UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA DEL SOFTWARE

Incorporación de incertidumbre de la medida en modelos UML para la industria 4.0

Introducing measurement uncertainty into UML models for the industry 4.0

Realizado por
Juan José Guerrero Ruiz

Tutorizado por
Antonio Vallecillo Moreno
Lola Burgueño Caballero
Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, JUNIO DE 2019

Fecha defensa: octubre de 2019

Fdo. El/la Secretario/a del Tribunal

Resumen

En este proyecto, se realiza un estudio detallado de las ventajas que supone la utilización de incertidumbre de la medida en la simulación de sistemas reales y, en particular, para la industria 4.0. El objetivo principal de dicho procedimiento es obtener una representación más fidedigna al comportamiento de un sistema en un entorno físico. Para ello se utilizan simulaciones de modelos en la herramienta USE, que inicialmente serán funcionales sin incertidumbre en ellos. Sobre estos sistemas se trabaja en la introducción de incertidumbre de la medida en las que lo que lo necesiten, realizando un estudio previo de que elementos de los sistemas lo requieren.
Finalmente, se reflexiona sobre la dificultad de incluir la incertidumbre en los mismos, y se evalúa su impacto de en los sistemas, así como la calidad de los resultados de las pruebas realizadas con respecto a los de las que no incluyen incertidumbre de la medida.

Palabras clave:

- Incertidumbre

- Pruebas

- Modelo

- Medidas

- Comparación

Abstract

In this project, we are doing a detailed study about the advantages that entail the usage of measurement uncertainty in real-world system simulation and, in particular, in those for the Industry 4.0. The main goal of this approach is to obtain a more accurate representation of the behavior of a system in a physical environment. For this matter, we use system simulations created for its usage on the USE(UML Specification Environment) tool, which initially will not have measurement uncertainty. We will work later on the introduction of measurement uncertainty in the parts of the system that require it, making a study first to evaluate which parts of the system are suitable for that.
In the end, we analyze how costly it really is to introduce and use uncertainty in systems similar to the ones we proposed here, and we evaluate the impact such introduction made on the different system, as well as the quality of the executed tests' results in comparison to the ones that didn't Include measurement uncertainty.

Keywords:

- Uncertainty

- Tests

- Model

- Measurement

- Comparison

Índice

Contenido

1

Introducción

1.1 Motivación

En la actualidad, la humanidad se abre paso a través de la ya denominada Cuarta Revolución Industrial, caracterizada por la convergencia de avances tecnológicos físicos, digitales y biológicos. La principal característica que diferencia a ésta de las revoluciones anteriores es que se basa en la creación de nuevos sistemas combinando elementos de la anterior revolución. De aquí nace el concepto de Industria 4.0, que se refiere a la manera de organizar y controlar los medios de producción. La meta a conseguir es la creación de fábricas inteligentes, caracterizadas por la automatización e interconexión masiva de los distintos componentes de la cadena de montaje, sensores y sistemas de control. De alcanzarse los objetivos, se pueden obtener fábricas con un nivel de rapidez, calidad y estabilidad en la producción muy superiores a lo que la humanidad ha conseguido en los últimos siglos.

La completa virtualización de los sistemas de control de la cadena de montaje permite obtener datos de diferente índole durante la ejecución del proceso (distancias, tiempos, temperaturas, cantidades…). La utilización de dichos datos como guía nos posibilita realizar simulaciones virtuales de parte (o toda) la cadena de producción. Dichas simulaciones nos permitirían realizar pruebas relativas al funcionamiento del sistema, así como observar que efectos tendrían sobre el mismo cambios en las distintas partes que lo componen. No obstante, las pruebas realizadas sobre simulaciones del sistema presentan un inconveniente potencialmente peligroso para su traducción al mundo real: el uso de números exactos para definir los diferentes parámetros medidos en el sistema no reflejan la inexactitud inherente al funcionamiento de un sistema en el mundo real. El desgaste de una articulación robótica, la erosión en la parte interna de una válvula, la deformación milimétrica de los muelles de un medidor de temperatura bimetálico tras miles de horas de funcionamiento… provocan pequeñas desviaciones del valor real con respecto al valor medido en un sistema real. Dichas variaciones no son representables de una manera realista mediante números exactos, y pueden llevar a simulaciones del sistema que no se correspondan con el comportamiento en el mundo real. Esto puede llegar a suponer un peligro para la industria, ya que confiar en simulaciones que no tienen en cuenta este factor desde un principio puede llevar a la construcción de

sistemas que acaben defectuosos tras un periodo de tiempo o que directamente no funcionen.

La industria necesita tener en cuenta lo expuesto aquí a la hora de realizar simulaciones si quiere que éstas sean fiables, por lo que el primer paso consiste en encontrar una manera de representar estos datos de manera adecuada sin alterar el funcionamiento del sistema en sí. Una posible solución es la utilización de incertidumbre en las medidas que así lo requieran.

1.2 Objetivos

El principal objetivo de este trabajo es estudiar el impacto de la introducción de incertidumbre en la simulación de sistemas para la Industria 4.0 . Lo primero es proponer varios sistemas como ejemplo, que imiten sistemas del mundo real y no incluyan incertidumbre. Posteriormente, se procede a realizar un análisis de cómo y dónde se debe introducir incertidumbre, de manera que reflejemos la inexactitud de los datos adecuados, pero no estropeemos información que presuponemos que no necesita de ella. Se pone énfasis en reflejar la complejidad del proceso de introducción de incertidumbre en un sistema ya creado, así como el efecto que dicha introducción produce en los resultados obtenidos. Además, se realizan pruebas de los distintos sistemas con incertidumbre para demostrar que, efectivamente, estas simulaciones de sistemas con incertidumbres son de gran utilidad a la hora de planear que tipo de componentes son los más adecuados a la hora de construir nuestra cadena de montaje en el mundo real.

1.3 Estructura de la memoria

Comenzaremos la memoria describiendo las distintas tecnologías sobre las que realizaremos los diferentes sistemas y las pruebas sobre ellos. Una vez descrito esto, se procede a explicar los distintos sistemas desarrollados que imitan cadenas de montaje del mundo real. Se hace hincapié en por qué se han escogido estos sistemas en concreto, y en el potencial que tienen a la hora de probar el impacto de la introducción de incertidumbre.

Posteriormente, se observa el funcionamiento de los sistemas sin incertidumbre y se recopilan datos del mismo. Esto es de ayuda para realizar un análisis a continuación de dónde y cómo sería adecuado introducir incertidumbre en los sistemas, así como que parámetros relativos al rendimiento se verían afectados.

Una vez decidido qué cambiar y cómo hacerlo, se procede a adaptar los distintos sistemas para su correcto funcionamiento con variables que tengan cierta incertidumbre. Es importante que dicha adaptación no altere el funcionamiento esencial de los sistemas, pues el objetivo final es evaluar el impacto de la incertidumbre en la actividad de la cadena de montaje. Se hace una valoración además de la dificultad y tiempo necesario para dicha adaptación, y se tienen en cuenta a la hora de sopesar si la introducción de incertidumbre para la realización de pruebas merece la pena a escala industrial.

Finalmente, se recopilan datos de diferentes ejecuciones de sistema variando los distintos valores que influyen en la incertidumbre introducida en el sistema.

Dichos datos sirven para realizar un estudio acerca de cómo unos componentes más o menos precisos afectan a la eficiencia y fiabilidad de los sistemas planteados.

2

Tecnologías a utilizar

2.1 USE

Centraremos el desarrollo de los modelos sobre los que realizar el estudio en la herramienta USE (UML-based Specification Environment). USE es una herramienta desarrollada por el departamento de Sistemas de Bases de Datos de la Universidad de Bremen para la especificación de modelos de sistemas de información. Está implementada en Java, y basada en un subconjunto de UML, utilizando elementos como clases y asociaciones para realizar descripciones textuales de los modelos. Dichos modelos pueden ser simulados una vez especificados para comprobar su correcto comportamiento.

Para añadir integridad a los modelos descritos textualmente, se pueden definir restricciones a cumplir por el mismo en OCL (Object Constraint Language). Dichas restricciones se comprueban automáticamente en cada estado del sistema, con el fin de asegurar la solidez del modelo durante la evolución de la simulación.

Al trabajar con USE, lo primero que se debe hacer es definir el sistema mediante texto en una herramienta externa (ATOM, Notepad++…) ya que USE no tiene editor de texto incorporado. La estructura básica de un fichero USE es la siguiente:

- El nombre del modelo debe encabezar el archivo de texto.

- A continuación se definen las clases, que representan los distintos elementos que componen nuestro sistema. Las clases se definen mediante dos componentes principales, los atributos y las operaciones. Los atributos representan cualidades o valores de la clase en concreto, y las operaciones simulan el comportamiento de estos elementos en el sistema que simulan, pudiendo alterar o no el estado de sus atributos o los de los demás.

- Tras esto, se representan las diferentes asociaciones del sistema, que reflejan las relaciones entre las distintas clases del sistema. Dichas relaciones tienen asociadas en cada extremo una multiplicidad, que indica la cantidad de ese tipo de asociaciones que pueden referenciar a única instancia de esa clase. Desde una operación, una clase puede referenciar a atributos y operaciones de clases

que estén unidas a ella mediante asociaciones. Dicha multiplicidad puede ser 1, * (cualquier número) o [x..y] (cualquier número dentro de un rango determinado. Asimismo, las asociaciones pueden ser ordenadas en uno o ambos extremos (FIFO).

- Por último, se definen las restricciones que añaden integridad al sistema. Cada restricción lleva asociado un contexto, que hace referencia a la clase que se utilizará de "base" para referenciar a otras clases y sus elementos a través de relaciones.

Siguiendo esta sintaxis, podemos construir de manera sencilla modelos como el expuesto a continuación (Fig. 1).

Una vez construido el modelo, podemos empezar a trabajar con él en USE. Para ello, lo primero que tendremos que hacer es compilarlo. Si existe algún error de sintaxis o de consistencia en el modelo proporcionado a USE, éste devuelve un output señalando el/los error/es en cuestión, así como la línea que potencialmente los origina.

Con el modelo ya correctamente compilado, podremos comenzar a simularlo definiendo instancias del sistema en SOIL. SOIL (Simple OCL-based Imperative Language) es el lenguaje de programación imperativa de USE. En líneas generales, su uso se basa en la creación de instancias de clases y asociaciones del modelo mediante sentencias por línea de comandos, que definan el estado del sistema en un momento concreto. Dichas sentencias se pueden preparar en un archivo .soil, para ser ejecutadas en batería. Una vez se tiene la instancia del sistema, se pueden usar sentencias SOIL para ejecutar las distintas operaciones de las clases instanciadas, con el fin de simular el comportamiento del sistema y ver su evolución.

```
model Modelo

class Clase1
  attributes
    att1: String
    att2:Integer
  operations
    operation1(arg: Integer) : Boolean = self.att2<=arg;
end

class Clase2
end

association RelatesTo between
  Clase1[0..*] role clase1
  Clase2[0..*] role clase2
End

constraints
context Clase1 inv menorquetres : self.att2<3
```

Cod.1 – Ejemplo de fichero USE

2.2 Incertidumbre

Con el fin de representar mejor la inexactitud de las medidas en el mundo real, debido a la tolerancia y desgaste de los sistemas en el mismo, se pretende introducir en las variables pertinentes incertidumbre. La incertidumbre de la medida es la manera de expresar la dispersión estadística de los distintos valores obtenidos al realizar la medición de una cantidad. La manera de expresar dicha dispersión es mediante un par valor-incertidumbre. Para este proyecto necesitaremos representar la incertidumbre de dos tipos de medidas: de cantidades (UReal) y de predicados lógicos asociados a medidas con incertidumbre (UBoolean).

Las variables de tipo UReal se representan de la manera **UReal(valor, incertidumbre).** El valor representa la media de las cantidades medidas en un sensor determinado, y la incertidumbre es un valor probabilístico relativo al conjunto de medidas tomadas, en nuestro caso (y por lo general, siempre) la desviación típica. De esta manera, el valor de la variable no representa la medida en sí, sino una distribución normal sobre la que se mueven los posibles valores de la medida real. Dicho dato resulta mucho más útil que una variable exacta que no refleja la realidad, puesto que se puede utilizar para pruebas como comprobar la dispersión de cierto valor al utilizar elementos con diferente precisión, por ejemplo.

Las variables de tipo UBoolean se representan con la expresión **UBoolean(true, incertidumbre)**. En este caso, el valor siempre es true, y la incertidumbre refleja la probabilidad de que la variable sea realmente true (entre 0 y 1). Para utilizar estas variables a la hora de establecer una condición, se debe indicar una cota mínima de confianza a partir de la cual se considera como true el valor, y compararlo con él de la variable.

Las variables con incertidumbre permiten a su vez trabajar sin la misma en caso de que sea necesario. Si queremos representar una medida exacta, bastará con escribir **UReal(value, 0.0)**. De la misma manera, si se requiere que un valor definido como UBoolean sea necesariamente true o false se podrá representar mediante **UBoolean(true, 1.0)** o **UBoolean(true, 0.0)**, respectivamente.

3

Desarrollo del primer modelo: Bottle Filler

3.1 El modelo

Para empezar, imaginemos que queremos desarrollar un modelo que simule el comportamiento de una cadena de llenado de botellines de 20cl de capacidad. En concreto, la parte que queremos simular es el tramo en el que se procede al llenado de las botellas, siendo los principales actuadores la máquina de llenado y la cinta transportadora por la que avanzan las botellas.

Antes de representar las clases que constituyen el modelo sin más, debemos establecer ciertas clases que impongan un orden en la estructura interna de nuestro modelo, a fin de facilitar la creación de asociaciones posteriores y ahorrarnos la repetición de código en clases que comparten cierto funcionamiento. Nuestro tendrá un controlador central representado por la clase Control. Dicho controlador dará paso a los diferentes controladores de los objetos activos del sistema, caracterizados por heredar de la clase ActiveObject.

```
abstract class ActiveObject
  operations
    token() begin end
end
--------------------------------
class Control
  operations
    control(n: Integer)
    begin
      while n>0 do
        n:=n-1;
        for i in self.controlledobject do
          i.token();
        end
      end
    end
end
```

Cod.2 – Clases ActiveObject y Control

Algunos objetos activos, además de por el permiso dado por Control, también verán sus acciones restringidas por la necesidad del paso de un mínimo de tiempo entre ellas. Estos objetos, en vez de ActiveObject,
heredarán de TimedObject, que no es más que un ActiveObject que se guía a su vez por un objeto del tipo Clock. Clock es un objeto activo que cuenta un segundo cada vez que se le da paso, y representa el tiempo de ejecución transcurrido en nuestra simulación. Por último, para redondear ésta estructura interna de nuestro sistema, crearemos la clase PositionedElement, de la que extenderá todo elemento cuya posición en el espacio de trabajo sea relevante.

```
abstract class PositionedElement
  attributes
        x : Real init: 0.0
        y : Real init: 0.0
        z : Real init: 0.0
  operations
        distanceTo(p:PositionedElement) : Real =
 ((self.x-p.x).abs()+(self.y-p.y).abs()+(self.z-p.z).abs())

    moveTo(x:Real, y:Real, z:Real)
        begin
                self.x := x; self.y := y; self.z := z;
        end

        advance(x:Real)
    begin
     self.x := self.x + x;
     end

        isAt(x:Real, y:Real, z:Real) :Boolean = (self.x = x) and (self.y = y) and (self.z = z)
end
-----------------------------------
class Clock < ActiveObject
  attributes
        NOW: Integer init: 0
        tick:Integer init: 0
        ticksPerMs:Integer init: 1
  operations
  token()
  begin
        self.tick:=self.tick+1;
        if self.tick>=self.ticksPerMs then
                self.NOW:=self.NOW+1;
                self.tick:=0;
        end;
  end
end
------------------------------
abstract class TimedObject < ActiveObject
  attributes
    clock:Clock
    processingTime:Integer
    nextWakeUp:Integer init:0
end
```

Cod.3 – Clases PositionedElement, Clock y TimedObject

Una vez establecido el esqueleto del comportamiento de nuestro sistema, procedemos a definir las distintas clases que representan los elementos reales que intervienen en el sistema.

Por una parte, las botellas las proporcionará un sistema externo representado por la clase BottleProvider, que colocará las botellas (representadas por la clase pasiva Bottle) en el punto de inicio de nuestra cinta transportadora, especificada por en la clase MotorizedTray. El BottleProvider comprobará que el punto de inicio de la cinta está vacío antes de colocar una nueva botella. En el momento de su entrada al sistema, las botellas tendrán asociadas un objeto de la clase Tag, que recogerá diferentes datos del proceso como el tiempo de inicio y final, el contenido de la botellas y el tiempo de llenado. Estos datos serán la principal fuente de información para analizar el funcionamiento del sistema más adelante.

```
enum TrayState {Idle, Moving}

class MotorizedTray < TimedObject
  attributes
    state:TrayState init:#Moving
    TrayInputX: Real init=0.0
    TrayInputY: Real init=0.0
    TrayInputZ: Real init=0.0
  operations
  token()
    begin
      if self.clock.NOW>=self.nextWakeUp
      then for b in self.bottles do
        if self.positiondetector.check(b)
        then self.attacher.attachBottle(b); self.state:=#Idle
        else
          if self.state=#Moving
          then b.advance(5.0)
          else
            if b.fillerforbottle->size()>0 and b.isFull
            then self.attacher.detachBottle(b); self.state:=#Moving; b.advance(5.0);
b.tag.readyForProcessing:=true;
            end
          end
        end
        end; self.nextWakeUp:=self.nextWakeUp+self.processingTime;
      end
  end
    isAvailableForInput() : Boolean = PositionedElement.allInstances-
>select(p|p.oclIsKindOf(Bottle) and
p.isAt(self.TrayInputX,self.TrayInputY,self.TrayInputZ))->isEmpty
end
```

Cod.4 – Clase MotorizedTray

```
class BottleProvider < TimedObject
  operations
    provideBottle()
    begin
      declare b: Bottle, t: Tag;
      t:=new Tag;
      t.creationTime:=self.clock.NOW;
      b:= new Bottle;
      b.tag:=t;
      insert(self.providedtray, b) into BottlesInTray;
b.moveTo(self.providedtray.TrayInputX,self.providedtray.TrayInputY,
self.providedtray.TrayInputZ);
    end
    token()
    begin
    if self.clock.NOW>=self.nextWakeUp
    then if self.providedtray.isAvailableForInput()
          then self.provideBottle();
          self.nextWakeUp:=self.clock.NOW+self.processingTime
          end
      end
    end
end
----------------------------
class Bottle < PositionedElement
  attributes
    content:Real init= 0.0
    capacity: Real init=20
    isFull: Boolean init=false
    tag:Tag
end
--------------------------------
class Tag
  attributes
  creationTime: Real
  startFillingTime: Real
  stopFillingTime: Real
  endProcessTime: Real
  content: Real
  readyForProcessing: Boolean init:false
end
```

Cod.5 – Clases BottleProvider, Bottle y Tag

La cinta avanzará regularmente con cada tick del reloj para colocar las botellas debajo del dispositivo de llenado, representado por la clase Filler. Para saber si una botella está en posición para ser rellenada, simulamos un sensor mediante la clase BottlePositionDetector, que comprobará si la posición de llenado y la de alguna botella coincide. Tras esto, la clase BottleAttacher se encargará de fijar la botella al dispositivo de llenado, causando la detención de la cinta hasta el desacople. Un sensor simulado por la clase PhotoeyeDetector determinará si la botella está llena, comprobando si su contenido supera un cierto límite algo menor que la capacidad de la botella. Tras el llenado, la botella se desacoplará y la cinta proseguirá su movimiento.

```
class Filler < ActiveObject
  operations
    fill()
    begin
     self.bottletofill.content := self.bottletofill.content + 2.0
    end
    hasBottleToFill(): Boolean = self.bottletofill->size()>0
    token()
    begin
      if  self.sensor.isBottleNotFull() and self.hasBottleToFill()
      then self.fill()
      else if not self.sensor.isBottleNotFull()
          then self.bottletofill.isFull:=true
          end
      end
    end
end
-----------------------------
class PhotoeyeDetector
  attributes
    threshold: Real init=20.0
    operations
      isBottleNotFull() : Boolean = (self.bottletodetect.content<=self.threshold)
end
-----------------------------
class BottlePositionDetector
  attributes
           ReadyToFillX: Real init=20.0
           ReadyToFillY: Real init=0.0
           ReadyToFillZ: Real init=0.0
  operations
    check(b: Bottle): Boolean =
b.isAt(self.ReadyToFillX,self.ReadyToFillY,self.ReadyToFillZ) and
 b.fillerforbottle->size()=0
end
-------------------------------
class BottleAttacher < TimedObject
   operations
   attachBottle(b: Bottle)
   begin
     insert(b, self.filler) into BottleToFill;
     insert(b, self.filler.sensor) into BottleToDetect;
     b.tag.startFillingTime:=self.clock.NOW;
   end

   detachBottle(b:Bottle)
   begin
           delete(b, self.filler) from BottleToFill;
           delete(b, self.filler.sensor) from BottleToDetect;
           delete(self.trayforattacher,b) from BottlesInTray;
           b.tag.stopFillingTime:=self.clock.NOW;
           b.tag.content:=b.content;
   end
 end
```

Cod.6 – Clases Filler, PhotoeyeDetector, BottlePositionDetector y BottleAttacher

Al sobrepasar la máquina de llenado, las botellas salen de la parte del sistema que estamos simulando Con el fin de no ocupar excesiva memoria con las botellas y tags que han finalizado su ciclo de vida, creamos una clase BottleProcesser, que recoge los distintos datos de los tags de las botellas y los utiliza para crear estadísticas en la clase StatisticsLog, como el número de objetos procesados, la media de contenido de las botellas, la media de tiempo de llenado o la media de tiempo de vida. Posteriormente, destruye los objetos ya procesados.

```
class BottleProcesser <TimedObject
  attributes
    statistics: StatisticsLog
  operations
  token()
  begin
    for b in Bottle.allInstances do
      if b.tag.readyForProcessing = true
      then            b.tag.endProcessTime:=self.clock.NOW;
            self.statistics.addValuesMean(self.clock.NOW-
b.tag.creationTime,b.tag.content, b.tag.stopFillingTime-b.tag.startFillingTime);
            destroy(b.tag); destroy(b);
            end
            end
  end
end
------------------------------
class StatisticsLog
  attributes
    TimeElapsedMean: Real init: 0
    AmountFilledMean: Real init: 0
    TimeFillingMean:Real init: 0
    ItemsProcessed: Integer init:0
    BottlesLost: Integer init: 0
  operations
    addValuesMean(timeElapsed: Real, amountFilled:Real, timeFilling: Real)
    begin
      self.ItemsProcessed := self.ItemsProcessed+1;
      self.AmountFilledMean := self.AmountFilledMean+((amountFilled-
self.AmountFilledMean)/self.ItemsProcessed);
      self.TimeElapsedMean:=self.TimeElapsedMean+((timeElapsed-
self.TimeElapsedMean)/ self.ItemsProcessed);
      self.TimeFillingMean:=self.TimeFillingMean+
            ((timeFilling-self.TimeFillingMean)/ self.ItemsProcessed);
    end
    addLostBottle()
    begin
     self.BottlesLost:=self.BottlesLost+1;
    end
end
```

Cod.7 – Clases BottleProcesser y StatisticsLog

Para terminar, necesitaremos las diferentes asociaciones que relacionan unas clases con otras, con el fin de cohesionar nuestro sistema. Las asociaciones necesarias son las siguientes:

```
association BottleToFill between
  Bottle [0..1] role bottletofill
  Filler [0..1] role fillerforbottle
end
------------------------------
association BottleToDetect between
  Bottle [0..1] role bottletodetect
  PhotoeyeDetector [0..1] role detectorforbottle
end
------------------------------
association FillerSensor between
  Filler[0..1] role filler
  PhotoeyeDetector[0..1] role sensor
end
---------------------------------
association AoControl between
  Control[1] role control
  ActiveObject [0..*] role controlledobject
end
--------------------------------
association BottlesInTray between
  MotorizedTray[0..1] role tray
  Bottle[0..*] role bottles ordered
end
--------------------------------
association TrayBottleProvider between
  BottleProvider[0..1] role provider
  MotorizedTray[0..1] role providedtray
end
---------------------------------
association AttacherFromTray between
  BottleAttacher[0..1] role attacher
  MotorizedTray[0..1] role trayforattacher
end
---------------------------------
association TraySensor between
  MotorizedTray[0..1] role traymonitored
  BottlePositionDetector[0..1] role positiondetector
end
---------------------------------
association AttacherFiller between
  BottleAttacher[0..1] role attacher
  Filler[0..1] role filler
end
---------------------------------
association ProcessorForTray between
  MotorizedTray[0..1] role trayforprocesser
  BottleProcesser[0..1] role processerfortray
end
```

Cod.8 – Asociaciones de BottleFiller

Si compilamos el sistema descrito en USE, podremos visualizar el diagrama de clases resultante (fig.1).

Lo siguiente es preparar un archivo .soil para realizar pruebas sobre el sistema. Dicho archivo contendrá los comandos pertinentes para la creación de instancias de las diferentes clases implicadas en el funcionamiento del sistema, así como las relaciones entre ellas por medio de asociaciones. También incluirá inicialización de variables en caso de necesitarlo. el .soil generado es el siguiente:

```
!create c:Control

!create f: Filler

!create clock:Clock

!create mt:MotorizedTray

!create ps: PhotoeyeDetector

!create bp: BottleProvider

!create ba: BottleAttacher

!create sensor: BottlePositionDetector

!create pr: BottleProcesser

!create st: StatisticsLog

!f.statistics:=st
!pr.statistics:=st
!mt.clock:=clock
!ba.clock:=clock
!bp.clock:=clock
!pr.clock:=clock
!bp.processingTime:=3;
!mt.processingTime:=2;

!insert(ba,f) into AttacherFiller;
!insert(bp,mt) into TrayBottleProvider;
!insert(f,ps) into FillerSensor;
!insert(c,bp) into AoControl;
!insert(c,mt) into AoControl;
!insert(c,f) into AoControl;
!insert(ba,mt) into AttacherFromTray;
!insert(mt,sensor) into TraySensor;
!insert(c,clock) into AoControl;
!insert(mt,pr) into ProcessorForTray;
!insert(c,pr) into AoControl;
```

Cod.9 – Fichero .soil para la incialización de un sistema BottleFiller

Al ejecutar este archivo, se genera un diagrama de objetos completamente funcional, listo para realizar pruebas sobre él (Fig.2).

Debido al control centralizado de nuestro sistema, solo tenemos que ejecutar la operación control(n: Integer), de la clase con el mismo nombre, para que el sistema empiece a funcionar de manera automática. El parámetro n representa el número de ciclos (ticks del reloj) que queremos que se realicen hasta la detención del sistema.
El diagrama de objetos actualiza en tiempo real todos los cambios recibidos sobre sí mismo debido a las operaciones realizadas durante la ejecución.

Lo primero que haremos será ejecutar el sistema un gran número de ciclos (100000, por ejemplo) para comprobar su correcto funcionamiento.

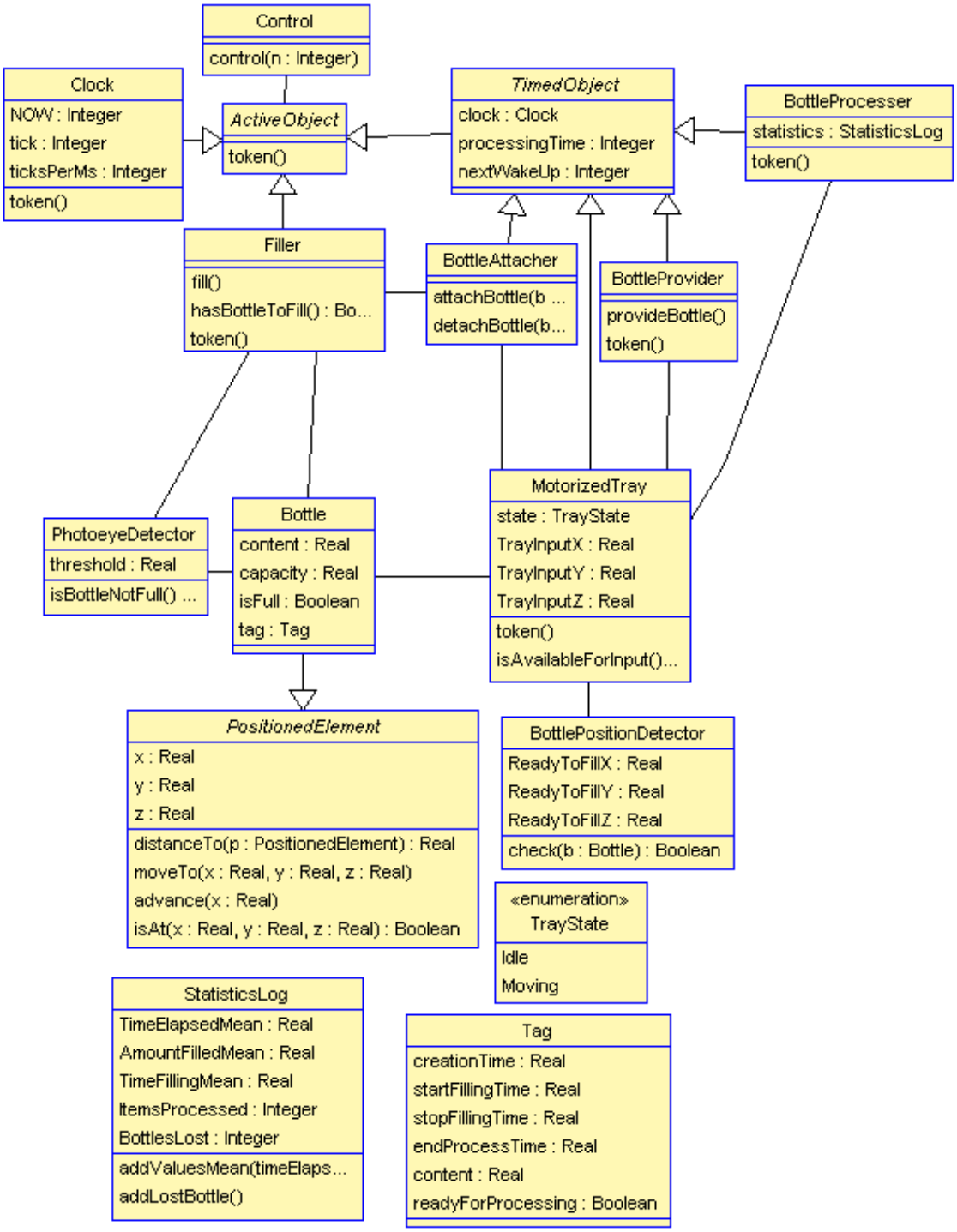

Fig. 1 - Diagrama de clases del modelo BottleFiller

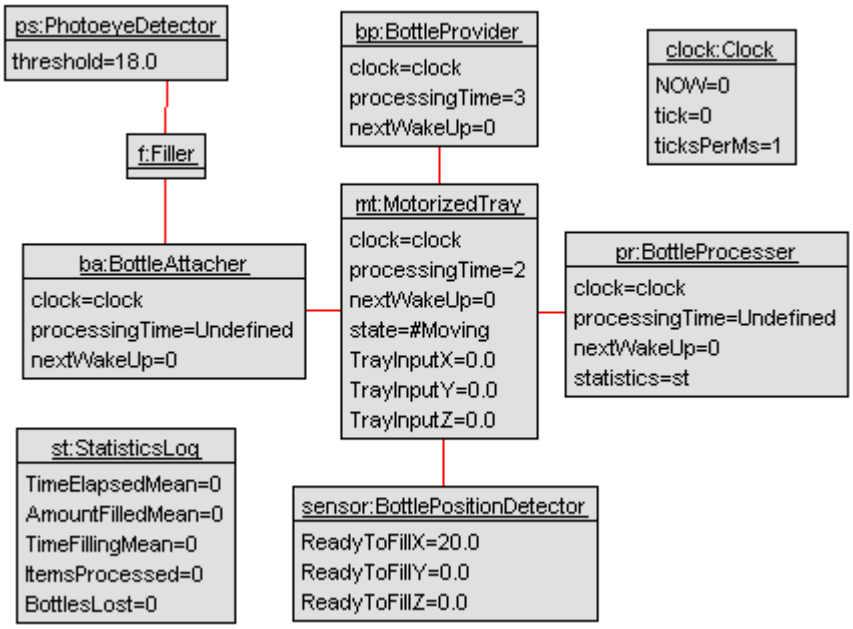

Fig. 2 - Diagrama de objetos de ejemplo para BottleFiller sin incertidumbre

Tras la ejecución, podemos comprobar las distintas estadísticas del sistema si comprobamos los distintos valores almacenados en la instancia de StatisticsLog.

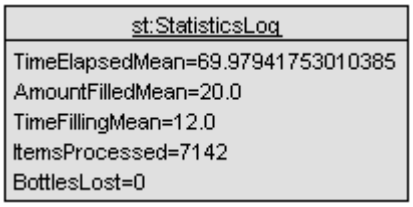

Fig. 3 - StatisticsLog de BottleFiller tras 100000 ticks del reloj (sin incertidumbre)

La ejecución, tal y como esperábamos, es completada sin errores. Sin embargo, si miramos los resultados, podemos llegar a la conclusión de que los datos recogidos nos sirven de poco rápidamente. Debido al carácter determinista de todo nuestro sistema, el tiempo de llenado y la cantidad de líquido en las botellas es siempre el mismo, algo que nunca podrá ocurrir en la realidad debido a la imperfección inherente a los sistemas de la vida real. Además, ninguna botella se pierde jamás por el desgaste de la cinta mecánica o el mal posicionamiento de ésta en la cinta. Por tanto, podemos concluir que las pruebas realizadas sobre el modelo actual no son lo suficientemente sólidas como para determinar que nuestro funcionará de manera fiable en el mundo real.

3.2 La introducción de incertidumbre

Una vez expuesta la falta de consistencia con la realidad de un modelo de carácter determinista, lo siguiente es estudiar cómo podemos adaptarlo para que sea más fidedigno al comportamiento real de una cadena de procesado. Para ello, lo mejor que podemos hacer es cambiar los valores fijos por valores con incertidumbre en aquellos parámetros del sistema sujetos a errores físicos.

Para este fin debemos realizar un estudio del sistema clase a clase, repasando todos los valores que pueden ser medidos erróneamente en el sistema real, y teniendo cuidado de no estropear valores que no requieran de dicha incertidumbre.

Repasando el modelo en el mismo orden en el que lo hemos presentado, el primer parámetro candidato a presentar incertidumbre es la posición (x, y, z) de cualquier objeto de la clase PositionedElement. En nuestro sistema, el único objeto que extiende esta clase es la clase Bottle. Si repasamos el ciclo de vida de una botella, ésta es colocada en la cinta transportadora por un brazo externo en una posición dada, y posteriormente llevada por la cinta hacia el objeto Filler. En un sistema real, al posicionar la botella en la cinta, esta puede quedar ligeramente descentrada con respecto a la posición dada como base. Además, el uso continuado de la cinta puede llevar a un desgaste del mecanismo que haga que avance algo más o menos de lo indicado por software. Por tanto, podemos concluir que la posición de una botella es una información que requiere de incertidumbre. Dicho cambio no sólo afecta a las variables mencionadas, sino también a todas las operaciones que la usan, requiriendo la refactorización de prácticamente todas ellas, así como el añadido de otras que serán explicadas cuando nos adentremos más en la transformación.

```
abstract class PositionedElement
  attributes
        x : UReal init: UReal(0.0, 0.0)
        y : UReal init: UReal(0.0, 0.0)
        z : UReal init: UReal(0.0, 0.0)
  operations
        distanceTo(p:PositionedElement) : UReal = ((self.x-p.x).abs()+(self.y-
p.y).abs()+(self.z-p.z).abs())

    moveTo(xx:UReal, yy:UReal, zz:UReal)
        begin
    declare M : Real;
  M := ((self.x-xx).abs()+(self.y-yy).abs()+(self.z-zz).abs()).toReal()/100.0;
  self.x := xx+UReal(0.0,0.0).setUncertainty((1.0+M)*self.x.uncertainty());
  self.y := yy+UReal(0.0,0.0).setUncertainty((1.0+M)*self.y.uncertainty());
  self.z := zz+UReal(0.0,0.0).setUncertainty((1.0+M)*self.z.uncertainty());
        end
```

```
        advance(x:UReal)
   begin
    self.moveTo(self.x+x,self.y,self.z);
   end

        isAt(x:UReal, y:UReal, z:UReal) :Boolean =
                self.sameAs(self.x,x) and self.sameAs(self.y,y) and
self.sameAs(self.z,z)

        sameAs(x:UReal,y:UReal):Boolean =
                ((y.value()-y.uncertainty()<=x.value()) and
(y.value()+y.uncertainty()>=x.value())) or
                ((x.value()-x.uncertainty()<=y.value()) and
(x.value()+x.uncertainty()>=y.value()))

   isInPositionToGrasp(x:UReal, y:UReal, z:UReal) :Boolean =
     self.isInGraspRangeAxis(self.x,x) and self.isInGraspRangeAxis(self.y,y) and
self.isInGraspRangeAxis(self.z,z)


   isInGraspRangeAxis(x: UReal, axisboundaries: UReal):Boolean =
       (axisboundaries.value()=x.value()) and
(axisboundaries.uncertainty()>=x.uncertainty())
end
```

Cod.10 – Clase PositionedElement modificada

En segundo lugar, tenemos la clase Clock. Si lo pensamos detenidamente, no tiene sentido que esta clase presente incertidumbre, puesto que representa simplemente un reloj que envía una señal cada segundo. El único fallo que podría presentar un reloj en el sistema real es su detención, que causaría un error fatal del sistema. DE la misma manera, en la clase TimedObject, tampoco tiene sentido que los parámetros processingTime y nextWakeUp presenten incertidumbre.

La siguiente clase a revisar es la clase Bottle. Ésta, además de heredar la incertidumbre en su posición de la clase PositionedElement, tiene otras variables que son claras candidatas a tener incertidumbre asociada. En concreto, si pensamos en el contenido de la botella, no parece plausible que en el mundo real podamos rellenar 7142 botellas (Fig.3) con la misma cantidad exacta de líquido, sin una desviación de centésimas de mililitro siquiera. Por ello, podemos asumir que la variable content requerirá la introducción de una cierta incertidumbre asociada. Por otra parte, las variables capacidad y isFull, puesto que representan tanto un límite como una cualidad del objeto, no deberían presentar incertidumbre asociada. La variable isFull, no obstante, verá modificada la forma en que es calculada debido a que ahora la fórmula incluirá números de tipo UReal. Dicha fórmula será explicada más adelante. La clase Tag, por otro lado, no recibirá incertidumbre alguna en sus variables aparte de la variable content, paralela a la variable del mismo nombre de la clase Bottle, ya que éstas representan simplemente marcas de tiempo relativas a distintos eventos ocurridos sobre la botella.

```
class Bottle < PositionedElement
  attributes
    content:UReal init= UReal(0.0,0.0)
    capacity: Real init=20.0
    isFull: Boolean init=false
    tag:Tag
end
-------------------------------
class Tag
  attributes
  creationTime: Real
  startFillingTime: Real
  stopFillingTime: Real
  endProcessTime: Real
  content: UReal
  readyForProcessing: Boolean init:false
end
```

Cod.11 – Clases Bottle y Tag modificadas.

Puesto que la posición de las botellas ha sido modificada para introducir incertidumbre, es lógico que todo valor que repercuta en el movimiento de las mismas lleve reflejada a su vez esa incertidumbre asociada. En la clase TrayState, concretamente, añadiremos cierta incertidumbre asociada al movimiento de la cinta en cada tick del reloj. Dicha incertidumbre será igual a un valor aleatorio mediante distribución exponencial con un umbral que podrá ser variable, y que se irá degradando con el paso del tiempo para simular el desgaste de la máquina. Las coordenadas de introducción de botellas, a su vez, se convertirán también en UReal aunque con incertidumbre igual a 0, con el fin de facilitar las operaciones aritméticas y comparativas con otras coordenadas. La clase BottleProvider, sin embargo, puesto que se encarga únicamente de crear las instancias de botellas y colocarlas en las coordenadas dadas por la clase TrayState, no se verá afectada.

La clase Filler también se verá afectada en gran medida, puesto que su funcionamiento se basa en el flujo continuado de una sustancia medible, que no se detiene hasta que un sensor de una señal concreta. Puesto que dicha medida puede estar sujeta a fallos por diversas razones (desgaste del grifo, malfuncionamiento del sensor…), y que el valor que representa el contenido de la botella también posee cierta incertidumbre, es lógico concluir que el llenado de la misma también implique esta cualidad. al igual que con el funcionamiento de la cinta transportadora, cada tick del reloj se rellenará la botella una cierta cantidad, que será representada mediante una distribución con poca variación para simular un flujo constante pero no exacto de líquido. además, existirá una incertidumbre que podrá ser variable (en función de la calidad del grifo, por ejemplo) y que se irá desgastando por el uso, pudiendo ser reiniciada si se desajusta hasta un punto inaceptable (representando el cambio de la pieza reguladora).

Otro cambio notable de dicha clase reside en las condiciones de las sentencias if, puesto que ahora comparan número de tipo UReal y generan expresiones de tipo UBoolean, no aptas para dichas sentencias. Lo que haremos en este caso es establecer un límite mínimo de probabilidad a partir del cual consideramos la expresión como verdadera (en nuestro caso 90%).

Ambos sensores, a su vez, será adaptados para trabajar con valores con incertidumbre, ya que controlan posición y contenido de las botellas.

```
enum TrayState {Idle, Moving}

class MotorizedTray < TimedObject
  attributes
    state:TrayState init:#Moving
    TrayInputX: UReal init=UReal(0.0, 0.0)
    TrayInputY: UReal init=UReal(0.0, 0.0)
    TrayInputZ: UReal init=UReal(0.0,0.0)
    uncertainty: Real init=0.001
    degradation: Real init=0.0000001

  operations
  token()
    begin
    declare u: Real;
    u:=self.uncertainty.expDistr();
      if self.clock.NOW>=self.nextWakeUp
      then for b in self.bottles do
        if self.positiondetector.check(b)
        then self.attacher.attachBottle(b); self.state:=#Idle
        else
          if self.state=#Moving
          then b.advance(UReal(5.0,0.0).setUncertainty(u))
          else
            if b.fillerforbottle->size()>0 and b.isFull
            then self.attacher.detachBottle(b); self.state:=#Moving;
b.advance(UReal(5.0,0.0).setUncertainty(u)); b.tag.readyForProcessing:=true;
            end
          end
        end
        end; self.nextWakeUp:=self.nextWakeUp+self.processingTime;
self.uncertainty:=self.uncertainty+self.degradation;
      end
  end
    isAvailableForInput() : Boolean = PositionedElement.allInstances-
>select(p|p.oclIsKindOf(Bottle) and
p.isAt(self.TrayInputX,self.TrayInputY,self.TrayInputZ))->isEmpty
end
class PhotoeyeDetector
  attributes
    threshold: UReal init=UReal(18.0, 0.0)
    operations
      isBottleNotFull() : UBoolean = (self.bottletodetect.content<self.threshold)
end
```

```
class BottlePositionDetector
  attributes
  ReadyToFillX: UReal init=UReal(20.0, 0.1)
  ReadyToFillY: UReal init=UReal(0.0, 0.1)
  ReadyToFillZ: UReal init=UReal(0.0, 0.1)
  operations
    check(b: Bottle): Boolean =
b.isInPositionToGrasp(self.ReadyToFillX,self.ReadyToFillY,self.ReadyToFillZ) and
b.fillerforbottle->size()=0
end
class Filler < ActiveObject
  attributes
    uncertaintybase: Real init: 0.05
    uncertainty: Real init: self.uncertaintybase
    uncertaintythreshold: Real init: 0.1
    degradation: Real init: 0.0002
            calibrations:Integer init: 0
    statistics: StatisticsLog
  operations
    fill()
    begin
     if self.uncertainty>self.uncertaintythreshold then
                    self.uncertainty:=self.uncertaintybase;
                    self.calibrations:=self.calibrations+1;
    else self.bottletofill.content := self.bottletofill.content +
           UReal(1.expDistr(), uncertainty);
    self.uncertainty:=self.uncertainty+self.degradation;
     end
    end
    hasBottleToFill(): Boolean = self.bottletofill->size()>0
    token()
    begin
      if  (self.sensor.isBottleNotFull().uncertainty()>0.9) and self.hasBottleToFill()
      then self.fill()
      else if not (self.sensor.isBottleNotFull().uncertainty()>0.9)
         then self.bottletofill.isFull:=true;
self.statistics.calibrationsNeeded(self.calibrations);
         end
      end
    end
end
```

Cod.12 – Clases Filler, MotorizedTray, BottlePositionDetector y PhotoeyeDetector modificadas.

Con la adición de incertidumbre al sistema, existe la posibilidad de que el sensor que comprueba si hay botellas en posición para ser llenadas no detecte cierta botella por poder encontrarse en una zona demasiado alejada de la posición adecuada. Dada esta posibilidad de perder botellas, debemos adecuar nuestra clase BottleProcesser para que encuentre botellas que ya han pasado de largo el sistema de llenado sin ser rellenadas y, por tanto, se han perdido. BottleProcesser encontrará dichas botellas mediante un umbral de tiempo máximo de vida, tras el cual se considera que dicha botella, si no ha empezado a llenarse, es porque no se encuentra colocada adecuadamente en la cinta. Nuestra clase StatisticsLog, a su vez, deberá adaptarse para poder recoger el

número de botellas perdidas en la estadística, así como poder trabaja con números con incertidumbre.

```
class BottleProcesser <TimedObject
  attributes
    statistics: StatisticsLog
    threshold: Integer init:120
  operations
  token()
  begin
    for b in Bottle.allInstances do
      if b.tag.readyForProcessing = true
      then            b.tag.endProcessTime:=self.clock.NOW;
           self.statistics.addValuesMean(self.clock.NOW-
b.tag.creationTime,b.tag.content, b.tag.stopFillingTime-b.tag.startFillingTime);
           destroy(b.tag); destroy(b);
      else if self.clock.NOW - b.tag.creationTime > self.threshold and
           b.fillerforbottle ->size()=0
        then self.statistics.addLostBottle(); destroy(b.tag); destroy(b);
                    end
           end
        end
  end
end
--------------------------------
class StatisticsLog
  attributes
    TimeElapsedMean: Real init: 0
    AmountFilledMean: UReal init: UReal(0.0,0.0)
    TimeFillingMean:Real init: 0
    ItemsProcessed: Integer init:0
    Calibrations: Integer init: 0
    BottlesLost: Integer init: 0
  operations
    addValuesMean(timeElapsed: Real, amountFilled:UReal, timeFilling: Real)
    begin
      self.ItemsProcessed := self.ItemsProcessed+1;
      self.AmountFilledMean :=
self.AmountFilledMean+((amountFilled-self.AmountFilledMean)/self.ItemsProcessed);
      self.TimeElapsedMean:=
self.TimeElapsedMean+((timeElapsed-self.TimeElapsedMean)/ self.ItemsProcessed);
      self.TimeFillingMean:=
           self.TimeFillingMean+((timeFilling-self.TimeFillingMean)/ self.ItemsProcessed);
    end
    calibrationsNeeded(calibrations: Integer)
    begin
    self.Calibrations:=calibrations;
    end
    addLostBottle()
    begin
     self.BottlesLost:=self.BottlesLost+1;
    end
end
```

Cod.13 – Clases BottleProcesser y StatisticsLog modificadas.

Las asociaciones, puesto que no tienen tipo, no se verán afectadas por la introducción de incertidumbre en el sistema.

3.3 Pruebas con incertidumbre

Ahora que ya tenemos nuestro sistema con incertidumbre añadida en todas las variables y operaciones que así lo requerían, lo primero que debemos hacer es volver a probar el sistema igualando todos los valores de incertidumbre de UReal a 0, y los de UBoolean, si se requiriera, a 1. De esta manera estaremos probando el sistema de nuevo sin incertidumbre, y podremos cerciorarnos de que el funcionamiento básico del sistema no se ha visto alterado por los cambios realizados. También será necesario para esta prueba cambia aquellos valores que ahora se calculaban con una distribución a valores fijos, con el fin de ver con más claridad si el funcionamiento sigue siendo el mismo.

Con estos cambios en cuenta, la ejecución del sistema durante 50.000 ticks del reloj da el siguiente resultado:

	Media de tiempo de procesado (segundos)	Media de cantidad de llenado (cl)	Media de tiempo de llenado (segundos)	Botellas procesadas	Calibraciones	Ticks del reloj	Porcentaje de producto defectuoso
Sin incertidumbre	109,89	18,00	20,00	2272	0	50000	0,00%

Fig. 4 - Prueba del sistema final sin incertidumbre.

El resultado obtenido es, además, determinista. Sin importar cuantas veces se ejecute el sistema, sin la introducción de incertidumbre el resultado de la ejecución no varía.

Una vez comprobado que el sistema, efectivamente, es determinista, procederemos a pensar con qué elementos del sistema real se corresponden los diferentes valores con incertidumbre, cuales son valores lógicos para ésta y cómo afectarían cambios en la calidad y precisión de estos componentes en la productividad del sistema. Si asumimos que un componente más preciso es siempre más caro, el objetivo de nuestras pruebas es conseguir el sistema más barato posible sin perjudicar de manera notable la productividad del mismo.

El primer componente que vamos a evaluar es la válvula o grifo que regula la salida de líquido, representado en nuestra clase Filler. Dicho grifo tendrá una precisión concreta, y se irá deteriorando poco a poco con el uso, hasta llegar a un punto en el que necesite ser recalibrado o reemplazado. Obviamente, un grifo más preciso y resistente implica un gasto mayor de dinero, por lo que nuestro objetivo es minimizar gastos sin afectar en medida de lo posible al tiempo y la calidad del producto.

Para medir cómo afecta la precisión del grifo comparamos cómo influye la variación de tanto la incertidumbre base como la velocidad de degradación en el llenado de las botellas durante 50000 ticks del reloj. Se considera que una botella producida es defectuosa si la cantidad de líquido contenido no alcanza los 18cl (9/10 de la botella) o sobrepasa los 20cl (desborda la botella).

	Tiempo de procesado medio (s)	cantidad de llenado media (cl)	tiempo de llenado medio (s)	Botellas procesadas	Calibraciones	Ticks del reloj	Porcentaje de producto defectuoso
0.0	109,89	18,00	20,00	2272	0	50000	0,00%
0.025	112,86	UReal(18.9231, 0.1019)	20,59	2212	548	50000	0,00%
0.05	112,36	UReal(18,7865, 0.1138)	20,49	2222	208	50000	0,00%
0.075	112,27	UReal(18.7071,0.1308)	20,48	2223	129	50000	0,00%
0.1(base)	111,53	UReal(18.6661, 0.1287)	20,33	2238	93	50000	0,00%
0.15	110,79	UReal(18.5529,0.1629)	20,18	2253	60	50000	0,03%
0.2	110,08	UReal(18.3896,0.1608)	20,04	2267	44	50000	0,70%
0.25	109,51	UReal(18.2462,0.1771)	19,92	2280	34	50000	8,20%

Fig. 5 - Estadísticas de funcionamiento del sistema variando la incertidumbre base.

Como podemos observar, el cambio en la mínima precisión para nuestro grifo afecta principalmente al número de calibraciones necesarias (a menor precisión, mayor el número de calibraciones y, por tanto, más tiempo perdido) y a la dispersión de la cantidad de líquido en las botellas (lo cual indirectamente causa que las botellas se llenen más deprisa). Vemos además que, precisamente por el efecto complementario que producen estos dos factores sobre el tiempo de llenado, éste no varía uniformemente a lo largo de las distintas ejecuciones, si no que se mantiene razonablemente estable.

El dato más importante que nos devuelve esta serie de simulaciones es que una precisión mínima demasiado alta puede causar que el sistema produzca un número inaceptable de botellas defectuosas. En conclusión, se puede prescindir de cierta precisión sin afectar al funcionamiento del sistema pero, con los requisitos actuales, no es recomendable que dicha precisión baje de 0.15ml.

	Tiempo de procesado medio (s)	cantidad de llenado media (cl)	tiempo de llenado medio (s)	Botellas procesadas	Calibraciones	Ticks del reloj	Porcentaje de producto defectuoso
0.0	109,89	18,00	20,00	2272,00	0	50000	0,00%
0.00005	111,00	UReal(18.6676, 0.0927)	20,20	2249,00	23	50000	0,00%
0.0001	111,87	UReal(18,6679, 0.1096)	20,39	2232,00	46	50000	0,00%
0.00015	111,10	UReal(18,6852, 0.1551)	20,24	2247,00	69	50000	0,00%
0.0002 (base)	110,61	UReal(18.6411, 0.1424)	20,15	2256,00	93	50000	0,00%
0.0003	111,10	UReal(18.6605, 0.1533)	20,24	2247,00	139	50000	0,00%
0.0005	111,37	UReal(18.6622, 0.1937)	20,30	2242,00	232	50000	0,03%
0.00075	112,15	UReal(18.6606, 0.2142)	20,45	2226,00	348	50000	0,10%
0.001	111,94	UReal(18.6929, 0.239)	20,41	2230,00	463	50000	0,19%

Fig. 6 - Estadísticas de funcionamiento del sistema variando la velocidad de degradación.

Podemos observar una tendencia similar en las diferentes ejecuciones variando la velocidad de degradación del grifo. El tiempo se mantiene razonablemente estable, pero la dispersión de la cantidad de líquido y el número de calibraciones requeridas crece a la vez que la velocidad de degradación. Asimismo, una velocidad de degradación excesiva hará que la cantidad de líquido que llega a las botellas sea más incierto durante más tiempo, llevando el número de botellas defectuosas a un nivel significativo.

El dato más relevante de esta tabla, sin embargo, viene al compararla con la anterior. Podemos observar que el número de calibraciones necesarias en el sistema se multiplica 5 veces más rápido al aumentar la velocidad de

degradación que el aumentar el límite de degradación, haciendo a este primero mucho más costoso en términos de tiempo. Esto se traduce en un acusado aumento de los tiempos medios de llenado en la parte final de la segunda tabla en comparación con la tendencia de la primera.
No obstante, la tendencia del sistema al llegar a valores de degradación más altos no debe preocuparnos puesto que, como refleja la tabla, cualquier valor de degradación por encima de 0.0003cl produce una cantidad inaceptable de botellas defectuosas.

Una vez elegido el grifo, podemos también tratar de averiguar cuanto podemos retrasar la calibración del mismo sin afectar a la calidad del producto. Para ello, observamos como el sistema se comporta ante un mayor límite de degradación.

	Tiempo de procesado medio (s)	cantidad de llenado media (cl)	tiempo de llenado medio (s)	Botellas procesadas	Calibraciones	Ticks del reloj	Porcentaje de producto defectuoso
0.0	109,89	18,00	20,00	2272,00	0	50000	0,00%
0.0025	111,23	UReal(18.717, 0.1019)	20,27	2244,00	86	50000	0,00%
0.005	111,67	UReal(18.6876,0.1114)	20,36	2235,00	88	50000	0,00%
0.0075	111,18	UReal(18.7174,0.1335)	20,26	2245,00	90	50000	0,00%
0.01(base)	110,80	UReal(18.6633, 0.1375)	20,18	2253,00	93	50000	0,00%
0.015	111,02	UReal(18.6721 ,0.1489)	20,23	2248,00	98	50000	0,00%
0.02	111,08	UReal(18.6562, 0.234)	20,24	2247,00	107	50000	0,26%
0.03	110,82	UReal(18.6645 ,0.2683)	20,19	2253,00	119	50000	0,66%
0.05	110,74	UReal(18.5558, 0.4063)	20,17	2254,00	167	50000	8,57%

Fig. 7 - Estadísticas de funcionamiento del sistema variando el límite de degradación.

Tal y como esperábamos, un aumento en el límite de degradación produce una reducción significativa en el tiempo de llenado y procesado, al disminuir drásticamente el número de calibraciones necesarias, así como aumentar ligeramente la dispersión de llenado. No obstante, un aumento excesivo del límite de degradación lleva a la pérdida de calidad del producto rápidamente, por lo que deberemos tener cuidado de no aumentar éste por encima de 1ml.

Una vez recogidos todos estos datos, resulta sencillo determinar los límites en cuanto a calidad/precio que queremos establecer a la hora de escoger el tipo de grifo que utilizaremos para nuestro sistema en la vida real, realizando pruebas en un entorno seguro y fiable, puesto que tiene en cuenta la dispersión inherente a las medidas reales.

El otro elemento a probar de nuestro sistema es la cinta transportadora. Para la elección de la cinta tendremos en cuenta solamente la incertidumbre base de su movimiento, ya que la variación en los valores de degradación será demasiado pequeña como para influir de manera significativa en los resultados (menos de una millonésima parte de un centímetro por ciclo).

A la hora de probar la cinta transportadora, además, no tendremos en cuenta posibles recalibraciones ya que, a diferencia de con el grifo, sustituir la cinta supondría desmontar la cadena de montaje al completo, puesto que a ella van asociados todos los componentes.

El producto defectuoso es, en este caso, aquel que no es rellenado al encontrarse en una posición incorrecta.

	Media de tiempo de procesado (segundos)	Botellas procesadas	Botellas perdidas	Ticks del reloj	Porcentaje de producto defectuoso
Sin incertidumbre	109,89	2272,00	0,00	50000	0,00%
0.0000001	112,65	1110,00	1,00	25000	0,09%
0.00000025	110,76	1126,00	0,00	25000	0,00%
0.0000005	111,64	1113,00	4,00	25000	0,36%
0.000001	106,61	1123,00	59,00	25000	4,99%

Fig. 8 - Estadísticas de ejecución con variación de la incertidumbre base de la cinta transportadora.

Como podemos observar, la subida de la incertidumbre base a un nanómetro hace que en un periodo tan corto de tiempo como 50000 segundos se empiece a perder una gran cantidad de botellas. Por otra parte, vemos que la pérdida de botellas se puede dar en cualquier caso, ya que incluso con la incertidumbre más baja posible perdemos una botella en el tiempo de prueba. sin embargo, se puede observar que existe una clara correlación entre la incertidumbre base del movimiento de la cinta transportadora y el número de botellas perdidas.

Aunque 0,25 nanómetros parece una incertidumbre base aceptable para el correcto funcionamiento del sistema, puesto que para la recalibración de la cinta debemos parar por completo y reiniciar el sistema, cuanto menor sea la incertidumbre base de la cinta, más tiempo podrá estar la misma funcionando sin entorpecer la productividad de nuestro sistema.

4

Desarrollo del segundo modelo: Productor-Consumidor

4.1 El modelo

Nuestro segundo modelo a construir simula un sistema productor-consumidor, en el cual objetos de diferente tamaño y peso producidos por cierta máquina es colocado en una bandeja de producción a la espera de ser recogido por un brazo mecánico. Dicho brazo transportará nuestro recién creado objeto hasta una bandeja de consumición, a la cual tendrá acceso el consumidor para procesar dicho objeto.

Las bases del sistema será las mismas que con el anteriormente desarrollado BottleFiller. Una clase Control actuará de "cerebro", dando paso al funcionamiento de los distintos ActiveObject de manera ordenada (Round Robin). Dichos ActiveObject adaptarán su funcionamiento al estado del sistema en el momento de su activación.

De la misma manera, existirán objetos que requieran de un tiempo de procesado entre acción y acción. Éstos serán representados mediante la clase TimedObject, que extiende a la clase ActiveObject pero lleva asociado un objeto de la clase Clock, utilizado para controlar el tiempo de procesado.

Además, al igual que en nuestro primer modelo, existirán objetos cuya posición sea información vital para el funcionamiento del sistema. Dicha información se verá reflejada extendiendo desde éstos la clase PositionedElement.

Como última base del sistema a simular, existen ciertos parámetros físicos que son comunes a todas las clases del sistema, como los límites físicos del brazo robótico, los tiempos de cierre y apertura de su pinza y su velocidad, tanto

horizontal como vertical. Dichos datos se verán reflejados en la clase Dashboard, que actuará como fuente de información para el sistema.

```
class Dashboard
attributes
  xMax:Real init: 100.0 -- max difference between gantry and base
  yMax:Real init: 100.0 -- max difference between gantry and base
  zMax:Real init: 100.0 -- max difference between gantry and base
  wMax:Real init: 100.0 -- max weight of items a gantry is able to carry
  hSpeed:Real init: 1.0 -- horizontal speed
  vSpeed:Real init: 1.0 -- vertical speed
  gTime:Integer init: 1 -- grasping time
  dTime:Integer init: 1 -- dropping time
end
```

Cod.14 – Clase Dashboard.

Nuestro proceso a simular da comienzo con un productor creando objetos que serán procesados al final por un consumidor. El productor está representado en nuestro sistema mediante la clase SimpleProducer, extendiendo de una clase genérica Producer a fin de hacer el modelo escalable.

```
abstract class Producer < TimedObject
attributes
  producedItems:Integer init:0
  toProduce: Integer init:100
  processingTime:Integer init:2 -- in ms
end
----------------------------------------
enum ProducerState {ProducerReady, Producing}
class SimpleProducer < Producer
attributes
          state:ProducerState init:#ProducerReady
operations
    produce()
      begin
        declare i: Item;
        i:=new Item;
                    i.weight:=1.0+(15.0).rand();
        self.outTray.put(i);
              self.producedItems:=self.producedItems+1;
      end
            pre  OutTrayNotFull: not self.outTray.isFull()
      post ElementAdded: self.outTray.items->excluding(self.outTray.items->last()) =
self.outTray.items@pre
            post CounterIncremented: self.producedItems = self.producedItems@pre + 1
    token()
      begin
        if self.state=#ProducerReady and not self.outTray.isFull() and
(self.producedItems<self.toProduce) then
                              self.produce(); self.state:=#Producing;
self.nextWakeUp:=self.clock.NOW+self.processingTime;
                    else if self.state=#Producing then
                                        if self.clock.NOW >= self.nextWakeUp then
                                                  self.state:=#ProducerReady;
                                        end end end end end
```

Cod.15 – Clases Producer y SimpleProducer.

Dicho productor crea un objeto de la clase Item, y lo asocia a su bandeja de salida, de la clase Tray, invocando una operación de la misma. Cada Item tendrá un peso asociado, calculado de momento de manera aleatoria en el momento de la creación.

Las bandejas, tanto del productor como del consumidor, tendrán una capacidad predeterminada que, de ser alcanzada, bloquearán esa parte del sistema hasta que se vacíen parcialmente. A fin de controlar adecuadamente el funcionamiento de la operación token de varias de los objetos activos, se crearán enumeraciones que representen los distintos posibles estados de las máquinas.

```
class Item
attributes
  weight:Real init: 1.0
end
class Tray < PositionedElement
  attributes
    capacity:Integer
          size:Integer derive = self.items->size()
  operations
    put(p:Item)
      begin
         insert( self, p ) into BufferedItems;
      end
      pre notFull: (self.items->size() < capacity)
      post ElementAdded: (self.items = self.items@pre->append(p))

    get():Item
      begin
        result:=self.items->at(1);
        delete( self, result) from BufferedItems;
      end
      pre notEmpty: self.items->size()>0
      post FirstElementRemoved:
        result = self.items@pre->at(1) and
        self.items@pre=self.items->prepend(result)

    size():Integer = self.items->size()

          isFull():Boolean = (self.size() = self.capacity)

          isEmpty():Boolean = self.size() = 0

          hasElements():Boolean = self.size() > 0

end
```

Cod.16 – Clases Item y Tray

El consumidor será representado en nuestro sistema mediante la clase Consumer. Dicha clase comprobará en cada ciclo su bandeja de entrada en busca de objetos listos para su procesado, y tratará uno de ellos (escogido por orden de llegada, FIFO) cada vez que despierte. Si no existen objetos en la bandeja de entrada al despertar, el consumidor pasará a estado de espera, ConsumerReady.

```
enum ConsumerState {ConsumerReady, Consuming}

class Consumer < TimedObject
  attributes
    consumedItems:Integer init:0
    processingTime:Integer init:2 -- in ms
          state: ConsumerState init:#ConsumerReady
  operations
    consume()
      begin
        declare i: Item;
        i:=self.inTray.get();
        destroy i ;
              self.consumedItems:=self.consumedItems+1;
      end
            pre  InTrayNotEmpty: not self.inTray.isEmpty()
      post ElementRemoved: self.inTray.items@pre=self.inTray.items-
>prepend(self.inTray.items@pre->at(1))
            post CounterIncremented: self.consumedItems =
                    self.consumedItems@pre + 1

    token()
      begin
        if self.state=#ConsumerReady and self.inTray.hasElements() then
                              self.consume(); self.state:=#Consuming;
self.nextWakeUp:=self.clock.NOW+self.processingTime;
                    else
                              if self.state=#Consuming then
                                        if self.clock.NOW >= self.nextWakeUp then
                                                  self.state:=#ConsumerReady;
                                        end
                              end
        end
      end

end
```

Cod.17 – Clase Consumer.

La parte principal y más compleja de nuestro sistema es la clase Gantry, que representa un brazo mecánico que trasladará los diferentes objetos de la bandeja asociada al productor a la bandeja del consumidor. Dicho brazo tendrá asociada una clase GantryBase, que no es más que la posición base (Home) del

brazo robótico, a la que volverá en caso de reinicio. El brazo tendrá dos modos, automático y manual, representados en un enum. En modo automático, el brazo realizará la acción requerida sin restricciones. En modo manual, el brazo robótico solo actuará si se encuentra en reposo en el momento de la petición. De estar realizando otra acción, ignorará el comando requerido. La operación token() tendrá un comportamiento diferente dependiendo de si el brazo se encuentra en ese momento en modo manual o no.

```
class GantryBase < PositionedElement
end
----------------------------------------
enum GantryState {Idle, PickingUp, WithItem, Releasing, -- states in #Automatic mode
                                  GoingUp, GoingDown, Travelling, GoingBackToBase,
Grasping, Dropping  -- states in #Manual mode
                                 }
enum GantryMode {Automatic, Manual}
----------------------------------------
class Gantry < TimedObject, PositionedElement
  attributes
    state:GantryState init:#Idle
    item:Item
          base:GantryBase
          movedItems:Integer init:0
          mode:GantryMode init:#Automatic
          processingTime:Integer init:1

  operations

          up(h:Real) -- relative coordinates
            begin
                  if self.mode=#Automatic then
                            self.moveTo(self.x,self.y,self.z+h);
                  else if self.state=#Idle then
                                    self.state:=#GoingUp;
                                    self.processingTime:=
          ((h/self.config.vSpeed)*(if self.item=null then 1.0 else
          self.item.weight/self.config.wMax endif)).round.toInteger;
                                    self.nextWakeUp:=self.clock.NOW+self.processingTime;
                                    self.moveTo(self.x,self.y,self.z+h);
                             end;
                  end;
            end
            pre hOk: (h >= 0)
            pre PositionNotTaken:
PositionedElement.allInstances->excluding(self)->select(p|p.oclIsKindOf(Gantry) and
p.isAt(self.x,self.y,self.z+h))->isEmpty
            post AtFinalPosition: (self.mode=#Automatic or (self.mode=#Manual and
self.state@pre=#Idle))implies self.isAt(self.x@pre,self.y@pre,self.z@pre+h)

            post NoOutOfBoundsTest: ((self.x-self.base.x).abs()<=self.config.xMax and
(self.y-self.base.y).abs()<=self.config.yMax and
(self.z-self.base.z).abs()<=self.config.zMax)
```

```
        down(h:Real) -- relative coordinates
          begin
                if self.mode=#Automatic then
                        self.moveTo(self.x,self.y,self.z-h);
                else if self.state=#Idle then
                                self.state:=#GoingDown;
                                self.processingTime:=
                        ((h/self.config.vSpeed)*(if self.item=null then 1.0 else
                         self.item.weight/self.config.wMax endif)).round.toInteger;
                                self.nextWakeUp:=self.clock.NOW+self.processingTime;
                                self.moveTo(self.x,self.y,self.z-h);
                         end
                end;
          end
          pre hOk: (h >= 0)
          pre PositionNotTaken:
PositionedElement.allInstances->excluding(self)->select(p|p.oclIsKindOf(Gantry) and
p.isAt(self.x,self.y,self.z-h))->isEmpty
          post AtFinalPosition: (self.mode=#Automatic or (self.mode=#Manual and
self.state@pre=#Idle))implies self.isAt(self.x@pre,self.y@pre,self.z@pre-h)

          post NoOutOfBoundsTest: ((self.x-self.base.x).abs()<=self.config.xMax and
(self.y-self.base.y).abs()<=self.config.yMax and
 (self.z-self.base.z).abs()<=self.config.zMax)

        travel(x:Real,y:Real)
          begin
                if self.mode=#Automatic then
                        self.moveTo(x,y,self.z);
                else if self.state=#Idle then -- if it is busy, it ignores the command
                                self.state:=#Travelling;
                                self.processingTime:=
((((self.x-x).abs+(self.y-y).abs)/self.config.hSpeed)*(if self.item=null then 1.0 else
self.item.weight/self.config.wMax endif)).round.toInteger();
                                self.nextWakeUp:=self.clock.NOW+self.processingTime;
                                self.moveTo(x,y,self.z);
                        end
                end;
    end
          pre PositionNotTaken
: PositionedElement.allInstances->excluding(self)->select(p|p.oclIsKindOf(Gantry) and
p.isAt(x,y,self.z))->isEmpty
          post AtFinalPosition: (self.mode=#Automatic or (self.mode=#Manual and
self.state@pre=#Idle))implies self.isAt(x,y,self.z@pre)
          post NoOutOfBoundsTest: ((self.x-self.base.x).abs()<=self.config.xMax and
 (self.y-self.base.y).abs()<=self.config.yMax and
 (self.z-self.base.z).abs()<=self.config.zMax)

        grasp(t:Tray)
          begin
                if self.mode=#Automatic then
                        self.item:=t.get();
                else if self.state=#Idle then -- if it is busy, it ignores the command and
returns null
                                self.state:=#Grasping;
                                self.processingTime:=self.config.gTime;
```

```
                                self.nextWakeUp:=self.clock.NOW+self.processingTime;
                                self.item:=t.get();
                        end
                end;
          end
          pre WithoutItem: self.item=null
          pre onTray: self.isAt(t.x,t.y,t.z)
          pre: ((t.items->at(1)).weight<=self.config.wMax)
          post ItemOK: (self.mode=#Automatic or (self.mode=#Manual and
self.state@pre=#Idle))implies (self.item.weight<=self.config.wMax)
          post Grasped: (self.mode=#Automatic or (self.mode=#Manual and
self.state@pre=#Idle))implies  self.item = t.items@pre->at(1) and
t.items@pre=t.items->prepend(self.item)


        drop(t:Tray)
          begin
                if self.mode=#Automatic then
                        t.put(self.item);
                        self.item:=null;
                else if self.state=#Idle then -- if it is busy, it ignores the command
                                self.state:=#Dropping;
                                self.processingTime:=self.config.dTime;
                                self.nextWakeUp:=self.clock.NOW+self.processingTime;
                                t.put(self.item);
                                self.item:=null;
                        end
                end;
          end
          pre WithItem: self.item<>null
          pre onTray: self.isAt(t.x,t.y,t.z)
          post WithoutItem: (self.mode=#Automatic or (self.mode=#Manual and
self.state@pre=#Idle))implies self.item=null
          post Put: (self.mode=#Automatic or (self.mode=#Manual and
self.state@pre=#Idle)) implies t.items = t.items@pre->append(self.item@pre)

        reset() -- returns to base
          begin
                if self.mode=#Automatic then
                        self.processingTime:=
((((self.base.x-self.x).abs+(self.base.y-self.y).abs)/self.config.hSpeed+
 ((self.base.z-self.z).abs/self.config.vSpeed))*(if self.item=null then 1.0 else
self.item.weight/self.config.wMax endif)).round.toInteger();
                        self.moveTo(self.base.x,self.base.y,self.base.z);
                else if self.state=#Idle then -- if it is busy, it ignores the command
self.state:=#GoingBackToBase;
self.processingTime:=
((((self.base.x-self.x).abs+(self.base.y-self.y).abs)/self.config.hSpeed+
 ((self.base.z-self.z).abs/self.config.vSpeed))*(if self.item=null then 1.0 else
self.item.weight/self.config.wMax endif)).round.toInteger;
        self.nextWakeUp:=self.clock.NOW+self.processingTime;
        self.moveTo(self.base.x,self.base.y,self.base.z);
                        end
                end;
          end
```

```
        post backToBase: (self.mode=#Automatic or (self.mode=#Manual and
self.state@pre=#Idle)) implies self.isAt(self.base.x,self.base.y,self.base.z)

      pickup(t:Tray)
        begin
            self.processingTime:=
((((t.z-self.z+5.0).abs/self.config.vSpeed)+
((t.x-self.x).abs+(t.y-self.y).abs)/self.config.hSpeed+
((5.0).abs/self.config.vSpeed))*(if self.item=null then 1.0 else
self.item.weight/self.config.wMax endif)).round.toInteger+self.config.gTime;
            self.up(t.z-self.z+5.0);
            self.travel(t.x,t.y);
            self.down(5.0);
            self.grasp(t);
        end
        pre OnlyInAutomaticMode: self.mode=#Automatic
        pre PositionNotTaken:
PositionedElement.allInstances->excluding(self)->select(p|isAt(t.x,t.y,t.z))->isEmpty
        pre inTrayNotEmpty: not t.isEmpty()
        pre WithoutItem: self.item=null
        pre: ((t.items->at(1)).weight<=self.config.wMax)
        post ItemOK: (self.item.weight<=self.config.wMax)
        post Grasped:
self.item = t.items@pre->at(1) and t.items@pre=t.items->prepend(self.item)

      release(t:Tray)
        begin
            self.processingTime:=
((((t.z-self.z+5.0).abs/self.config.vSpeed)+
((t.x-self.x).abs+(t.y-self.y).abs)/self.config.hSpeed+
((5.0).abs/self.config.vSpeed))*(if self.item=null then 1.0 else
self.item.weight/self.config.wMax endif)).round.toInteger+self.config.dTime;
            self.up(t.z-self.z+5.0);
            self.travel(t.x,t.y);
            self.down(5.0);
            self.drop(t);
            self.movedItems:=self.movedItems+1;
        end
        pre OnlyInAutomaticMode: self.mode=#Automatic
        pre PositionNotTaken:
PositionedElement.allInstances->excluding(self)->select(p|isAt(t.x,t.y,t.z))->isEmpty
        pre outTrayNotFull: not t.isFull()
        pre WithItem: self.item<>null
        post WithoutItem: self.item=null
        post Put: t.items = t.items@pre->append(self.item@pre)

------ MAIN OPERATION

      token()
        begin
            if self.mode=#Automatic then self.automaticToken() else
self.manualToken() end
        end

      automaticToken()
    begin
```

```
                if self.state=#Idle and not self.inTray.isEmpty() and
Gantry.allInstances->select(p|p.isAt(self.inTray.x,self.inTray.y,self.inTray.z))->isEmpty
and
                ((self.inTray.items->at(1)).weight<=self.config.wMax)
                then
                        self.pickup(self.inTray);
                        self.state:=#PickingUp; -- it stays there until next token()
                        self.nextWakeUp:=self.clock.NOW+self.processingTime;
            else
                        if self.state=#PickingUp and self.clock.NOW >= self.nextWakeUp
then if not self.outTray.isFull() and
Gantry.allInstances ->select(p|p.isAt(self.outTray.x,self.outTray.y,self.outTray.z))->
isEmpty then
                                        self.release(self.outTray);
                                        self.state:=#Releasing; -- it stays there until next
token()

        self.nextWakeUp:=self.clock.NOW+self.processingTime;
                                else
                                        self.reset();
                                        self.state:=#WithItem; -- goes back to base if it
cannot drop the item

        self.nextWakeUp:=self.clock.NOW+self.processingTime;
                                end
                        else
                if self.state=#Releasing and self.clock.NOW >= self.nextWakeUp then
                                        self.reset();
                                        self.state:=#Idle;

        self.nextWakeUp:=self.clock.NOW+self.processingTime;
                                else
                        if self.state=#WithItem and not self.outTray.isFull() and
self.clock.NOW >= self.nextWakeUp and
Gantry.allInstances->select(p|p.isAt(self.outTray.x,self.outTray.y,self.outTray.z))->
isEmpty then
                        self.release(self.outTray);
                        self.state:=#Releasing;

        self.nextWakeUp:=self.clock.NOW+self.processingTime;
                                        end
                                end
                        end
                end
          end
          manualToken()
          begin
                if (self.state<>#Idle) and (self.clock.NOW >= self.nextWakeUp) then
self.state:=#Idle end
          end
end
```

Cod.18 – Clase Gantry

Las siguientes asociaciones son las requeridas para dar cohesión a las distintas clases de nuestro sistema. Es importanto que la relación entre las bandejas y los

objetos creados sea ordenada, para conseguir que los mismos sean tratados en orden First In First Out (FIFO).

```
association AoControl between
  Control[1] role control
  ActiveObject [0..*] role controlledobject
end
----------------------------------------
association ProducerTray between
  ActiveObject[0..*] role producer
  Tray[0..1] role outTray
end
----------------------------------------
association ConsumerTray between
  ActiveObject[0..*] role consumer
  Tray[0..1] role inTray
end
----------------------------------------
aggregation BufferedItems between
  Tray[0..1] role buffer
  Item[0..*] role items ordered
end
```

Cod.19 – Asociaciones de ProducerConsumer.

4.2 La Introducción de Incertidumbre

Ahora que tenemos el modelo completamente definido, y una vez comprobado que su funcionamiento es el correcto, procederemos a estudiar la manera adecuada de introducir la incertidumbre en el sistema, ahí donde sea necesaria.

Para empezar, al igual que en el modelo anterior, sabemos que una de las principales causas de error de un sistema como este en el mundo real reside en la precisión a la hora de colocar y mover los distintos elementos dependientes de su posición. Es por esto que, tal y como hicimos anteriormente, las coordenadas de las distintas instancias de PositionedElement incluyan dicha incertidumbre. De la misma manera, todas las operaciones que incluyan las coordenadas en el cálculo del resultado también pasarán a devolver UReal.

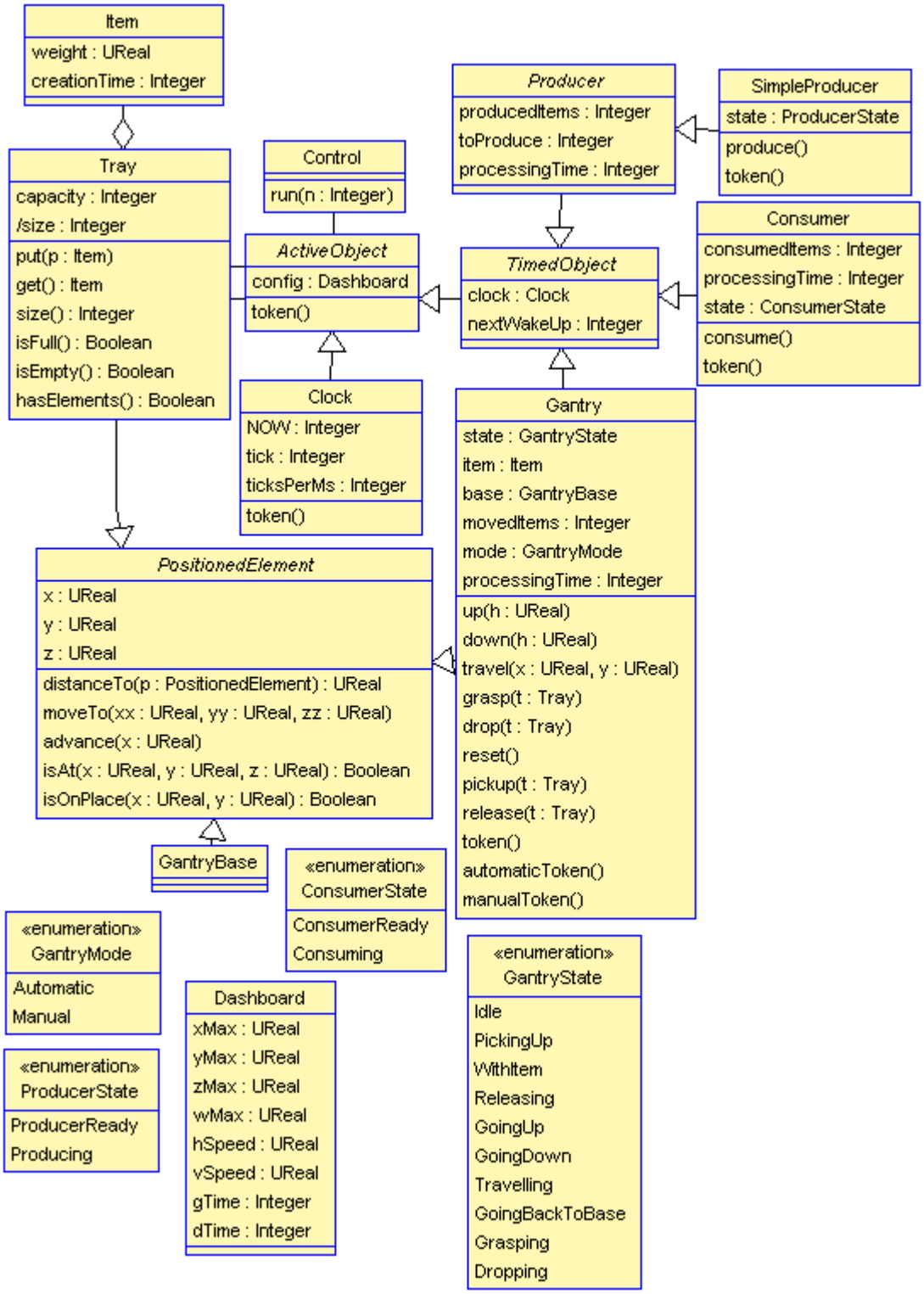

Fig. 9 - Diagrama de clases del modelo Producer-Consumer

Si observamos los efectos que tiene en el modelo el cambio anteriormente descrito, nos daremos cuenta de que el modelo pasa a tener un comportamiento inadecuado. Esto se debe a que varias comparaciones que eran entre dos números reales en el modelo original han pasado a ser entre un UReal y un número real, que en casi todos los casos pertenece a la clase Dashboard. Para que las comparaciones descritas en el modelo se comporten adecuadamente al tratar con UReal (Devuelvan un UBoolean representando la probabilidad de que la expresión sea cierta), será necesario que las cotas establecidas en la clase Dashboard pasen a ser de tipo UReal también.

```
class Dashboard
attributes
  xMax:UReal init: UReal(100.0, 0.01) -- max difference between gantry and base
  yMax:UReal init: UReal(100.0, 0.01) -- max difference between gantry and base
  zMax:UReal init: UReal(100.0, 0.01) -- max difference between gantry and base
  wMax:UReal init: UReal(100.0, 0.01) -- max weight of items a gantry is able to carry
  hSpeed:UReal init: UReal(1.0, 0.01) -- horizontal speed
  vSpeed:UReal init: UReal(1.0, 0.01) -- vertical speed
  gTime:Integer init: 1 -- grasping time
  dTime:Integer init: 1 -- dropping time
end
```

Cod.20 – Clase Dashboard modificada.

Ahora que las operaciones de comparación devuelven adecuadamente un valor de tipo UBoolean, , debemos tratar estos para que puedan ser utilizados como guardas en sus respectivas sentencias. Como ya mencionamos anteriormente, un valor UBoolean no puede actuar de guarda por si mismo, ya que representa la probabilidad de una expresión de ser cierta, no el valor de la expresión en sí. Existen varias maneras de trabajar con UBoolean, pero nosotros lo haremos transformando las expresiones en booleanas mediante la operación toBooleanC() que, aplicada sobre un predicado UBooleano, y dada como parámetro una cota entre 0 y 1, devolverá true si la probabilidad del predicado de ser cierta es al menos la de la cota dada. En caso contrario, devolverá false. De esta manera podemos solventar de manera fiable todos los usos de expresiones UBooleanas en nuestro sistema.

Por último, el otro valor que podemos someter a incertidumbre es el peso de los objetos creados. Dado que el productor crea objetos de diferente peso de una manera aleatoria, y el brazo robótico tiene un límite de peso con el que puede cargar, es lógico que a la hora de simular esta parte del sistema obtengamos unos resultados más realista si en vez de utilizar números exactos introducimos cierta incertidumbre. Esta incertidumbre se extenderá una vez más a distintas operaciones de la clase Gantry, lo que nos obligará a tratar las expresiones que ahora son de tipo UBoolean de la misma manera que hicimos con las anteriormente mencionadas.

```
        up(h:UReal) -- relative coordinates
          begin
                if self.mode=#Automatic then
                        self.moveTo(self.x,self.y,self.z+h);
                else if self.state=#Idle then
                        self.state:=#GoingUp;
                        self.processingTime:=((h/self.config.vSpeed)*(if self.item=null
then 1.0 else self.item.weight/self.config.wMax endif)).round.toInteger;
                                self.nextWakeUp:=self.clock.NOW+self.processingTime;
                                self.moveTo(self.x,self.y,self.z+h);
                         end;
                end;
          end
          pre hOk: (h >= 0).toBooleanC(0.85)
          pre PositionNotTaken:
PositionedElement.allInstances->excluding(self)->select(p|p.oclIsKindOf(Gantry) and
p.isAt(self.x,self.y,self.z+h))->isEmpty
          post AtFinalPosition: (self.mode=#Automatic or (self.mode=#Manual and
self.state@pre=#Idle))implies self.isAt(self.x@pre,self.y@pre,self.z@pre+h)
          post NoOutOfBoundsTest: ((self.x-self.base.x).abs()<=self.config.xMax and
                (self.y-self.base.y).abs()<=self.config.yMax and
                (self.z-self.base.z).abs()<=self.config.zMax).toBooleanC(0.85)
```

Cod.21 – Clase de Gantry modificada.

4.3 Pruebas con incertidumbre

Al igual que con el modelo anterior, lo primero que debemos hacer es realizar una prueba poniendo las distintas igualando la incertidumbre en los diferentes valores que la poseen a cero, así como hacer que los UBoolean siempre den verdadero o falso de manera determinista. De esta manera, comprobamos que el funcionamiento del sistema no ha sido variado involuntariamente a causa de su modificación.

Como podemos observar en el diagrama, el sistema parece presentar un comportamiento adecuado, tal y como esperábamos. Ningún objeto parece haberse perdido por errores de posicionamiento o peso durante la ejecución, lo que era esperable debido al determinismo inherente a este tipo de pruebas sin incertidumbre.

Una vez controlado esto, podemos empezar a trabajar en las pruebas sobre la calidad de los distintos componentes y como afectan a la productividad de la cadena de producción. Para ello, de manera similar a como hicimos en el sistema anterior, necesitaremos una clase encargada de llevar la cuenta y procesar los objetos perdidos por el camino, a causa de su mal emplazamiento en las bandejas o de un peso excesivo. Dicha clase será nombrada GarbageCollector. Esta clase se destruirá los elementos cuyo peso sea superior al máximo que puede transportar nuestro brazo mecánico, así como aquellos que lleven un tiempo excesivo dentro del sistema. Para esto último será necesario asociar cada Item creado con su tiempo de creación, según el reloj del sistema. Esta clase no representa ningún elemento físico del sistema, sino que es una herramienta de mantenimiento para nuestra simulación.

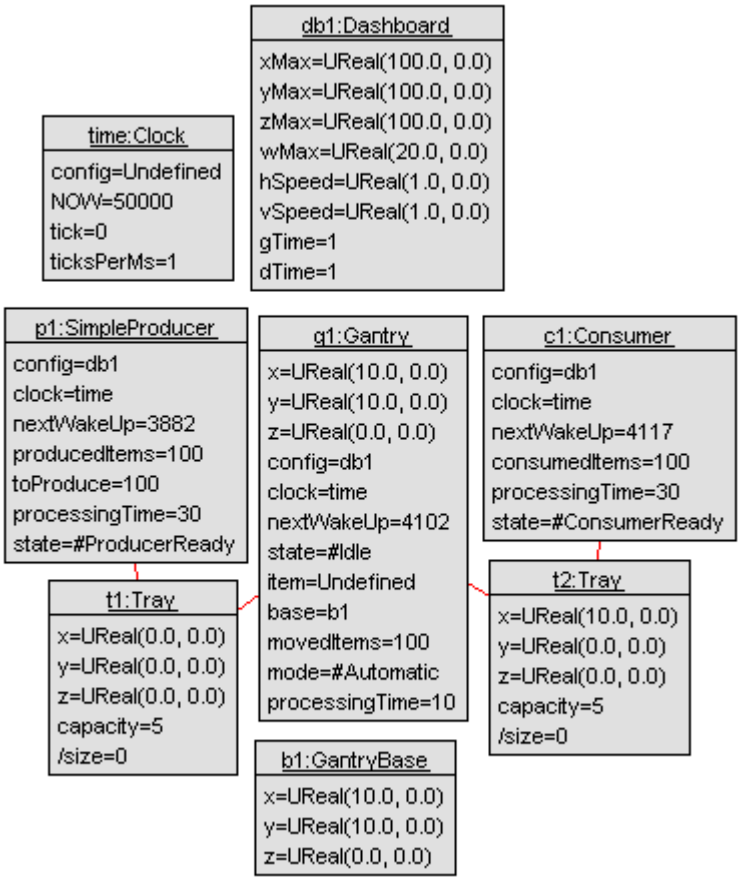

Fig. 10 - Estado del sistema tras 50000 ticks de reloj tras la modificación, sin incertidumbre.

```
class GarbageCollector < TimedObject
  attributes
    processingTime: Integer init=20
    itemTimeout: Integer init=500
    faultyItems: Integer init=0
  operations
    token()
    begin
      self.nextWakeUp:=self.clock.NOW+self.processingTime;
      for t in self.traytocheck do
        for i in t.items do
          if (i.weight>self.config.wMax).toBooleanC(0.9) or
            (self.clock.NOW-i.creationTime>self.itemTimeout) then destroy i;
self.faultyItems:=self.faultyItems+1;
          end
          end
          end
    end
end
```

Cod.22 – Clase GarbageCollector.

El siguiente objetivo es, una vez más, estudiar qué cualidades de los diferentes componentes del sistema afectan a su productividad y fiabilidad, y hasta que punto podemos sacrificar precisión o calidad sin que estas características se vean afectadas o, en todo caso, perjudicadas lo menos posible.

La primera de estas cualidades, ya mencionada anteriormente, se refiere a la calidad de los objetos producidos por nuestro sistema, más concretamente la precisión en su masa al ser creados. En el sistema descrito, el productor genera objetos de entre 1 y 16 kilogramos de peso. Aunque a priori pueda parecer que con un brazo mecánico que soporte hasta 16 kilogramos de peso el problema esté resuelto, es posible que diversos defectos en el acabado del producto hagan que este exceda el peso límite que el brazo puede soportar, causando un fallo en el sistema e imposibilitando su retirada de la bandeja pertinente. Con el objetivo de controlar este problema, vamos a estudiar como afecta la precisión en el acabado de los diferentes productos a la productividad de nuestra cadena, variando la incertidumbre de los objetos de la clase Item en el momento de su creación. (Nótese que los resultados de la tabla son la media truncada de 3 ejecuciones, para una mayor precisión en las pruebas).

	Objetos producidos	Objetos Procesados	Objetos perdidos	Objetos movidos	Porcentaje de éxito
0,005	2000	2000	0	2000	100,00%
0,01	2000	1998	2	1998	99,90%
0,025	2000	1997	3	1998	99,85%
0,03	2000	1990	10	1990	99,50%
0,05	2000	1986	14	1986	99,30%
0,07	2000	1984	16	1984	99,20%
0,075	2000	1984	16	1984	99,20%
0,09	2000	1982	18	1982	99,10%
0,1	2000	1978	22	1978	98,90%

Fig. 11 - Resultados de la ejecución variando la precisión en el acabado de los objetos producidos.

Observando los resultados de las distintas pruebas, podemos apuntar que mientras nos movamos por debajo de los 25 gramos de incertidumbre el número de objetos defectuosos es casi despreciable (apenas 1 de cada 1000). Sin embargo, si queremos asegurarnos de que el peso de los objetos no sea un factor de riesgo para el funcionamiento de nuestro sistema, deberemos utilizar un productor cuya incertidumbre en la masa de los objetos producidos sea inferior a 0.005.

El otro gran factor cuya precisión afecta al funcionamiento de nuestro sistema es la colocación de los objetos en la bandeja del consumidor por parte del brazo mecánico. El consumidor de nuestro sistema requiere que los objetos que lleguen hasta él estén colocados en un lugar concreto de la cinta para poder ser procesados de manera adecuada. Por tanto, nuestro brazo deberá tener cierta precisión a la hora de colocar los mismos en la cinta, con el fin de evitar perderlos. Esta es una prueba que se dificulta debido a la definición de nuestro sistema hasta ahora, y que requerirá de algunos cambios en la estructura del mismo para ser realizada. En concreto, la naturaleza de las pruebas a realizar a continuación entra en conflicto con el funcionamiento de las pre y postcondiciones en USE. Cuando una pre o postcondición falla, USE detiene automáticamente el funcionamiento del sistema. Esto nos perjudica, debido a que con nuestras pruebas queremos forzar el sistema hasta el punto de hacerlo fallar, puesto que estamos probando límites. Es por ello que distintas pre y postcondiciones referidas al posicionamiento de objetos y del propio brazo

deberán ser adaptadas dentro del código de las operaciones como condiciones para sentencias o bien borradas totalmente. Así mismo, con el fin de centrar las pruebas a realizar en el posicionamiento de los objetos en concreto, asumiremos una incertidumbre de 0,005 en el peso de los objetos y que el reinicio del brazo mecánico lo devuelve a la posición exacta de su base.

Para hacer la prueba no determinista en este caso, haremos que la incertidumbre en los desplazamientos del brazo siga una distribución exponencial de la que iremos variando la base, frente a una incertidumbre fija de aceptación por parte del consumidor, 0.1.

.expDistr()	Objetos producidos	Objetos Procesados	Objetos perdidos	Objetos movidos	Porcentaje de éxito
0,005	2000	2000	0	2000	100,00%
0,0075	2000	1999	1	1999	99,95%
0,01	2000	1995	5	1999	99,75%
0,025	2000	1619	381	1998	80,95%
0,05	2000	967	1033	1999	48,35%

Fig. 12 - Resultados de la ejecución variando la precisión en el movimiento del brazo robótico con respecto a la bandeja del consumidor.

Aunque no pueda parecernos intuitivo a primera vista, debido al número de movimientos necesarios para transportar un objeto de una bandeja a otra se nos puede acumular un error bastante abultado pese a tener un brazo robótico mucho más preciso que la tolerancia de nuestro consumidor. Como podemos observar en los resultados de las pruebas realizadas, una incertidumbre 10 veces menor que la del consumidor parece ser el límite a permitir si necesitamos un funcionamiento razonablemente efectivo del sistema. Cualquier cosa a partir de ahí acumula en unos pocos movimientos un margen de error inadmisible.

5

Conclusiones

Gracias a todas las pruebas realizadas sobre los dos modelos previamente expuestos, podemos empezar a elucubrar si realmente la introducción de incertidumbre al sistema merece la pena en términos de la obtención de unos resultados significativamente más útiles con respecto al esfuerzo que supone la introducción de la misma, el tiempo empleado y la dificultad de la traducción de un modelo a otro.

En términos de utilidad, no cabe duda de que los modelos con incertidumbre producen resultados de las distintas pruebas mucho más fieles a la realidad que los que nos proporcionan las pruebas con valores reales. Esto es latente, por ejemplo, en el caso del posicionamiento de las botellas en la cinta en el primer modelo. En el modelo inicial, dado que la cinta presenta un movimiento a priori constante a través del tiempo, y el punto de introducción de las botellas al sistema es siempre el mismo, ni siquiera se plantea la posibilidad de que fallos mecánicos del sistema afecten a la posición de las botellas.

Sin embargo, en nuestro modelo modificado con incertidumbre, resulta sencillo tener en cuenta factores del mundo real que condicionan fallos en sistema, como el desgaste en la precisión de la cinta transportadora con el paso del tiempo. De esta manera, una prueba que con números reales apenas servía para estimar de manera poco realista tiempos de llenado y tratado, y que ofrece apenas nada como base para construir el sistema físico, se convierte en una prueba que además de cubrir esto es capaz de medir otros factores como el efecto del desgaste de los distintos componentes en el sistema, así como la falta de exactitud de las medidas en el mundo real.

Respecto a la dificultad en la introducción y utilización de incertidumbre en modelos, se ha visto que con la preparación previa adecuada esto no supone un problema. Las diferencias entre ambos modelos en este apartado, no obstante, han sido notables. En el primer modelo, el cual se apoyaba en gran medida en condiciones descritas en el interior de las operaciones de los distintos objetos activos, no hemos tenido apenas complicación a la hora de introducir la incertidumbre. Ha sido un proceso simple y directo, sin necesidad apenas refactorizar nada.

No obstante, en el segundo modelo, el funcionamiento de las distintas operaciones venía dado por una serie de pre y postcondiciones que, si bien funcionan perfectamente a la hora de probar un modelo sin incertidumbre, han probado ser una traba a la hora de traducir el modelo para la utilización de la misma, requiriendo todas ellas bien una transformación al convertirse en UBoolean, bien una refactorización para su introducción como condición en el código, o bien su supresión, debido a entrar en conflicto con la filosofía de las pruebas con incertidumbre realizadas (si un elemento falla, tomamos los datos y continuamos la ejecución, no la detenemos).

En conclusión, las utilización de incertidumbre de la medida puede representar, sin duda alguna, el siguiente paso en la evolución de la realización de pruebas para sistemas industriales. No existe ningún inconveniente a la hora de utilizarla a excepción del tiempo extra empleado en realizar los modelos actualizados, el cual puede verse reducido drásticamente si construyes los modelos base con vista a que la transformación sea lo más liviana posible (no crear pre y postcondiciones con hechos que puedan fallar en el sistema real, en su lugar introducir dichos hechos como condiciones en las operaciones, por ejemplo).

Si se siguen pautas como las descritas para los modelos anteriores adecuadamente, y se realizan las pruebas sobre los parámetros adecuados, el beneficio económico que pueden suponer las mismas en calidad de ahorro en componentes y fallos del sistema es de un valor mucho mayor al invertido en su realización.

6

Bibliografía

1.-Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Sci. Comput. Program. 69(1-3), 27–34 (2007). https://doi.org/10.1016/j.scico.2007.01.013

2.-M. F. Bertoa, N. Moreno, L. Burgueño, A. Vallecillo. “Incorporating Measurement Uncertainty into OCL/UML Primitive Datatypes”. Software and Systems Modeling (Sosym), 2019. https://doi.org/10.1007/s10270-019-00741-0

3.-Büttner, F., Gogolla, M.: On OCL-based imperative languages. Sci. Comput. Program. 92, 162–178 (2014)

4.-Loli Burgueño, Tanja Mayerhofer, Manuel Wimmer, Antonio Vallecillo "Specifying quantities in software models." Information & Software Technology 113: 82-97 (2019)