Hacia la Composición Paralela de Programas en DynAlloy

Nazareno Aguirre, María Marta Novaira y Sonia Permigiani
Departamento de Computación
Facultad de Ciencias Exactas, Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto
Ruta 36 Km. 601, Río Cuarto (5800),
Córdoba, Argentina
{naguirre, mnovaira, spermigiani}@dc.exa.unrc.edu.ar

Resumen

En este artículo estudiamos una extensión del lenguaje DynAlloy con un operador de composición paralela de acciones. DynAlloy es una extensión al lenguaje de especificaciones Alloy, con soporte para la definición de acciones (expresables originalmente con funciones el Alloy), acciones compuestas y aserciones de corrección parcial. El objetivo principal de DynAlloy es incorporar a Alloy soporte para la validación de propiedades de safety de ejecuciones de sistemas.

En su versión original, DynAlloy soporta composición de acciones mediante composición secuencial, elección no determinista, iteración no acotada, etc, pero no incluye un operador de composición paralela. La composición paralela es un operador de importancia en el ámbito de los sistemas reactivos y concurrentes, por lo cual analizamos aquí las dificultades asociadas a a la extensión de DynAlloy con un operador para composición paralela.

1 Introducción

La creciente aplicación de la computación en sistemas críticos [7], es decir, sistemas cuyo mal funcionamiento puede causar daños económicos importantes o la pérdida de vidas humanas, hace necesaria la aplicación de métodos rigurosos de desarrollo de software. Los métodos formales de desarrollo están sustentados por sólidas bases matemáticas, y con frecuencia involucran lógicas de algún tipo. Por esto, su utilización requiere en general la manipulación de fórmulas en la lógica correspondiente al formalismo utilizado, y la deducción en ésta. Esta característica de los *métodos formales* hace que éstos sean difíciles de aplicar en la práctica, ya que en general se hace necesario contar con desarrolladores altamente capacitados en matemática (o, más precisamente, lógica matemática). Aún con desarrolladores con capacidades sobresalientes en lógica matemática, la aplicación de métodos formales de desarrollo en la construcción de sistemas es difícil, debido a la normal extensión de las actividades de verificación de propiedades.

Por estas razones, los métodos formales o semi-formales con herramientas de soporte, que permiten automatizar las actividades de verificación o validación, han ganado notoria importancia. Alloy es uno de estos lenguajes. Alloy es un lenguaje

formal de especificaciones con una semántica clara, basada en la bien conocida noción matemática de relación [4, 6]. El lenguaje Alloy ha ganado interés en la comunidad de métodos formales, principalmente debido a su simplicidad y el hecho de que el mismo ha sido diseñado con el objetivo de que las sus especificaciones sean automáticamente analizables. De hecho, Alloy cuenta con una herramienta de soporte, denominada Alloy Analyzer, que permite analizar especificaciones de manera completamente automática. Esta herramienta funciona utilizando una técnica basada en SAT solving. Suponiendo que se cuenta con una especificación de un sistema y una propiedad a validar del mismo, el Alloy Analyzer permite buscar contraejemplos de la propiedad, bajo las suposiciones del sistema. Dado que la lógica de primer orden no es decidible, y la lógica en la cual se basa Alloy es una extensión propia de ésta, este mecanismo no es exhaustivo, sino que los contraejemplos sólo pueden buscarse en interpretaciones acotadas por cierta constante k, en el número de elementos en los dominios. Sin embargo, al parecer la técnica es útil en la práctica, ya que, según observa D. Jackson en [6], si una especificación tiene contraejemplos, con frecuencia éstos son de tamaños reducidos.

Si bien Alloy es un lenguaje interesatne, con una herramienta potente, tanto el lenguaje como la herramienta no son apropiados para la validación de propiedades de ejecuciones de los sistemas especificados. Por esta razón, Frias et al. propusieron una extensión a Alloy con soporte para la validación de propiedades de ejecuciones, más precisamente, propiedades de safety de ejecuciones [2]. DynAlloy es una extensión al lenguaje de especificaciones Alloy, con soporte para la definición de acciones (expresables originalmente con funciones el Alloy), acciones compuestas y aserciones de corrección parcial. En su versión original, DynAlloy soporta composición de acciones mediante composición secuencial, elección no determinista, iteración no acotada, etc. Pero, DynAlloy no incluye un operador de composición paralela. La composición paralela es un operador de importancia en el ámbito de los sistemas reactivos y concurrentes, por lo cual analizamos aquí las dificultades asociadas a a la extensión de DynAlloy con un operador para composición paralela.

2 Modelando Arquitecturas de Software

Nuestro trabajo se realiza en el contexto de la validación de arquitecturas de software [3], un área de la Ingeniería de Software que pone especial énfasis en la definición de software en términos de *componentes*, las unidades de cómputo, y *conectores*, las interacciones entre las componentes. Presentamos aquí, brevemente y a través de un ejemplo, de qué manera se pueden modelar arquitecturas de software en (Dyn)Alloy. El ejemplo nos servirá además como vehículo para ilustrar problemas, y definir las extensiones de DynAlloy con composición paralela.

Alloy posee una construcción que puede aprovecharse directamente para el modelado de componentes: `las signaturas`. Las signaturas pueden usarse para definir dominios de datos y componentes con estructuras internas. Consideremos, por ejemplo, que deseamos modelar una arquitectura conocida,`Producer-Consumer` Podemos comenzar diciendo que necesitaremos modelar los elementos que los productores producen y los consumidores consumen, es decir los datos que intercambian los productores y consumidores. En caso de requerir multiples instancias de productores y consumidores para escenarios particulares, podríamos también pensar en dotar los productores y consumidores con un nombre. Los datos y los nombres de componentes pueden modelarse usando signaturas básicas (es decir, sin estructura interna):

```
sig Data {}
```

```
sig Name {}
```

Los productores y consumidores también pueden modelarse mediante signaturas, pero a diferencia de los nombres y los datos, éstas contarán con una estructura interna:

```
sig Producer {
   n: Name,
   queue: set Data,
   outgoing: Data
}

sig Consumer {
   n: Name,
   consumed: set Data,
   incoming: Data
}
```

Note que las signaturas denotan la estructura de productores y consumidores, y no un productor y un consumidor (de alguna manera, describen el *tipo* de los consumidores y productores, de los cuales podemos tener varias instancias diferentes). Un productor tiene un nombre (el campo o atributo `n`), una cola modelada con un conjunto de datos (el campo `queue`) y un elemento listo para ser enviado (el campo `outgoing`). Un consumidor consta de un nombre (el campo `n`), un conjunto de elementos ya consumidos (el historial de consumo de un consumidor, modelado por el campo `consumed`) y un elemento de entrada, listo para ser consumido (el campo `incoming`). Note que, respetando la filosofía de las arquitecturas de software, tanto `Producer` como `Consumer` están libres de referencias a otras componentes. La interacción entre componentes se modela de manera externa a las componentes, mediante conectores.

Las componentes, como las hemos definido, encapsulan información, modelada por los campos de las signaturas. Sin embargo, éstas no incluyen comportamiento. Podemos modelar comportamiento, es decir, operaciones de las componentes, mediante *funciones* en Alloy. Por ejemplo, podemos considerar operaciones para producir y enviar datos en un productor, o para consumir datos en un consumidor:

```
fun produce ( p, p' : Producer, d: Data ){
 p'.n = p.n
 p'.queue = p.queue + d
 p'.outgoing = p.outgoing
}

fun send ( p, p' : Producer, d: Data ){
 d in p.queue
 p'.n = p.n
 p'.queue = p.queue - d
 p'.outgoing = d
}

fun receive ( c, c' : Consumer, d: Data ){
 c'.n = c.n
```

```
  c'.consumed = c.consumed + c.incoming
  c'.incoming = d
}
```

Las variables primadas entre los parámetros de las funciones denotan el estado de las componentes correspondientes después de la ejecución de las operaciones, aunque esto es sólo una convención [2]. Este no es el enfoque seguido en DynAlloy, en el cual las operaciones se modelan con acciones [2], como mostraremos más adelante.

Gracias a que Alloy está basado en relaciones, y que las signaturas pueden contener campos de tipo relación, podemos modelar conectores mediante relaciones (binarias, en principio). En nuestro caso, podemos requerir conectar productores con consumidores. Luego, un sistema, de productores y consumidores, constaría, en principio, de un conjunto de productores, un conjunto de consumidores, una relación binaria entre éstos y dos conjuntos unitarios que representarán canales de comunicación entre el mensaje enviado por el productor y el recibido por el consumidor:

```
sig System {
  ccs: set Consumer,
  pps: set Producer,
  conn: pps -> ccs
  chan_c: set Consumer,
  chan_d: set Producer
}
```

Con el sistema definido, podemos utilizar funciones en Alloy para modelar operaciones del sistema. A diferencia de otras operaciones, como por ejemplo las de productores y consumidores, podemos definir las operaciones del sistema de manera *incremental*, a partir de las definiciones que ya tenemos (esto puede notarse en la definición de la función *syssend*, descripta más abajo). Podemos definir *operaciones de reconfiguración* fácilmente, es decir, operaciones que modifican la arquitectura del sistema, agregando productores, por ejemplo:

```
fun sysnewprod (s, s' : System, p: Producer ){
 s'.pps = s.pps + p
 s'.ccs = s.ccs
 s'.conn = s.conn
 s'.chan_c = s.chan_c
 s'.chan_d = s.chan_d
}
```

Otra operación de reconfiguración, importante para nuestro ejemplo es la de conectar o desconectar productores con consumidores dinámicamente. Consideremos, por ejemplo, la operación que conecta un productor vivo con un consumidor vivo:

```
fun connect(s,s': System, p: Producer, c: Consumer) {
 p in s.pps && c in s.ccs
 s'.pps = s.pps
```

```
 s'.ccs = s.ccs
 s'.conn = s.conn + (p -> c)
 s'.chan_c = s.chan_c
 s'.chan_d = s.chan_d
}
```

En la definición de la signatura *System*, definimos conectores mediante una relación entre productores y consumidores. Pero esto no modela la interacción entre las instancias. podemos definir la interacción, ligada a la existencia de una conección entre componentes (representado por un par ordenado en la relación `conn`, en nuestro ejemplo) mediante funciones de System.

```
fun syssend(s, s' : System, p,p':Producer, c,c':Consumer, d: Data){
 p in s.pps && c in s.ccs &&  s'.conn = s.conn &&
 s'.chan_c = s.chan_c && s'.chan_d = s.chan_d
 && (p -> c) in s.conn
 send(p,p',d)
 receive(c,c',d)
 p.n = p'.n
 c.n = c'.n
 p' in s'.pps && c' in s'.ccs
}
```

Nótese que, como parte de la precondición de esta función, tenemos que el par ordenado `(p -> c)` tiene que pertenecer a `s.conn`.

Se pueden imponer restricciones de integridad al modelado de una arquitectura de software mediante el uso de *hechos* en Alloy. Estas restricciones, debido a las limitaciones de Alloy, pueden referirse, en principio, sólo a restricciones estructurales (*estáticas*), pero *no* pueden referirse a ejecuciones. Veremos más adelante que existen mecanismos para caracterizar propiedades de trazas de ejecución, pero esto requiere una especificación complicada de trazas dentro del modelo de la arquitectura [2]. A modo de ejemplo, consideremos la restricción de que los productores y consumidores no comparten el espacio de nombre. Esto puede ser fácilmente expresado de la siguiente forma:

```
fact NoSharedNamesPC  {
 all s: System | all p : s.pps | all c : s.ccs | ! (p.n = c.n)
}
```

Podemos también requerir que diferentes productores (resp. consumidores) tienen asignados diferentes nombres:

```
fact NoSharedNamesP  {
 all s: System | all p1,p2 : s.pps | p1.n = p2.n => p1 = p2
}

fact NoSharedNamesC  {
```

```
  all s: System | all c1,c2 : s.ccs | c1.n = c2.n => c1 = c2
 }
```

2.1 Validando Propiedades de Trazas de Ejecución

Como se observa en [1, 2], Alloy no es apropiado para la validación de propiedades mirando ejecuciones de trazas del sistema. Estas propiedades son importantes, especialmente en arquitecturas dinámicas. Así mismo Alloy está limitado en este sentido, en [5], los autores proponen un mecanismo para representar y validar, por medio del Alloy Analyzer y con el standard Alloy, propiedades de traza. Este mecanismo consiste, esencialmente, en la complementación de la especificación de un sistema con una especificación explícita de traza. La principal desventaja de este mecanismo es que la definición de traza de ejecución depende fuertemente de la propiedad de traza que uno desearía validar.

Por ejemplo, si quisiéramos comprobar que la secuencia de aplicaciones de `connect` y `sysnewprod` preservan la funcionalidad de `conn`, las trazas del sistema deberían ser definidas de la siguiente forma:

```
sig Tick {}

sig SystemTrace {
        ticks: set Tick,
        first, last :ticks,
        next: (ticks-last) ! -> !(ticks-first),
        state: ticks ->! System
}
   {
        first.*next=ticks
        all t:ticks - last |
          some s = t.state, s' =t.next.state |
          some p: Producer, c:Consumer | connect (s,s',p,c)
          || (some p: Producer| sysnewprod(s,s',p))
   }
```

Puede ser entonces, que intentemos validar que las secuencias de aplicaciones de `connect` y `sysnewprod` preservan la funcionalidad de `conn` para decir que, para cada traza, si `conn` es funcional en el estado inicial, entonces es también funcional en cualquier estado (inicial, intermedio o final) de la traza.

Este enfoque no es considerado una buena práctica, pues las trazas de ejecución no deberían tener que ser especificadas explícitamente, sino que deberían estar determinadas por la combinación de operaciones objeto del análisis. Por esto, se ha provisto una alternativa en [2], la cual básicamente reemplaza funciones, como descriptores de operaciones, por *acciones*, que pueden combinarse para formar programas, usando composición secuencial, elección no determinista, iteración, etc. Uno puede luego especificar acciones, o validar propiedades, mediante aserciones de corrección parcial. Luego, operaciones tales como `produce` o `send`, se expresan de la siguiente manera:

```
act produce(p: Producer, d: Data) {
    pre { }
    pos { (p'.n = p.n) and (p'.queue = p.queue + d)
           and (p'.outgoing = p.outgoing)
   }
}

act send(p: Producer, d: Data) {
    pre { d in p.queue }
    pos { (p'.n = p.n) and (p'.queue = p.queue - d)
          and (p'.outgoing = d) }
}
```

He aquí un ejemplo de na acción más compleja:

```
 act sysnewprod ( s : System, p: Producer ) {
  pre { }
  pos {  s'.ccs = s.ccs and
         s'.conn = s.conn and
         s'.pps = s.pps + p }
 }
```

Note que las variables primadas no son necesarias en la definición de las operaciones, ya que las acciones tienen un significado de input/output, reflejado explícitamente en la semántica. Podemos combinar acciones básicas y formar programas:

```
syssend(s, p, c, d) = send(p,d) ; receive(c,d)
```

y escribir aserciones para los mismos, las cuales pueden luego ser validadas:

```
assert syssendSemantics {
    assertCorrectness {
        pre = { some s : System | p in s.pps and c in s.ccs
        and (p -> c) in s.conn}
        prg = { send(p,d) ; receive(c,d) }
        pos = { p'.n = p.n and p'.queue = p.queue - d and
                p'.outgoing = d and c'.n = c.n and
                c'.consumed = c.consumed + c.incoming
                and c'.incoming = d}
    }
}
```

3 Composición Paralela En DynAlloy

En esta sección describimos la extensión a DynAlloy que proponemos. Es sabido en el ámbito de la programación concurrente que la composición paralela de procesos es definible en términos de *interleaving* de las acciones atómicas de los procesos

intervinientes. Basados en esta idea, podemos pensar en definir el operador de composición paralela de acciones en DynAlloy como la elección no determinista de todas las combinaciones posibles de acciones atómicas, compuestas con composición secuencial. Así, si tenemos dos procesos `P1 = A1; A2` y `P2 = A3;A4`, la composición paralela `P1 II P2` correspondería al programa DynAlloy siguiente:

```
A1;A2;A3;A4 + A1;A3;A4;A2 + A1;A3;A2;A4 + A3;A1;A2;A4 + ...
```

Este ejemplo es simple, y en este caso la composición paralela es fácil de "expandir". Sin embargo, cuando uno de los procesos involucrados contiene un operador de iteración, la expansión puede corresponder a un programa infinito. Consideremos los programas `P1 = A1` y `P2 = (A2;A3)*`. La composición paralela `P1 II P2` correspondería al programa DynAlloy infinito siguiente:

```
A1 + A2;A3;A1 + A2;A3;A2;A3;A1 + ...
```

Por este motivo, se hace necesario considerar una cota en la iteración (es decir, interpretar "*" como iteración acotada) para poder expandir la composición paralela. Si bien, a primera vista, esta restricción puede considerarse demasiado restrictiva, es importante notar que, debido a los mecanismos de análisis automático asociados a (Dyn)Alloy, la restricción es absolutamente normal. La razón de esto es que los mecanismos de análisis de (Dyn)Alloy exigen imponer tales cotas.

Incluimos entonces soporte para la composición paralela en DynAlloy. La gramática para programas DynAlloy es extendida a la siguiente:

```
program ::= (formula,formula) /*"atomic action"*/
  | formula? /*"test"*/
  | program + program /*"non-deterministic choice"*/
  | program;program /*"sequential composition"*/
  | program* /*"iteration"*/
  | program II program /*"interleaving"*/

formula ::= ...| { formula } program {formula }
/*"partial correctness"
```

Para evaluar el uso de este operador, extendemos el ejemplo presentado anteriormente con programas compuestos que lo utilizan. Debemos primero definir algunas acciones que servirán de base para el ejemplo.

```
act sproduce ( s : System ) {
   pre { }
   pos { some p: s.pps, p':s'.pps, d:Data | (p'.n = p.n) and
         (p'.queue = p.queue + d) and
         (p'.outgoing = p.outgoing) and !(d in p.queue) }
}


act ssend ( s : System ) {
   pre { no ( s.chan_c ) and no (s.chan_d)}
   pos { some p: s.pps,p':s'.pps d: Data, c: s.ccs| (p'.n = p.n)
         and (p'.queue = p.queue - d) and (p'.outgoing = d) and
         (p -> c) in s.conn and (s'.chan_c = s.chan_c + c)
```

```
            and no(s'.chan_c - c) and (s'.chan_d = s.chan_d + d)
            and no(s'.chan_d - d) }
    }
```

Como podemos observar, en la precondición de la acción necesitamos que los conjuntos (unitarios) chan_c y chan_d de System no tengan elementos. Esto nos será de utilidad para asegurarnos que no se sobreescriban los mensajes que envía el sistema

```
act srec ( s : System ) {
   pre { !no( s.chan_c ) and !no(s.chan_d)}
   pos {some c: s.ccs, c':s'.ccs, d: s.chan_d | (c'.consumed =
         c.consumed + c.incoming) and (c'.incoming = d) and
         (s'.chan_c= s.chan_c - c) and (s'.chan_d= s.chan_d - d)}
}
```

DynAlloy nos permite unificar estas operaciones en un programa, que ejecuta en forma secuencial las acciones definidas, de la siguiente forma:

```
sssend(s) = ssend(s) ; srec(s)
```

Estamos ahora en condición de definir un programa que utilice el operador `II`. Consideremos el siguiente programa:

```
(sssend(s) II sproduce (s))*
```

Este programa corresponde a aplicaciones intercaladas de producciones en los productores y envíos. Podríamos ahora intentar validar la siguiente aserción:

> Luego de aplicaciones intercaladas de producciones en los productores y envíos, no quedan items en ningún productor del sistema por enviarse.

Esta aserción es claramente falsa, y es relativamente simple para DynAlloy encontrar contraejemplos de la misma. Para validarla es necesario representar la aserción de la siguiente manera:

```
fun emptyqueue(s:System){
   all p:s.pps| no (p.queue)
}

assert Interleaving {
     assertCorrectness {
          pre = { s : System }
          prg = { (sssend(s) II sproduce (s))* }
          pos = {emptyqueue(s')  }
     }
}
```

El programa `(sssend(s) II sproduce (s)*` es traducido por nuestro preprocesador en un programa DynAlloy standard. El programa `(sssend(s) II sproduce (s))*` es reescrito como el siguiente programa DynAlloy::

```
(ssend(s) ; srec(s); sproduce (s) + ssend(s); sproduce (s);
srec(s) + sproduce (s) ; ssend(s) ; srec(s))*
```

Muchas de las fallas en sistemas reactivos se deben a la incorrecta interpretación de la composición paralela de programas. Agregar soporte en DynAlloy para tal operación de composición es entonces, a nuestro entender, importante. Además la creciente aplicación de conceptos de programación paralela y concurrente (y la inclusión de soporte para éstos en lenguajes de programación de propósito general, como por ejemplo el soporte de threads en java), hace a la composición paralela un operador que ya forma parte del conocimiento común de los desarrolladores. Es metodológicamente incorrecto forzar al desarrollador a "implementar" manualmente tal operador para realizar las actividades de validación en (Dyn)Alloy.

Como nuestra extensión es definible en término de los operadores ya existentes en DynAlloy, no debemos preocuparnos de haber introducido inconsistencias en el mismo. La interpretación de acciones atómicas, según se expresa en [2], prohibe otra interpretación distinta de interleaving para la composición paralela. En particular, no es posible, mediante la composición de programas, que dos acciones atómicas distintas se ejecuten simultáneamente (deberán ejecutarse en secuencia, en algún orden posible). Por esto, la semántica de operaciones de comunicación, como por ejemplo la operación `sssend` definida más arriba, no puede implementarse como comunicación *rendevous*, sino asincrónicamente.

4 Conclusiones y Trabajos Futuros

Hemos estudiado los problemas asociados a la introducción de un operador de composición paralela de programas en DynAlloy. La dificultad principal que identificamos tiene que ver con que, para poder desplegar una composición paralela usando composición secuencial y elección no determinista (semántica de entrelazado para la composición paralela), necesitamos imponer una cota en la iteración. Esto sin embargo no constituye una complicación, ya que el mecanismo de análisis principal de (Dyn)Alloy necesita imponer cotas en los dominios, y en la longitud de las trazas. Hemos implementado soporte en DynAlloy para composición paralela, la cual, en presencia de una cota para iteración, se traduce en una especificación DynAlloy standard, mediante la interpretación clásica de composición paralela mediante interleaving de acciones atómicas de los programas intervinientes. Experimentamos con un caso de estudio relativamente simple, basado en un modelo de productor-consumidor.

Como indicamos en el comienzo de este artículo, nuestro trabajo se centra en la validación y verificación de arquitecturas de software. En este contexto, estamos explorando el uso de DynAlloy en combinación con técnicas de deducción automática, principalmente para la verificación de propiedades de liveness, no expresables en DynAlloy, en su forma actual. También planeamos proveer una traducción de un lenguaje de descripción de arquitecturas de alto nivel, tal como Acme o Darwin, a especificaciones (Dyn)Alloy, para facilitar las tareas de validación y verificación para usuarios no familiarizados con Alloy o su lógica subyacente. Particularmente, pretendemos utilizar un lenguaje de descripción de arquitecturas que soporte la especificación de reconfiguración dinámica, una característica que es considerada actualmente de gran importancia, y que muchos desarrolladores exigen. (Dyn)Alloy, como hemos mostrado en los ejemplos, es lo suficientemente expresivo para representar tal tipo de arquitecturas.

Referencias

[1] M. Frias, C. López Pombo, G. Baum, N. Aguirre and T. Maibaum, *Taking Alloy to the Movies*, in Proceedings of the International Symposium on Formal Methods FM 2003, Pisa, Italy, Lecture Notes in Computer Science, Springer-Verlag, 2003.

[2] M. Frias, J. Galeotti, C. López Pombo and N. Aguirre, *DynAlloy: Upgrading Alloy with Actions*, in Proceedings of the International Conference on Software Engineering ICSE 2005, St. Louis, USA, ACM Press, 2005.

[3] D. Garlan and M. Shaw, *An Introduction to Software Architecture*, in V. Ambriola and G. Tortora (eds), Advances in Software Engineering and Knowledge Engineering, Series on Software Engineering and Knowledge Engineering, Vol 2, World Scientific Publishing Company, 1993.

[4] D. Jackson, *Lightweight Formal Methods*, in Proceedings of the International Symposium on Formal Methods Europe FME 2001, Berlin, Germany, Lecture Notes in Computer Science, Springer-Verlag, 2001.

[5] D. Jackson, I. Shlyakhter and M. Sridharan, *A Micromodularity Mechanism*, in Proceedings of the 8th European Software Engineering Conference held jointly with the 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering ESEC/FSE 2001, Viena, Austria, 2001.

[6] D. Jackson, *Alloy: a lightweight object modelling notation*, in ACM Transactions on Software Engineering and Methodology (ACM TOSEM), Vol. 11, Nro. 2, 2002.

[7] N. Storey, *Safety-Critical Computer Systems*, Addison-Wesley, 1996.