E-Machines

Pablo Castro* Gabriel Baum**

Resumen

Durante el proceso de desarrollo de un sistema de software es común que se utilicen diseños reusables, los cuales debido al carácter iterativo e incremental de este proceso son modificados recurrentemente. Tales modificaciones (llamadas "evoluciones" por cierta comunidad de Ingeniería de Software) pueden tener efectos no deseados en los diseños. En este trabajo presentamos un formalismo llamado E-Machines que permite analizar el proceso de evolución de una manera rigurosa.

Palabras clave: Diseños Reusables, Ingeniería de Software, Métodos Formales, Evolución de Diseño.

1. Introducción

El diseño de un sistema de software cambia constantemente en el tiempo, ya sea por la metodología de desarrollo (desarrollo iterativo e incremental) o bien debido a los ajustes y adaptación del sistema a requerimientos cambiantes. Esta situación se ve agravada aun más con la utilización de conceptos de "orientación a objetos" que ponen fuerte énfasis en la reutilización de partes y en la modularidad.

Estos cambios sobre los diseños (a veces llamados "evoluciones" [16][17]) pueden afectar de forma negativa en la calidad del producto final. En este trabajo se propone abordar este problema utilizando métodos formales y herramientas basadas en ellos que posibiliten control riguroso sobre las evoluciones.

En [17] se formulan las siguientes preguntas :

- ¿ Cuáles aspectos de la evolución del software deben ser automatizados mediante herramientas?
- ¿Dónde y de qué forma los formalismos pueden ayudar?
- ¿Cómo podemos construir herramientas con fundamentos formales que sean lo más generales y flexibles posibles?

En el presente trabajo se intenta responder parcialmente alguna de estas cuestiones en el contexto de las evoluciones que se producen en los diseños reusables de los sistemas, y para lograr tal propósito se presenta un formalismo llamado *E-Machines* que se utilizará como fundamento en la descripción de una herramienta semi-automática. En la sección 2 se introduce

*Departamento de Computación, Universidad Nacional de Río Cuarto. e-mail:pcastro@dc.exa.unrc.edu.ar
**Laboratorio de Investigación y Formación en Informática Avanzada, Universidad Nacional De La Plata. e-mail:gbaum@sol.info.unlp.edu.ar

el concepto de *E-Machines*, en la sección 3 se discute la utilización de la teoría en la práctica, en las últimas secciones se describen los posibles usos de dichos conceptos así como posibles trabajos futuros.

2. E-Machines

En esta sección se presenta una breve introducción a los conceptos lógicos necesarios para luego abordar la definición del formalismo que es objeto del presente artículo; finalmente se ilustran los conceptos descritos con algunos ejemplos.

2.1. Fundamentos Lógicos

El formalismo básico que se utilizará a lo largo de la presentación es la lógica de primer orden con sorts (*many sorted logic*, para una presentación más detallada ver [10]) Los conceptos necesarios de dichas lógicas para este trabajo son los siguientes:

- Un lenguaje L con sorts (*many sorted language*) consiste de un alfabeto de símbolos, un conjunto de sorts y una declaración asignando a cada símbolo sus argumentos y resultados.
- Una teoría con sorts (solo teoría de aquí en adelante) consiste de un lenguaje con sorts junto con un conjunto de sentencias (sus axiomas).
- Una L-estructura para un lenguaje L es una estructura con una aplicación asignando a cada sort un conjunto, a cada símbolo funcional una función que respeta su perfil y a cada símbolo relacional una relación que respeta su perfil.
- Dada una teoría T=(L,A), una L-Estructura $\mathfrak{U}$ es modelo de T si y solo si $\mathfrak{U} \models T$, esto es, si todas las sentencias de A se cumplen en $\mathfrak{U}$.
- Dados dos lenguajes L_1 y L_2, se dice que $L_1 \subseteq L_2$ (L_2 extiende a L_1) si y solo si L_2 puede obtenerse agregándole símbolos extra-lógicos o sorts a L_1.
- Dadas dos teorías $T_1 = (L_1, A_1)$ y $T_2 = (L_2, A_2)$, se dice que $T_1 \subseteq T_2$ (T_2 extiende a T_1) si y solo si L_2 extiende a L_1 y cualquier consecuencia de A_1 es consecuencia de A_2.
- Dadas dos teorías T_1 y T_2 se dice que $T_1 \leq T_2$ (T_2 es una extension conservativa de T_1) si y solo si T_2 extiende a T_1 y además toda sentencia sobre L_1 que es consecuencia de A_2 es también consecuencia de A_1.

Los conceptos sobre modelos y teorías pueden ser consultados en [2].

2.2. Definición de E-Machines

Una E-machine es, intuitivamente, un mecanismo abstracto que pretende capturar procesos de cambio a nivel sintáctico y semántico de diversos artefactos involucrados en el proceso de desarrollo de software, sean estos especificaciones, diseños, programas, etc. En este sentido las E-machines están estrechamente ligadas a las *Abstract State Machines*, introducidas por Gurevich y sus colaboradores ([6],[7],[8]), aún cuando las E-machines han sido motivadas por el análisis de la evolución de los diseños, en tanto las ASMs provienen del campo de la

computabilidad. De todos modos, ambos formalismos comparten similares bases filosóficas y lógico-matemáticas, en el sentido que retoman las ideas fundamentales de Turing en cuanto a la concepción del proceso de computación como un proceso de pequeños cambios elementales (o atómicos) y las ideas de Tarski en cuanto a la semántica de las teorías lógicas.

En las siguientes definiciones se considerarán solo teorías con un conjunto finito de axiomas. Sean los siguientes conjuntos no vacíos de operaciones sobre teorías y estructuras:

- Ω_L, un conjunto de operaciones atómicas sobre teorías que producen cambios sobre sus lenguajes.
- Ω_A, un conjunto de operaciones atómicas sobre teorías que realizan cambios sobre sus axiomas.
- $\Omega_{\mathfrak{U}}$, un conjunto de operaciones atómicas que realizan cambios sobre estructuras. Estas operaciones son(en el sentido clásico) "hasta isomorfismo", es decir, no "distinguen" estructuras isomorfas.

No se discutirá en detalle sobre el contenido de estos conjuntos para no sobrecargar de notación al lector. En la sección 3.6 se presentan ejemplos de dichos operadores, solamente se reitera que deben considerarse aquellos operadores que permiten realizar modificaciones básicas. Sobre estos conjuntos se definen, de la manera usual, el conjunto $(\Omega_L \cup \Omega_U)^*$ de las secuencias de operaciones que pertenecen a Ω_L ó Ω_A, y análogamente, ${\Omega_{\mathfrak{U}}}^*$.

Definición 2.1. *Una **evolución atómica** es un elemento de $(\Omega_L \cup \Omega_A)$ tal que tiene asociado un elemento de ${\Omega_{\mathfrak{U}}}^*$. Si e es una evolución atómica , $e^{\mathfrak{U}}$ denota un elemento asociado con e.*

Definición 2.2. *Una **evolución** es una secuencia de evoluciones atómicas*

Naturalmente, se puede identificar una evolución con la secuencia de evoluciones atómicas que la conforman, por ejemplo $e_1; e_2; ...; e_n$.

Definición 2.3. *Un estado es un par (T,$\mathfrak{U}$), donde T es una teoría y $\mathfrak{U}$ es un modelo de T ($\mathfrak{U} \models T$).*

Dado un estado S se denota con S^T al primer componente del estado, y con $S^{\mathfrak{U}}$ al segundo componente.

Definición 2.4. *Una **aplicación** de una evolución atómica **e** a un estado (T,$\mathfrak{U}$), es el par $(e(T), e^{\mathfrak{U}}(\mathfrak{U})))$. Por otra parte, e(s) denota la aplicación de la evolución e al estado s.*

Debe notarse que la aplicación de una evolución atómica en un estado puede no ser un estado.

Definición 2.5. *Una **E-Machine** sobre una teoría T consistente viene dada por:*

- *Una evolución $e_1; e_2; ...; e_n$, con $n \in \mathbb{N}$.*
- *Un conjunto de estados $\{S_i\}$ con i = 0,...,n.*

Donde los estados cumplen las siguientes condiciones:

- $\forall 1 \leq i \leq n : S_i = e_i(S_{i-1})$, *donde e(X) es la aplicación de e al estado X.*
- $\forall 0 \leq i \leq n : T \subseteq {S_i}^T$, *es decir todos los estados son extensiones de T.*

S_0 es llamado estado inicial y S_n estado final.

Es interesante volver ahora sobre la idea intuitiva expresada al comienzo de esta sección. En términos de las definiciones previas, una e-machine es una máquina abstracta cuyos estados están conformados por teorías y modelos de estas teorías, y cuya función de transición es un conjunto de modificaciones sobre las teorías y sus modelos, donde cada modificación realiza cambios sobre ambos componentes de algún estado.

Es interesante notar algunos rasgos inmediatamente observables de estas máquinas abstractas:

- En cada paso de una evolución se modifica o bien el lenguaje de una teoría o bien sus axiomas, pero nunca ambos.
- En cualquier estado S de una E-Machine sobre un teoría T se cumple que el primer componente de S (es decir una teoría) es una extensión consistente de T. Es razonable pensar a T como un invariante.

En las próximas secciones se discute la importancia de estas propiedades.

3. La aplicación de las E-machines

En esta sección se discute la factibilidad de utilizar las E-machines como soporte teórico para estudiar y verificar evolución de diseños; naturalmente, la noción de diseño y mas aún la de evolución de diseños son básicamente informales. Por este motivo el objetivo de los párrafos que siguen solamente buscan dar al lector argumentos convincentes acerca de la razonabilidad de construir una herramienta (o un conjunto de herramientas) sobre la base de estas máquinas abstractas. A tal fin, se discuten algunas ideas que apuntan a formalizar la noción de reuso y evolución de diseños, se propone la utilización de una herramienta de verificación (Alloy) y se presentan algunos ejemplos.

Un diseño de software puede definirse como una especificación acerca de como un sistema a desarrollar satisfará ciertos requisitos deseados, particularmente en un diseño orientado a objetos hay un énfasis sobre la definición de los objetos lógicos que luego serán implementados por algún lenguaje de programación. Es decir, de una manera indirecta, durante la etapa de diseño de un sistema queda definido el "aspecto" que van a tener los estados del sistema (por estados entenderemos la configuración de los objetos en un tiempo dado). Cuando además se utilizan lenguajes como UML (*Unified Modeling Language* [9][11]) este "aspecto" queda expresado en términos de diversos diagramas: el de clases, para definir la estructura lógica del diseño, el diagrama de objetos para expresar instancias de los diagramas de clases y otros para explicar la interacción entre objetos. Sin mucho esfuerzo es bastante claro que un diagrama de clases define un lenguaje con sorts como el presentado en la Sección 2.1 (asociando sorts con clases y relaciones entre estos), a veces estos diagramas tienen restricciones que pueden representarse como axiomas, y mas aún, los llamados "diagramas de objetos" corresponden de manera bastante directa a la noción de modelo de una teoría. Resumiendo, la idea de conectar los diseños informales de un sistema de software con las ideas matemáticas surge de una manera natural. En las próximas secciones se analiza muy brevemente una traducción de diseños reusables a teorías de un lenguaje de primer orden llamado Alloy, y finalmente la evolución de estas teorías en términos de E-machines.

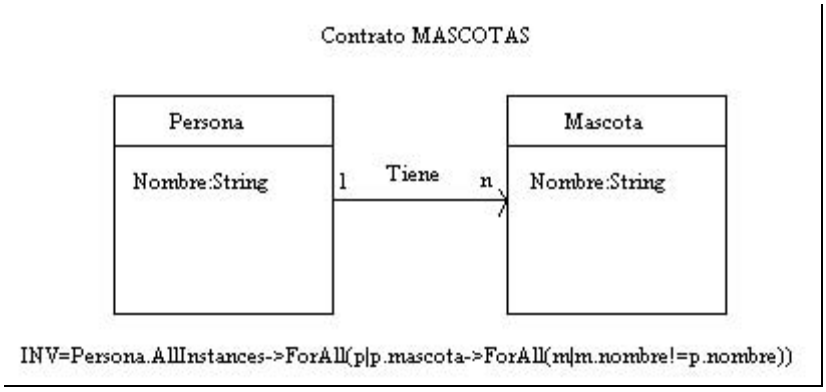

Figura 1: un ejemplo de contrato.

3.1. Contratos de reuso con semántica de comportamiento

Los contratos de reuso fueron creados para documentar sistemáticamente las partes reusables de un sistema [15], es decir aquellas partes del sistema que serán modificadas y reutilizadas posteriormente durante el desarrollo; los contratos de reuso con semántica de comportamiento [3][4][5], están conformados por participantes relacionados que poseen métodos y restricciones. Es importante resaltar que estos contratos pueden ser expresados en UML y sus restricciones mediante OCL (*Object Constraint Language* [18]).

Para describir los cambios que se pueden introducir en un contrato existen los denominados *operadores de reuso*, que permiten documentar las modificaciones realizadas. En la Figura 1 se muestra un pequeño ejemplo de contrato de reuso, que modela la relación entre una persona y sus mascotas, e impone la restricción de que una persona no puede compartir su nombre con alguna de sus mascotas.

3.2. Alloy

Alloy [12][13][14] es un lenguaje de primer orden extendido con clausura reflexo-transitiva sobre relaciones. y provee un herramienta para realizar verificación automática sobre especificaciones utilizando la técnica conocida como *model finding* [13]. Debido a los problemas de decibilidad de la lógica de primer orden los chequeos se realizan sobre un *scope* que limita la cantidad de átomos en el universo. Una especificación Alloy esta dividida en párrafos, lo cual favorece notoriamente su modularidad y claridad.

En la Figura 2 se muestra la gramática, tipos y semántica del kernel de Alloy. Un ejemplo de una especificación Alloy correspondiente al contrato de la figura 1 es la siguiente:

```
Sig Persona{nombre: Nombre, mascotas:set Mascota}
Sig Mascota{nombre:Nombre}
Fact{all p: Persona | no p.nombre & p.mascota.nombre}
```

Es interesante observar como se definen los tipos (Persona, Mascota) y como se impone la restricción de que no existen personas que comparten el nombre con alguna de sus mascotas. También es bastante directa la analogía con el ejemplo de la figura 1: hay un tipo por cada clase, y el invariante en el contrato expresa exactamente lo mismo que el *fact* de la especificación. En la siguiente sección se discute esta relación de manera más general.

problem::=decl*formula decl::=var:typexpr typexpr::= type | type→type | type⇒typexpr formula ::= expr in expr (subconj.) |!formula (neg.) |formula && formula (conj.) |formula || formula (disj.) |all v:type|formula(univ.) |some v:type|formula(exist.) expr ::= expr + expr (union) | expr & expr (inter.) | expr - expr (difer.) | ẽxpr (transp.) | +expr (clausura) | {v:t|formula} (def.conj.) | var Var::= var(variable) | Var[var](aplicacion)	$\frac{E\vdash a:S, E\vdash b:S}{E\vdash a\,in\,b}$ $\frac{E,v:T\vdash F}{E\vdash all\,v:T\,\vert\,F}$ $\frac{E\vdash a:S\to T, E\vdash b:S\to T}{E\vdash a+b:S\to T}$ $\frac{E\vdash a:S\to T, E\vdash b:S\to U}{E\vdash a.b:U\to T}$ $\frac{E\vdash a:S\to T}{E\vdash\ a:T\to S}$ $\frac{E\vdash a:T\to T}{E\vdash +a:T\to T}$ $\frac{E,v:T\vdash F}{E\vdash a[v]:t}$ $\frac{E\vdash a:T\Rightarrow t, E\vdash v:T}{E\vdash a[v]:t}$	M:formula→ env → Boolean X:expr → env → value env = (var+type)→value value = P(atom × atom) + (atom → value) M[a in b]=X[a]e⊆X[b]e M[!F]e=¬M[F]e M[F&&G]e= M[F]e ∧ M[g]e M[F||G]e=M[F]e∨M[G]e M[all c:T|F]= ⋀ { M[F](e⊕ v ↦ x)|(x,unit)∈ e(t)} M[some v:T|F]= ⋁ { M[F](e⊕ v → x) | (x,unit) ∈ e(t)) X[a+b]=X[a]e∪X[b]e X[a & b]e=X[a]e ∩ X[b]e X[a-b]e=X[a]e \ X[b]e X[a]e={ (x,y) | (y,x) ∈ X[a]e } X[+a]e= el r mas chico tal que r;r⊆ r ∧ X[a]e ⊆ r X[{ v:t|F}]e= {(x,unit)∈ e(t) | M[F](e⊕v → x) X[v]e=e(v) X[a[v]]e=(e(a))(e(v))

Figura 2: gramática, tipos y semántica del kernel de Alloy.

3.3. Interpretando los contratos de reuso en Alloy

En [1] se define una traducción de los contratos de reuso con semántica de comportamiento a Alloy. Por esta vía, queda definida implícitamente una semántica formal para los mencionados contratos; es decir, si τ:*Contratos*→ *Alloy* es la traducción, y σ es la función de interpretación para Alloy, entonces la función $\sigma \circ \tau$ provee una semántica para los contratos de reuso. De esta manera, sobre la base de aceptar la interpretación de los contratos (expresados con diagramas de UML y restricciones en OCL) como teorías Alloy, es factible su verificación automática. Es importante observar que la función τ puede extenderse fácilmente para traducir, además de contratos de reuso, diagramas de objetos que sean compatibles con estos contratos. Esto es factible debido a que en una especificación Alloy puede introducirse (en forma de *facts* y extensiones de signaturas) un determinado modelo de la especificación.

Este hecho está en la base de la propuesta para expresar las evoluciones.

3.4. La idea detrás de las e-machines

Una evolución sobre un contrato de reuso puede imaginarse como la aplicación reiterada de operadores de evolución (ver por ejemplo [3]) que producen ciertos cambios sobre los contratos originales. Estos cambios pueden producir diversos efectos sobre los diseños y algunos de estos pueden introducir inconsistencias. Esta es, sin dudas, la primer obligación de prueba a la que debe someterse un diseñador en el contexto de un proceso de desarrollo evolutivo. Otra petición interesante consiste en exigir que durante las evoluciones se conserven ciertos rasgos y propiedades del contrato original (se pueden ver a estas propiedades como los invariantes del proceso de evolución). Esta idea es exactamente la expresada por las *e-machines*. La relación entre la dos visiones (la formal y la informal) viene dada por la traducción de los contratos de reuso a Alloy; el contrato inicial junto con una instancia (un diagrama de objetos consistente

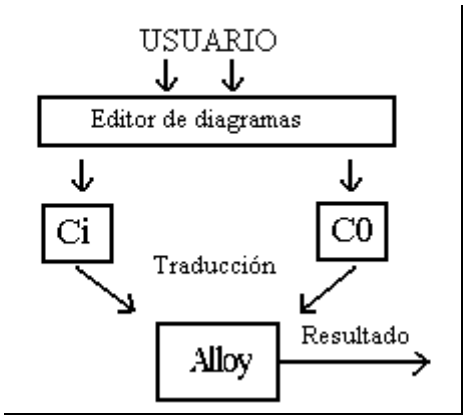

Figura 3: esquema para el chequeo de evoluciones

con las clases participantes del contrato) pueden verse como el estado inicial; los operadores de evolución sobre los contratos y las modificaciones sobre los diagrama de objetos definen las operaciones sobre los estados y los rasgos que se mantienen fijos durante el proceso de evolución pueden verse como la teoría inicial (el invariante).

En resumen, la noción de E-machine, y su evidente factibilidad práctica sobre la base de la disponibilidad de Alloy, captura todos los elementos necesarios para sentar las bases de una herramienta semi-automática que verifique el proceso de evolución

3.5. Una verificación automática sobre evoluciones

La posibilidad de construir una herramienta basada en el concepto de E-machine está íntimamente relacionada con la idea de que el proceso de evolución está compuesto de pequeños cambios que afectan a los distintos aspectos de una especificación. Estos cambios corresponden a transiciones de una máquina abstracta y deben poder expresarse como "pasos" en un proceso. El primer requisito para la posibilidad de existencia de la herramienta está expresado en el siguiente corolario de la definición de extensión de una teoría, introducido en la sección 2:

> *Dadas dos teorías* $T_1 = (L_1, A_1)$ *y* $T_2 = (L_2, A_2)$ *tal que* $L_1 \subseteq L_2$ *ocurre* $T_1 \subseteq T_2$ *si y solo si para cualquier* $a \in A_1 : A_2 \models a$

Este resultado autoriza a verificar que dos especificaciones Alloy E_1 y E_2 cumplen $E_1 \subseteq E_2$ (notar que las especificaciones en Alloy son teorías en el sentido de 2.1) de la siguiente manera:

- El lenguaje de E_1 es un sublenguaje del lenguaje de E_2.
- Los *facts* de E_1 son teoremas en E_2.

Sobre esta base, una herramienta de verificación automática sobre evoluciones solo tendría que traducir los diseños reusables e ir chequeando mediante Alloy los requisitos que se deben cumplir para que una evolución resulte ser *e-machine*, es decir:

- El actual contrato es consistente.
- Se cumple $\tau(C_0) \subseteq \tau(C_i)$, siendo C_0 el invariante y C_i el contrato resultado de un paso de evolución.

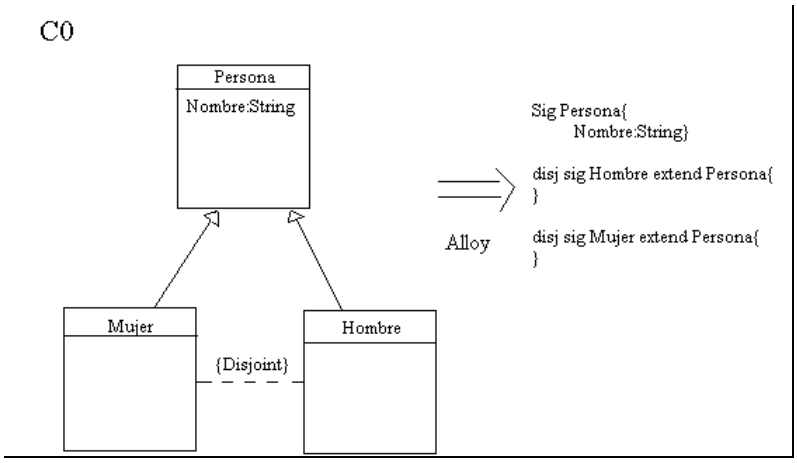

Figura 4: El contrato invariante y su traducción a Alloy.

En la figura 3 se puede observar ilustrado el proceso de verificación.

Claramente, la herramienta propuesta es sencilla, y puede interactuar con alguna herramienta gráfica de edición de diagramas, de manera que el proceso de verificación de una evolución se realice de una manera simple y sin complicaciones técnicas.

3.6. Un ejemplo

En esta sección se presenta un ejemplo que muestra la utilización práctica de los conceptos descritos anteriormente. En la figura 4 se puede observar un contrato C_0 que será el invariante de la evolución. La figura 5 muestra el contrato C_1 con su correspondiente instancia, es decir, el estado inicial de la evolución. Las siguientes podrían considerarse operaciones atómicas sobre contratos:

- C[I'], agrega como conjunción I' en el invariante de C.
- NewObject(M,O), agrega el nuevo objeto O en la instancia M.

Estos dos operadores definen los siguientes operadores sobre teorías:

- T:=A, agrega el axioma A en la teoría T.
- Add(M,e,C), agrega el nuevo elemento e de tipo C al modelo M.
- skip(M), no realiza cambios sobre el modelo.

Si bien se pueden agregar y definir muchas más operaciones, estas serán suficientes para el ejemplo. Mediante la función de traducción τ obtenemos:

- $C_0 \mapsto E_0$, siendo E_0 la especificación obtenida de C_0 por medio de τ.
- $(C_1, O_1) \mapsto E_1$, siendo E_1 la especificación Alloy que contiene la teoría obtenida de C_1 y el modelo obtenido de O_1.

Las especificación E_0 y E_1 fueron chequeadas exitosamente con Alloy. La siguiente modificación fortalece el invariante de C_1:

$$C_1[Hombre.AllInstances \rightarrow ForAll(h|h.esposa \rightarrow noEmpty())],$$

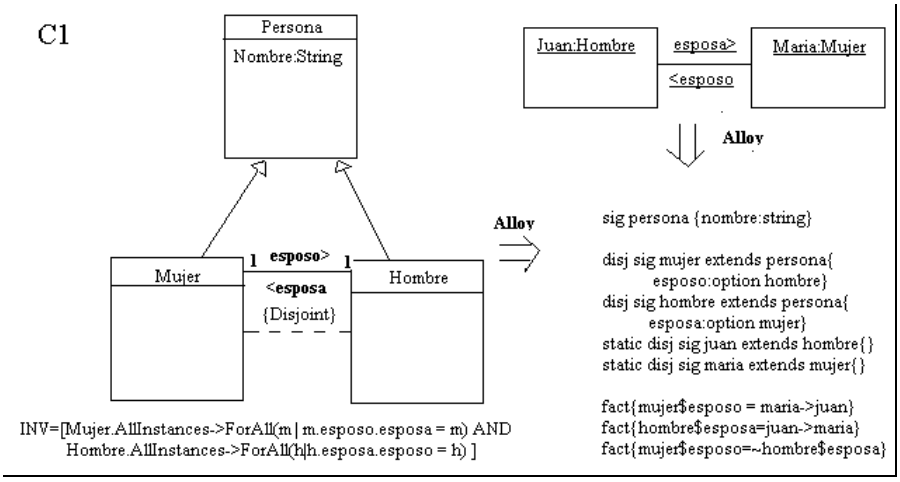

Figura 5: El contrato C1, su modelo y la traducción a Alloy.

imponiendo la restricción "todo hombre tiene al menos una esposa". Dicho cambio puede ser expresado formalmente como la siguiente evolución para la e-machine:

(T_1 := "all h:Hombre | some h.esposa" ,skip(M_1))

que resulta en la teoría T_2 =T_1∪ "all h:Hombre|some h.esposa " y deja intacto el modelo M_1. Dicha teoría fue chequeada mediante Alloy siendo otra vez los resultados satisfactorios.

Un caso interesante para analizar es la evolución;

(skip(T_2),AddObject(M,Jorge,Hombre))

en este caso se deja la teoría T_2 intacta y se agrega un elemento al conjunto que corresponde a la Signatura Hombre en el modelo, se puede observar que el estado resultado es inconsistente ya que existe un hombre que no tiene esposa lo que contradice uno de los axiomas de la teoría.

En el ejemplo dado se recurrió a una notación sencilla y se realizó un solo paso de evolución, en la practica las evoluciones son conformadas por varios pasos y se puede definir su notación al estilo de un lenguaje de programación sencillo. En la figura 6 se puede observar un esquema del paso de evolución realizado.

3.7. Discusión

De acuerdo a lo discutido en las anteriores secciones, tanto el método descrito como la definición de las *e-machines* parecen hacer hincapié en los aspectos estructurales de los diseños, es decir, como cambia su estructura a través del tiempo; no menos importante es el aspecto de la interacción de los componentes de un diseño (es decir, como interactuan sus participantes). Esta último no sólo es interesante para expresar el comportamiento de los participantes sino para el análisis de los cambios que se pueden producir durante la ejecución del sistema. Ahora bien, ¿Es posible expresar la interacción entre los objetos mediante e-machines? En efecto, es posible, debido a que en un estado no solo están las teorías sino los modelos, es por esto que mediante la aplicación de operaciones atómicas sobre los modelos se puede lograr una simulación de la interacción entre los objetos. En este caso, el requisito a tener en cuenta es que todas las modificaciones del primer modelo deben ser consistentes con la teoría. Hablando en términos de diseños esto puede entenderse como que todos los cambios producidos durante una interacción deben respetar las restricciones impuestas en el diseño original.

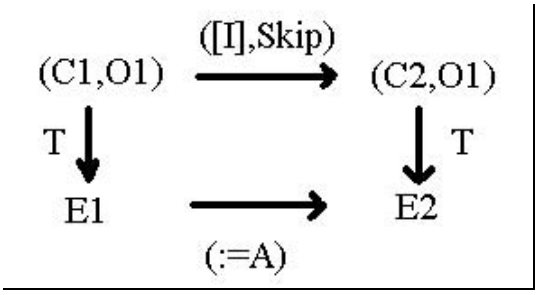

Figura 6: Representación de la evolución realizada.

4. Conclusión

En el presente trabajo se ha esbozado un formalismo que permite identificar y expresar de una manera rigurosa las evoluciones de componentes reusables de diseño que cumplen con las condiciones mínimas que un buen diseñador exigiría. Además se plantean las bases para la construcción de una herramienta semi-automática que permite al desarrollador chequear los cambios sobre sus diseños de una manera sencilla y sin tecnicismos. En la introducción de este articulo se formulan tres preguntas, en este el artículo se intenta responder, a veces implícitamente, a dichas cuestiones:

- Es factible automatizar parcialmente el proceso de evolución de componentes reusables.
- La definición de un formalismo adecuado (como el planteado en sección 2) permite a los desarrolladores realizar un control riguroso sobre los cambios producidos sobre sus diseños.
- En la sección 3.5 se argumenta que es posible la construcción de herramientas semi-automáticas basadas en formalismos y en la experiencia de los desarrolladores.

Estas respuestas son parciales, en parte debido a la amplitud de las preguntas, que todavía siguen siendo problemas abiertos para la comunidad de la ingeniería de software.

5. Trabajos Futuros

Las ideas expuestas en las secciones anteriores representan un interesante acercamiento a la formalización de las evoluciones, a la vez que se propone un método para chequear los posibles errores cometidos durante el desarrollo de un sistema de software. De estos conceptos se desprenden dos líneas de trabajos futuros a tener en cuenta:

- La implementación de la herramienta automática propuesta en la sección 3.5, y su utilización en casos interesantes que permitan analizar su performance y testear su funcionamiento.
- La construcción de un cálculo que permita derivar mediante reglas evoluciones correctas; por ejemplo, pueden plantearse reglas como la siguiente:

$$\frac{S_0 \rightarrow_e^T S_n, S_n \rightarrow_{e'}^T S_m}{S_0 \rightarrow_{e;e'}^T S_m}$$

La cual intuitivamente significa que si S_0 evoluciona correctamente (es un e-machine) a S_n por e y S_n evoluciona correctamente a S_m por e', entonces S_0 evoluciona correctamente a S_m por e;e'.

Referencias

[1] Castro P. y Baum G., *Utilizando Contratos de Reuso con Alloy*, Anales del CACIC 2001, Universidad Nacional de la Patagonia.

[2] C.C.Chang and H.J.Keisler, *Model Theory*, second ed., North Holland, 1977.

[3] Giandini R.S., *Documentación y Evolución de Componentes Reusables*, Tesis de Magister en Ingeniería de Software, Facultad de Ciencias Exactas, UNLP, 1999.

[4] Giandini R.S and Pons C. *Una formalización para Documentación y Evolución de Componentes Reusables*, Anales Cacic´99, Universidad Nacional del Centro de Pcia. Bs. As., Argentina, del 26 al 30 de octubre de 1999.

[5] Giandini R., Baum G. and Pons C. *Manejando Formalmente Evolución de Contratos de Reuso con Semántica de Comportamiento*, En actas de las III Jornadas en Ingeniería del Software realizadas en Murcia España, del 11 al 13 de noviembre de 1998.

[6] Gurevich Y., *Evolving Algebras: An Attemp to Discover Semantics*, Published in Bulletin of European Assoc. for Theor. Computer Science, no. 43, Feb. 1991.

[7] Gurevich Y., Del Castillo G. and Stroetmann K., *Typed Abstract State Machines*, avalaible at www..eecs.umich.edu/gasm/.

[8] Gurevich Y., *Sequential Abstract State Machines Capture Sequential Algorithms*, ACM Transactions on Computational Logic, vol. 1, no. 1 (july 2000).

[9] Grady Booch, James Rumbaugh and Ivar Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[10] Enderton H.B, *A mathematical introduction to logic.* Academic Press; New York, 1972.

[11] Hans-Erik Eriksson and Magnus Penker, *UML Toolkit*, John Wiley & Sons, 1998.

[12] Jackson D., *Automating First-Order Relational Logic*, ACM SIGSOFT Proc. Conf. Foundations of Software Engineering, San Diego, November 2000.

[13] Jackson D., *A Micromodularity Mechanism*, avalaible at http//:sdg.lcs.mit.edu/dng/publications.

[14] Jackson D., *A Lightweight Object Modelling Notation*, To appear in ACM Transaction on Software Engineering and Methodology. Available at http//:sdg.lcs.mit.edu/dng/publications.

[15] Carine Lucas, *Documenting Reuse and Evolution with Reuse Contracts, Chapter 2*, PhD Thesis, Department of Computer Science Vrije Universiteit Brussel, Belgium, Set.1997.

[16] Mens T. and Wermelinger M., *Formal Foundations of Software Evolution: Workshop Report*, March 2001.

[17] Mens T., Demeyer S. and Wermelinger M., *Towards a Software Evolution Benchmark*, Jan.2001.

[18] Warmer-Kleppe, *The Object Constraint Language*, 1998.