Implementación en Java de un Modelo de Consistencia Temporal para Datos de Tiempo Real

Carlos E. Buckle, Damián P. Barry, José M. Urriza

Facultad de Ingeniería, Departamento de Informática
Universidad Nacional de La Patagonia San Juan Bosco - Puerto Madryn, Argentina
cbuckle@unpata.edu.ar, damian_barry@unpata.edu.ar, josemurriza@unp.edu.ar

Resumen. Este trabajo presenta una implementación en JAVA de un modelo de consistencia temporal para datos de tiempo real. Los Sistemas de Bases de Datos de Tiempo Real, incorporan el concepto de transacciones y datos con tiempo de vencimiento. El vencimiento de una transacción, es el tiempo máximo para el cual todos los resultados deben haberse alcanzado. Algunos de los datos involucrados, representan elementos variables del ambiente y su valor tiene un tiempo de vigencia limitado en el sistema. Estos datos tienen una restricción adicional de vencimiento, el cual puede ocurrir en el lapso de ejecución de la transacción. El modelo desarrollado facilita la representación de datos de tiempo real y encapsula la validación de consistencia temporal. La implementación se verifica mediante un conjunto de pruebas y el código fuente resultante se publica en un sitio web.

Palabras Clave: Datos de Tiempo Real, Consistencia Temporal, Bases de Datos de Tiempo Real, Ingeniería de Software de Tiempo Real, Transacciones de Tiempo Real.

1 Introducción

En los Sistemas de Tiempo Real (*RTS*) un requerimiento significativo es ser consistente con el ambiente monitoreado. Los objetos del sistema deben reflejar el estado del entorno. Esto significa garantizar la consistencia entre el estado de los objetos externos y los datos del sistema que los representan. Estos datos obtienen su valor desde sensores, mediante actualizaciones periódicas acorde a los cambios externos y dicho valor tiene un tiempo de vigencia limitado. Una vez superado el vencimiento, su valor generalmente se torna inválido. El vencimiento puede ocurrir mientras transcurre la tarea que los opera. Los datos con estas características son llamados *Datos de Tiempo Real (RTD Real-Time Data).*

Los *RTS* convencionales trabajan con una cantidad limitada de *RTD.* Estos se conocen a priori y usualmente cada *RTD* es utilizado en una única tarea del sistema, la cual es responsable de garantizar su validez temporal. El diseño de estos *RTS* se basa en el modelado de tareas. Sin embargo, existen otros tipos de *RTS* abocados al uso intensivo de datos con requerimientos de persistencia, como por ejemplo, sitios web

de subastas electrónicas, sistemas de supervisión industrial, tableros de comando para toma de decisiones empresariales, etc. Estos sistemas requieren incorporar el uso de un *Sistema de Bases de Datos* (*DBS*) y a su vez, contemplar el concepto de *transacción* ([1]). En el diseño no solo se debe considerar el modelado de tareas para las transacciones, sino también el modelado de datos *RTD*.

En la disciplina de *DBS*, la gestión de datos con tiempo de vigencia es desarrollado por las *Bases de Datos Temporales* ([1]). Estas definen el dominio *tiempo* y su estructura de representación. Además, existen diferentes líneas de tiempo para los datos como: estampilla de tiempo del evento real que lo genera, intervalo de vigencia del dato en el sistema y estampilla de tiempo de la transacción que lo registra. Para plantarse el diseño de este tipo de datos, se han extendido algunos lenguajes de modelado conceptual, para incorporar características temporales. Algunas de estas extensiones se han realizado sobre el enfoque relacional ([2]) y otras sobre el enfoque orientados a objetos ([3]). No obstante, las *Bases de Datos Temporales* no han contemplado el contexto específico de Tiempo Real. En los *RTS* los valores de los datos pueden perder vigencia durante la transacción. Un *RTD* válido al inicio de la transacción puede caducar antes del final. Las transacciones a su vez, también tienen la exigencia de responder antes de un determinado tiempo de vencimiento (*deadline*). Esto conlleva a que el *DBS* deba planificar transacciones considerando las constricciones de tiempo involucradas.

Por otro lado, los *RTS* convencionales, contemplan naturalmente el concepto de tareas con vencimiento, pero no contemplan el concepto de transacción como unidad de consistencia lógica de datos y no consideran el problema de acceder a un número impredecible de datos desde un DBS. Existen importantes trabajos de ingeniería de software aplicables al diseño de *RTS,* como los patrones de Douglas en [4], o las recomendaciones de *MARTE* ([5]) (*OMG: Modeling and Analysis of Real-Time Embedded systems*). Estas propuestas son amplias y abarcan un amplio espectro de *RTS* a muy alto nivel. Sin embargo, no se enfocan hacia el modelado de datos.

La brecha no cubierta por los *DBS* ni por los *RTS* ha sido desarrollada en la subdisciplina de los *Sistemas de Bases de Datos de Tiempo Real* (*RTDBS*) ([6, 7]). Los *RTDBS,* además de considerar transacciones con tiempo de vencimiento, incorporan el concepto de tiempo de vencimiento de los datos (*Datadeadline* [8]). Las transacciones, además de garantizar la consistencia lógica de los datos involucrados, deben garantizar su *consistencia temporal* ([9]). En [7], Stankovic enuncia que las constricciones de tiempo para las transacciones en un *RTDBS* están definidas por los intervalos de validez de los *RTD* involucrados en ella y por características propias de la transacción, como periodicidad o vencimiento. Consecuentemente, el concepto de *RTD* es natural dentro de los *RTDBS.* En estas, se consideran a su vez, dos tipos de *RTD*: *base (RTDBase)* y *derivados* (*RTDDerived*).

Los *RTDBase* reflejan el estado de un objeto externo y permiten detectar los cambios del ambiente. Estos cambios externos pueden tener un comportamiento *continuo* o *discreto* [10]. Una entidad externa de cambio *continuo* genera variaciones continuas en el tiempo, por ejemplo, el monitoreo de un sensor de temperatura. Una entidad externa de cambio *discreto* genera variaciones en instantes puntuales y espaciados, por ejemplo, el arribo de una nueva oferta en un sistema de subastas vía

web. Debe garantizarse que las transacciones siempre utilicen *RTDBase* correctamente actualizados, y en caso de detectarse datos envejecidos debe deshacerse la transacción, con posibilidades de re-ejecución (*rollback-redo*).

Los *RTDDerived* son datos derivados, en base a cálculos que se obtienen operando sobre un conjunto de otros *RTD*. El re-cálculo de derivaciones, es un importante problema a resolver en la planificación de transacciones, pues significa un compromiso entre predictibilidad y eficiencia. Los *RTDDerived* pueden recalcularse siempre que se actualiza un *RTDBase* (más predecible) o solo cuando una transacción lo requiere (más eficiente). Pero en ambos casos es necesario recalcular el vencimiento del dato derivado.

Es importante para los desarrolladores, contar con herramientas de librería que soporten estos conceptos y faciliten la construcción de aplicaciones con uso de *RTD*.

Este trabajo retoma un modelo desarrollado en trabajos anteriores ([11]) y realiza su implementación en JAVA. El objetivo que se persigue, es poner a disposición de la comunidad de desarrolladores, un paquete (*JAVA package*) verificado que contenga las clases *RTD* implementadas ([12]).

El resto de este documento está organizado de la siguiente manera: en la sección 2 se presenta el tipo de dato abstracto *RTD*. En la sección 3 se presenta la implementación realizada en JAVA. En la sección 4 se muestran las verificaciones realizadas. En la sección 5 se elaboran las conclusiones y se plantean los trabajos a futuro.

2 El Modelo *RTD*

En la subdisciplina de *RTDBS*, se han desarrollado modelos de datos con constricciones de tiempo real. Un trabajo de referencia es *Real Time Semantic Objects Relationships And Constraints* (*RTSORAC*) ([13]) donde se definen y se describen los componentes de un *RTDBS.* Dichas definiciones son modeladas de forma concreta y posteriormente, en un paquete *UML* que especifica objetos de tiempo real (*RT-Object*) ([14]). El trabajo referido anteriormente, ha sido tomado como base para desarrollar el perfil de diseño *UML-RTDB(*[15]). En éste, se presenta un diseño estructural para un *RTDBS,* enfocado al modelado estático de objetos con atributos de tiempo real.

Los trabajos antes mencionados, han motivado el desarrollo de un *Tipo de Dato Abstracto para Bases de Datos de Tiempo Real* ([11]), el cual permite definir cualquier tipo de atributo de tiempo real. En este modelo, se incorpora la sub-clasificación de *RTD* base continuos ó discretos y se pone especial énfasis en los *RTD* derivados. Se trata de un modelo conceptual que define el tipo abstracto parametrizable *RTD*, el cual encapsula las propiedades de tiempo y las validaciones de consistencia temporal, de manera que el programador de aplicaciones se pueda desligar de estas responsabilidades. Utilizando este modelo, el programador sólo debe focalizarse en identificar y definir correctamente los atributos de tiempo real. Además, puede operar con ellos normalmente dentro del código de una transacción y estar preparado para deshacer y rehacer en caso de obtener una excepción por *RTD vencido*. Asimismo, al ser parametrizable, permite que un objeto de cualquier clase pueda ser definido como un *RTD*, para esto, implementa un envoltorio (*wrapper*) que

incorpora características y constricciones propias de un atributo de tiempo real. Por ejemplo:

```
INIT:
    d RTD<float>        //Define RTD as float
    d.setMaxAge(6 seg)  //set Datadeadline
    init_sensing(d)     //init Periodic Update
END INIT
```

```
BEGIN TRANSACTION:
  …(any)…
  Res = d.getData() *0.25    //RTD access
  …(any)…
  commit;                    //commit work
ON EXCEPTION RTDExpired: rollback;
END TRANSACTION
```

3 Implementación en JAVA

El presente trabajo continúa con el desarrollo anterior mencionado ([11]), ampliando algunas especificaciones y generando una implementación en JAVA para el tipo de dato *RTD*, cuyo modelo se muestra en la *Figura 1*.

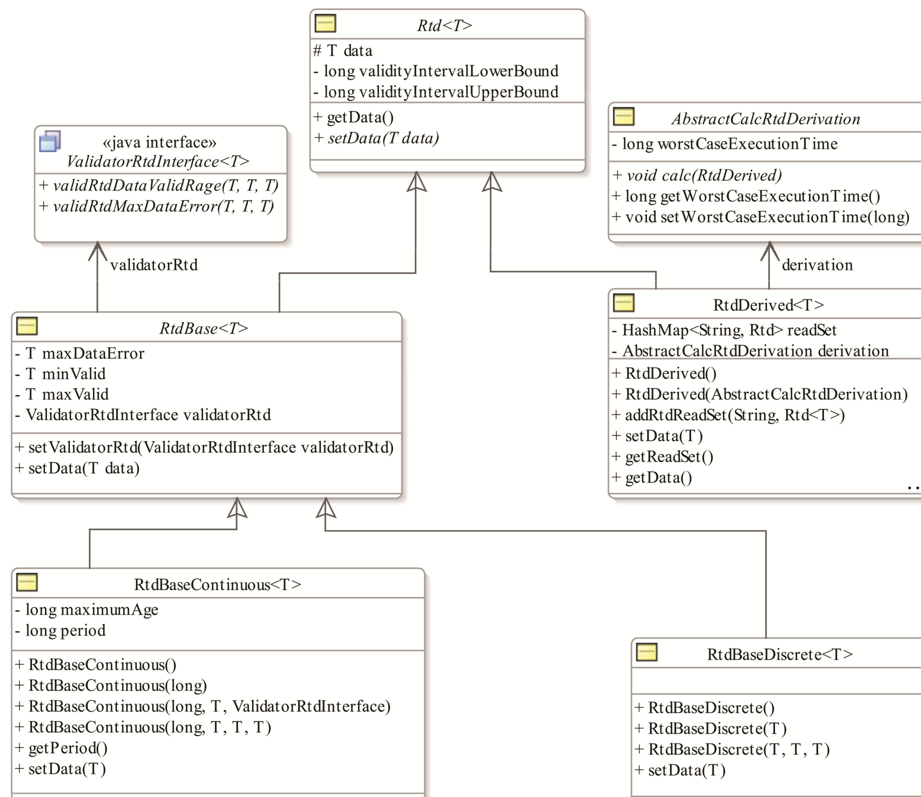

Fig. 1. Diagrama de Clases Java del Tipo de Dato Abstracto *RTD*

En esta sección se presentan los principales detalles de la implementación. El código JAVA completo puede descargarse del sitio de *Google-Code* [12]. También

está disponible un *CVS* (*Concurrent Versioning System*) para que la comunidad de desarrolladores pueda colaborar con la evolución del proyecto.

El propósito del modelo es definir un *RTD* como un dato genérico, de cualquier tipo, con validación de *consistencia temporal*.

Sea d un *RTD*, llamaremos VI_d (*Validity Interval*) al intervalo de validez de d en el tiempo. El límite inferior del intervalo (*VILB:Validity Interval Lower Bound*), es el instante en el que se actualiza el dato d, y el límite superior (*VIUB:Validity Interval Upper Bound*) es el instante en que dicho dato pierde vigencia (*datadeadline*). Un *RTD d* satisface *consistencia temporal* si el instante en el que se accede (*now*(*t*)) pertenece al intervalo VI_d. Esto es:

$$VILB_d(t) \leq now(t) \leq VIUB_d(t)$$

En el modelo presentado, esta validación se implementa en la clase principal *RTD* la cual se define con un atributo *data* genérico y con su VI_d representado por los atributos *validityIntervalLowerBound* y *validityIntervalUpperBound.* La validación de consistencia temporal se realiza cada vez que se accede al dato utilizando *getData*(). A continuación se muestra el código principal de la clase *RTD*:

```
abstract public class Rtd<T> {
    protected  T data;
    private long validityIntervalLowerBound = -1;
    private long validityIntervalUpperBound = -1;
  public T getData() throws RtdException{
     // valid data? check temporal restriction
     long now = System.currentTimeMillis();
     if (validityIntervalLowerBound <= now && now <= validityIntervalUpperBound) {
        return data;
     } else {
        throw new RtdDataHasExpired("Rtd Data has expired within lower and upper bound limits.");
     } …
```

La determinación de los límites *VILB* y *VIUB* se realiza en el momento que se actualiza el dato utilizando *setData*(). Pero esta determinación depende del tipo de *RTD*, el cual, como se comentó antes, se clasifica en *RTDBase* y *RTDDerived.*

3.1 RTD Base

Los *RTDBase* reflejan el estado de un objeto externo y actualizan su valor desde sensores o publicadores. Estas tareas, en un *RTDBS,* se implementan en transacciones de ejecución periódica o esporádica según corresponda. Estas transacciones renuevan el valor de un *RTDBase*, lo cual puede significar el re-cálculo de datos derivados relacionados. Debe considerarse, que un cambio en un dato base puede generar cadenas de actualizaciones de datos derivados. Cuando los cambios en el objeto externo son poco significativos puede ser innecesario generar sobrecarga, por ello, el modelo introduce el concepto de *Error Máximo* (*maxDataError*) ([16]), el cual permite descartar la actualización del valor de un *RTDBase* si la variación entre el valor registrado del dato y el nuevo valor no es suficientemente significativa. El valor del *RTDBase d* no se actualiza si: $|ValorNuevo_d - ValorAnterior_d| \leq maxDataError$.

Además de contemplar el máximo error *maxDataError*, se contempla la definición opcional de un rango de valores válidos para descartar posibles anomalías en las lecturas sobre los sensores (*minValid* y *maxValid*). El modelo genera una excepción *NotValidRTDData* sobre un *RTDBase d* si: $\neg(minValid \leq ValorNuevo_d \leq maxValid)$.

Al momento de la implementación, se presentó el problema de realizar comparaciones y operaciones de valor *absoluto* sobre un dato genérico *RTD<T>*, pues *T* podría ser de una clase escalar comparable (integer, float, etc.) o de una clase más compleja como un punto en el espacio, un pixel, una imagen, etc. Este problema fue resuelto agregando un atributo *validatorRtd* para implementar las mencionadas validaciones. *validatorRtd* respeta la interface *ValidatorRtdInterface* la cual prevé dos métodos: *validRtdDataValidRange*() para validar el rango del dato *d* y *validRtdMaxDataError*() para validar *maxDataError*. Estas validaciones se llevan a cabo en el método *setData*() de *RTDBase*, donde además se actualiza el *VILB* con el instante actual de asignación (*now*). De la siguiente manera:

```
abstract public class RtdBase<T> extends Rtd<T>  {
…
  public void setData(T data) throws RtdException {
    …
    if (validatorRtd != null) {
      // data berween min and max values only if defined
      if (minValid != null && maxValid != null) {
        if (!validatorRtd.validRtdDataValidRage(minValid, maxValid, data)) {
          throw new NotValidRtdData("Data is not valid in range.");
        }
      }
      // only set data change if delta is greater than MaxDataError
      if (maxDataError != null && this.data != null){
        if (validatorRtd.validRtdMaxDataError(this.data, data, maxDataError)) {
          this.data = data;
        }
       } …
```

En la implementación se incluyen una serie de validadores pre-construidos para los tipos de datos escalares más comunes, implementando la interface *ValidatorRtdInterface*. Ellas son: *ValidatorRtdByte*, *ValidatorRtdInteger*, *ValidatorRtdFloat*, *ValidatorRtdDouble* y *ValidatorRtdLong*.

Como se comentó en la sección 1, los *RTDBase* se clasifican, a su vez, de acuerdo al objeto externo que reflejan. Pueden ser continuos (*RTDBaseContinuous*) ó discretos (*RTDBaseDiscrete*). Los continuos son actualizados con muestras periódicas mientras que los discretos son actualizados en forma esporádica, solo cuando cambia el valor externo. En el modelo, ambos se definen como especializaciones de la clase *RTDBase*. En un *RTDBaseContinuous* su validez depende de su *edad*. La *edad* es el tiempo desde su última actualización, hasta el instante actual. El vencimiento del dato (*datadedline*) se define en base a establecer la *edad máxima* (*maximumAge*) ([17]) de un determinado *RTD d*. Superada esa edad, el dato pierde vigencia. Este concepto permite calcular el límite superior del VI_d como: $VIUB_d(t) = VILB_d(t) + \text{maximumAge}_d$. De esta manera es posible validar la *consistencia temporal absoluta* ([10]).

En el caso de transacciones periódicas, es posible garantizar la *consistencia temporal absoluta* de un *RTDBaseConinuous d* si se considera un período $P \leq MA(b)/2$ ([8]). Para esto, el modelo calcula un atributo *period*, que puede ser útil al planificador de transacciones. Es posible recuperarlo invocando a *getPeriod*().

3.2 RTD Derivados

Los *RTDDerived* son datos *calculados,* cuyo valor se determina a partir de un *Conjunto-Lectura* (*Read-set*) de otros *RTD*. Consecuentemente, el intervalo de validez del *RTDDerived* resulta de la intersección de todos los intervalos de valides de los datos involucrados en el cálculo, cuando dicho intervalo es no vacío. Esto se denomina consistencia temporal relativa ([10]). Para un *RTDDerived d*, el *Conjunto-Lectura* cumple con la consistencia temporal relativa, si existe intersección entre los VI_d de sus elementos, $\bigcap\{VI_x(t)|x \in \text{ReadSet}_d\} \neq \varnothing$ y se define un intervalo de validez relativa VI_d con:

$$VILB_d(t) = Max\{VILB_x(t)|x \in \text{ReadSet}_d\}$$
$$VIUB_d(t) = Min\{VIUB_x(t)|x \in \text{ReadSet}_d\}$$

El re-cálculo de datos derivados y la validación de consistencia temporal relativa son de importante impacto en un *RTDBS*. El modelo re-calcula el VI_d de un *RTDDerived* al momento de recuperar su valor en el método *getData*() y valida que no se haya superado su vencimiento (*datadeadline*).

El modelo prevé dos formas para actualizar el valor de un *RTDDerived d*. La primera es que la transacción sea la responsable de realizar el cálculo de la derivación y actualizar el valor utilizando *setData*(). La segunda es que el diseñador pueda definir la estrategia de cálculo de la derivación y su peor caso de tiempo de ejecución (*WCET: Worst Case Execution Time*). Esto permite validar si es posible cumplir con el vencimiento antes de realizar el cálculo. Para esto se ofrece una clase abstracta *AbstractCalcRtdDerivation* con un método *calc*() donde se debe definir el algoritmo de cálculo, y un atributo *worstCaseExecutionTime* donde registrar el *WCET*. De esta manera, el método *calc*() de *RTDDerived*, puede realizar la siguiente validación:

```
…
if (derivation != null){
  if (this.getValidityIntervalUpperBound()- this.derivation.getWorstCaseExecutionTime() <= now) {
       throw new RtdDataHasExpired("Rtd Derivation it's not possible prior to DataDeadline.");
  }
  derivation.calc(this);
}…
```

4 Test y verificación de la implementación

Para la verificación se ha definido una clase de test *RtdTest* que representa a un objeto de tiempo real con atributos *RTD* continuos, discretos y derivados utilizado para verificación también en [11]. La clase *RtdTest* se muestra en la *Figura 2*.

El código de la clase *RtdTest* puede descargarse del sitio de *Google-Code* [12]. Los atributos *RTDBase* de la clase se inicializan como sigue:

cpuWorkLoad: maxAge = 4 seg,   maxDataError = 5,   minVal=0, maxVal=100
memWorkLoad: maxAge = 6 seg,   maxDataError = null, minVal=0, maxVal=100
loadTrend: (RTD discreto),   maxDataError = null, minVal=0, maxVal=100

Además se define un *RTDDerived measureVal* que se calcula como:

measureVal = (cpuWorkLoad X 0.5) + (memWorkLoad X 0.3) + (loadTrend X 0.2).

Para esta derivación se implementa una clase de cálculo *measureValCalc* (extendiendo *AbstractCalcRtdDerivation*) con un *WCET* de 5 milisegundos.

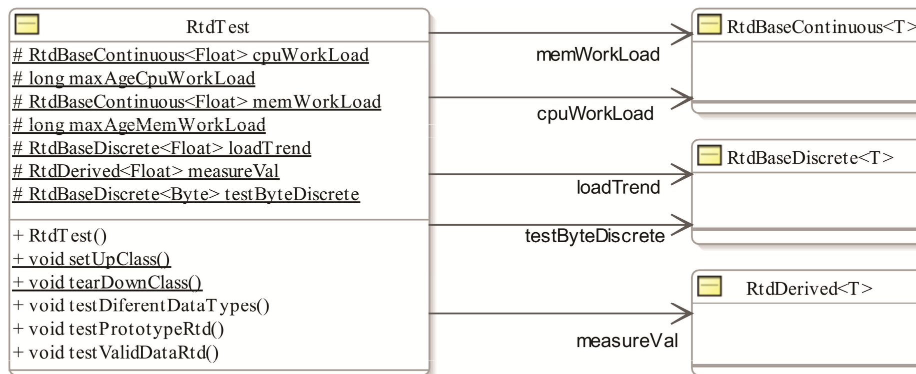

Fig. 2. Diagrama de Clases Java para pruebas de *RTD.*

En base a estas definiciones previas se diseñan dos unidades de prueba que serán ejecutadas utilizando *JUnit* ([18]), herramienta para pruebas de unidad en *JAVA*:

1. *testValidDataRtd*: para verificar maxDataError y valores válidos (minVal, maxVal)
2. *testPrototypeRtd*: para verificar la validación de consistencia temporal.

4.1 Unidad de Prueba *testValidDataRtd*

En primer lugar esta unidad de prueba, verifica la implementación del intervalo de validez de un *RTDBase*. Considerando los valores configurados para *cpuWorkLoad*, realiza tres casos de prueba, asignando los siguientes valores float: 100.0001 (fuera de rango superior), -0.00001 (fuera de rango inferior) y 70 (valor correcto). Las salidas de la ejecución de la prueba son las siguientes:

```
cpuWorkLoad=    100.0001;  Not valid data for cpuWorkLoad<Float> type
cpuWorkLoad=    -0.0001;   Not valid data for cpuWorkLoad<Float> type
cpuWorkLoad=    70.0;      Valid data for cpuWorkLoad<Float> type
```

Luego, se generan tres casos de prueba para verificar la funcionalidad de *maxDataError*. Para esto, se asigna un valor inicial de *cpuWorkLoad* de 35. Y se prueba posteriormente asignando los siguientes valores: 39.9999 (no cambia el valor pero refresca el *VI*) y 45 (asigna nuevo valor y refresca *VI*). A continuación se muestran las salidas de la ejecución:

```
set=35;       cpuWorkLoad=35.0;now=1339543696175;VILB=1339543696175;VIUB=1339543700175
set=39.9999;cpuWorkLoad=35.0;now=1339543696176;VILB=1339543696176;VIUB=1339543700176
set=45;       cpuWorkLoad=45.0;now=1339543696185;VILB=1339543696185;VIUB=1339543700185
```

4.2 Unidad de Prueba *testPrototypeRtd*

Esta Unidad de Prueba, verifica la validación de consistencia temporal absoluta y relativa. Para esto asigna los siguientes valores iniciales para los *RTDBase*: *cpuWorkLoad*=60, *memWorkLoad*=35 y *loadTrend*=50. Posteriormente, se verifica el cálculo correcto del *RTDDerived measureVal* que corresponde con el valor 50.5.

Luego, se deja pasar 3 segundos de tiempo y se analiza la vigencia temporal de los datos. Verificando que no se ha alcanzado el vencimiento de ninguno de los *RTD:*

```
3 secs after; Initial=1339545462036; Current=1339545465045
cpuWorkLoad=60.0;        VILB=1339545462036;        VIUB=1339545466036
memWorkLoad=35.0;        VILB=1339545462036;        VIUB=1339545468036
loadTrend=50.0;          VILB=1339545462036;        VIUB=9223372036854775807
mesurableVal=50.5;       VILB=1339545462036;        VIUB=1339545466036
```

Inmediatamente, se deja pasar 1 segundo de tiempo y verifica que *cpuWorkLoad* se ha vencido y que no es posible calcular la derivación de *measureVal* antes del vencimiento*:*

```
1 sec after; Initial=1339545462036; Current=1339545466046
cpuWorkLoad: Rtd Data has expired within lower and upper bound limits.
VILB=1339545462036;      VIUB=1339545466036
memWorkLoad=35.0;        VILB=1339545462036;        VIUB=1339545468036
loadTrend=50.0;          VILB=1339545462036;        VIUB=9223372036854775807
mesurableVal: Rtd Derivation it's not possible prior to DataDeadline.
VILB=1339545462036;      VIUB=1339545466036
```

Posteriormente, se actualiza solo el valor de *cpuWorkLoad* a 40 y se deja pasar 2 segundos de tiempo. Se verifica que *memWorkLoad* se ha vencido, pero que *cpuWorkLoad* está vigente:

```
2 secs after; Initial=1339545462036; Current=1339545468048
cpuWorkLoad=40.0;        VILB=1339545466047;        VIUB=1339545470047
memWorkLoad: Rtd Data has expired within lower and upper bound limits.
VILB=1339545462036;      VIUB=1339545468036
loadTrend=50.0;          VILB=1339545462036;        VIUB=9223372036854775807
mesurableVal: Rtd Derivation it's not possible prior to DataDeadline.
VILB=1339545466047;      VIUB=1339545468036
```

5 Conclusiones y Trabajos Futuros

Este trabajo presenta una implementación del *Tipo de Dato Abstracto Parametrizable RTD*. En esta implementación, se manifestaron cuestiones relativas a la codificación en JAVA de las entidades conceptuales. Estas cuestiones han impulsado la revisión y ampliación del modelo teórico. Se ha generado una solución extensible que da mayor cobertura a la solución. Se ha definido un conjunto de unidades de prueba que permiten verificar la funcionalidad. El paquete JAVA resultante, debidamente verificado, ha sido publicado en un sitio web.

A futuro se ampliará sobre el tratamiento de *RTD derivados y consistencia temporal relativa*, expandiendo además la implementación en otros lenguajes.

Referencias

[1] R. Elmasri and S. Navathe, *Fundamentals of Database Systems 5/E*, 5/E ed.: Addison-Wesley, 2007.

[2] C. Combi*, et al.*, "Capturing Temporal Constraints in Temporal ER Models," in *Proceedings of the 27th International Conference on Conceptual Modeling*, ed Berlin, Heidelberg: Springer-Verlag, 2008, pp. 397-411.

[3] Ellen Rose and A. Segev, "TOODM - A Temporal Object-Oriented Data Model with Temporal Constraints," in *Proceedings of the 10th International Conference on Entity-Relationship Approach (ER'91)*, San Mateo, California, USA, 1991.

[4] B. P. Douglass, *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*: Addison-Wesley, 2002.

[5] O. M. G. (OMG). (2009, OMG Document Number: formal/2009-11-02). *A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems*. Available: http://www.omg.org/spec/MARTE/1.0/

[6] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," in *Proceedings of the 14th VLDB Conference*, 1988.

[7] J. A. Stankovic*, et al.*, "Misconceptions About Real-Time Databases," *IEEE Computer,* vol. 32, pp. 29-36, 1998.

[8] M. Xiong*, et al.*, "Maintaining Temporal Consistency: Issues and Algorithms," in *Proceedings of International Workshop on Real-Time Database Systems*, 1996, pp. 2-7.

[9] K. Ramamritham, "Real Time Databases," *International Journal of Distributed and Parallel Databases,* vol. 1, pp. 199-226, 1993.

[10] K. Ben*, et al.*, "Maintaining temporal consistency of discrete objects in soft real-time database systems," *Computers, IEEE Transactions on,* vol. 52, pp. 373-389, 2003.

[11] C. Buckle*, et al.*, "Abstract Data Type for Real-Time Database Systems," in *XVII Congreso Argentino de Ciencias de la Computación*, UNLP. La Plata, Argentina, 2011.

[12] D. Barry and C. Buckle. (2012, *real-time-db Java Project*. Available: http://code.google.com/p/real-time-db/

[13] J. J. Prichard*, et al.*, "RTSORAC: A Real-Time Object-Oriented Database Model," *In The 5th International Conference on Database and Expert Systems Applications,* pp. 601-610, 1994.

[14] L. C. DiPippo and L. Ma, "A UML Package for Specifying Real-Time Objects," *Computer Standards & Interfaces* vol. 22, pp. 307-321, 2000.

[15] N. Idoudi*, et al.*, "Structural Model of Real-Time Databases: An Illustration," in *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, 2008, pp. 58-65.

[16] M. Amirijoo*, et al.*, "Specification and management of QoS in real-time databases supporting imprecise computations," *Computers, IEEE Transactions on,* vol. 55, pp. 304-319, 2006.

[17] B. Adelberg*, et al.*, "Applying update streams in a soft real-time database system," *Proceedings of the 1995 ACM SIGMOD,* vol. 24, pp. 245-256, 1995.

[18] K. Beck and E. Gamma. (2008, *JUnit Cookbook*. Available: http://junit.sourceforge.net