entre los componentes de una compañía, a saber, "Compañia", "Persona", "Empleado" y "Área". El participante

"Compañia" esta relacionado con sus empleados(participante "Empleado") mediante la relación emplea, el participante "Empleado" es una subclase de "Persona", además existe una relación de empleado a "Area" (relación pertenece) que relaciona cada empleado con el área en la cual trabaja. Los empleados también están relacionados con la empresas en las cuales trabajan por medio de la relación "trabaja_en". En el participante "Compañia" se define una operación cuya funcionalidad es agregar un empleado a la compañía, y finalmente un invariante que expresa que todos los empleados de una compañía deben trabajar para esta(notar que pueden trabajar para mas de una empresa),.

ContratoCompañia



INV:
Compania.allInstances->forAll(c1|c1.emplea.->forAll(e1|e1.trabaja_en.intersection(c1)->notEmpty())

Fig. 3

Traducción Alloy:

```
Sig Persona {
      Nombre:String
      DNI:Int
}
    Sig Area {
            Especilidad: String
    }
```

```
Sig Empleado Extend Persona{
        Sueldo: Float
        Trabaja_en : Set Compañia
        Pertenece: Area        }
          Sig Compañia{
            Nombre:String
             Emplea:+ Empleado
           }
```

```
 fun Incorporar (c,c':Compania,e,e':Empleado,a:Area){
   c'.emplea = c.emplea + e
   e'.trabaja_en = s;
   e'.dedicado_a = a
  }
```

```
Sig ContratoComp {
        Part1:Set Persona
        Part2:Set Empleado
        Part3:Set Compañia
        Part4:Set Area
}
```

```
fun INV1 (c:Compania){
  all e: c.emplea | c in e.trabaja_en
 }

assert IncorporarPresINV{
    all e:ContratoComp.Part3 | all e:contratoComp.Part2 | all a:ContratoComp.Part4 | INV(c) &&
    Incorporar(c,c'e,e',a) => Inv1(c') }
```

El contrato anterior fue chequeado con la herramienta de Alloy y se obtuvieron los siguientes resultados que describen una estructura de primer orden, compuesta por tres dominios y tres relaciones, correspondiendo directamente a los elementos definidos en el contrato.

```
Instance found:
Domains:
  Area = {A0,A1,A2}
  Compania = {C0,C1,C2}
  Empleado = {E1,E2,E3}
Relations:
  pertenece = {E1 -> {A2}, E2 -> {A1}, E3 -> {A0}}
  emplea = {C0 -> {E3}, C1 -> {E2}, C2 -> {E1}}
  trabaja_en = {E1 -> {C2}, E2 -> {C1}, E3 -> {C0}}
```

Los resultados describen una instancia de la especificación que permite asegurar que el diseño es consistente, es decir, posee un modelo. Además se verificó que la operación Incorporar preserva el invariante, en este caso la herramienta no pudo encontrar contraejemplos en un alcance(*scope*) de 3 átomos para todos los dominios.

Una posible evolución del anterior contrato consiste en la definición de una nueva operación en el participante Compañía, la operación esta definida por:

```
Renuncia(e:empleado){
   Post : e.trabaja_en' = e.trabaja_en – self
}
```

Intuitivamente, se la puede interpretar como una operación que desliga un empleado de una empresa; la traducción al lenguaje Alloy de la evolución realizada viene dada por la extensión de la traducción anterior, más el siguiente texto :

```
 fun Renunciar(c,c':Compania,e,e':Empleado){
   e'.trabaja_en = e.trabaja_en - c
   c'.emplea = c.emplea
  }

 assert RenunciarPresInv {
 all e:ContratoComp.Part3 | all e:contratoComp.Part2 | INV(c) && Renunciar(c,c'e,e') => Inv1(c')
```

}

Donde el ultimo párrafo es la aserción que debe probarse para asegurar que la nueva operación preserva el invariante. Sin embargo, es fácil comprobar que esto no sucede pues la operación "Renunciar" no actualiza la relación "c'.emplea"; si se verifica este nuevo ejemplo, Alloy construye un contraejemplo que muestra que "Renunciar" rompe con lo establecido en el invariante. Para comprender el contraejemplo, debe notarse que el efecto de la "ejecución" de Renunciar modifica las relaciones pertenece y emplea, indicadas como "pertenece" y "emplea".

Counterexample found:
Domains:
Area = {A1,A2}
Compania = {C1,C2}
Empleado = {E3,E4}
Relations:
pertenece = {E3 -> {A2}, E4 -> {A1}}
emplea = {C1 -> {E4}, C2 -> {E3}}
trabaja_en = {E3 -> {C2}, E4 -> {C1}}
pertenece' = {E3 -> {A2}, E4 -> {A1}}
emplea' = {C1 -> {E4}, C2 -> {E3}}
trabaja_en' = {E3 -> {C2}}
Parameters:
c = {C1}
e = {E4}

En la instancia anterior se puede observar que el empleado E4, después de la ejecución de la operación, no trabaja para ninguna empresa (observar la relación "trabaja_en") pero sin embargo la empresa C1 lo tiene como empleado(observar la relación "emplea"), lo que contradice el invariante.

La segunda evolución del diseño original consiste en agregar la operación Renunciar, pero correctamente definida para evitar los problemas con el invariante, la definición esta dada por:

Renunciar(e:empleado)
Post : (e.trabaja_en' = e.trabaja_en – self) and (self.emplea' = self.emplea- e)
}
Cuya traducción viene dada por :

Fun Renunciar(c,c':Compania,e,e':Empleado) {
e'.trabaja_en = e.trabaja_en - c
c'.emplea = c'.emplea – e}

Utilizando nuevamente la herramienta de Alloy para chequear que la nueva operación preserva el invariante, el resultado es: "No counterexample found" (no se encontraron contraejemplos) por lo que se puede asegurar que el invariante se preserva para un *scope* de 3 atomos.

5. Conclusiones

La técnica de verificación de evoluciones presentada, si bien en un estado embrionario, permite razonar rigurosamente sobre contratos de reuso y diseños de software orientado a objetos expresados en UML. El mérito fundamental del trabajo consiste en haber definido una traducción simple, modular y eficiente de dichos diseños gráficos y semi-formales a un lenguaje formal de

primer orden, que posibilita el uso de una herramienta automática, en este caso Alloy. La modularidad de la traducción, como es posible observar en los ejemplos, posibilita verificar de manera incremental las sucesivas evoluciones, es decir, permite al desarrollador detectar automáticamente inconsistencias que pueden introducir los cambios en sus diseños, con lo cual se obtiene una manera rápida y segura de ir verificando las diferentes evoluciones de un modelo durante el desarrollo de un sistema de software.

6. Trabajos Futuros.

Las ideas y resultados presentados en las secciones previas permiten resolver dos problemas básicos: una traducción consistente, razonable y eficiente de contratos expresados en UML a Alloy, y la verificación automática de consistencia y otras propiedades de los modelos. Sin embargo, aún queda pendientes dos cuestiones de la mayor utilidad e importancia: por una parte, la captura del proceso de evolución en si mismo y la verificación de su corrección; por otro lado, la definición de una metodología que permita, siguiendo cierto conjunto de reglas o esquemas, orientar el proceso de evolución de modo que se asegure la preservación de las propiedades fundamentales de los diseños o contratos. A partir de estas ideas aparecen dos líneas de trabajos futuros:

- Expresar formalmente evoluciones de contratos. En este punto no es evidente que el poder expresivo de Alloy sea suficiente para expresar el proceso de evolución; ciertamente Alloy es básicamente un lenguaje de primer orden y, en el estado actual del trabajo, la evolución aparece como una transformación de segundo orden. Sin embargo, aún resta estudiar que partes del proceso pueden expresarse en el lenguaje y cuales no. Por otro lado, dada la sencillez y la modularidad de la traducción definida, parece factible definir una técnica de verificación modular que tome en cuenta las componentes que sufren cambios en cada paso de evolución y verifique solamente las partes necesarias de los contratos. Esta aproximación al problema, si bien no captura el proceso de evolución en un formalismo, representa una enorme ganancia en eficiencia en tanto “hace solamente el trabajo necesario” y, por otra parte, toma ventaja del carácter también modular de los operadores de reuso definidos en [4].

- Definir un calculo de refinamientos sobre los modelos de diseño. Un calculo de refinamientos es básicamente un conjunto de reglas definidas sobre un lenguaje que permiten, dada una especificación aplicar un conjunto de reglas o heurísticas formalmente definidas, hasta obtener otra especificación que preserva ciertas propiedades fundamentales de la primera, brevemente se puede expresar del siguiente modo:

  Sean C1,C2 contratos, C1 refina C2 <=> Sem(C1) $\subseteq$ Sem(C2), donde Sem es la función semántica de los contratos.

  La idea básica es definir un conjunto de reglas de transformación sobre el metalenguaje de los modelos conjuntamente con la definición de una relación de refinamiento(puede ser la anterior pero no necesariamente), y de esta manera obtener una metodología que nos permita partir de un diseño y obtener otro mediante la utilización de reglas que garantizan la consistencia. Esta línea de trabajo esta inspirada en el trabajo de Ralph Back [5], que define un calculo de refinamientos sobre especificaciones de programas.

Bibliografia

[1]Daniel Jackson, A micromodularity mechanism, Available at http//: sdg.lcs.mit.edu/dng/publications.

[2] Daniel Jackson, Alloy: A Lightweight Object Modelling Notation. To appear in ACM Transaction on Software Engineering and Methodlogy.Available at http//: sdg.lcs.mit.edu/dng/publications.

[3] Daniel Jackson, Automating First-Order Relational Logic, ACM SIGSOFT Proc. Conf. Foundations of Software Engineering, San Diego, Novemeber 2000.

[4] Roxana S.Giandini, Documentación y evolución de componentes reusables. Tesis de Magister en Ingeniería de Software, Facultad de Ciencias Exactas, UNLP, 1999.

[5] Ralph Back and Joakim von Wright Refinement Calculus, Springer, 1997

[6] Warmer-Kleppe, The Object Constraint Lenguage, 1998

[7] Jacobson, I, Booch, G, Rumbaugh, J. The Unified Software Development Process, Addison Wesley. ISBN 0-201-57169-2 (1999)

[8] Breu,R., Hinkel,U., Hofmann,C., Klein,C., Paech,B., Rumpe,B. y Thurner,V., Towards a formalization of the unified modeling language. ECOOP'97 procs., Lecture Notes in Computer Science vol.1241, Springer (1997).

[9] Evans,A., France,R., Lano,K. y Rumpe, B., Developing the UML as a formal modeling notation, UML'98 Beyond the notation, Muller and Bezivin editors, Lecture Notes in Computer Science 1618, Springer-Verlag (1998).

[10] Evans,A., France,R., Lano,K. y Rumpe,B., Towards a core metamodelling semantics of UML, Behavioral specifications of businesses and systems, H,Kilov editor, Kluwer Academic Publishers (1999).

[11] France, R. y Rumpe, B. editors, Proceedings of the UML´99 conference, Beyond the Standar, Colorado, USA, Lecture Notes in Computer Science 1723, Springer-Verlag (1999).

[12] Goldsack,S. y Kent,S., Formal Methods and Object Technology", Chapter 3: LOTOS in the Object-oriented analysis process. Editors S.J. Goldsack, S.J.H. Kent. Serie FACIT, Springer-Verlag (1996).

[13]Helm, R. Y Holland, I. Contracts: specifying behavioral composition in object-oriented systems, proceedings of OOPSLA´90 (1990).

[14]Jungclaus,R., Saake,G., Hartmann,T., Sernadas,C., TROLL- a language for o-o specifications of information systems, ACM Transactions on IS, vol.14 no.2. (1996).

[15] Kim, S. y Carrington,D., Formalizing the UML Class Diagrams using Object-Z, proceedings UML´99 Conference, Lecture Notes in Computer Sciencie 1723 (1999).

[16]Lano,K., y Biccaregui,J., Formalizing the UML in Structured Temporal Theories, Second ECOOP Workshop on Precise Behavioral Semantics, TUM-I9813, Technische Universitat Munchen (1998).

[17] Muller. P. y Bezivin. J. editors, Proceedings of the UML´98 conference, Beyond the notation, Mulhouse, France, Lecture Notes in Computer Science 1618, Springer-Verlag (1998).

[18] Pons,C., Baum,G., Felder,M., Foundations of Object-oriented modeling notations in a dynamic logic framework, Fundamentals of Information Systems, Chapter 1, T.Polle,T.Ripke,K.Schewe Editors, Kluwer Academic Publisher (1999).

[19] Pons,C., Tesis de doctorado, Facultad de Ciencias Exactas, Universidad Nacional de La Plata, Argentina (1999).

Anexo

En esta sección se definen los aspectos fundamentales de la función de traducción de contratos de reúso a Alloy. La definición está expresada en un lenguaje funcional –similar a Haskell-. En primer lugar se describen los dominios de los lenguajes como tipos de datos y posteriormente se definirán funciones que traducen elementos de un dominio al otro.

1. Definición del dominio de los contratos de reúso con semántica de comportamiento

Contrato = [Part] × [Con] × [Interf] × Inv

Part = String , dominio de los nombres de los participantes

Interf = Part × Op × [Param] × Res × Cespec × Cond

Op = String, dominio de los nombres de operaciones
Param = Nombre ':' Type | σ, donde σ significa el elemento vacío de los parámetros; aquí hace falta un digresión: el parámetro [] significa que el elemento es un atributo, en cambio el parámetro [σ] indica que el método carece de argumentos.

Res = Type Type = String, dominio de los tipos

Cespec = Invoc '||' Cespec | Invoc ; Cespec | Invoc

Invoc = String, es la invocación de un método.

Cuerpo = ExpOcl, donde ExpOcl es un término que pertenece al lenguaje definido por la gramática de OCL [6].

El dominio del lenguaje Alloy esta definido por:

AlloyExp = Los términos pertenecientes al lenguaje de expresiones de Alloy [2].

2. La función de traducción

Trad : Contratos × Nombre→ Alloy
Trad (P,C,I.Inv) = if P = [] then ([],[],[],[])
else
(GetSig(P,C,I):GetContracSig(P),GetFunc(I)+Π1(TransInv(Inv)),Π2(TransInv(Inv)),[])

donde Πn es la n-ésima proyección y GetContractSig retorna la signatura del contrato

GetSig: [Part] → [Con] → [Interf] → [Sig]
GetSig [ ] C I = []
GetSig P C I = if (first(P),GetField(C,P)) : GetSig(P-first(P),C,I)

Esta función obtiene la signaturas de los Participantes teniendo en cuenta las relaciones de conocimientos e interfaces.

Las demás funciones utilizadas se pueden definir de una forma parecida, se pueden destacar dos funciones,

TransAlloy: OclExp → Alloy Exp, que transforma las expresiones Ocl a expresiones Alloy,

TransInv : Inv → ([Fun],[Assert]), que dado un invariante nos devuelve la expresión análoga en Alloy, esta función devuelve una tupla debido a que los invariantes en Alloy se expresan con funciones y se controla su validez con aserciones.