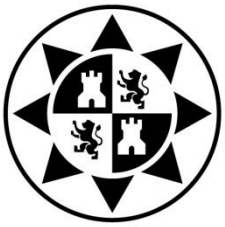

Universidad
Politécnica
de Cartagena

industriales
etsii UPCT

IMPLEMENTACION DE UN FRAMEWORK DE COMPONENTE EN EL LENGUAJE DE PROGRAMACION ADA

Titulación: Ingeniería en Automática y Electrónica Industrial
Intensificación:
Alumno: Francisco José Hernández Villalón
Director: Diego Alonso Cáceres

Cartagena, 15 de Diciembre de 2012

Contenido

Tabla de Ilustraciones

1. Introducción.

1.1. Contexto

La memoria del trabajo que aquí se presenta esta enmarcada en el Proyecto Explore, un *Framework* estructurado siguiendo la forma de ciertos patrones de diseño cuyo objetivo principal son los sistemas con fuertes requisitos temporales (*hard Real-Time Requirements*).

El uso de patrones de diseño para solventar problemas de diseño viene motivado por que es una rama muy madura de las ciencias de la comunicación que aporta soluciones robustas y bien documentadas. Por ello, que cada conjunto de problemas de diseño y contexto de desarrollo, los patrones definen un conjunto de soluciones para tales problemas en dicho contexto [Gamma 95]. Cada patrón, por tanto, está orientado a resolver un problema específico y la elección de uno u otro es lo que marcará el éxito o el fracaso del software en su conjunto.

El *lenguaje de patrones* debe dejar claro cuáles son los *problemas claves* que hay que resolver, qué patrones se deben emplear para resolverlos, por dónde empezar, qué camino seguir y qué alternativa elegir cuando el camino se bifurca. En este sentido, un lenguaje de patrones nos guía que decisiones tenemos que tomar.

En el Proyecto Explore, dos son los objetivos que se persiguen. El primero es proporcionar una solución al problema de cómo asignar las actividades de las máquinas de estados de los componentes de V3CM a un conjunto de tareas planificables. El segundo objetivo es comenzar a definir los lenguajes de patrones contemplados en este proyecto. Para alcanzar el segundo objetivo se definieron una serie de objetivos parciales:

- La definición de lenguaje de patrones para el diseño de aplicaciones con requisitos de tiempo real estricto.
- La definición de estrategias de transformación de modelos de componentes a módulos orientados a objetos y a lenguajes de implementación final (Ada, C/C++/C#, Java, etc.) conforme a los patrones de diseño seleccionados.

En el Proyecto Explores se utiliza un lenguaje de modelado ya mencionado en párrafos anteriores conocido como V3CM [Alonso 08] es un lenguaje de modelado para diseñar aplicaciones según un enfoque de desarrollo basado en componentes. Este enfoque establece que el software debe ser construido mediante el ensamblado de partes predefinidas (los componentes software) siguiendo un conjunto de reglas de composición. V3CM define tres vistas complementarias: estructural, de coordinación y algorítmica.

- La vista estructural describe la estructura estática de las aplicaciones, los componentes que la forman y cómo se interconectan a través de sus puertos.

- La vista de coordinación describe el comportamiento interno de cada componente simple en función de su estado y de la ocurrencia de diferentes tipos de eventos, por ejemplo cambios en sus puertos de entrada, invocación de sus servicios, etc. La vista de coordinación se define mediante máquinas de estado jerárquicas.
- La vista algorítmica describe el algoritmo o algoritmos ejecutados por el componente. La vista algorítmica mediante diagramas de actividad.

El comportamiento de un componente complejo se construye (o se deduce) a partir del comportamiento de los componentes que lo forman. Un componente simple no es válido hasta que se le ha asociado una máquina de estados válida y una máquina de estados no es válida hasta que no se asocien actividades y acciones de entrada y salida a cada uno de sus estados. De esta manera cada componente simple tiene una máquina de estados y cada estado de dicha máquina una actividad definida en la vista algorítmica.

En la ilustración 1 se representa las tres vistas de V3CM.

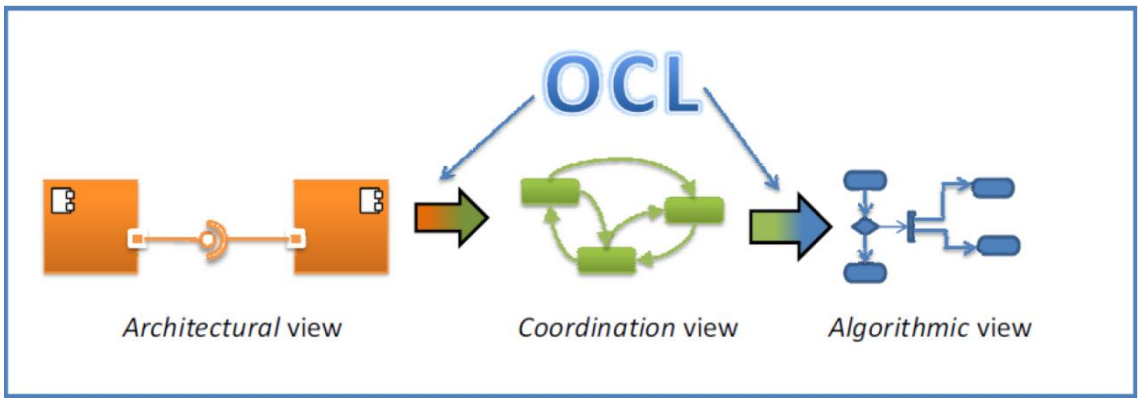

Ilustración 1: Vista del diseño en V3CM

Las tres vistas de V3CM proporcionan un nivel de abstracción mucho más alto que el ofrecido por los lenguajes de modelado o programación orientados a objetos.

Inicialmente se definió una arquitectura *Framework* de ejecución de V3CM mediante un caso de estudio desarrollado en lenguaje *JAVA*, el siguiente paso fue separar el *Framework* en dos partes, una parte tenía que tener el *Framework* como tal y la otra parte tenia que ser un caso de estudio que utilizara el *Framework* para su ejecución y demostrar el buen funcionamiento del conjunto.

El Lenguaje V3CM esta pensado para modelar aplicaciones con requisitos de tiempo real (*Hard Real-Time Requirements*). *Java* no resulta totalmente adecuado para implementar el *Framework*, ya que no soporta características de tiempo real estricto. Por ello se decidió realizar un cambio radical en la implementación y se opto por un lenguaje que estuviera en contacto directo con el sistema operativo. Se eligió *C++* al satisfacer los requisitos pedidos. Se realizo la traducción de *Framework* original en *Java* a uno escrito en *C++*, posteriormente se ha traducido a *Ada 2005*, que es el objeto de este proyecto. Donde los requisitos temporales se ajustan con éxito [FASL 2010].

Para realizar la integración de la programación de *Java* a *C++* se ha realizado una serie de modificaciones importantes en el funcionamiento de *Framework* que se proceden a listar.

- Se modelaron las transiciones entre estados y las condiciones para esas transiciones.
  Las transiciones modelan los cambios entre estados y las condiciones las guardas necesarias para que estas se produzcan.

- Se modelaron los eventos y las condiciones para ello.
  Los eventos modelan situaciones que pueden desembocar en un cambio de estado repentino, como puede ser la finalización de una tarea o la recepción de un mensaje desde otro componente.

- Se implementaron las conexiones por Sockets entre componentes.
  La conexión por Socket aporta una vía de comunicación en el despliegue de los componentes así como cierta flexibilidad para el manejo de conexiones.

- Se implemento un *deployer* [ICSOFT] que maneja la distribución de los componentes entre diferentes procesos y la conexión de esos componentes dentro del mismo proceso.

- Este *deployer* es capaz de realizar el despliegue de todos y cada uno de los componentes, instanciando sus regiones, puertos, actividades, estados, y transiciones.

- Se modela un proceso por el cual se recibe las descripciones de un componente en XML y se instancia dinámicamente dicho componente, enlazándose con las actividades en ejecución. Este proceso forma parte del *deployment* o despliegue.

- Se implementa la serialización de los mensajes en forma XML. Esta serialización es fundamental para garantizar la máxima compatibilidad entre sistemas.

En la Traducción del *Framework* escrito en C++ a *Ada2005*que es el objeto de este proyecto, no se han tenido que realizar grandes transformaciones del *Framework*, por seguir la misma estructura de clases, los únicos cambios realizados que merecen una mención se describen en el apartado 2.3.

1.2. Objetivos.

El objetivo principal de este proyecto es la traducción de *FrameworkAda 2005* y mantener la compatibilidad entre las dos implementaciones de *Framework* que se realizaron anteriormente *C++* y *Java*. Cabe destacar que la implementación en *Java* es el *Framework* de partida de donde derivan estas dos últimas *C++* y *Ada 2005*.

Los tres objetivos de este proyecto son.

- Traducción de *Framework* en *C++* a uno escrito en *Ada 2005* Para ello se asegura que:
    - Se implementaran las clases siguiendo la misma estructura de la anterior versión *C++*.
    - Se considera que el proyecto no alcanza su madurez hasta que no quede perfectamente documentadas sus funcionalidades. Por ello se detallan extensamente cada una de las clases que componen el *Framework*
- Demostrar la validez de dicha implementación desarrollando un caso de estudio, haciendo uso de *Framework* expuesto.
- Revisión final del código y de la filosofía de desarrollo seguida.

Cabe destacar que algunos aspectos no se han tenido en cuenta en este proyecto. Como son los despliegues de componentes mediante *Sockets* (*deployment*), Implementación de al serialización de los mensajes en formato XML, y modelado de procesos por el cual se recibe la descripción de un componente serializado en XML y la instanciación dinámica de dicho componente.

1.3. Contenido y estructura del documento.

- Introducción:

  Se describe cuales han sido los antecedentes y la motivación para la realización de este proyecto y cuál es el contexto general en el cual esta enmarcado. Se presentan los objetivos del proyecto explicando brevemente como se lograran cada uno de ellos. Por último se da una relación del contenido de esta memoria.

- Software de partida y cambios realizados:

  Breve descripción de las versiones anteriores que es la base de partida para la realización de este proyecto y los cabios realizados para una correcta traducción a *Ada 2005*.

- Descripción del código en *Ada 2005*:

  Se describe las diferentes estructuras de las clases y de los objetos implementados en *Ada 2005*, para que tenga una estructura similar a las versiones anteriores del *Framework*.

- Documentación de cada una de las clases implementadas en *Ada 2005*:

  Se provee de una documentación sobre todas las clases que componen el *Framework*.

- Caso de Estudio:

  En este apartado se explica en detalle en qué consiste la aplicación que se ha utilizado para probar el *Framework*. Se describen de forma detallada la estructura general de la aplicación y se analizan los resultados obtenidos.

- Conclusiones

  Se comprobará que los objetivos marcados para este proyecto se han cumplido.

2. Software de partida y cambios realizados.

En esta sección se describen las características principales del software anteriores del *Framework*, tanto de *Java* como de *C++*. Cabe destacar que esta documentación no hace una descripción detallada de las anteriores versiones (puesto que ya existe sendas documentaciones), se realiza una breve descripción para entender los pasos seguidos hasta llegar a las actuales versiones del *Framework*.

2.1. Descripción del software original Java.

Para el desarrollo del *Framework* original se establecieron una serie de requisitos a cumplir cuya meta era garantizar que se alcanzara los objetivos de tiempo real y planificación impuestos desde el principio. Dichos requisitos se ilustran en la *Tabla 1*.

Requisitos Sobre las actividades.
R1. Las ejecuciones de las actividades asociados a los estados deben ser asignadas a diferentes tareas.
R2. Las actividades deben ser auto-contenidas. Se trata de minimizar el acoplamiento entre la tarea que ejecuta la actividad y la actividad misma. Cuanto menos tenga que conocer la una de la otra mejor.
R3. Se debe modelar el máximo intervalo de tiempo posible entre dos ejecuciones sucesivas de la actividad asociada a un estado.
R4. La actividad a de ser lo más corta posible. El tiempo de computación de las tareas a de ser inferior a su periodo de ejecución.
Requisitos sobre las tareas.
R5. A las tareas debe poder asignarles las actividades que deben ejecutarse.
R6. Las tareas deben de poder planificar la ejecución de sus actividades.
R7. Las tareas pueden ser a su vez planificadas. Debe poder aplicarse una planificación *rms*.
Requisitos sobre los puertos y las comunicaciones.
R8. Deben soportarse las interfaces y los tipos de comunicación definidos en los puertos de los componentes V3CM
R9. El intercambio de datos en los puertos debe ser seguro garantizando la consistencia de los datos y la ausencia de interbloqueo.
R10. El acceso a los puertos ha de ser eficiente.
R11. La implementación ha de ser sencilla, para no complicar en la medida que sea posible el sincronismo entre tareas y/o componentes.
R12. Debe ser viable extender la solución para soportar la distribución de componentes.
Requisitos sobre máquinas de estado.
R13. Las actividades asociadas a sus estados deben cumplir los requisitos R1 a R4.
R14. Deben considerarse explícitamente las acciones de entrada y salida de los estados.
R15. Deben implementarse regiones compuestas por máquinas de estados no jerárquicas.
R16. Los cambios que se producen por acciones realizadas en una región pueden disparar transiciones en otras regiones.
R17. Debe proporcionarse un medio para animar las máquinas de estados, en lazando el código de la máquina de estados con las fuentes de los eventos que provocan sus transiciones.

Tabla 1: Requisitos impuestos sobre el Framework

De los requisitos expuestos anteriormente, cabe resaltar:

- Los requisitos que se imponen sobre las tareas para permitir la ejecución de aplicaciones con requisitos de tiempo real.
- Los requisitos que se imponen a la implementación de las máquinas de estado, pues en ellas radica el éxito de la planificación RMS (*Rate Monotonic Scheduling*).
- Los requisitos que se imponen sobre la distribución de componentes.

El soporte a las características de tiempo real de las aplicaciones es uno de los factores claves desde el punto de vista de la implementación de *Framework*.

El problema principal para conseguir este objetivo radica en la diferencia de diseños entre V3CM y los sistemas de tiempo real. Mientras que el primero (V3CM) se diseña definiendo máquinas de estados asociadas a un componente, los sistemas de tiempo real se orienta hacia la ejecución de tareas procesos y su distribución. Además, el hecho de cada aplicación (o incluso dentro de la misma, según los recursos disponibles, etc.) no hacia más que complicar esta tarea de diseño. Por ello la solución que se adoptase debía de ser muy flexible y robusta. Son precisamente estas dos características las que hacían que se tomaran soluciones más inmediatas.

La solución ideal debería recoger las actividades asociadas a todos los estados de todas las máquinas de estado de todos los componentes, barajarlas, asignarlas arbitrariamente a un numero también arbitrario de tareas, ejecutar dichas tareas y que todo funcione en la forma definida por las máquinas de estados originales. En la práctica la asignación de actividades a las tareas no seria arbitraria, sino en función de la imposición del método de planificación seleccionado y de los heurísticos de definición y agrupación de tareas que suele emplearse para aumentar el rendimiento y facilitar la planificación. Pero como los algoritmos y los heurísticos son muy variados, la pretensión de poder barajar las actividades y repartirlas arbitrariamente en un arbitrario numero de tareas no era del todo exagerada. La figura 2 muestra gráficamente los problemas que se le hizo frente.

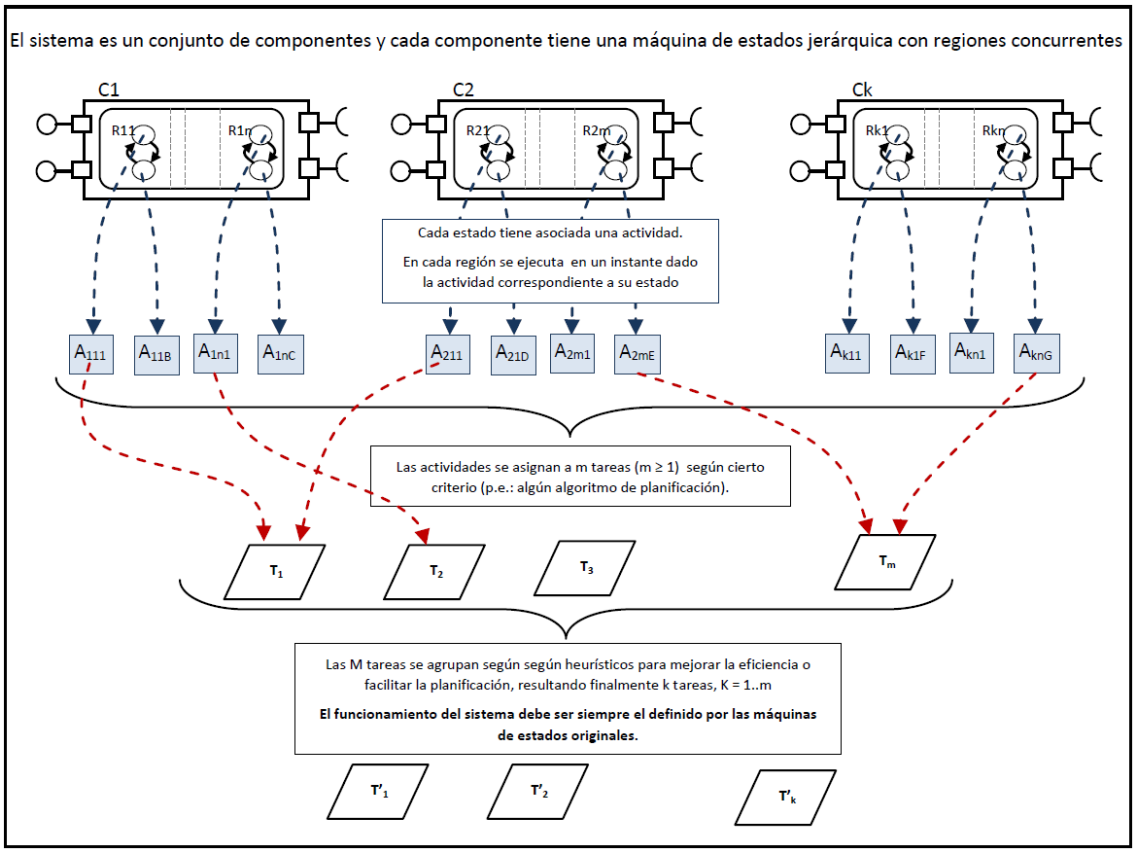

Ilustración 2: Reparto de actividades en tareas

Una vez elegida la arquitectura que tendría el sistema, y los requisitos que debe de cumplir, se realizo la implementación de *Framework* en base a ellos. De manera que el código de una aplicación formada mediante dicho *Framework* pueda ser clasificado en uno de estos tres conjuntos:

- CS1: Código que proporciona el soporte de ejecución conforme a los requisitos de las aplicaciones de dominio. CS1 debe soportar características como diferentes políticas de concurrencia, planificación de tiempo real, procesamiento de eventos y señales, manejo de la memoria, etc., incluso si es necesario sacrificar otras características.

- CS2: Código que provee una interpretación orientada a objetos de los conceptos de V3CM. Por ejemplo como se relacionan los componentes y los puertos a objetos, los *state-chart*, puertos de comunicaciones, etc.

- CS3: Código correspondiente a la funcionalidad definida en el modelado V3CMde la aplicación.

Estos tres conjuntos están dispuestos de tal manera que CS1 y CS2 constituyen el *Framework* en si, mientras que CS3 debe ser integrado con el objeto de obtener el código final para una aplicación específica. Los puntos de especialización, o *Hot-Sport*, están definidos en CS2 que proporciona un cierto desacoplamiento entre CS1 y CS2 permitiendo una evolución por separado.

Para Ilustrar el resultado final. Se muestra en la Figura 3. Un diagrama de clases cuyo objetivo no es otro sino clarificar lo expuesto anteriormente.

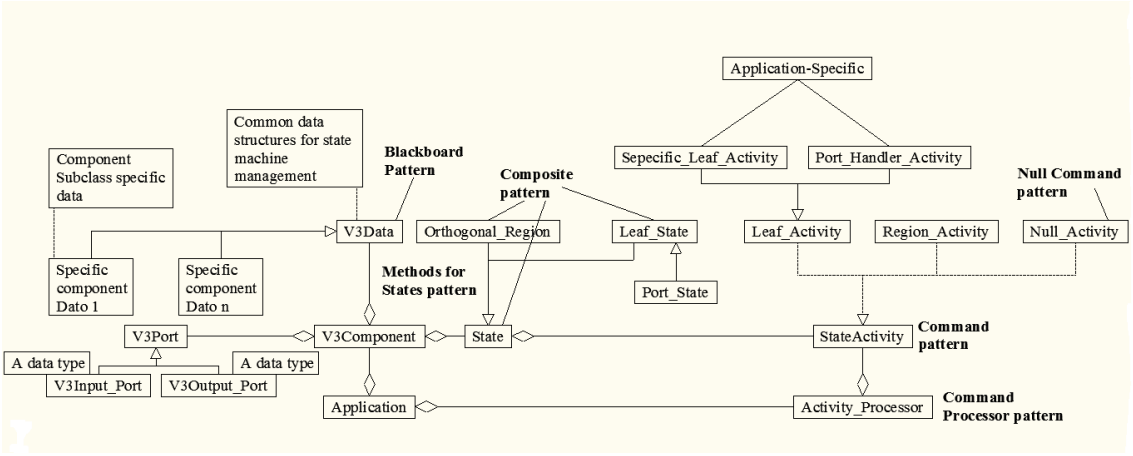

Ilustración 3: Diagrama de clases versión Java

- Los patrones *command processor* y *command* se utilizan para separar la petición de un servicio de su ejecución, de forma que el cliente y servidor sean ejecutados de manera asíncrona. En el *Framework* se definen actividades (StateActivity) asociadas a estados (State), que son entendidas como comandos, así, estas pueden ser asignadas a procesos de comandos, tarea desempeñada por la clase (ActivityProcessor).

- El patrón *composite* nos permite tratar de manera homogénea componentes y contenedores. En nuestro caso los componentes son las regiones (OrthogonalRegion) y los componentes son los estados no jerárquicos (State) y (LeafState).

- Se opta por el patrón *methods for states* (en lugar de *object for states*) que nos permite gestionar los cambios de estado. Según este patrón, el comportamiento asociado a cada estado se organiza en tablas de funciones que son llamadas de acuerdo al estado en el que se encuentran la máquina de estados. (State) y (V3Component) son las clases de realizar este cometido.

- El patrón *blackboard* se usa para proporcionar una estructura de datos globales al componente sobre la que puedan trabajar de forma concurrente diferentes tareas. Este patrón suele traer consigo ciertos problemas añadidos, como son el sincronismo y la consistencia de los datos, pero, dado que tratamos con espacio de memoria relativamente pequeño (cada componente tiene su propia pizarra) estos inconvenientes no suponen un problema real en la practica.

- Puestos que alguno de los estados modelados pueden no tener asociada ninguna actividad, se define una actividad (NullActivity) siguiendo el patrón *null object* para, así, poder tratar de forma regular todos los estados.

2.2. Descripción del software C++.

Para entender completamente la estructura y funcionalidad del *Framework*, es necesario obtener una idea bastante precisa de los cambios que se han realizado en el *Framework* a lo largo de sus etapas. Se pretende enumerar y describir las modificaciones sufridas en la traducción al lenguaje C++.

El primer esbozo que se realizo del *Framework*, pese a que aparecen la mayoría de clases definidas y la mayor parte de los patrones aplicados, faltan por perfilar ciertos detalles de la implementación final de *Framework*. De hecho la primera versión se corresponde con una aplicación concreta [DT Explore 2010], marco en el cual surgió el interés del desarrollo de un *Framework* con requisitos de tiempo real.

Los cambios que se hicieron en esta fase, fuero el cambio de concepto de región ortogonal por el de estados jerárquicos desaparece (OrthogonalRegion en pos de HierarchicalState y HierarchicalActivity). Cada región ahora aparece representada por una clase denominada HierarchicalState, que contiene la información relativa a ella. Cada estado posee una actividad que se ejecuta por el procesador de comandos. La actividad relativa al estado jerárquico es HierarchicalActivity encargada de ejecutar cada una de las actividades de los estados que la componen.

A la hora de instanciar el estado jerárquico se hace a través de su implementación final (Region) que permite la construcción de las regiones

tomando como parámetros de entrada un vector de estados (State), la información relativa al componente V3ComponentData, el nombre de la región y una serie de identificadores que permite identificar unívocamente a ese componente y a esa región.

Siguiendo el modelo de máquinas de estado jerárquico clásico, se definen las condiciones (Condition) como los guardas para producir una transición. Para el caso de los eventos, se hace uso de las pizarras comunes para los componentes, de manera que cualquier actividad en cualquier estado pueda introducir en la pizarra un mensaje (EventMsg) que conlleva el disparo de una transacción. Esto provee de cierta flexibilidad al *Framework* dado que proporciona un mecanismo de subscrición selectivo a todo aquel interesado en la recepción de un mensaje en concreto de tipo EventMsg.

Relativo a los mensajes cabe destacar que las clases Message y MessageHeader, ahora se combinan en una sola MSG, que en lugar de serializar los mensajes como una ristra de byte ahora lo hace como cadena de caracteres expresados según el lenguaje XML.

A demás se crean dos clases que derivan de MSG, EventMsg y V3PortMsg, especializaciones que tienen como objeto representar dos tipos diferentes de mensajes, los relativos a los puertos de comunicaciones y los relativos a los eventos.

Cuando se diseño el *Framework* se decidió a toda costa que varios elementos mantuvieran una referencia común a un objeto compartido. El objetivo, por tanto, es reducir al máximo las estructuras tales como semáforos, monitores y objetos protegidos, por ello, se optó por aplicar un patrón de diseño bien conocido, *copied value*. En lugar de pasar por referencia entre métodos, se pasan copia de un objeto.

El resultado final se puede observar en la figura 4, podemos observar como se relaciona y heredan las clases anteriormente descritas.

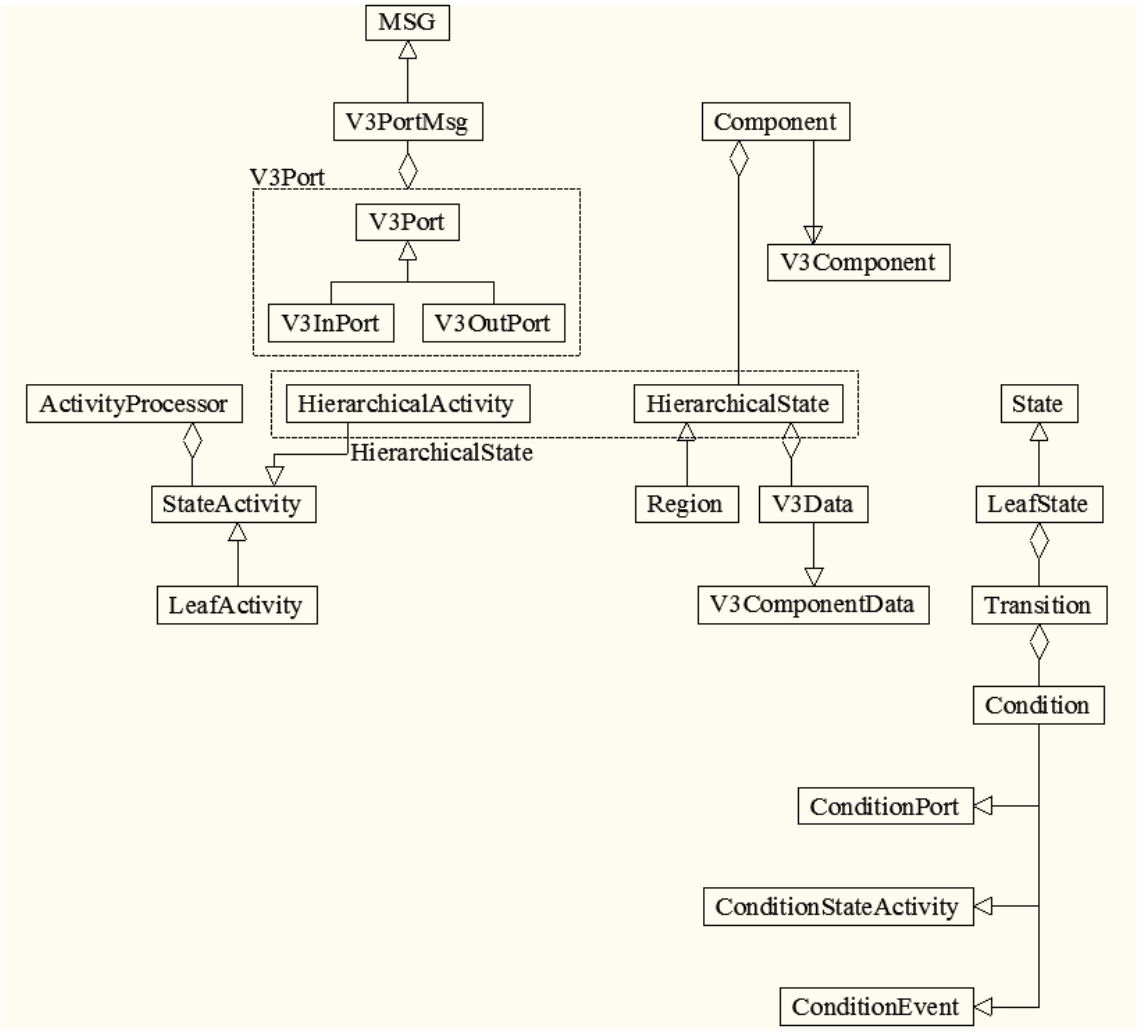

Ilustración 4: Diagrama de clases versión C++

2.3. Cambios realizados en el proyecto durante su traducción Ada 2005.

Se cambia el inicio de los nombres de las clases que empiezan por **V3** por **MinFr**, **V3_Component** por **MinFr_Component**, **V3_Component_Data** por **MinFr_Component_Data**, **V3_DataporMinFr_Data**, etc.

Otro de los cambios que se ha realizado en los objetos **Condition**, **Minfr_Code, Minfr_Component**, **Minfr_Component_Data**, **Sobrecargable** y **State_Activity**. Estos objetos en *C++* sus funciones y procedimientos y el propio objeto son de tipo abstracto (tienen definición pero no tiene implementación). Para dejar el código de una forma más elegante en *ADA* estos objetos se declaran de tipo *interface*. Los Objetos *interface* permiten que otros objetos hereden todos los método y funciones que define el objeto *interface* pero estos últimos no heredan la implementación, la tienen que implementar ellos o sus hijos. También hay que destacar que los objetos tipo *interface* no definen la implementación de sus métodos y funciones.

Los objetos **MinFr_Port**, **MinFr_State**, **Hierarchical_State**, **MSG**, **Leaf_Activity** en *ADA* pasan a ser objetos de tipo abstracto "*abstract*", porque no representan

ningún objeto concreto, se utilizan para ser padres de los objetos que heredan de ellos. Estos objetos tienen unas características especiales que se enumeran a continuación.

- No se permite crear objetos de tipo abstracto.
- Puede tener procedimientos y funciones primitivas abstractas (Sin Cuerpo).
    - Sera obligatorio definirlas en los hijos no abstractos.

Por lo mencionado en los puntos anteriores, que no esta permitido construir objetos de tipo abstracto y en C++ si esta permitido tenemos que realizar unas modificaciones para poder conseguir acceder y modificar los parámetros de estos objetos. Esto se consigue creando un procedimiento y pasándole por referencia el puntero (Dirección del Objeto) del objeto abstracto, posteriormente este procedimiento será llamado en los objetos hijos por las funciones constructoras de los objetos hijos, pasándole por referencia el puntero (Dirección del objeto hijo) ya creado por el constructor del objeto hijo. Esto se podrá ver de una forma más clara con el siguiente ejemplo.

Declaración del objeto padre con un procedimiento constructor.

```
Pakage Padre is

      Type Padre is abstract tagged private;
      Type Ptr_Padre is Access all Padre'class;-- Dirección del
      Padre y de los objetos de este tipo

      Procedure Constr_Padre (this: in out Ptr_Padre;····);

Private

      Type Padre is abstract tagged record
            ---
            ---
      End record;

End Padre;
```

Declaración del objeto hijo con la función constructora.

```
Pakage Padre.Hijo is

      Type Hijo is new Padre with private;
      Type Ptr_Hijo is Access all Hijo;-- Dirección del Hijo

      Function Constr_Hijo (·····) return Ptr_Hijo;

Private

      Type Hijo is new Padre with record
            ---
            ---
      End record;

End Padre.Hijo;
```

Cuerpo del objeto hijo.

```
Pakage body Padre.Hijo is

      Function Constr_Hijo (·····)return Ptr_Hijo is

      Begin
            Return Tmp:Ptr_Hijo do
                  Tmp := new Hijo; --Creamos el Objeto Hijo
                  ---
                  ---
                  Const_Padre(Ptr_Padre(Tmp),····);--Llamamos
                  al constructor padre y le pasamos por
                  referencia la dirección del objeto hijo
            End return;

      End Constr_Hijo;

End Padre.Hijo;
```

Al finalizar la traducción a *ADA* y comprobar el funcionamiento del *Framework*, se detectaron dos comportamientos erróneos de mal funcionamiento de *Framework*, el primero de ellos fue un exceso de consumo de los recursos de CPU, y el segundo un aumento imparable de memoria que iba utilizando el Framework en su ejecución. Para la solución de estos dos comportamientos se analizo detalladamente el código hasta dar con una solución.

En el caso de exceso de consumo de CPU el problema estaba a la hora de calcular el tiempo de ejecución de las actividades, una vez realizada la implementación correcta los niveles de consumo de CPU se redujeron a niveles normales y se mantuvieron estables.

Para el problema de aumento imparable de memoria, estaba en la creación de objetos. En Ada cuando se crea un objeto con la sentencia **`New,`** este objeto se almacena en la memoria no volátil y esta activa durante el tiempo en el que se esta la ejecución del *software.* Tras la modificación y eliminación de creación de objetos en bucles infinitos, se ha conseguido que el aumento de consumo de memoria se detenga y se mantenga en un nivel estable.

En los códigos *Framework* anteriores de Java y C++, no se verificaba si la conexión entre los distintos puertos era coherente. En esta versión de ADA a la hora de realizar la conexión de los puertos se verifica si dicha conexión es correcta o no, evitando a si que se produzcan conexiones entre dos puertos del mismo tipo, entrada o de salida. También se evita que se conecte el puerto así mismo.

Para verificar todo lo que se ha comentado en el párrafo anterior se ha utilizado el mecanismo que ofrece *Ada 2005* de depuración de errores con **excepciones**. Dichas excepciones se pueden definir como nueva excepciones que indiquen por pantalla los distintos errores mencionados en el apartado anterior y provocar la terminación del programa.

La clase que se ha modificado y gestiona estos errores con excepciones es "*MinFr_Port ver 4.3*" y más concretamente el procedimiento "*Set_Conjugate ver 4.3.3*".

En la figura 5 se puede observar la estructura final de ADA, como se relacionan y heredan las clases mencionadas anteriormente.

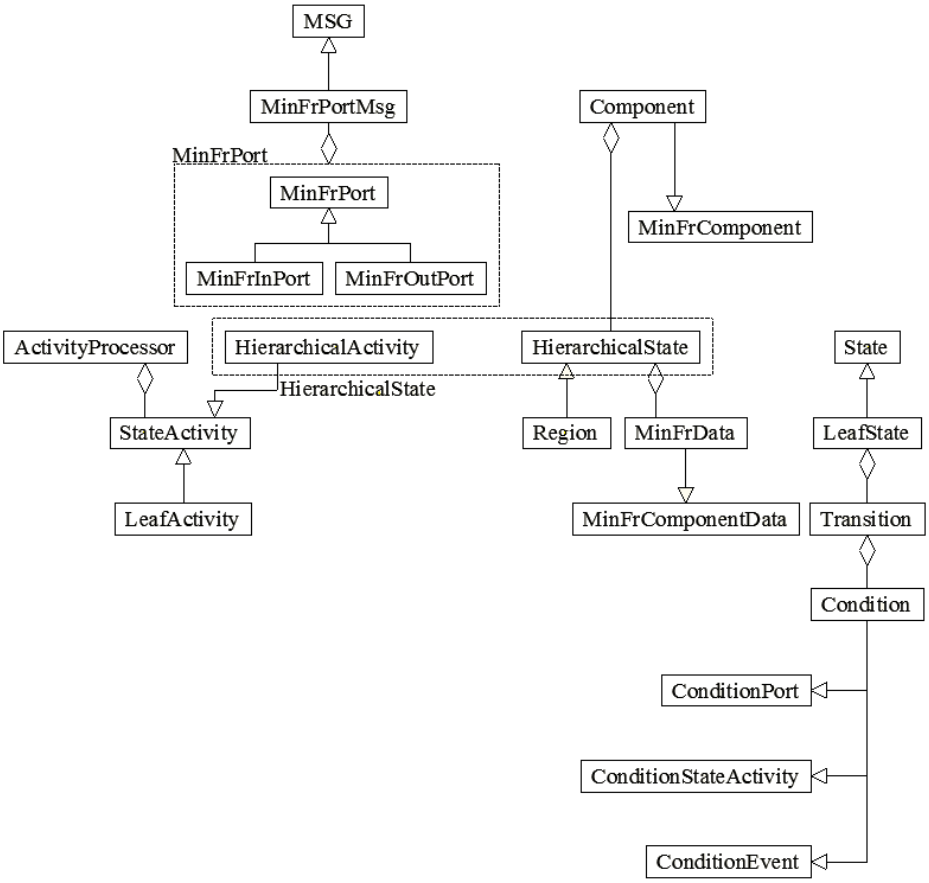

Ilustración 5: Diagrama de clases versión ADA

3. Descripción Código en *ADA 2005*.

Descripción de la estructura del código de *ADA*, cómo se estructuran las clases y cómo se crean los objeto para que tengan una estructura similar al código del *Framework* de la versión anterior de *C++*.

3.1. Estructura de las clases en ADA.

3.1.1. Especificación de las clases.

La estructura de la especificación de una clase es la siguiente.

```
Pakage <Nombre_de_la_Clase>is

      <Elementos públicos>
            <Variables>
            <Objetos>
            <Punteros> --Dirección de variables y objetos
            <Función Constructora del objeto>
            <Funciones y Procedimientos que interactúan con el
            objeto>
[Private]
      <Elementos privados>
            <Variables>
            <Definición de objetos>
            <Funciones y Procedimientos>

End <Nombre_de_la_Clase>;
```

Hay tres tipos de estructura de especificaciones de clases que se han utilizado para la traducción del *Framework*. Estos tres tipos se pueden clasificar en función del tipo de objeto que implementan, a continuación se enumeran y se pondrá un ejemplo de cada una de ellas.

- Clases con Objetos tipo *interface*

```
Pakage <Nombre_de_la_Clase>is

      type objeto_Interface is interface;

      Procedure Nombre_Proc_Abs (this : in out objeto_Interface;...)is
      abstract;

      Function Nombre_fucti_Abs (this : in out objeto_Interface; ...)
      return ...is abstract;

End <Nombre_de_la_Clase>;
```

Todos los procedimientos y funciones de esta clase que utilicen el objeto tipo interface son de tipo abstracto, esto quiere decir que estos métodos y funciones no tienen implementación, los encargados de implementarlos son las clases que heredan de ellas.

- Clases con Objetos tipo *abstracto*

Estas clases se pueden subdividir en dos tipos.

- Clases con objetos tipo abstractos que heredan de otros objetos.

```
Pakage <Nombre_de_la_Clase>is

        type Objeto_Hijo_Abs is abstract new Objeto_Padrewith private;

        Procedure Nombre_Proc_Abs (this : in out Objeto_Hijo_Abs;...)is
        abstract;

        Function Nombre_fucti_Abs (this : in out Objeto_Hijo_Abs; ...)
        return ...is abstract;

        Procedure Nombre_Proc_Padre (this : in out Objeto_Hijo_Abs;...)
        is;
        --Este es un procedimiento abstracto definido por el objeto
        padre que se define en los objetos hijos que heredan de él.
        Function Nombre_fucti_Padre (this : in out Objeto_Hijo_Abs;...)
        return ...;
        --Este es un función abstracta definido por el objeto padre que
        se define en los objetos hijos que heredan de él.
        Procedure Nombre_Proc (this : in out Objeto_Hijo_Abs;...)is;

        Function Nombre_fucti (this : in out Objeto_Hijo_Abs;...)return
        ...;

[Private]

        type Objeto_Hijo_Abs is abstract new Objeto_Padre with record
                ...
                ...
        end record;

End <Nombre_de_la_Clase>;
```

Como se puede observar en el ejemplo anterior si el objeto padre es un objeto abstracto o de tipo interface, las funciones y procedimientos y funciones de tipo abstracto del padre, están obligados de implementarlos los hijos que heredan de ellos o los hijos que heredan de estos últimos.

Un ejemplo de ello, en el *Framework* son los objetos **Minfr_State**, **Hierarchical_State** y **Region**. El objeto **Minfr_State** es un objeto abstracto que no tiene herencias y tiene definidos tres procedimientos también abstractos, el objeto **Hierarchical_State** es un objeto también abstracto que hereda del objeto padre **Minfr_State**, y este define uno de los procedimientos del padre **Minfr_State**. Por último el objeto **Region** es hijo de objeto **Hierarchical_State** e implementa los dos procedimientos del padre **Minfr_State** que quedaban por implementar. En la figura 6 se puede ver de forma esquematizada la relación de herencia de los tres objetos.

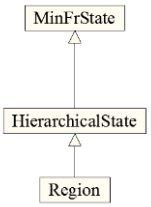

Ilustración 6: Herencia de clases abstractas

- Clases con objetos tipo abstractos que no tienen herencia.

```
Pakage <Nombre_de_la_Clase>is

       type Objeto_Abs is abstract tagged private;

       Procedure Nombre_Proc (this : in out Objeto_Abs;...)is abstract;

       Function Nombre_fucti (this : in Objeto_Abs;...)return ...is
       abstract;

       Procedure Nombre_Proc (this : in out Objeto_Abs;...)is;

       Function Nombre_fucti (this : in Objeto_Abs;...)return ...;
[Private]

       type Objeto_Abs is abstract tagged record
              ...
              ...
       end record;

End <Nombre_de_la_Clase>;
```

Estos objetos son siempre los padres de otros objetos, otros objetos heredan de ellos la estructura del objeto padre y sus procedimientos. Estos objetos nunca se puede utilizar para construir el objeto, quien construye el objeto siempre son los objetos hijos que heredan de él.

- Clases con Objetos simples.

  Estas clases son las que construyen el objeto que va a utilizar el *Framework*. A la hora de declarar estos objetos, hay que obligar que se llame a una función que haga las veces de constructor, que cree el objeto e inicialice los parámetros que componen al objeto, para que la estructura del código en *ADA* sea similar a *C++*. Esto se consigue poniendo el operador "`(<>)`" en la declaración del objeto como se indica a continuación.

  ```
  Type Objeto(<>)is tagged private;
  ```

  Con el operador "`(<>)`" obligamos que siempre que se quiera cree el objeto hay que llamar a una función que lo crea y lo define, esta funciones constructoras devuelve el puntero (dirección) de donde se creo el objeto. Con esto conseguimos que la programación en *Ada* funcione de forma similar a *C++*, la función mencionada anteriormente se comporta como la función constructora de *C++*.

```
Function Cons_Objeto (...) return Ptr_Objeto;
```

El ejemplo anterior indica como se definiría esta función constructora, como se menciono en el apartado anterior esta función devuelve el puntero (dirección) del objeto creado que es **Ptr_Objeto**.

Estas clases se pueden subdividir en dos tipos.

- Clases con Objetos Simples que no tiene herencia.

```
Pakage <Nombre_de_la_Clase>is

       type Objeto(<>) is tagged private;;

       type Ptr_Objeto is access all Objeto;

       Function Cons_Objeto (...) return Ptr_Objeto;

       Procedure Nombre_Proc (this : in out Objeto;...)is;

       Function Nombre_fucti (this : in Objeto;...)return ...;
[Private]

       type Objeto is tagged record
              ...
              ...
       end record;

End <Nombre_de_la_Clase>;
```

- Clases con Objetos Simples que tiene herencia.

```
Pakage <Nombre_de_la_Clase>is

       type Objeto(<>) is new Objeto_Padre with private;

       type Ptr_Objeto is access all Objeto;

       Function Cons_Objeto (...) return Ptr_Objeto;

       Procedure Nombre_Proc (this : in out Objeto;...)is;

       Function Nombre_fucti (this : in Objeto;...)return ...;
[Private]

       type Objeto is new Objeto_Padre with record...
              ...
       end record;

End <Nombre_de_la_Clase>;
```

3.1.2. Cuerpo de las clases.

La especificación de la mayoría de las clases necesita un cuerpo, donde se define la implementación de las funciones y procedimientos que se definieron en la especificación de la clase. Hay que destacar que las clases que tienen objetos de tipo *interface*, todos sus procedimientos y funciones, definidos en la especificación no necesitan implementación, por lo tanto este tipo de clases no necesitan un cuerpo. La estructura del cuerpo de las clases es la siguiente.

```
Pakage Body<Nombre_de_la_Clase>is

      <Desarrollo de las Funciones y Procedimientos declarados
      en la especificación de la clase>

End <Nombre_de_la_Clase>;
```

4. Descripción de clase

En los siguientes apartados se irán describiendo detalladamente cómo están compuestas las clases de *Framework.* Describiendo cada uno de los objetos y los procedimientos y funciones que los definen.

4.1. Mensajes Genéricos Clase MSG.

La clase *Msg* define un único objeto de tipo abstracta, este objeto es el encargado de construir y manejar los mensajes genéricos. Está constituido por un mapa ordenado "Ordered Maps" y el nombre del objeto en la siguiente figura se ve de forma esquemática como está constituido este objeto.

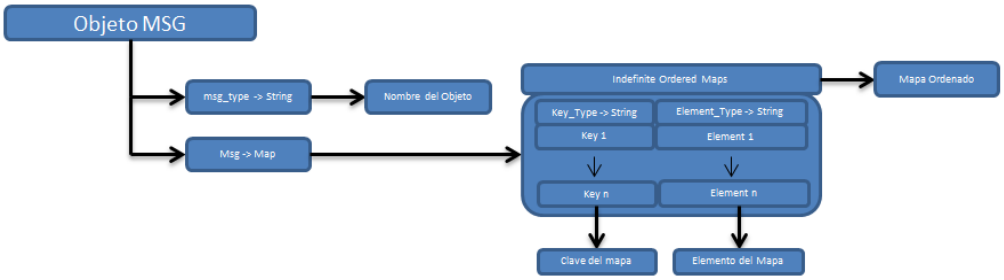

Ilustración 7: Objeto MSG

Estos son los métodos y funciones de la clase *MSG*

set_Value
UpDate_Value
get_Msg_Type
get_Value

4.1.1. Procedimiento Set_Value.

Este procedimiento inserta en el mapa ordenado un dato con una clave que lo identifica, estos dos datos son de tipo *String.*

procedure set_Value(this:in out Msg; Key,value:string);

4.1.2. Procedimiento UpDate_Value.

Este procedimiento busca la clave que identifica el valor que se quiere actualizar, si no se encuentra dicha clave no se realiza dicha actualización del dato y el procedimiento devuelve un valor falso indicando que no se ha podido actualizar el valor de dicha clave por no existir dicha clave.

procedure UpDate_Value(this:in out Msg;Key,value:String;Res :in out Boolean);

4.1.3. Función get_Msg_Type.

Esta Función devuelve el nombre del objeto

function get_Msg_Type (this:in Msg) return string;

4.1.4. Función Get_Value.

Esta Función busca la clave que se quiere obtener su valor si encuentra dicha clave devuelve el valor relacionado, si la clave no es encontrada devuelve un carácter "0"

function get_Value(this:in Msg;Key:string) return string;

4.2. Clase Minfr_Msg.

Esta clase hereda de la clase abstracta *MSG ver 4.1*, los métodos que se le agregan aumentan la funcionalidad y gestionan el origen y el destino del mensaje, además gestiona si se ha producido algún error a la hora de gestionar dicho mensaje.

Constructores.

Constr_Minfr_Msg .
Constr_Minfr_Msg (Name_Type : string).
Constr_Minfr_Msg (msg_Type : msg).

Los métodos y funciones que agrega esta clase son los siguientes.

Get_Origin_Id
Set_Origin_Id
Get_Destination_Id
Set_Destination_Id
Get_Error_Code
Set_Error_Code
Get_Name
Set_Name

4.2.1. Constructor Constr_Minfr_Msg.

Esta función crea el objeto tipo *Minfr_Msg ver 4.2*, donde se crea un mensaje inicial con la clave "MinifrMsg" y el elemento relacionado con la clave que es una cadena vacía.

function Constr_Minfr_Msg return Ptr_Minfr_Msg;

4.2.2. Constructor Constr_Minfr_Msg.

Esta función crea el objeto tipo *Minfr_Msg ver 4.2*, donde se crea un mensaje inicial con la clave "*MinfrMsg*" y el elemento relacionado con la clave que es la cadena de caracteres que se le pasa como argumento al constructor *"Name_Type".*

function Constr_Minfr_Msg (Name_Type : string) return Ptr_Minfr_Msg.

4.2.3. Constructor Constr_Minfr_Msg.

Esta función crea el objeto tipo *Minfr_Msg ver 4.2*, donde se crea un mensaje inicial con el objeto "*MSG ver 4.1*" que se le pasa como argumento al constructor *"msg_Type"*.

function Constr_Minfr_Msg (msg_Type : msg) return Ptr_Minfr_Msg.

4.2.4. Función Get_Origin_Id.

Esta función busca el mensaje en la tabla con la clave *"OPortId"*y devuelve su valor, si no existe devuelve el valor *"0"*.
Esta función indica cual es el origen del mensaje del objeto *"Minfr_Msg ver 4.2"*.

function Get_Origin_Id (this:in Minfr_Msg) return O_Port_ID;

4.2.5. Procedimiento Set_Origin_Id.

Este procedimiento inserta un mensaje en la tabla con la clave *"OPortId"*y el elemento relacionado que se le pasa como argumento al procedimiento *"Port_Id"*. Si dicha clave ya existe en la tabla de mensajes se actualizara el elemento con el argumento del procedimiento *"Port_Id"*.
Este procedimiento establece cual va a ser el origen de los mensajes del objeto *"Minfr_Msg ver 4.2"*.

procedure Set_Origin_Id (this:in out Minfr_Msg; Port_Id: O_Port_ID);

4.2.6. Función Get_Destination_Id.

Esta función busca el mensaje en la tabla con la clave *"DPortId"*y devuelve su valor, si no existe devuelve el valor *"0"*.
Esta función indica cual es el destino del mensaje del objeto *"Minfr_Msg ver 4.2"*.

procedure Set_Origin_Id (this:in out Minfr_Msg; Port_Id: O_Port_ID);

4.2.7. Procedimiento Set_Destination_Id.

Este procedimiento inserta un mensaje en la tabla con la clave *"DPortId"*y el elemento relacionado que se le pasa como argumento al procedimiento *"Port_Id"*. Si dicha clave ya existe en la tabla de mensajes se actualizara el elemento con el argumento del procedimiento *"Port_Id"*.
Este procedimiento establece cual va a ser el destino de los mensajes del objeto *"Minfr_Msg ver 4.2"*.

procedure Set_Destination_Id (this:in out Minfr_Msg; Port_Id: O_Port_Id);

4.2.8. Función Get_Error_Code.

Esta función busca el mensaje en la tabla con la clave *"ErrorCode"*y devuelve su valor, si no existe devuelve el valor *"OK"*.
Esta función indica si es ha producido algún error en el manejo del objeto *"Minfr_Msg ver 4.2"*.

function Get_Error_Code (this:in Minfr_Msg) return Port_Erro_Code;

4.2.9. Procedimiento Set__Error_Code.

Este procedimiento inserta un mensaje en la tabla con la clave *"ErrorCode"*y el elemento relacionado que se le pasa como argumento al procedimiento *"Error_Code"*. Si dicha clave ya existe en la tabla de mensajes se actualizara el elemento con el argumento del procedimiento *"Error_Code"*.
Este procedimiento establece cual es el error que se ha producido en el manejo del objeto *"Minfr_Msg ver 4.2"*.

procedure Set_Error_Code (this:in out Minfr_Msg; Error_Code: Port_Erro_Code);

4.2.10. Función Get_Name.

Esta función busca el mensaje en la tabla con la clave *"Name"*y devuelve su valor, si no existe devuelve el valor *"0"*.
Esta función indica cual es el nombre del objeto *"Minfr_Msg ver 4.2"*.

function Get_Name(this:in Minfr_Msg) return string;

4.2.11. Procedimiento Set_Name.

Este procedimiento inserta un mensaje en la tabla con la clave *"Name"*y el elemento relacionado que se le pasa como argumento al procedimiento *"name"*. Si dicha clave ya existe en la tabla de mensajes se actualizara el elemento con el argumento del procedimiento *"name"*.
Este procedimiento establece el nombre del objeto *"Minfr_Msg ver 4.2"*.

procedure Set_Name(this:in out Minfr_Msg; name:string);

4.3. Clase Minfr_Port.

La clase *Minfr_Port ver 4.3*es una clase abstracta, es la encargada de construir y gestionar los puertos de conexión, Esta clase crea un objeto que está constituido por identificador del puerto, identificador del componente que pertenece el puerto, dirección del puerto donde se conecta el puerto y nombre del puerto, en la siguiente figura se ve de forma esquemática como esta constituida esta clase.

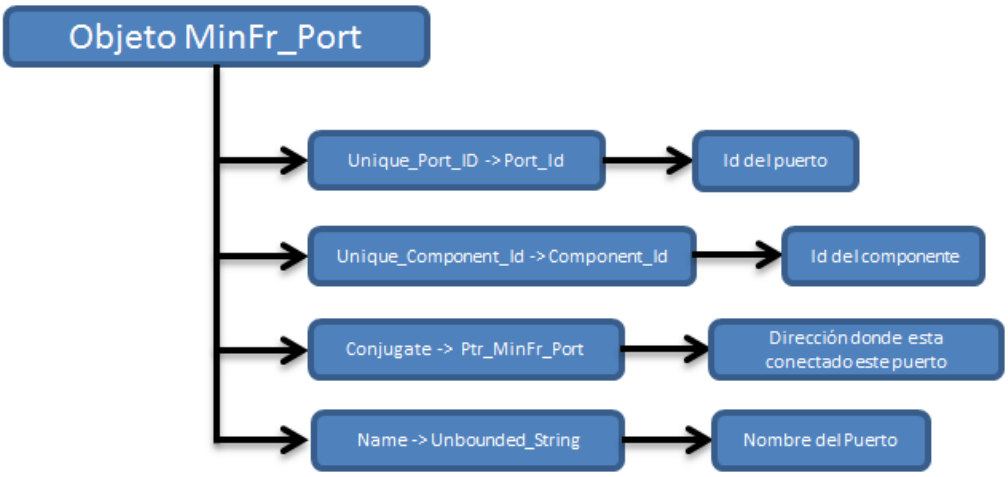

Ilustración 8: Objeto Minfr_Port

Estos son los métodos y funciones de la clase *Minfr_Port4.3*

Get_Unique_Id
UpDate
Set_Conjugate
Get_Conjugate
Is_Connected
Disconnect
Get_Name

4.3.1. Función Get_Unique_Id.

Esta función nos devuelve el valor del Id del objeto puerto *"Minfr_Port ver 4.3"*.

function Get_Unique_Id (this:in MinFr_Port) return Port_ID;

4.3.2. Procedimiento UpDate.

Este procedimiento es abstracto y se define en las clases que heredarán de esta clase, las clases que heredan de esta clase son *"MinFr_Input_Port ver 4.4"* y *"MinFr_OutPut_Port ver 4.5",* En la *Ilustración 5*Se puede ver el esquema de bloques de las clases.

procedure UpDate (this:in out MinFr_Port;D_Minfr_Msg :in Ptr_Minfr_Msg) is abstract;

4.3.3. Procedimiento Set_Conjugate.

Este procedimiento establece la conexión con la que se conectara el objeto puerto *"Minfr_Port ver 4.3"*. Si la conexión es a si mismo o a un objeto del tipo puerto *"Minfr_Port ver 4.3"* del mismo tipo, este procedimiento generara una excepción indicando cual es el error que se ha producido y terminado la ejecución del programa. Porque estos dos casos anteriormente mencionados no son posibles en la estructura del Framework.

procedure Set_Conjugate (this:in out MinFr_Port'Class;P_Port : Ptr_MinFr_Port);

4.3.4. Función Get_Conjugate.

Esta función nos indica la dirección del objeto puerto *"Minfr_Port ver 4.3"* al que esta conectado el objeto puerto *"Minfr_Port ver 4.3"*.

function Get_Conjugate (this:in MinFr_Port'Class) return Ptr_MinFr_Port;

4.3.5. Función Is_Connected.

Esta función nos indica si el objeto puerto *"Minfr_Port4.3"* esta conectado a otro objeto puerto *"Minfr_Port ver 4.3"*.

function Is_Connected (this:in MinFr_Port) return Boolean;

4.3.6. Procedimiento Disconnect.

Este procedimiento desconecta el objeto puerto*"Minfr_Port ver 4.3"* de cualquier otro objeto puerto *"Minfr_Port ver 4.3"*.

procedure Disconnect (this:in out MinFr_Port);

4.3.7. Función Get_Name.

Esta función indica cual es el nombre del objeto *"Minfr_Port ver 4.3"*.

function Get_Name (this:in MinFr_Port) return String;

4.4. Clase MinFr_Input_Port.

Esta clase hereda de la clase abstracta *"Minfr_Port ver 4.3"*. Y los procedimientos y funciones que se agregan definen el comportamiento del objeto como un puerto de entrada. Esta clase crea un objeto de tipo *"Minfr_Port ver 4.3"*y además agrega al objeto la dirección del objeto donde se almacena y se gestiona los datos que se van a recibir por el puerto de entrada, en la siguiente figura se ve de forma esquemática como esta constituida esta clase.

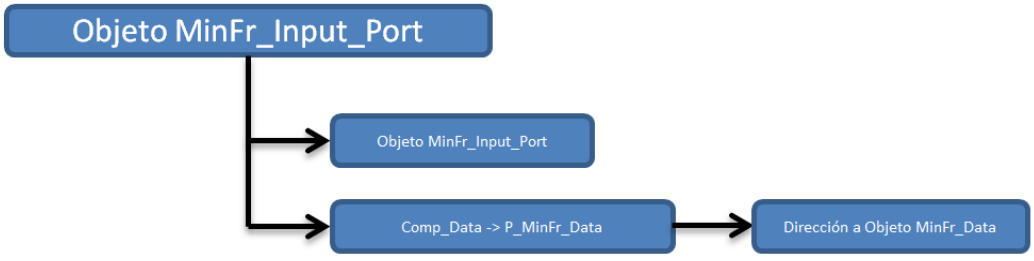

Ilustración 9: Objeto MinFr_Input_Port.

Constructores.

Constr_MinFr_Input_Port (Id :Port_ID;Unique_Component_Id :Component_Id; Name :String;Comp_Data : P_MinFr_Data) function Constr_MinFr_Input_Port

Los métodos y funciones que agrega esta clase son los siguientes.

UpDate

4.4.1. Constructor Constr_MinFr_Input_Port

Esta función crea el objeto tipo *"MinFr_Input_Port ver 4.4"*, Donde se le asigna el identificador del puerto *"Unique_Port_ID"* que es la suma de los parámetros que se le pasan por referencia a esta función *"Id"* más *"Unique_Component_Id"* multiplicado por 1000, se le asigna una dirección del puerto que esta conectado una dirección nula *"null"*, esto indica que este puerto no esta conectado a ningún puerto. Se le indica la dirección del objeto que almacena los datos que va a recibir el puerto de entrada que es el parámetro que se le pasa por referencia de esta función *"Comp_Data"* y por ultimo se indica el nombre del puerto mediante el parámetro de referencia *"Name"*.

```
function Constr_MinFr_Input_Port (Id :Port_ID; Unique_Component_Id :Component_Id; Name:String; Comp_Data : P_MinFr_Data) return Ptr_MinFr_Input_Port;
```

4.4.2. Constructor Constr_MinFr_Input_Port.

Esta función crea el objeto tipo *"MinFr_Input_Port ver 4.4"*. No realiza ninguna operación más.

```
function Constr_MinFr_Input_Port return Ptr_MinFr_Input_Port;
```

4.4.3. Procedimiento UpDate.

Este procedimiento inserta un mensaje con el identificador del objeto puerto*"MinFr_Input_Port ver 4.4"* en el objeto "*Minfr_Msg ver 4.2"*, que se le pasa por referencia a este procedimiento "*D_Minfr_Msg*", e inserta en el objeto "*MinFr_Data ver 4.7*" con el procedimiento "*Push_In_Port_Msg ver 4.6.6*" el nombre del puerto y el objeto mensaje "*Minfr_Msg ver 4.2"* Para que sea enviados.

```
procedure UpDate (this:in out  MinFr_OutPut_Port;D_Minfr_Msg :in Ptr_Minfr_Msg);
```

4.5. Clase MinFr_OutPut_Port.

Esta clase hereda de la clase abstracta *"Minfr_Port ver 4.3"*. Y los procedimientos y funciones que se agregan definen el comportamiento del objeto como un puerto de salida.

Constructores.

Constr_MinFr_OutPut_Port (Id :Port_ID;Unique_Component_Id :Component_Id; Name :String)
Constr_MinFr_OutPut_Port

Los métodos y funciones que agrega esta clase son los siguientes.

UpDate

4.5.1. Constructor Constr_MinFr_OutPut_Port

Esta función crea el objeto tipo *"MinFr_OutPut_Port ver 4.5"*, Donde se le asigna el identificador del puerto *"Unique_Port_ID"* que es la suma de los parámetros que se le pasan por referencia a esta función *"Id"* más *"Unique_Component_Id"* multiplicado por 1000, se le asigna una dirección del puerto que esta conectado una dirección nula *"null"*, esto indica que este puerto no esta conectado a ningún puerto, y por ultimo se indica el nombre del puerto mediante el parámetro de referencia *"Name"*.

```
function Constr_MinFr_OutPut_Port (Id :Port_ID;Unique_Component_Id :Component_Id; Name :String) return Ptr_MinFr_OutPut_Port;
```

4.5.2. Constructor Constr_MinFr_OutPut_Port

Esta función crea el objeto tipo *"MinFr_OutPut_Port ver 4.5"*. No realiza ninguna operación más.

```
function Constr_MinFr_OutPut_Port return Ptr_MinFr_OutPut_Port;
```

4.5.3. Procedimiento UpDate.

Este procedimiento inserta un mensaje con el identificador del objeto puerto*"MinFr_OutPut_Port ver 4.5"* en el objeto "*Minfr_Msg ver 4.2"*, que se le pasa por referencia a este procedimiento "*D_Minfr_Msg*". El siguiente paso se determina si el puerto esta conectado con un puerto de entrada "*MinFr_Input_Port ver 4.4",* si es así ejecuta el procedimiento *"UpDate4.4.3"*del puerto de entrada que esta conectado con este puerto pasándole el objeto "*Minfr_Msg ver 4.2"* que contiene el mensaje que se va a enviar.

```
procedure UpDate (this:in out  MinFr_OutPut_Port;D_Minfr_Msg :in Ptr_Minfr_Msg);
```

4.6. Clase MinFr_Component_Data.

Esta clase es de tipo interface. Es donde se declaran todos los métodos y funciones que van a ser visibles y pueden ser utilizados por otras clase que utilice esta clase. Estos métodos y funciones son los que define y gestionan el

comportamiento de la base de datos de todos los mensajes que se envía por el *Framework*.

Estos son los métodos y funciones de la clase "*MinFr_Component_Data ver 4.6*".

Pop_OutPort_Msg
Push_OutPort_Msg
Has_OutPort_Msg
Create_OutPort_Buffer
Subscribe_To_InPort_Msg
Push_In_Port_Msg
Pop_InPort_Msg
Has_InPort_Msg
Pop_UntilLast_InPort_Msg
Subscribe_To_Event_Msg
Push_Event_Msg
Pop_Event_Msg
Has_Event_Msg
Pop_UntilLast_Event_Msg
Get_Complete_Name
Get_Partial_Name

4.6.1. Función Pop_OutPort_Msg.

Esta función es abstracta y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*MinFr_Data ver 4.7*". En la *Ilustración 5*se puede ver el esquema de bloques de las clases y la dependencia entre clase.

```
function Pop_OutPort_Msg (This : MinFr_Component_Data;Name: String) return Minfr_Msg is abstract;
```

4.6.2. Procedimiento Push_OutPort_Msg.

Este procedimiento es abstracto y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*MinFr_Data ver 4.7*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

```
procedure Push_OutPort_Msg (This : MinFr_Component_Data;Name: String;Msg :Ptr_Minfr_Msg) is abstract;
```

4.6.3. Función Has_OutPort_Msg.

Esta función es abstracta y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*MinFr_Data ver 4.7*". En la

Ilustración 5 se puede ver el esquema de bloques de las clases y la dependencia entre clase.

```
function Has_OutPort_Msg (This : in MinFr_Component_Data;Name : String) return Boolean is abstract;
```

4.6.4. Procedimiento Create_OutPort_Buffer.

Este procedimiento es abstracto y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*MinFr_Data ver 4.7*". En la *Ilustración 5*se puede ver el esquema de bloques de las clases y la dependencia entre clase.

```
procedure Create_OutPort_Buffer (This : in out MinFr_Component_Data;Name : String;Bool : in out Boolean) is abstract;
```

4.6.5. Procedimiento Subscribe_To_InPort_Msg.

Este procedimiento es abstracto y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*MinFr_Data ver 4.7*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

```
procedure Subscribe_To_InPort_Msg (This : in out MinFr_Component_Data;Name : String;Id : in out LastTable_Id) is abstract;
```

4.6.6. Procedimiento Push_In_Port_Msg.

Este procedimiento es abstracto y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*MinFr_Data ver 4.7*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

```
procedure Push_In_Port_Msg (This : MinFr_Component_Data;Name: String;Msg :Ptr_Minfr_Msg) is abstract;
```

4.6.7. Función Pop_InPort_Msg.

Esta función es abstracta y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*MinFr_Data ver 4.7*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

```
function Pop_InPort_Msg (This : MinFr_Component_Data;Name: String;Subs_Id : LastTable_Id) return Minfr_Msg is abstract;
```

4.6.8. Función Has_InPort_Msg.

Esta función es abstracta y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*MinFr_Data ver 4.7*". En la

Ilustración 5 se puede ver el esquema de bloques de las clases y la dependencia entre clase.

function Has_InPort_Msg (This : in MinFr_Component_Data;Name : String;Subs_Id : LastTable_Id) return Boolean is abstract;

4.6.9. Función Pop_UntilLast_InPort_Msg.

Esta función es abstracta y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*MinFr_Data ver 4.7*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

function Pop_UntilLast_InPort_Msg (This : MinFr_Component_Data;Name: String;Subs_Id : LastTable_Id) return Minfr_Msg is abstract;

4.6.10. Procedimiento Subscribe_To_Event_Msg.

Este procedimiento es abstracto y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*MinFr_Data ver 4.7*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

procedure Subscribe_To_Event_Msg (this : in out MinFr_Component_Data; Name : String; Id : in out LastTable_Id) is abstract;

4.6.11. Procedimiento Push_Event_Msg.

Este procedimiento es abstracto y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*MinFr_Data ver 4.7*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

procedure Push_Event_Msg (This : MinFr_Component_Data;Name: String;Msg :Ptr_Minfr_Msg) is abstract;

4.6.12. Función Pop_Event_Msg.

Esta función es abstracta y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*MinFr_Data ver 4.7*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

function Pop_Event_Msg (This : MinFr_Component_Data;Name: String;Subs_Id : LastTable_Id) return Minfr_Msg is abstract;

4.6.13. Función Has_Event_Msg.

Esta función es abstracta y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*MinFr_Data ver 4.7*". En la

Ilustración 5 se puede ver el esquema de bloques de las clases y la dependencia entre clase.

function Has_Event_Msg (This : in MinFr_Component_Data;Name : String;Subs_Id : LastTable_Id) return Boolean is abstract;

4.6.14. Función Pop_UntilLast_Event_Msg.

Esta función es abstracta y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*MinFr_Data ver 4.7*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

function Pop_UntilLast_Event_Msg (This : MinFr_Component_Data;Name: String;Subs_Id : LastTable_Id) return Minfr_Msg is abstract;

4.6.15. Función Get_Complete_Name.

Esta función es abstracta y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*MinFr_Data ver 4.7*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

function Get_Complete_Name (this : in MinFr_Component_Data;Name : in string; Type_Msg : Msg_Type;Subs_Id : LastTable_Id) return String is abstract;

4.6.16. Función Get_Partial_Name.

Esta función es abstracta y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*MinFr_Data ver 4.7*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

function Get_Partial_Name (this : in MinFr_Component_Data;Name : in string; Type_Msg : Msg_Type) return string is abstract;

4.7. Clase MinFr_Data.

Esta clase hereda de la clase interface *"MinFr_Component_Data ver 4.6"*. Esta clase crea, define y gestiona las bases de datos que almacenan todos los mensajes que se envían por el *Framework*. En la siguiente figura se ve de forma esquemática como esta constituida esta clase.

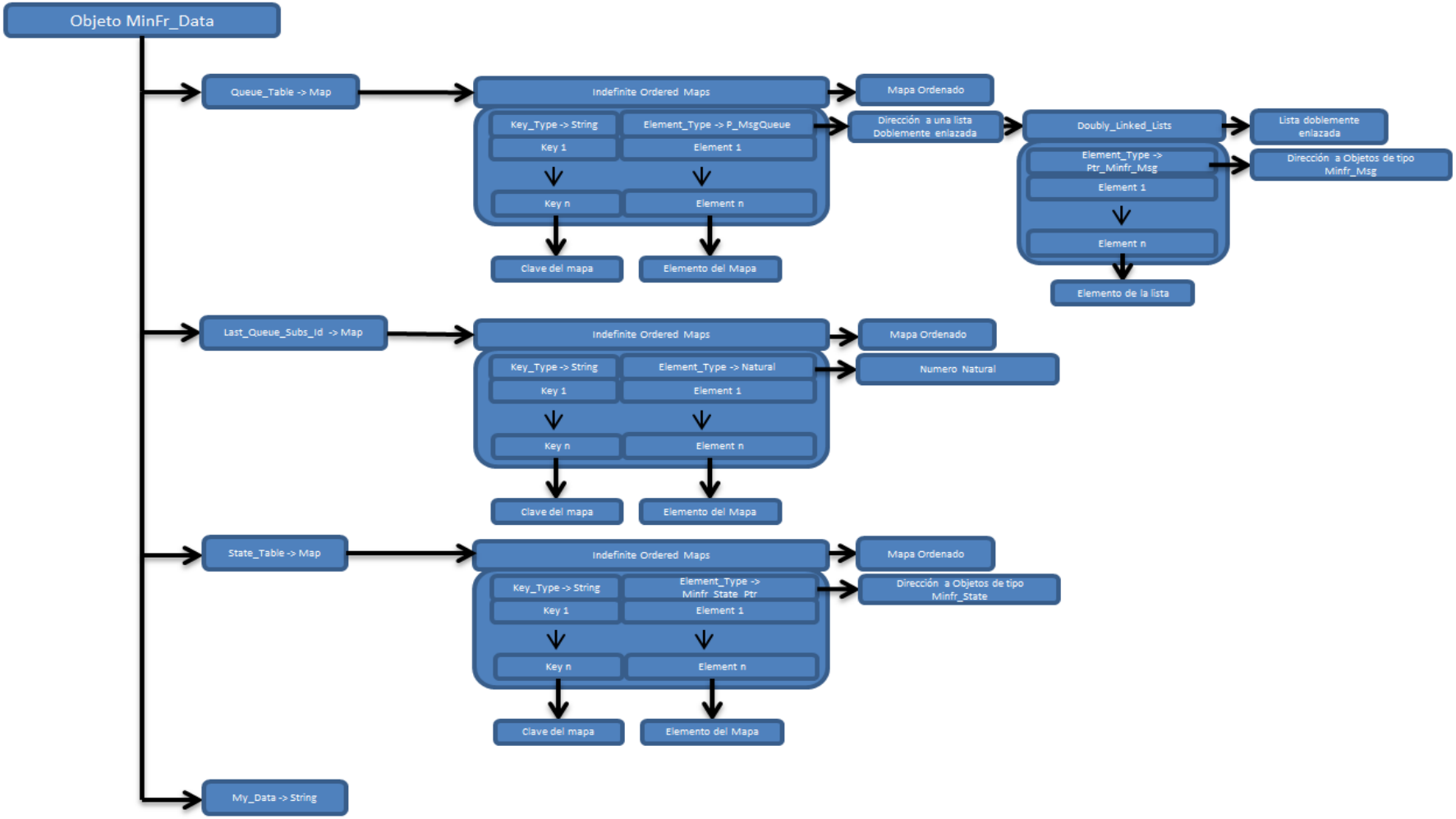

Ilustración 10: Objeto MinFr_Data.

Constructores.

Constr_MinFr_Data

Los métodos y funciones que agrega esta clase que son públicos son los siguientes.

Get_Complete_Name
Get_Partial_Name
Register_State get_State
Pop_OutPort_Msg
Push_OutPort_Msg
Has_OutPort_Msg
Create_OutPort_Buffer
Subscribe_To_InPort_Msg
Push_In_Port_Msg
*Pop_InPort_*
Has_InPort_Msg
Pop_UntilLast_InPort_Msg
Subscribe_To_Event_Msg
Push_Event_Msg
Pop_Event_Msg
Has_Event_Msg
Pop_UntilLast_Event_Msg

Los métodos y funciones que agrega esta clase que son privados son los siguientes.

Subscribe_To_Minfr_Msg
Push_Minfr_Msg
Pop_Minfr_Msg
Has_Minfr_Msg
Pop_Until_Last

4.7.1. Constructor Constr_MinFr_Data.

Esta función crea el objeto tipo *"MinFr_Data ver 4.7"*. No realiza ninguna operación más.

```
function Constr_MinFr_Data return P_MinFr_Data;
```

4.7.2. Función Get_Complete_Name.

Esta función devuelve una cadena de texto tipo "*string*" con los parámetro que se le pasan por referencia a esta función con los nombres "*Name*", "*Type_Msg*" y "*Subs_Id*". La Cadena que forma es "*Name + * + Type_Msg + Subs_Id"*.

```
function Get_Complete_Name (this : in MinFr_Data;Name : in string; Type_Msg : Msg_Type;Subs_Id : LastTable_Id) return string;
```

4.7.3. Función Get_Partial_Name.

Esta función devuelve una cadena de texto tipo "*string*" con los parámetro que se le pasan por referencia a esta función con los nombres "*Name*", "*Type_Msg*". La Cadena que forma es "*Name + * + Type_Msg".*

function Get_Partial_Name (this : in MinFr_Data;Name : in string; Type_Msg : Msg_Type) return string;

4.7.4. Procedimiento Register_State.

Este procedimiento inserta en le tabla de estados "*State_Table*" el Id del estado y la dirección del objeto "*Minfr_State ver 4.10*", estos dos parámetros se consiguen del parámetro "*State*" que se le pasa a esta función por referencia.

procedure Register_State (this : in out MinFr_Data; State : Minfr_State_Ptr);

4.7.5. Función get_State

Esta función busca en la tabla de estados "*State_Table*", si hay algún registro que coincida con el Id "*St_Id*" que se le pasa por referencia a esta función. Si existes dicho registro devuelve la dirección del objeto "*Minfr_State ver 4.10*" que esta almacenado en la tabla "*State_Table*".

function get_State (this : in MinFr_Data; St_Id : Type_MD_State_Id) return Minfr_State_Ptr;

4.7.6. Función Pop_OutPort_Msg

Esta función busca en la tabla "*Queue_Table*", si algún registro coincide con la cadena que devuelve la función "*Get_Partial_Name ver 4.7.3*", a esta función se le pasa los datos "*Name*" y "*OutPort*" donde "*Name*" se pasa por referencia en esta función y "*OutPort*" indica el tipo de mensaje y es un tipo de dato predefinido. Si se encuentra dicha cadena extrae la dirección de la lista doblemente enlazada "MsgQueue" relacionada a la cadena, se averigua si esta lista tiene algún elemento que contiene la dirección del objeto mensaje "*Minfr_Msg ver 4.2*", esta función devolvería el objeto mensaje "*Minfr_Msg ver 4.2*" pero en el caso de que la lista no contuviera ningún elemento devolvería el objeto "*Minfr_Msg ver 4.2*" indicando en este que se abría producido un error.

function Pop_OutPort_Msg (This :MinFr_Data;Name: String) return Minfr_Msg;

4.7.7. ProcedimientoPush_OutPort_Msg.

Este procedimiento busca en la tabla "*Queue_Table*", si algún registro coincide con la cadena que devuelve la función "*Get_Partial_Name ver 4.7.3*", a esta función se le pasa los datos "*Name*" y "*OutPort*" donde "*Name*" se pasa por referencia en este procedimiento y "*OutPort*" indica el tipo de mensaje y es un tipo de dato predefinido. Si se encuentra dicha cadena extrae la dirección de la lista doblemente enlazada "MsgQueue"

relacionada a la cadena e insertando la dirección del objeto "*Minfr_Msg ver 4.2*" en la lista doblemente enlazada "MsgQueue". La dirección del Objeto "*Minfr_Msg ver 4.2*" se pasa por referencia a este procedimiento con el nombre "*Msg*"

```
procedure Push_OutPort_Msg (This : MinFr_Data;Name: String;Msg :Ptr_Minfr_Msg);
```

4.7.8. Función Has_OutPort_Msg

Esta función busca en la tabla "*Queue_Table*", si algún registro coincide con la cadena que devuelve la función "*Get_Partial_Name ver 4.7.3*", a esta función se le pasa los datos "*Name*" y "*OutPort*" donde "*Name*" se pasa por referencia en esta función y "*OutPort*" indica el tipo de mensaje y es un tipo de dato predefinido. Si se encuentra dicha cadena extrae la dirección de la lista doblemente enlazada "MsgQueue" relacionada a la cadena y averigua si dicha lista esta vacía o llena. Si no encuentra la lista o esta lista esta vacía la función devuelve el valor boolean "False", en el caso de que exista la lista y contenga algún elemento devuelve el valor boolean "True".

```
function Has_OutPort_Msg (This : in MinFr_Data;Name : String) return Boolean;
```

4.7.9. Procedimiento Create_OutPort_Buffer

Este procedimiento se crea una lista doblemente enlazada "MsgQueue" e inserta la dirección de esta lista en la tabla "*Queue_Table*" con la referencia de la cadena de caracteres que devuelve la función "*Get_Partial_Name ver 4.7.3*", a esta función se le pasa los datos "*Name*" y "*OutPort*" donde "*Name*" se pasa por referencia en este procedimiento y "*OutPort*" indica el tipo de mensaje y es un tipo de dato predefinido.

```
procedure Create_OutPort_Buffer (This : in out MinFr_Data;Name : String;Bool : in out
Boolean);
```

4.7.10. Procedimiento Subscribe_To_InPort_Msg.

Este procedimiento ejecuta el procedimiento privado de esta clase "*Subscribe_To_Minfr_Msg ver 4.7.20*" pasándole por referencia los datos "*Name*" "*InPort*" y "*Id*", Donde "*Name*" y "Id" son pasados por referencia por la propio procedimiento "*Subscribe_To_InPort_Msg ver 4.7.10*" y "*InPort*" es un valor predefinido que indica que los mensajes son de tipo entrada.

```
procedure Subscribe_To_InPort_Msg (This : in out MinFr_Data;Name: String;Id : in out
LastTable_Id);
```

4.7.11. Procedimiento Push_In_Port_Msg.

Este procedimiento ejecuta el procedimiento privado de esta clase "*Push_Minfr_Msg ver 4.7.21*" pasándole por referencia los datos "*Name*", "*Msg*" y "*InPort*", Donde "*Name*" y "Msg" son pasados por referencia por

la propio procedimiento "*Push_In_Port_Msg ver 4.7.11*" y "*InPort*" es un valor predefinido que indica que los mensajes son de tipo entrada.

procedure Push_In_Port_Msg (This : MinFr_Data;Name: String;Msg :Ptr_Minfr_Msg);

4.7.12. Función Pop_InPort_Msg.

Esta función devuelve un objeto de tipo "*Minfr_Msg ver 4.2*" que es el que devuelve tras ejecutar la función privado de esta clase "*Pop_Minfr_Msg ver 4.7.22*" pasándole por referencia los datos "*Name*", "*Subs_Id*" y "*InPort*", donde "*Name*" y "*Subs_Id*" son pasados por referencia por la propia función "*Pop_InPort_Msg ver 4.7.12*" y "*InPort*" es un valor predefinido que indica que los mensajes son de tipo entrada.

function Pop_InPort_Msg (This : MinFr_Data;Name: String;Subs_Id : LastTable_Id) return Minfr_Msg;

4.7.13. Función Has_InPort_Msg.

Esta función devuelve un valor "*Boolean*" (Verdadero o Falso) que es el que devuelve tras ejecutar la función privado de esta clase "*Has_Minfr_Msg ver 4.7.23*" pasándole por referencia los datos "*Name*", "*Subs_Id*" y "*InPort*", donde "*Name*" y "*Subs_Id*" son pasados por referencia por la propia función "*Has_InPort_Msg ver 4.7.13*" y "*InPort*" es un valor predefinido que indica que los mensajes son de tipo entrada.

function Has_InPort_Msg (This : in MinFr_Data;Name : String;Subs_Id : LastTable_Id) return Boolean;

4.7.14. Función Pop_UntilLast_InPort_Msg.

Esta función devuelve un objeto de tipo "*Minfr_Msg ver 4.2*" que es el que devuelve tras ejecutar la función privado de esta clase "*Pop_Until_Last ver 4.7.24*" pasándole por referencia los datos "*Name*", "*Subs_Id*" y "*InPort*", donde "*Name*" y "*Subs_Id*" son pasados por referencia por la propia función "*Pop_UntiLast_Port_Msg ver 4.7.14*" y "*InPort*" es un valor predefinido que indica que los mensajes son de tipo entrada.

function Pop_UntilLast_InPort_Msg (This :MinFr_Data;Name: String;Subs_Id : LastTable_Id) return Minfr_Msg;

4.7.15. Procedimiento Subscribe_To_Event_Msg.

Este procedimiento ejecuta el procedimiento privado de esta clase "*Subscribe_To_Minfr_Msg ver 4.7.20*" pasándole por referencia los datos "*Name*" "*Evento*" y "*Id*", Donde "*Name*" y "Id" son pasados por referencia por la propio procedimiento "*Subscribe_To_Event_Msg ver 4.7.15*" y "*Event*" es un valor predefinido que indica que los mensajes son de tipo evento.

procedure Subscribe_To_Event_Msg (this : in out MinFr_Data; Name : String; Id : in out LastTable_Id);

4.7.16. Procedimiento Push_Event_Msg.

Este procedimiento ejecuta el procedimiento privado de esta clase "*Push_Minfr_Msg ver 4.7.21*" pasándole por referencia los datos "*Name*", "*Msg*" y "*Event*", Donde "*Name*" y "*Msg*" son pasados por referencia por la propio procedimiento "*Push_Event_Msg ver 4.7.16*" y "*Event*" es un valor predefinido que indica que los mensajes son de tipo evento.

```
procedure Push_Event_Msg (This : MinFr_Data;Name: String;Msg :Ptr_Minfr_Msg);
```

4.7.17. Función Pop_Event_Msg.

Esta función devuelve un objeto de tipo "*Minfr_Msg ver 4.2*" que es el que devuelve tras ejecutar la función privado de esta clase "*Pop_Minfr_Msg ver 4.7.22*" pasándole por referencia los datos "*Name*", "*Subs_Id*" y "*Event*", donde "*Name*" y "*Subs_Id*" son pasados por referencia por la propia función "*Pop_Event_Msg ver 4.7.17*" y "*Event*" es un valor predefinido que indica que los mensajes son de tipo evento.

```
function Pop_Event_Msg (This : MinFr_Data;Name: String;Subs_Id : LastTable_Id)
return Minfr_Msg;
```

4.7.18. Función Has_Event_Msg.

Esta función devuelve un valor "*Boolean*" (Verdadero o Falso) que es el que devuelve tras ejecutar la función privado de esta clase "*Has_Minfr_Msg4.7.23*" pasándole por referencia los datos "*Name*", "*Subs_Id*" y "*Event*", donde "*Name*" y "*Subs_Id*" son pasados por referencia por la propia función "*Pop_Event_Msg ver 4.7.18*" y "*Event*" es un valor predefinido que indica que los mensajes son de tipo evento.

```
function Has_Event_Msg (This : in MinFr_Data;Name : String;Subs_Id : LastTable_Id)
return Boolean;
```

4.7.19. Función Pop_UntilLast_Event_Msg.

Esta función devuelve un objeto de tipo "*Minfr_Msg ver 4.2*" que es el que devuelve tras ejecutar la función privado de esta clase "*Pop_Until_Last ver 4.7.24*" pasándole por referencia los datos "*Name*", "*Subs_Id*" y "*Event*", donde "*Name*" y "*Subs_Id*" son pasados por referencia por la propia función "*Pop_UntiLast_Event_Msg ver 4.7.19*" y "*Event*" es un valor predefinido que indica que los mensajes son de tipo evento.

```
function Pop_UntilLast_Event_Msg (This : MinFr_Data;Name: String;Subs_Id :
LastTable_Id) return Minfr_Msg;
```

4.7.20. Procedimiento Subscribe_To_Minfr_Msg.

Este procedimiento inserta en la tabla "Last_Queue_Subs_Id" el valor "*LastTable_Id*" que es el numero de la tabla que se a crear del tipo "*Type_Msg*", este numero se va incrementando en uno cada vez que se suscribe este tipo de mensaje "*Type_Msg*", con la referencia de la cadena de caracteres que devuelve la función "*Get_Partial_Name ver 4.7.3*", a esta función se le pasa los datos "*Name*" y "*Type_Msg*" donde se pasan por referencia en este procedimiento.
Crea una lista doblemente enlazada "MsgQueue" e inserta la dirección de esta lista en la tabla "*Queue_Table*" con la referencia de la cadena de caracteres que devuelve la función "*Get_Complete_Name ver 4.7.2*", a esta función se le pasa los datos "*Name*", "*Type_Msg*" y "*LastTable_Id*", donde "*Name*" y "*Type_Msg*" se pasan por referencia en este procedimiento, "*LastTable_Id*" es el numero de la tabla que se a creado del tipo "*Type_Msg*".
El dato que se le pasa por referencia "*Id*" contiene el valor de "*LastTable_Id*" que como se menciono en el párrafo anterior es el numero de la tabla que se a crear del tipo "*Type_Msg*".

```
procedure Subscribe_To_Minfr_Msg (This : in out MinFr_Data;Name : String; Type_Msg
:Msg_Type;Id : in out LastTable_Id);
```

4.7.21. Procedimiento Push_Minfr_Msg.

Este procedimiento busca en la tabla "Last_Queue_Subs_Id" un elemento que coincida con la cadena de caracteres que devuelve la función "*Get_Partial_Name ver 4.7.3*", a esta función se le pasa los datos "*Name*" y "*Type_Msg*" donde se pasan por referencia en este procedimiento. Si dicha tabla contiene dicha coincidencia se obtiene una copia del dado relacionado a esta cadena, este dado indica el numero de listas doblemente enlazas que existen en la tabla "*Queue_Table*" del tipo "*Type_Msg*".
En el siguiente etapa se realiza un barrido de todas las listas doblemente enlazadas en la tabla "*Queue_Table*" buscando la cadena de caracteres que coincida con la que devuelve la función "*Get_Complete_Name ver 4.7.2*", a esta función se le pasa los datos "*Name*" , "*Type_Msg*" y "*LastTable_Id*", donde "*Name*" y "*Type_Msg*" se pasan por referencia en este procedimiento, "*LastTable_Id*" es el numero de listas doblemente enlazadas que se esta haciendo el barrido, cuando aparece una coincidencia se le inserte a dicha lista la dirección del objeto "*Minfr_Msg ver 4.2*" que se le pasa por referencia a este procedimiento con el nombre "*Msg*".

```
procedure Push_Minfr_Msg (This : MinFr_Data; Name :String; Msg : Ptr_Minfr_Msg;
Type_Msg :Msg_Type);
```

4.7.22. Función Pop_Minfr_Msg

Esta función busca en la tabla "*Queue_Table*" un elemento que coincida con la cadena de caracteres que devuelve la función "*Get_Complete_Name ver 4.7.2*", a esta función se le pasa los datos "*Name*", "*Type_Msg*" y "*LastTable_Id*", donde estos datos se pasan por referencia en esta función

"*Pop_Minfr_Msg ver 4.7.22*". Si aparece una coincidencia con la cadena de caracteres se extrae la dirección de la lista doblemente enlazada "MsgQueue" relacionada a la cadena, se averigua si esta lista tiene algún elemento. Si contiene algún o varios elementos, se extrae el último elemento introducido a la lista, devolviéndolo en la función el contenido del elemento que contiene la dirección del objeto mensaje "*Minfr_Msg ver 4.2*". Pero en el caso de que esta lista no contuviera ningún elemento esta función devolvería el objeto mensaje "*Minfr_Msg ver 4.2*" indicando en el que se había producido un error.

function Pop_Minfr_Msg (This : MinFr_Data;Name :String;SubsId : LastTable_Id;Type_Msg :Msg_Type) return Minfr_Msg;

4.7.23. Función Has_Minfr_Msg.

Esta función busca en la tabla "*Queue_Table*", si algún registro coincide con la cadena que devuelve la función "*Get_Complete_Name*", a esta función se le pasa los datos "*Name*", "*Type_Msg*" y "*LastTable_Id*", donde estos datos se pasan por referencia en esta función "*Has_Minfr_Msg ver 4.7.23*". Si se encuentra dicha cadena extrae la dirección de la lista doblemente enlazada "MsgQueue" relacionada a la cadena y averigua si dicha lista esta vacía o llena. Si no encuentra la lista o esta lista esta vacía la función devuelve el valor boolean "False", en el caso de que exista la lista y contenga algún elemento devuelve el valor boolean "True".

function Has_Minfr_Msg(This : MinFr_Data; Name :String; SubsId : LastTable_Id; Type_Msg :Msg_Type) return Boolean;

4.7.24. *Función Pop_Until_Last.*

Esta función saca todos los objetos mensaje "*Minfr_Msg ver 4.2*" almacenados en la lista doblemente enlazada del nombre, sud índice y tipo que se le pasa por referencia desde esta función "*Pop_Until_Last ver 4.7.24*". Devuelve el objeto mensaje "*Minfr_Msg ver 4.2*" que contiene el primer mensaje que se almaceno en la lista.

4.8. Clase MinFr_Component.

Esta clase es de tipo interface. Es donde se declaran todos los métodos y funciones que van a ser visibles y pueden ser utilizados por otras clase que utilice esta clase. Estos métodos y funciones son los que define y gestionan el comportamiento de los componentes Framework. Definiendo número de regiones y puertos.

Estos son los métodos y funciones de la clase "*MinFr_Component ver 4.8*".

Set_ME
Get_Activities
Get_Component_Name
Get_Ports
Get_Port_In

Get_Port_Out
Add_Port
Add_Region
Get_Region
Get_Unique_Id
Get_Component_Data

4.8.1. Procedimiento Set_ME.

Este procedimiento es abstracto y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*Component4.9*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

4.8.2. Función Get_Activities

Esta función es abstracta y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*Component ver 4.9*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

4.8.3. Función Get_Component_Name.

Esta función es abstracta y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*Component ver 4.9*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

4.8.4. Función Get_Ports.

Esta función es abstracta y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*Component ver 4.9*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

4.8.5. Función Get_Port_In.

Esta función es abstracta y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*Component ver 4.9*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

4.8.6. Función Get_Port_Out.

Esta función es abstracta y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*Component ver 4.9*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

4.8.7. Procedimiento Add_Port.

Este procedimiento es abstracto y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*Component ver 4.9*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

4.8.8. Procedimiento Add_Region.

Este procedimiento es abstracto y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*Component ver 4.9*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

4.8.9. Función Get_Region.

Esta función es abstracta y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*Component ver 4.9*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

4.8.10. Función Get_Unique_Id.

Esta función es abstracta y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*Component ver 4.9*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

4.8.11. Función Get_Component_Data.

Esta función es abstracta y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*Component ver 4.9*". En la *Ilustración 5*Se puede ver el esquema de bloques de las clases y la dependencia entre clase.

4.9. Clase Component.

Esta clase hereda de la clase "*MinFr_Component ver 4.8*", al ser esta última una clase tipo interface la clase "*Component ver 4.9*" define todos los procedimientos y funciones definidos en la clase "*MinFr_Component ver 4.8*".
Esta clase define y gestionan el comportamiento de los componentes del *Framework*. Definiendo el número de regiones y puertos.En la siguiente figura se ve de forma esquemática como esta constituida esta clase.

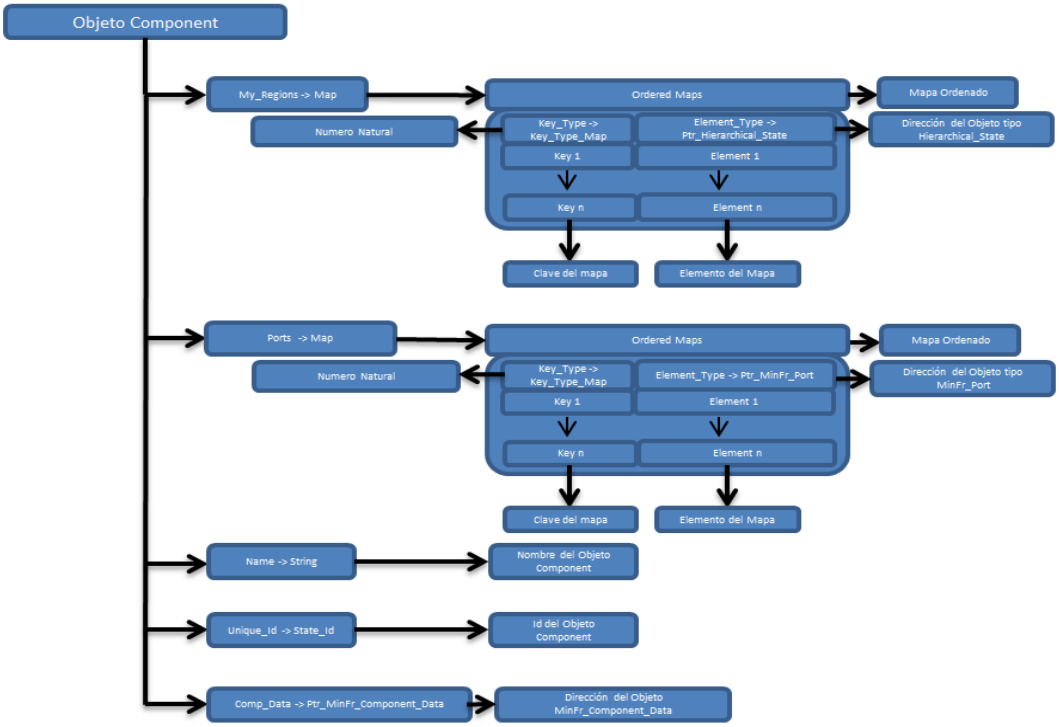

Ilustración 11: Objeto Component.

Constructores.

Constr_Component (Id : State_Id; name : string; compData : Ptr_MinFr_Component_Data).

Los métodos y funciones que agrega esta clase son los siguientes.

Set_ME
Get_Activities
Get_Component_Name
Get_Ports
Get_Port_In
Get_Port_Out
Add_Port
Add_Region
Get_Region
Get_Unique_Id
Get_Component_Data

4.9.1. Constructor Constr_Component.

Esta función crea el objeto tipo "*Component ver 4.9*", donde se define el "*Unique_Id*", "*Name*" y "*Comp_Data*" que son Id, nombre y la dirección de la base de datos "*MinFr_Component_Data ver 4.6*". Estos datos se pasan por referencia a esta función mediante los parámetros "*Id*", "*name*" y "*compData*" respectivamente.

function Constr_Component (Id : State_Id; name : string; compData : Ptr_MinFr_Component_Data) return Ptr_Component;

4.9.2. ProcedimientoSet_ME.

Este procedimiento no tiene ninguna tarea asignada.

procedure Set_ME (This : Component; me : Minfr_State_Ptr);

4.9.3. FunciónGet_Activities.

Esta función devuelve una lista doblemente enlazada con todas las direcciones del objeto del tipo "*State_Activity ver 4.15*", este objeto son las actividades que están ejecutando en el componente.

function Get_Activities (This : Component) return Pack_Vector.List;

4.9.4. FunciónGet_Component_Name.

Esta función devuelve una cadena de caracteres de tipo "*String*" que es el nombre "*Name*" que se le ha asignado al objeto cuando ha sido creado.

function Get_Component_Name (This : Component) return string;

4.9.5. FunciónGet_Ports.

Esta función retorna "*Ports*" que es la tabla ordenada que contiene la lista los puertos que contiene este objeto "*Component ver 4.9*".

function Get_Ports (This : Component) return Pack_Orde_Maps_Port.Map;

4.9.6. FunciónGet_Port_In.

Esta función busca en la tabla ordenada "*Ports*" si hay algún puerto de tipo entrada, el primero objeto "*MinFr_Input_Port ver 4.4*" que encuentre devuelve su dirección.

function Get_Port_In (This : Component) return Ptr_MinFr_Input_Port;

4.9.7. FunciónGet_Port_Out.

Esta función busca en la tabla ordenada "*Ports*" si hay algún puerto de tipo Salida, el primero objeto "*MinFr_OutPut_Port ver 4.5*" que encuentre devuelve su dirección.

function Get_Port_Out (This : Component) return Ptr_MinFr_OutPut_Port;

4.9.8. ProcedimientoAdd_Port.

Este procedimiento agrega al mapa ordenado "*Ports*" la dirección del objeto tipo puerto "*Minfr_Port ver 4.3*", que se le pasa por referencia a esta función con el nombre "*Port*".

```
procedure Add_Port (This : in out Component; Port : Ptr_MinFr_Port);
```

4.9.9. ProcedimientoAdd_Region.

Este procedimiento agrega al mapa ordenado "*My_Regions*" la dirección del objeto tipo región "*Hierarchical_State ver 4.11*", que se le pasa por referencia a esta función con el nombre "*Region*".

```
procedure Add_Region (This : in out Component; Region :  Ptr_Hierarchical_State);
```

4.9.10. FunciónGet_Region

Esta función busca en la tabla ordenada "*My_Regions*" si existe algún elemento que coincida con el identificado que se le pasa por referencia a esta función "*Id*". En el caso de encontrase, esta función devuelve la dirección del objeto "*Hierarchical_State ver 4.11*" relacionado con el identificador que se ha encontrado.

```
function Get_Region (This : Component;Id : State_Id) return Ptr_Hierarchical_State;
```

4.9.11. FunciónGet_Unique_Id

Esta función devuelve el identificado "*Unique_Id*" de este objeto.

```
function Get_Unique_Id (This : Component) return State_Id;
```

4.9.12. FunciónGet_Component_Data.

Esta función devuelve la dirección de la base de datos "*Comp_Data*" que es un objeto tipo "*MinFr_Component_Data ver 4.6*".

```
function Get_Component_Data (This : Component) return Ptr_MinFr_Component_Data;
```

4.10. ClaseMinfr_State.

La clase *Minfr_State ver 4.10* es una clase abstracta, es la encargada de construir definir y gestionar el comportamiento de los estados que están dentro de cada componente, en la siguiente figura se ve de forma esquemática como esta constituida esta clase.

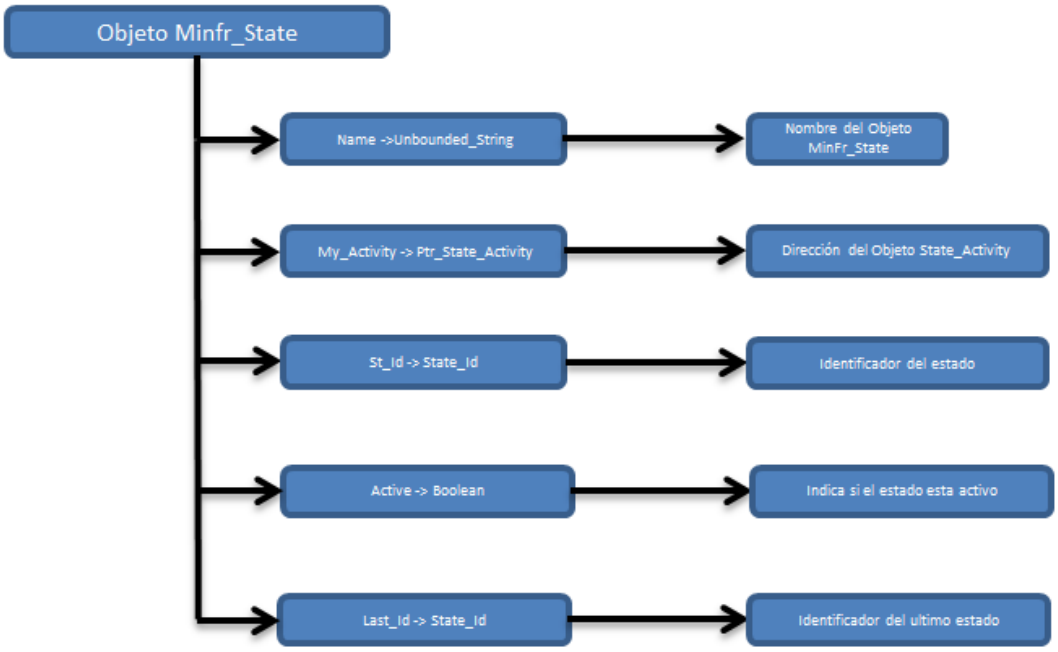

Ilustración 12: Objeto MinFr_State.

Constructores.

Constr_Minfr_State (This :in out Minfr_State_Ptr; Id : State_Id);

Los métodos y funciones que agrega esta clase son los siguientes.

On_Entry
On_Exit
Execute_Tick
Resolve_Next_State
Get_Activity
Get_Id
Is_Active
Get_Name
Is_Done

4.10.1. Constructor Constr_Minfr_State

Este procedimiento no es realmente un constructor, este procedimiento es llamado por las funciones constructores de las clases que heredan de este objeto abstracto.
A este procedimiento se le pasa por referencia la dirección del objeto "*Minfr_State ver 4.10*", que es en la cual va a actuar este procedimiento. Asignándole a "*St_Id*" el identificador pasado por referencia en este procedimiento "*Id*", crear el objeto "*State_Activity ver 4.15*" mediante el constructor "*Constr_Null_Activity*" y asignarle la dirección de este objeto a "*My_Activity*". Este objeto es una actividad nula, esto quiere decir que tiene todas las características de un objeto tipo "*State_Activity ver 4.15*" pero no realizan ninguna tarea todos sus procedimientos y funciones.

Asigna la cadena de caracteres "*NoName_st*" a "*Name*" y por último indica que este estado esta desactivado "*Active=false*".

```
procedure Constr_Minfr_State (This :in out Minfr_State_Ptr; Id : State_Id);
```

4.10.2. ProcedimientoOn_Entry.

Este procedimiento es abstracto y se define en las clases que heredarán de esta clase, las clase que heredan de esta clase son "*Hierarchical_State ver 4.11*" y "*Leaf_State ver 4.14*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

```
procedure On_Entry (This : in out Minfr_State) is abstract;
```

4.10.3. ProcedimientoOn_Exit.

Este procedimiento es abstracto y se define en las clases que heredarán de esta clase, las clase que heredan de esta clase son "*Hierarchical_State ver 4.11*" y "*Leaf_State ver 4.14*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

```
procedure On_Exit (This : in out Minfr_State) is abstract;
```

4.10.4. ProcedimientoExecute_Tick.

Este procedimiento ejecuta el procedimiento "*Execute_Tick*" del objeto tipo "*State_Activity ver 4.15*" que esta almacenada su dirección en "*My_Activity*"

```
procedure Execute_Tick (This : in out Minfr_State);
```

4.10.5. FunciónResolve_Next_State.

Esta función es abstracta y se define en las clases que heredarán de esta clase, las clase que heredan de esta clase son "*Hierarchical_State ver 4.11*" y "*Leaf_State ver 4.14*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

```
function Resolve_Next_State (This : Minfr_State) return State_Id is abstract;
```

4.10.6. Función Get_Activity.

Esta función devuelve la dirección del objeto tipo "*State_Activity ver 4.15*" que esta almacenada en "*My_Activity*".

```
function Get_Activity (This : Minfr_State) return Ptr_State_Activity;
```

4.10.7. Función Get_Id

Esta función devuelve el identificador del objeto "*Minfr_State ver 4.10*" que esta almacenada en "*St_Id*".

function Get_Id (This :Minfr_State) return State_Id;

4.10.8. FunciónIs_Active

Esta función devuelve el estado de activación del objeto "*Minfr_State ver 4.10*" que esta almacenada en "*Active*".

function Is_Active (This : Minfr_State) return Boolean;

4.10.9. FunciónGet_Name.

Esta función devuelve el nombre del objeto "*Minfr_State ver 4.10*" que esta almacenada en "*Name*".

function Get_Name (This : Minfr_State) return String;

4.10.10. FunciónIs_Done

Esta función devuelve lo que devuelve la función "*Is_Done*" del objeto tipo "*State_Activity ver 4.15*" que esta almacenada su dirección en "*My_Activity*"

function Is_Done (This : Minfr_State) return Boolean;

4.11. ClaseHierarchical_State.

Esta clase hereda de la clase "*Minfr_State ver 4.10*" y a demás es una clase abstracta. Esta clase aumenta las funcionalidades a los estados que están dentro de los componentes. En la siguiente figura se ve de forma esquemática como esta constituida esta clase.

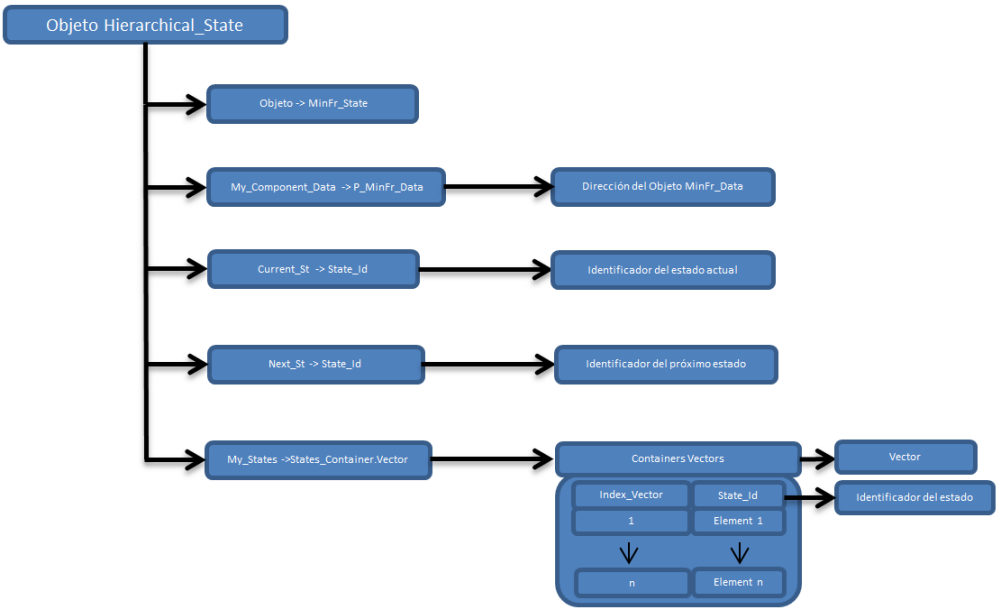

Ilustración 13: Objeto Hirarchical_State.

Constructores.

Constr_Hierarchical_State (tmp :in out Ptr_Hierarchical_State; data : P_MinFr_Data;Id :State_Id);

Los métodos y funciones que agrega esta clase son los siguientes.

Get_States
Set_Enable_State_Id
Get_Hierarchical_Activity
Resolve_Next_State

4.11.1. Constructor Constr_Hierarchical_State

Este procedimiento no es realmente un constructor, este procedimiento es llamado por las funciones constructores de las clases que heredan de este objeto abstracto.
Este constructor define como es el estado llamando al constructor "*Constr_Minfr_State ver 4.10.1*", define cual es la actividad que se va ejecutar llamando al constructor "*Cont_Hierarchical_Activity ver 4.12.1*", define cual es el id de la actividad que se va ejecuta y por ultimo indica cual es la dirección de la base de datos donde se van almacenar y consultar los datos.

procedure Constr_Hierarchical_State (tmp :in out Ptr_Hierarchical_State; data : P_MinFr_Data;Id :State_Id);

4.11.2. FunciónGet_States.

Esta función devuelve un vector que contiene los identificadores "*State_Id*" de todos los estados.

function Get_States (this : Hierarchical_State) return States_Container.Vector;

4.11.3. ProcedimientoSet_Enable_State_Id.

Este procedimiento habilita el estado que coincida con el id que se le pasa por referencia a este procedimiento "*St_Id*".

procedure Set_Enable_State_Id (this : in Hierarchical_State;St_Id : in State_Id);

4.11.4. FunciónResolve_Next_State

Esta función devuelve el id "*State_Id*" que indica cual es el siguiente estado que ha de ejecutarse.

function Resolve_Next_State (This : Hierarchical_State) return State_Id;

4.12. Clase Hierarchical_Activity.

Esta clase hereda de las clases "*Sobrecargable*" y "*State_Activity ver 4.15*". Esta clase es la encargada de gestionar y ejecutar las actividades que hay dentro de una región.En la siguiente figura se ve de forma esquemática como esta constituida esta clase.

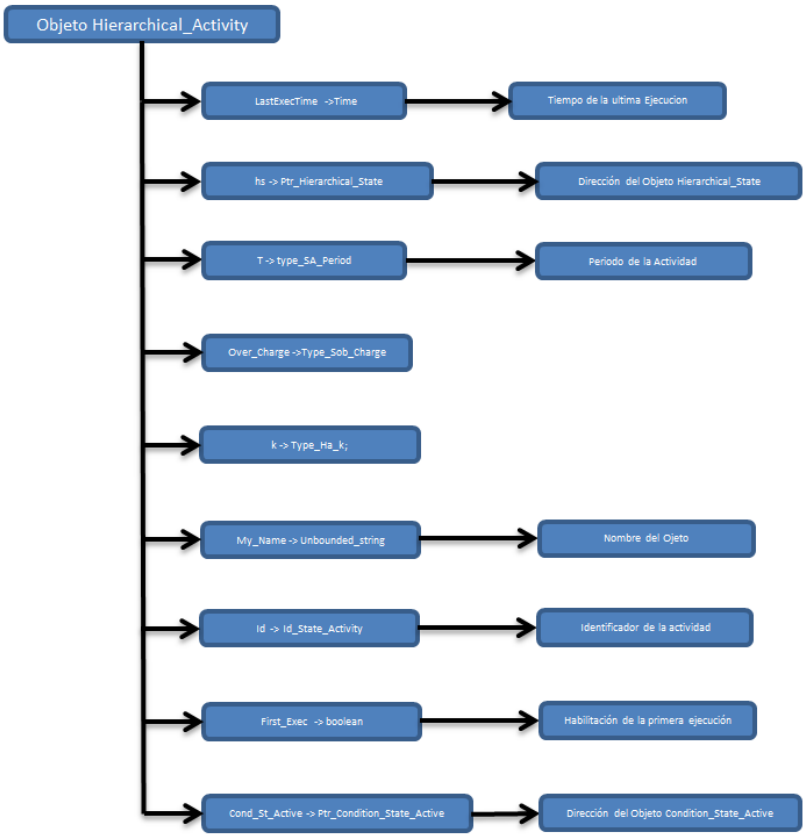

Ilustración 14: Objeto Hirarchical_Activity.

Constructores.

Cont_Hierarchical_Activity (hs :Ptr_Hierarchical_State) return Ptr_Hierarchical_Activity;

Los métodos y funciones que agrega esta clase son los siguientes.

Can_Execute
Set_Over_charge
Execute_Tick
Set_Period
Get_Period
Is_Done
Get_Name
Set_Name
Set_Id
Get_Id
Execute_On_Entry
Execute_On_Exit
Set_Enable_State_Id

4.12.1. Constructor Cont_Hierarchical_Activity.

Esta función crea el objeto tipo "*Hierarchical_Activity ver 4.12*". Esta función hay que pasarle por referencia la dirección del objeto "*Hierarchical_State ver 4.11*" para indicarle la jerarquía de las regiones que componen el componente.

```
function Cont_Hierarchical_Activity (hs :Ptr_Hierarchical_State) return Ptr_Hierarchical_Activity;
```

4.12.2. Función Can_Execute.

Evalúa si se puede ejecutar la actividad dentro de la región, evaluando las condiciones de los estados, y el tiempo de ejecución de las actividades.

```
function Can_Execute(this : Hierarchical_Activity) return boolean;
```

4.12.3. ProcedimientoSet_Over_charge.

Este procedimiento establece el valor de "*Over_Charge*".

```
procedure Set_Over_charge (This : in out Hierarchical_Activity; n : in Type_Sob_Charge);
```

4.12.4. ProcedimientoExecute_Tick.

Este procedimiento es el encargado de ejecutar y gestionar el orden de ejecución de todos los estados que componen una región del componente.

procedure Execute_Tick (this : in out Hierarchical_Activity);

4.12.5. ProcedimientoSet_Period.

Establece el periodo en milisegundos de la actividad que se va ejecutar en la región perteneciente a un componente.

procedure Set_Period (this : in out Hierarchical_Activity;Per_ms : type_SA_Period);

4.12.6. FunciónGet_Period.

Esta función devuelve el valor del periodo en milisegundos de la actividad que se esta ejecutando en una región perteneciente a un componente.

function Get_Period (this : Hierarchical_Activity) return type_SA_Period;

4.12.7. FunciónIs_Done.

Esta función indica si se ha realizado la actividad que se ejecutaba en la región.

function Is_Done (this : Hierarchical_Activity) return boolean;

4.12.8. FunciónGet_Name.

Esta función nos informa del nombre del objeto "*Hierarchical_Activity ver 4.12*".

function Get_Name (this : Hierarchical_Activity) return string;

4.12.9. ProcedimientoSet_Name.

Este procedimiento establece el nombre del objeto "*Hierarchical_Activity ver 4.12*".

procedure Set_Name (this : in out Hierarchical_Activity; str :string);

4.12.10. ProcedimientoSet_Id.

Este procedimiento establece cual va a ser el identificador de la región que se está definiendo en este objeto.

procedure Set_Id (This : in out Hierarchical_Activity;id : Id_State_Activity);

4.12.11. FunciónGet_Id.

Esta función devuelve el valor del identificador de la región que define este objeto.

function Get_Id (this : Hierarchical_Activity) return Id_State_Activity;

4.12.12. ProcedimientoExecute_On_Entry.

Este procedimiento no realiza ninguna tarea, se define por que este objeto hereda del objeto "*State_Activity ver 4.15*" y tiene esta función definida como abstracta.

procedure Execute_On_Entry (This :in out Hierarchical_Activity);

4.12.13. ProcedimientoExecute_On_Exit.

Este procedimiento no realiza ninguna tarea, se define por que este objeto hereda del objeto "*State_Activity ver 4.15*" y tiene esta función definida como abstracta.

procedure Execute_On_Exit (This : Hierarchical_Activity);

4.12.14. ProcedimientoSet_Enable_State_Id

Este procedimiento crea el objeto "*Condition_State_Active ver 4.22*".Este objeto es en cargado de evaluar y gestionar las condiciones que se utilizan para hacer un cambio de estado.

procedure Set_Enable_State_Id (This : in out Hierarchical_Activity;St_Id : Id_State_Activity);

4.13. ClaseRegion.

Esta clase hereda de la clase "*Hierarchical_State ver 4.11*". Esta la encargada de crear y definir cada una de las regiones que componen el componente. No se define estructuras adicionales a este objeto, las que se utilizan son las que hereda de la clase padre "*Hierarchical_State ver 4.11*".

Constructores.

Constr_Region (data : P_MinFr_Data; name : string; states : Vector_ptr_Minfr_State.Vector; id : State_Id; Component_Unique_Id : State_Id) return Ptr_Region;

Los métodos y funciones que agrega esta clase son los siguientes.

On_Entry
On_Exit

4.13.1. ConstructorConstr_Region.

Este constructor crea el objeto región, indica cual va a ser la dirección donde se van almacenar y gestionar los datos de la región y agrega los estados que componen dicha región.

function Constr_Region (data : P_MinFr_Data; name : string; states : Vector_ptr_Minfr_State.Vector; id : State_Id; Component_Unique_Id : State_Id) return Ptr_Region;

4.13.2. Procedimiento On_Entry.

Este procedimiento no realiza ninguna tarea, se define por que este objeto hereda del objeto "*Hierarchical_State ver 4.11*" y este a su vez de "*Minfr_State ver 4.10*" que tiene esta función definida como abstracta.

procedure On_Entry (This : in out Region);

4.13.3. Procedimiento On_Exit.

Este procedimiento no realiza ninguna tarea, se define por que este objeto hereda del objeto "*Hierarchical_State ver 4.11*" y este a su vez de "*Minfr_State ver 4.10*" que tiene esta función definida como abstracta.

procedure On_Exit (This : in out Region);

4.14. ClaseLeaf_State.

Esta clase hereda de la clase "*Minfr_State ver 4.10*".Esta clase en la encargada de asignar que actividad se va a ejecutar en cada región, y gestionar la ejecución de la actividad y de los estados que constituyen la región.En la siguiente figura se ve de forma esquemática como está constituida esta clase.

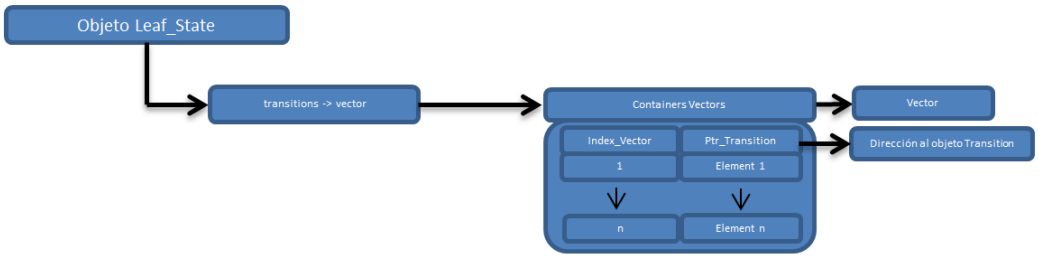

Ilustración 15: ObjetoLeaf_State.

Constructores.

Constr_Leaf_State (stID : State_Id; name : string) return Ptr_Leaf_State;

Los métodos y funciones que agrega esta clase son los siguientes.

Add_Transition
Set_Activity
On_Entry
On_Exit
Resolve_Next_State

4.14.1. ConstructorConstr_Leaf_State.

Esta función crea el objeto tipo "*Leaf_State ver 4.14*", y como este objeto hereda del objeto "*Minfr_State ver 4.10*" llama al procedimiento "Constr_Minfr_State *ver* 4.10.1", que define las características del objeto. Esta función también establece el nombre del objeto.

function Constr_Leaf_State (stID : State_Id; name : string) return Ptr_Leaf_State;

4.14.2. ProcedimientoAdd_Transition.

Este procedimiento agrega al vector de transiciones la dirección del objeto de tipo "*Transition ver 4.23*", que se le pasa por referencia "*trans*".

procedure Add_Transition(This : in out Leaf_State; trans : Ptr_Transition);

4.14.3. ProcedimientoSet_Activity.

Este procedimiento establece la dirección del objeto "*State_Activity ver 4.15*" que es la actividad que se va ejecutar en el estado, esta dirección se le pasa por referencia con el nombre "*State_Act*"

procedure Set_Activity(This : in out Leaf_State; State_Act : Ptr_State_Activity);

4.14.4. ProcedimientoOn_Entry.

Este procedimiento pone en activo la actividad.

procedure On_Entry (This : in out Leaf_State);

4.14.5. ProcedimientoOn_Exit.

Este procedimiento desactiva la actividad.

procedure On_Exit (This : in out Leaf_State);

4.14.6. FunciónResolve_Next_State.

Esta función devuelve el identificador "*State_Id*" del próximo estado que ha de ejecutarse.

function Resolve_Next_State (This : Leaf_State) return State_Id;

4.15. Clase State_Activity

Esta clase es de tipo interface, es donde se declaran todos los métodos y funciones que van a ser visibles y pueden ser utilizados por otras clase que utilice esta clase. Estos métodos y funciones son los que define y gestionan el

comportamiento de los estados *Framework*, definiendo el periodo del estado e identificador de este.

Estos son los métodos y funciones de la clase "*State_Activity ver 4.15*".

Execute_Tick
Set_Period
Get_Period
Is_Done
Get_Name
Set_Name
Set_Id
Get_Id
Execute_On_Entry
Execute_On_Exit
Set_Enable_State_Id

4.15.1. ProcedimientoExecute_Tick.

Este procedimiento es abstracto y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*Leaf_Activity ver 4.16*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

```
procedure Execute_Tick (This :in out State_Activity) is abstract;
```

4.15.2. Procedimiento Set_Period.

Este procedimiento es abstracto y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*Leaf_Activity ver 4.16*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

```
procedure Set_Period(This : in out State_Activity;Per : type_SA_Period) is abstract;
```

4.15.3. Función Get_Period.

Esta función es abstracta y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*Leaf_Activity ver 4.16*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

```
function  Get_Period(This :State_Activity)return type_SA_Period is abstract;
```

4.15.4. Función Is_Done.

Esta función es abstracta y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*Leaf_Activity ver 4.16*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

```
function  Is_Done(This :State_Activity) return boolean is abstract;
```

4.15.5. FunciónGet_Name.

Esta función es abstracta y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*Leaf_Activity ver 4.16*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

```
function  Get_Name (This :State_Activity) return string is abstract;
```

4.15.6. Procedimiento Set_Name.

Este procedimiento es abstracto y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*Leaf_Activity ver 4.16*". En la *Ilustración 5*Se puede ver el esquema de bloques de las clases y la dependencia entre clase.

```
procedure Set_Name(This :in out State_Activity;str : string) is abstract;
```

4.15.7. ProcedimientoSet_Id.

Este procedimiento es abstracto y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*Leaf_Activity ver 4.16*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

```
procedure Set_Id (This :in out State_Activity;id : Id_State_Activity) is abstract;
```

4.15.8. FunciónGet_Id.

Esta función es abstracta y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*Leaf_Activity ver 4.16*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

```
function  Get_Id (This :State_Activity) return Id_State_Activity is abstract;
```

4.15.9. ProcedimientoExecute_On_Entry.

Este procedimiento es abstracto y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*Leaf_Activity ver 4.16*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

```
procedure Execute_On_Entry (This :in out State_Activity) is abstract;
```

4.15.10. Procedimiento Execute_On_Exit.

Este procedimiento es abstracto y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*Leaf_Activity ver 4.16*". En

la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

procedure Execute_On_Exit(This :State_Activity)is abstract;

4.15.11. Procedimiento Set_Enable_State_Id

Este procedimiento es abstracto y se define en las clases que heredarán de esta clase, la clase que heredan de esta clase es "*Leaf_Activity ver 4.16*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clase.

procedure Set_Enable_State_Id (this : in out State_Activity;St_Id : in Id_State_Activity) is abstract;

4.16. Clase Leaf_Activity.

Esta clase abstracta hereda de la clase interface "*State_Activity ver 4.15*".esta clase define y gestionan el comportamiento de los estados *Framework*, definiendo el periodo del estado e identificador de este.En la siguiente figura se ve de forma esquemática como está constituida esta clase.

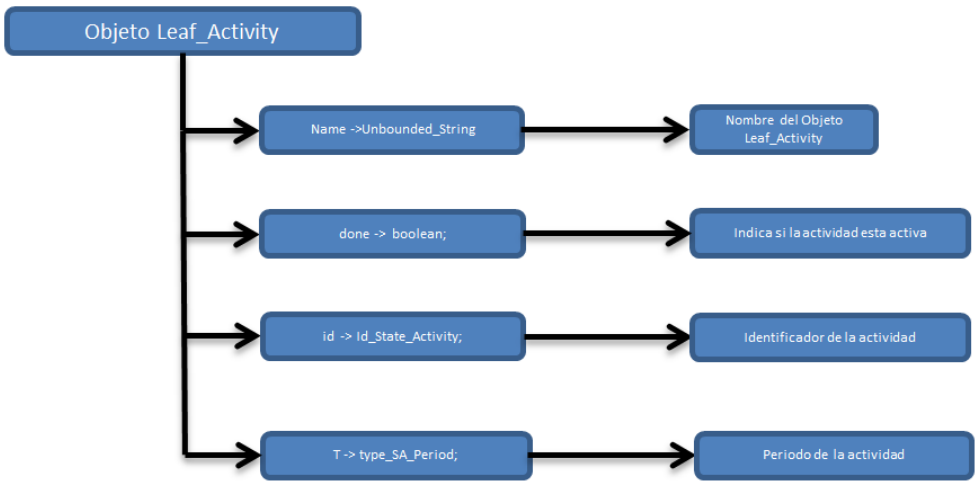

Ilustración 16: Objeto Leaf_Activity.

Constructores.

Constr_Leaf_Activity (tmp :in out Ptr_Leaf_Activity; Id :Id_State_Activity)
Constr_Leaf_Activity(tmp :in out Ptr_Leaf_Activity)

Los métodos y funciones que agrega esta clase son los siguientes.

Execute_Tick
Set_Period
Get_Period
Is_Done
Set_Done
Get_Name

Set_Name
Set_Id
Get_Id
Execute_On_Entry
Execute_On_Exit
Set_Enable_State_Id

4.16.1. Procedimiento Constr_Leaf_Activity.

Este procedimiento define las características del objeto "*Leaf_Activity ver 4.16*", pasado por referencia la dirección de este objeto "*tmp*". Definiendo el periodo de la actividad a 0 y especifica el identificador de la actividad pasado por referencia "Id".

```
procedure Constr_Leaf_Activity (tmp :in out Ptr_Leaf_Activity; Id :Id_State_Activity);
```

4.16.2. Procedimiento Constr_Leaf_Activity.

Este procedimiento define las características del objeto "*Leaf_Activity ver 4.16*", pasado por referencia la dirección de este objeto "*tmp*". Definiendo el periodo de la actividad a 0.

```
procedure Constr_Leaf_Activity(tmp :in out Ptr_Leaf_Activity);
```

4.16.3. ProcedimientoExecute_Tick.

Este procedimiento no realiza ninguna actividad, se ha tenido que definir por que el objeto "*Leaf_Activity ver 4.16*" hereda del objeto "*State_Activity ver 4.15*" y este procedimiento esta declarado como abstracto y esta obligado que todos los objetos hijos definan todos los procedimientos y funciones abstractas.

```
procedure Execute_Tick (This :in out Leaf_Activity);
```

4.16.4. Procedimiento Set_Period.

Este procedimiento establece el periodo que va a tener la actividad

```
procedure Set_Period (This : in out Leaf_Activity;Per : type_SA_Period);
```

4.16.5. FunciónGet_Period.

Esta función evuelve el periodo de la actividad.

```
function Get_Period (This :Leaf_Activity)return type_SA_Period;
```

4.16.6. FunciónIs_Done.

Esta función indica el estado de la actividad, indicando si esta activa o desactiva.

function Is_Done(This :Leaf_Activity) return boolean;

4.16.7. Procedimiento Set_Done.

Este procedimiento establece el estado de la actividad, poniendo la en activo o desactivo en función del valor pasado por referencia "".

procedure Set_Done(This :in out Leaf_Activity);

4.16.8. FunciónGet_Name.

Esta función devuelve el nombre que tiene la actividad.

function Get_Name (This :Leaf_Activity) return string;

4.16.9. Procedimiento Set_Name.

Este procedimiento establece el nombre de la actividad, mediante la variable "*srt*" pasada por referencia en este procedimiento.

procedure Set_Name (This : in out Leaf_Activity;str : string);

4.16.10. ProcedimientoSet_Id.

Este procedimiento establece el identificado numérico de la actividad, mediante la variable "*id*" pasada por referencia en este procedimiento.

procedure Set_Id (This : in out Leaf_Activity;id : Id_State_Activity);

4.16.11. FunciónGet_Id.

Esta función devuelve el valor numérico del identificador de la actividad.

function Get_Id (This :Leaf_Activity) return Id_State_Activity;

4.16.12. ProcedimientoExecute_On_Entry.

Este procedimiento establece el estado de la actividad desactivado

procedure Execute_On_Entry (This :in out Leaf_Activity);

4.16.13. Procedimiento Execute_On_Exit.

Este procedimiento no realiza ninguna actividad, se ha tenido que definir por que el objeto "*Leaf_Activity ver 4.16*" hereda del objeto "*State_Activity ver 4.15*" y este procedimiento esta declarado como abstracto y están

obligado que todos los objetos hijos definan todos los procedimientos y funciones abstractas.

procedure Execute_On_Exit(This :Leaf_Activity);

4.16.14. Procedimiento Set_Enable_State_Id.

Este procedimiento no realiza ninguna actividad, se ha tenido que definir por que el objeto "*Leaf_Activity ver 4.16*" hereda del objeto "*State_Activity ver 4.15*" y este procedimiento esta declarado como abstracto y están obligado que todos los objetos hijos definan todos los procedimientos y funciones abstractas.

procedure Set_Enable_State_Id (this : in out Leaf_Activity;St_Id : in Id_State_Activity);

4.17. Clase Cmd_Up_Date.

Esta clase hereda de la clase "*Leaf_Activity ver 4.16*".en todos los componentes se ha de definir este objeto, es el encargada de gestionar y transferir los mensajes de los puertos de salida del componente, a los puertos de entrada de los componentes que estén conectados a este.En la siguiente figura se ve de forma esquemática como está constituida esta clase.

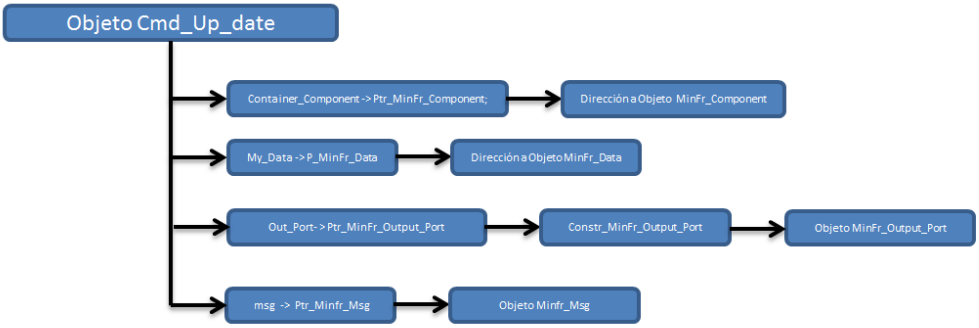

Ilustración 17: Objeto Cmd_Up_Date.

Constructores.

Constr_Cmd_Up_date (Id : Id_State_Activity;Container_Component : Ptr_MinFr_Component;Data : Ptr_MinFr_Component_Data) return Ptr_Cmd_Up_date

Los métodos y funciones que agrega esta clase son los siguientes.

Execute_Tick

4.17.1. FunciónConstr_Cmd_Up_date.

Esta función crea el objeto tipo "*Cmd_Up_Date ver 4.17*", definiendo la dirección del objeto "*MinFr_Component_Data ver 4.6*" que es base de

datos del componente y la dirección del objeto que define la arquitectura del componente "*MinFr_Component ver 4.8*".

function Constr_Cmd_Up_date (Id : Id_State_Activity;Container_Component : Ptr_MinFr_Component;Data : Ptr_MinFr_Component_Data) return Ptr_Cmd_Up_date;

4.17.2. Procedimiento Execute_Tick.

Este procedimiento es el encargado de gestionar y transferir los mensajes de los puertos de salida del componente, a los puertos de entrada de los componentes que estén conectados a este

procedure Execute_Tick (This :in out Cmd_Up_date);

4.17.2.1. Vista Algorítmica

Para quede mas clara el funcinamiento de este procemiento se describe a continuación el algoritmo que ejecuta este procedimento.

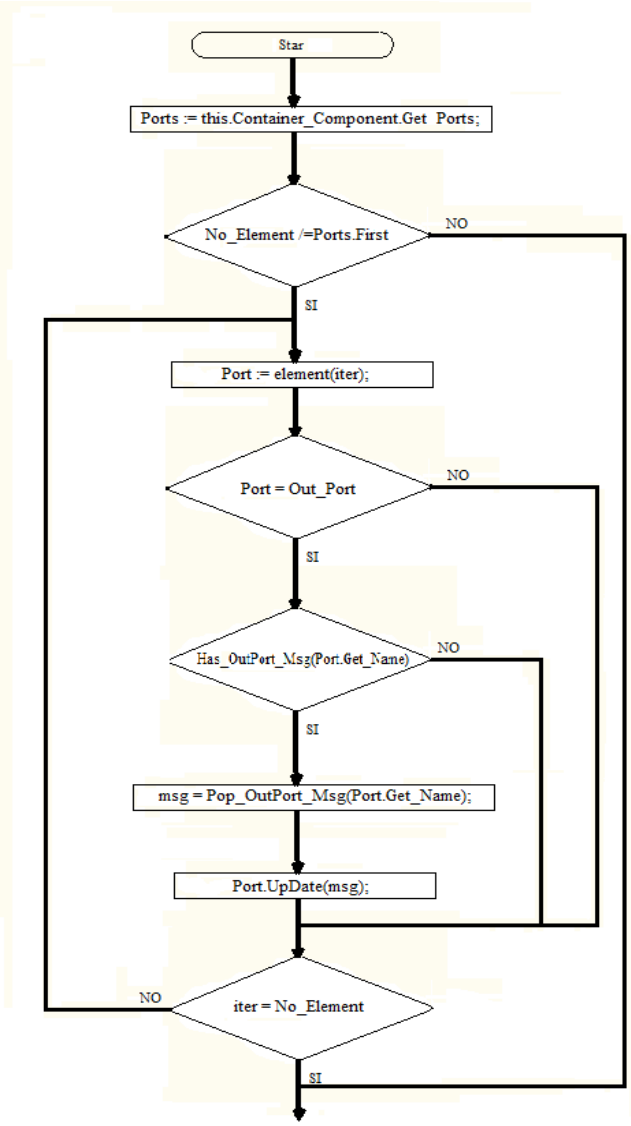

Ilustración 18: Vista algorítmica procedimientoExecute_Tick de Cmd_Up_Date.

Este algoritmo lo primero que hace es evaluar si hay algún objeto definido como pueto en el componente, si dicho componente tiene puertos de comunicaciones el siguente pasa es chequear todos los puertos y averiguara si son del tipo salida "*Out_Port*". Una vez localizados averiguamos si tiene algún mensaje en el buffer de salida, lo estraemos y lo enviamos al puerto de entrada que este conectado este puerto de salida, con el procedimento "*UpDate*"

4.18. clase Activity_Processor.

La clase "*Activity_Processor ver 4.18*" define un único objeto, este objeto es el alma de *Framework*, es el más importante de todos. Este objeto es el encargado de definir la estructura, gestionar y ejecutar todas las tareas en el *Framework*. En las siguientes figuras se ve de forma esquemática como está constituida esta clase.

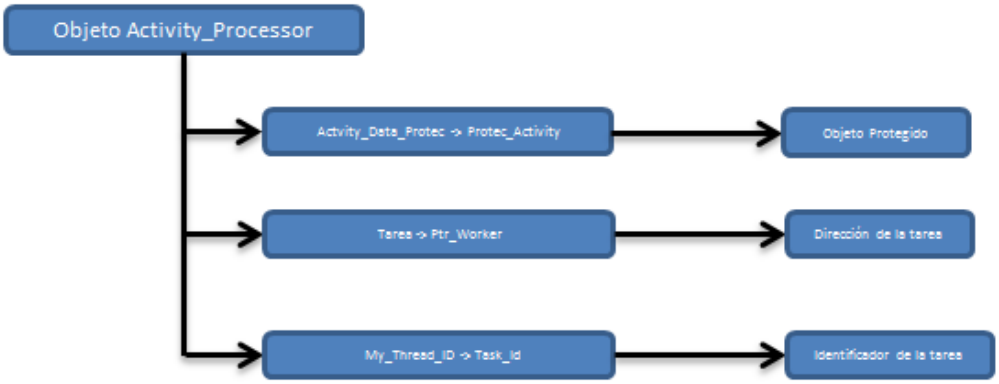

Ilustración 19: Objeto Activity_Processor.

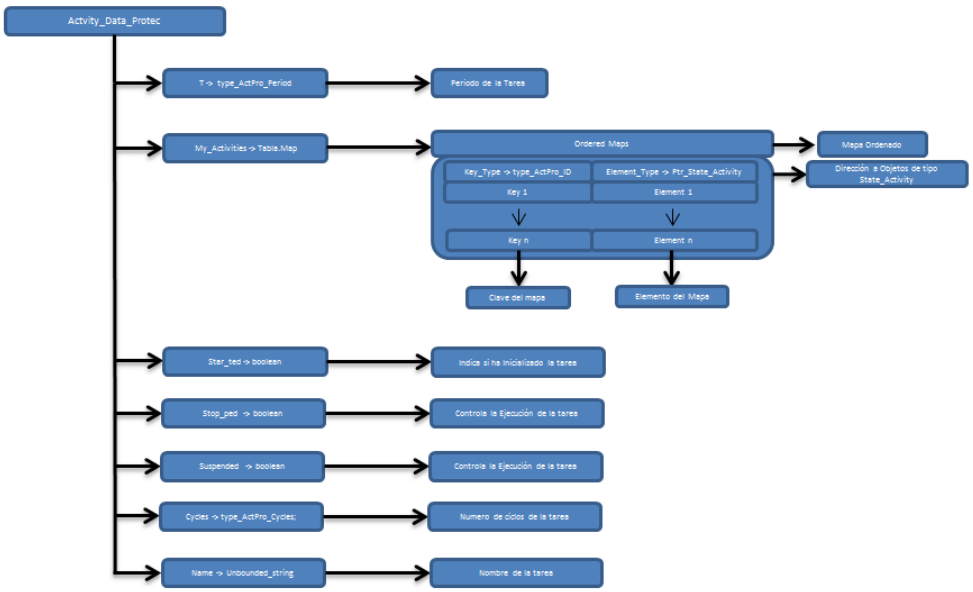

Ilustración 20: ObjetoProtegido Activity_Processor.

Constructores.

Constr_Activity_Processor return Ptr_Activity_Processor;
Constr_Activity_Processor (str : string) return Ptr_Activity_Processor;

Los métodos y funciones que agrega esta clase son los siguientes.

Get_State
Get_Name
Set_Name
Get_Cycles
Start
Suspend
Resume
Stop
Set_Period
Get_Period
Add_Activity
Remove_Activity
Reset_Activities
gcd

4.18.1. Constructor Constr_Activity_Processor.

Esta función crea el objeto tipo "*Activity_Processor ver 4.18*", e inicializa todos los valores del objeto a un valor predeterminado para un correcto funcionamiento.

function Constr_Activity_Processor return Ptr_Activity_Processor;

4.18.2. ConstructorConstr_Activity_Processor.

Esta función crea el objeto tipo "*Activity_Processor ver 4.18*", e inicializa todos los valores del objeto a un valor predeterminado para un correcto funcionamiento y establece el nombre de la tarea que es pasado por referencia "*str*" en esta función.

function Constr_Activity_Processor (str : string) return Ptr_Activity_Processor;

4.18.3. FunciónGet_State.

Esta función devuelve el nombre que tiene asignada la tarea.

function Get_State (this : Activity_Processor) return string;

4.18.4. FunciónGet_Name.

Esta función devuelve el nombre que tiene asignada la tarea.

function Get_Name (this : Activity_Processor) return string;

4.18.5. ProcedimientoSet_Name.

Este procedimiento establece el nombre de la tarea, pasado por referencia en esta función "*str*".

Procedure Set_Name (this : in out Activity_Processor; str :string);

4.18.6. FunciónGet_Cycles.

Esta función nos devuelve el número de ciclos que se han ejecutado en la tarea.

function Get_Cycles (this : Activity_Processor) return type_ActPro_Cycles;

4.18.7. ProcedimientoStart.

Este procedimiento inicia la ejecución de la tarea. Para que funcione correctamente hay que pasarle a este procedimiento la dirección de objeto "*Activity_Processor ver 4.18*".

Procedure Start (this : in out Activity_Processor; obj : Ptr_Activity_Processor);

4.18.8. ProcedimientoSuspend.

Este procedimiento suspende temporalmente la ejecución de la tarea que lo define.

Procedure Suspend (this : in out Activity_Processor);

4.18.9. ProcedimientoResume.

Este procedimiento reanuda la ejecución de la tarea si esta no esta ya.

Procedure Resume (this : in out Activity_Processor);

4.18.10. ProcedimientoStop.

Este procedimientofinaliza la ejecución de la tarea.

Procedure Stop (this : in out Activity_Processor);

4.18.11. ProcedimientoSet_Period.

Este procedimiento define el periodo de la tarea, mediante la variable "*Per*" que se le pasa por referencia.

Procedure Set_Period (this : in out Activity_Processor; Per :type_ActPro_Period);

4.18.12. FunciónGet_Period.

Esta función nos devuelve el valor del periodo de la tarea.

function Get_Period (this : Activity_Processor) return type_ActPro_Period;

4.18.13. ProcedimientoAdd_Activity.

Este procedimiento agrega las actividades que se van a ejecutar en la tarea, estas actividades se pasan por referencia en este procedimiento con el nombre "*Act*".

Procedure Add_Activity (this : in out Activity_Processor; Act : Ptr_State_Activity);

4.18.14. ProcedimientoRemove_Activity.

Este procedimiento elimina de la lista de actividades que se esta ejecutando en esta tarea, la actividad que coincide con el identificador "*Act_Id*" que se le pasa por referencia a este procedimiento.

Procedure Remove_Activity (this : in out Activity_Processor; Act_Id : type_ActPro_ID);

4.18.15. ProcedimientoReset_Activities.

Este procedimiento elimina todas las actividades que se están ejecutando en la tarea.

Procedure Reset_Activities (this : in out Activity_Processor);

4.18.16. Funcióngcd.

Esta función nos devuelve el máximo común divisor entre dos periodos. Esta función es utilizada por este objeto para determinar cual es el periodo que ha de tener la tarea, cuando tiene asignada varias actividades.

function gcd (this : Activity_Processor; P ,Q : type_ActPro_Period) return type_ActPro_Period;

4.19. Clase Condition.

Esta clase es de tipo interface, Es donde se declaran todos los métodos y funciones que van a ser visibles y pueden ser utilizados por otras clase que utilice esta clase. Esta clase define las funciones que gestiona el comportamiento de cambio de estados entre los estados

Estos son los métodos y funciones de la clase "*Condition*".

Evaluate_Condition

4.19.1. Función Evaluate_Condition

Esta función es abstracta y se define en las clases que heredarán de esta clase, las clases que heredan de esta clase son "*Condition_Event ver 4.20*", "*Condition_Port ver 4.21*" y "*Condition_State_Active ver 4.22*". En la *Ilustración 5* se puede ver el esquema de bloques de las clases y la dependencia entre clases.

function Evaluate_Condition (this : Condition) return boolean is abstract;

4.20. Clase Condition_Event.

Esta clase hereda de la clase interface "*Condition ver 4.19*"Esta clase define el comportamiento de los cambio de estados evaluando los mensajes del tipo "*conditiont_Event*" que se han producido, en el transcurso de la ejecución de una actividad. En las siguientes figuras se ve de forma esquemática como está constituida esta clase.

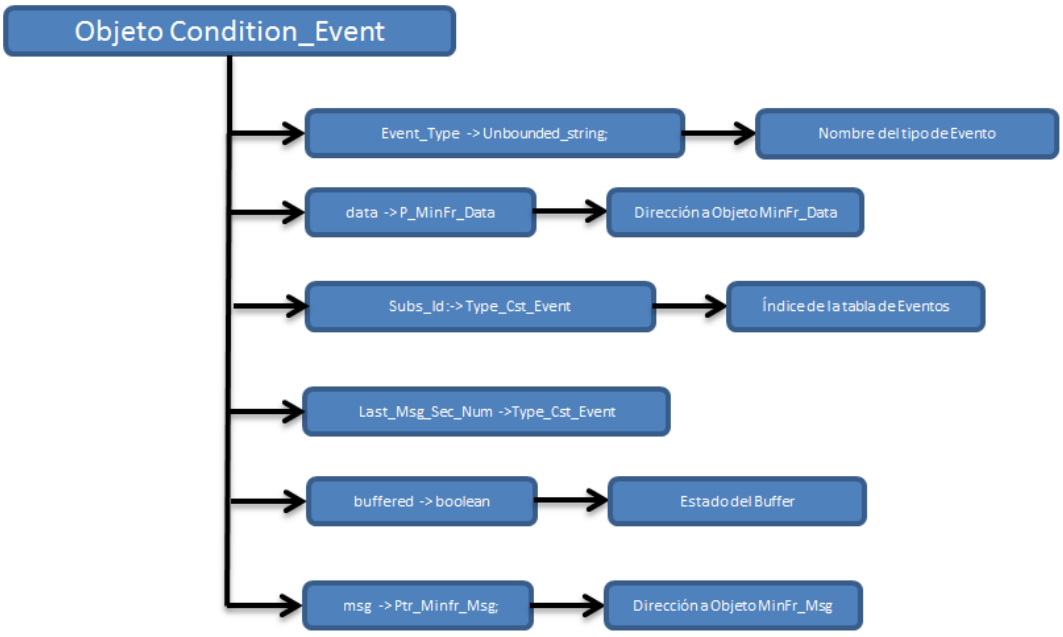

Ilustración 21: Objeto Condition_Event.

Constructores.

Constr_Condition_Event (data : P_MinFr_Data; Event_Type : string; buffered : boolean)

Los métodos y funciones que agrega esta clase son los siguientes.

Get_Name
Evaluate_Condition

4.20.1. Constructor Constr_Condition_Event.

Esta función crea el objeto tipo "*Condition_Event ver 4.20*", e inicializa todos los valores del objeto a un valor predeterminado para un correcto funcionamiento.Creando un buffer de datos en la base de datos que se le ha pasado por referencia "data", del tipo "*Event*".
La variable que se le pasa por referencia "buffered", se utiliza para establecer si se saca el último mensaje del buffer de la base de datos o se sacan todos los mensajes del buffer.

```
function Constr_Condition_Event (data : P_MinFr_Data; Event_Type : string; buffered : boolean)return Ptr_Condition_Event;
```

4.20.2. FunciónGet_Name

Esta función devuelve el nombre del mensaje del objeto "*Condition_Event ver 4.20*"

```
function Get_Name (this : Condition_Event) return String;
```

4.20.3. *FunciónEvaluate_Condition*

Esta función evalúa si hay algún mensaje en la base de datos del tipo "*Event*" de este objeto, en caso de existir dicho mensaje devuelve un valor verdadero, en caso contrario falso.

```
function Evaluate_Condition (this : Condition_Event) return boolean;
```

4.21. Clase Condition_Port.

Esta clase hereda de la clase interface "*Condition ver 4.19*"Esta clase define el comportamiento de los cambio de estados evaluando los mensajes del tipo "*Condition_Port ver 4.21*" que se han producido, en el transcurso de la ejecución de una actividad.En las siguientes figuras se ve de forma esquemática como está constituida esta clase.

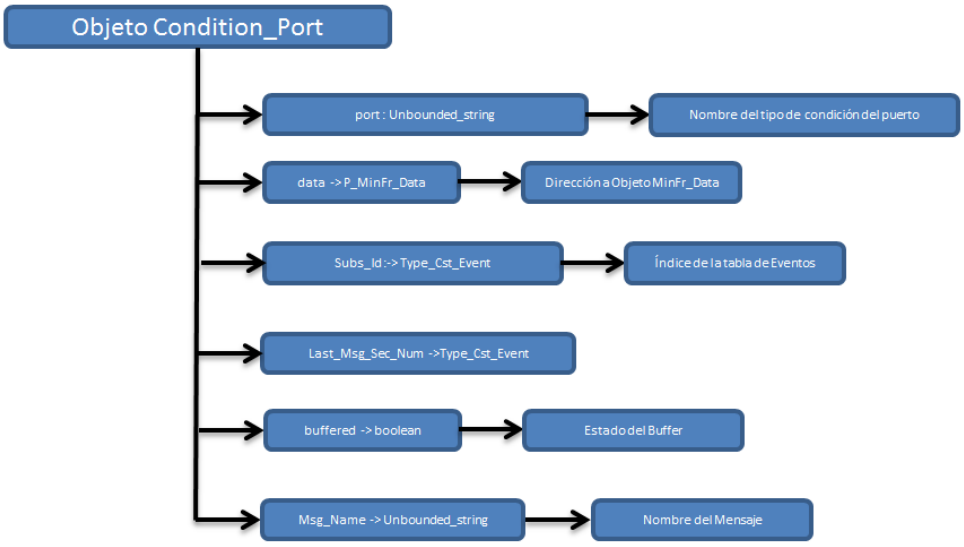

Ilustración 22: Objeto Condition_Port.

Constructores.

Constr_Condition_Port(data : P_MinFr_Data; port : string; msgName : string; buffered : boolean)

Los métodos y funciones que agrega esta clase son los siguientes.

Evaluate_Condition

4.21.1. Constr_Condition_Port.

Esta función crea el objeto tipo "*Condition_Port ver 4.21*", e inicializa todos los valores del objeto a un valor predeterminado para un correcto funcionamiento. Creando un buffer de datos en la base de datos que se le ha pasado por referencia "data", del tipo "*InPort*".
La variable que se le pasa por referencia "buffered", se utiliza para establecer si se saca el último mensaje del buffer de la base de datos o se sacan todos los mensajes del buffer.

function Constr_Condition_Port(data : P_MinFr_Data; port : string; msgName : string; buffered : boolean)return Ptr_Condition_Port;

4.21.2. FunciónEvaluate_Condition.

Esta función evalúa si hay algún mensaje en la base de datos del tipo "*InPort*" de este objeto, en caso de existir dicho mensaje devuelve un valor verdadero, en caso contrario falso.

function Evaluate_Condition (this : Condition_Port) return boolean;

4.22. Clase Condition_State_Active.

Esta clase hereda de la clase interface "*Condition ver 4.19*"Esta clase define el comportamiento de los cambio de estados evaluando los mensajes del tipo "*Condition_State_Active ver 4.22*" que se han producido, en el transcurso de la ejecución de una actividad.En las siguientes figuras se ve de forma esquemática como está constituida esta clase.

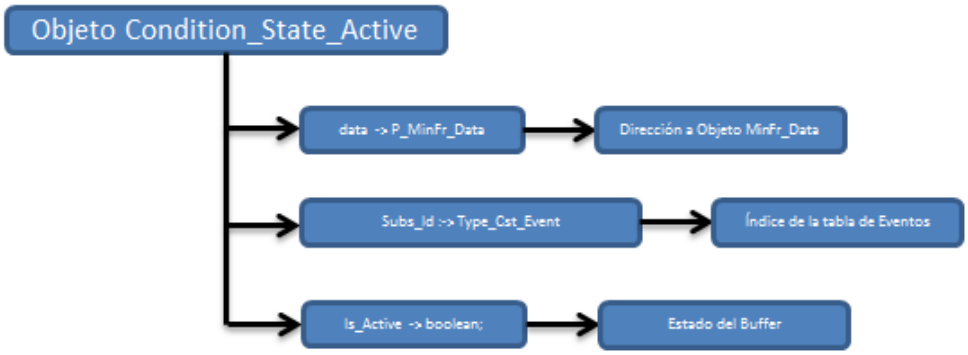

Ilustración 23: Objeto Condition_State_Active.

Constructores.

Constr_Condition_State_Active (data : P_MinFr_Data;St_Id : Type_CstAc_Id;Is_Active: boolean)

Los métodos y funciones que agrega esta clase son los siguientes.

Evaluate_Condition

4.22.1. Constr_Condition_State_Active.

Esta función crea el objeto tipo "*Condition_State_Active ver 4.22*", e inicializa todos los valores del objeto a un valor predeterminado para un correcto funcionamiento. Indica en la base de datos que se le ha pasado por referencia "data", el id del estado "*Condition_State_Active*".
La variable que se le pasa por referencia "Is_Activity", se utiliza para establecer como se devuelve el dato a la hora de evaluar la condición, negado o sin negar.

```
function Constr_Condition_State_Active (data : P_MinFr_Data;St_Id : Type_CstAc_Id;Is_Active: boolean)return Ptr_Condition_State_Active;
```

4.22.2. FunciónEvaluate_Condition.

Esta función evalúa si se ha producido un cambio de estado, en el estado relacionado con este objeto.

```
function Evaluate_Condition (this : Condition_State_Active) return boolean;
```

4.23. Clase Transition

Esta clase es la encargada de gestionar y evaluar todas las transiciones que se producen dentro de la región de un componente. En las siguientes figuras se ve de forma esquemática como está constituida esta clase.

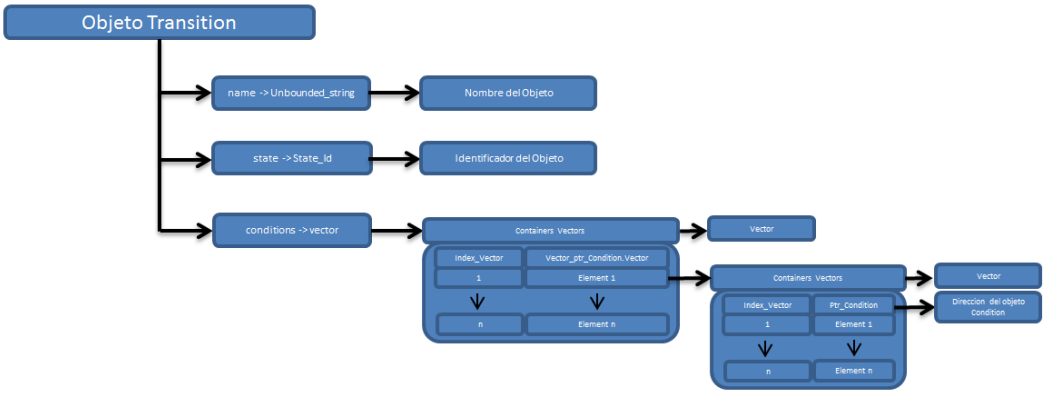

Ilustración 24: Objeto Transition.

Constructores.

Constr_Transition (Name : string)

Los métodos y funciones que agrega esta clase son los siguientes.

Evaluate_Transition
Add_Conditions
Set_State
Get_State
Get_Name

4.23.1. Constructor Constr_Transition (Name : string)

Esta función crea el objeto tipo "*Transition ver 4.23*", establece el Id delsiguiente estado que se tiene que ejecutar por defecto igual a "0", establece el nombre de este objeto mediante el valor "*Name*" que se le pasa por referencia a esta función.

```
function Constr_Transition (Name : string) return Ptr_Transition;
```

4.23.2. Función Evaluate_Transition.

Esta funciónevalúatodas las transiciones que se pueden producir dentro de una región si encuentra alguna activa devuelve el valor verdadero.

```
function Evaluate_Transition (this : Transition) return boolean;
```

4.23.3. ProcedimientoAdd_Conditions.

Este procedimiento agrega un vector de objetos del tipo "*Condition ver 4.19*"

procedure Add_Conditions (this : in out Transition; vec : Vector_ptr_Condition.Vector);

4.23.4. ProcedimientoSet_State.

Este procedimiento establece el id del estado siguiente que se tiene que ejecutar en el caso de que la transición sea verdadera.

procedure Set_State (this : in out Transition; id : State_Id);

4.23.5. FunciónGet_State.

Esta función devuelve el id delestado siguiente que se tiene que ejecutar en el caso de que la transición sea verdadera.

function Get_State (this : Transition) return State_Id;

4.23.6. Función Get_Name.

Esta función devuelve el nombre del objeto "*Transition ver 4.23*".

function Get_Name (this : Transition)return string;

5. Caso de Estudio.

Con el fin de demostrar todo lo expuesto anterior mente sobre el *Framework* y mejorar el conocimiento sobre el mismo, se ha realizado un caso de estudio, con el objetivo de introducir al lector de esta memoria en los procesos de instanciación y conexión de elementos, asignación de actividades a tareas, conexión de puertos y envió de mensajes.

Para este caso de estudio se va a seguir un guion:

- Descripción de la aplicación de prueba.
- Representación de las tres vistas de V3CM
    - Estructural.
    - De coordinación.
    - Algorítmica.
- Funcionalidad e interface de cada componente.
- Interacciones entre componentes (diagrama de secuencia).
- Aspectos de la implementación.
    - Clases instanciadas.
    - Subclases definidas.
    - Como se ensambla el código.
- Organización del código, instalación, ejecución y pruebas.
- Conclusiones.

5.1. Caso de Estudio-Envió de Mensajes entre componentes.

Este caso de estudio se ha diseñado con el objetivo de ser una introducción sencilla al *Framework* al completo. Representa dos componentes sencillos los cuales se están enviando mensajes de un valor numérico, y cuando lo reciben cada uno de ellos lo incrementa en uno o en cinco, según el tipo de mensajes y lo vuelve a enviar. Cuando uno de los componentes recibe el mensaje y después de incrementarlo su valor comprueba si es mayor de 1000, si es así lo pasa a cero y lo vuelve a enviar. Este proceso se esta repitiendo infinitamente.

La estructura de los dos componentes son prácticamente iguales, con la diferencia que una de ellos tiene dos estados iniciales que son los que lanzan las ejecuciones de envió de mensajes entre los dos componentes.La arquitectura de los componentes se podrá observar en los diagramas de estados.

5.1.1. Vista V3CM

La motivación de representar las tres vistas V3CM es clara, se entiende que gracias a estas tres representaciones, se puede comprender en gran medida las funcionalidades que ofrece el caso de estudio, la relación entre los componentes y el comportamiento de la aplicación.

5.1.2. Vista estructural.

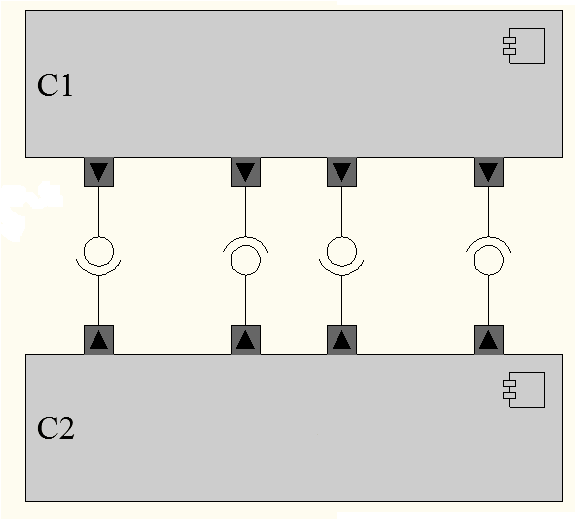

Ilustración 25: Vista estructural caso de estudio.

Dada la sencillez de la aplicación solo es necesaria modelar dos componentes y seis puertos, cuatro de entrada y otros cuatro de salida. Uno de los componentes los nombramos C1 (Componente 1) y el otro C2 (Componente 2).
Se interconecta los dos componentes mediante conexiones de puertos, de tal forma que las regiones del componente 1 le pueda enviar información a las regiones del componente 2 y la operación contraria, que las regiones componente 2 le pueda enviar información a lasregiones del componente 1.

5.1.3. Vista de coordinación.

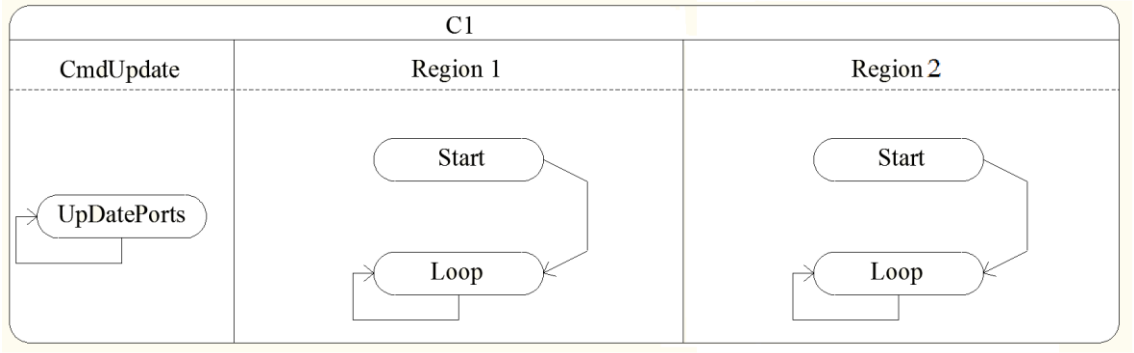

Ilustración 26: Vista coordinación caso de estudio Componente C1.

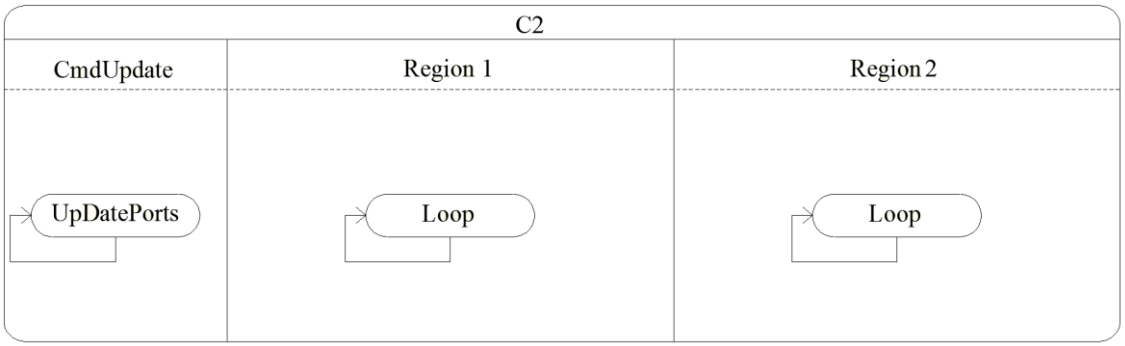

Ilustración 27: Vista coordinación caso de estudio Componente C2.

Aquí se representa las regiones que componen cada componente y las máquinas de estados asociadas a cada una de ellas. Como se puede observar ambos componentes tienen una estructura muy similar, contienen tres regiones ortogonales, una de ellas es encargada de la actualización de los puertos, y las otras dos de las actividades propias de cada componente.

La región encargada de la actualización de los puertos, la nombramos como "CmdUpdate", se encuentra continuamente en el mismo estado, y ejecutando la misma actividad asociada a ella.
La "Región 1" del componente 1 "C1", contiene 2 actividades una de ellas es la encargada de iniciar el proceso de envió de datos "Start", la otra actividad "Loop", es la encargada de leer los datos recibidos dirigidos a ella por el puerto de entrada, actuar sobre ellos e enviar estos datos al puerto de salida. La otra región "Región 2"es exactamente igual que la "Región 1", la única diferencia es que incrementa los datos de cinco en cinco.

La "Region 1" y "Region 2"del componente 2 "C2", contienen una sola actividad "Loop", esta actividad tiene la misma funcionalidad que las actividades "Loop" del componente 1 explicada en el apartado anterior.

5.1.4. Vista algorítmica.

Como su propio nombre indica la vista algorítmica describe los algoritmos ejecutados por cada actividad de cada estado. Este caso de estudio tiene seis estados actividades funcionales iguales llamado “Loop” y “Start”, a continuación se presentan los diagramas con la representación algorítmica de cada uno de ellos.

El algoritmo que representa el estado de actualización de puertos “CmdUpDate”, no se describe pues ya se describió en el apartado 4.17.

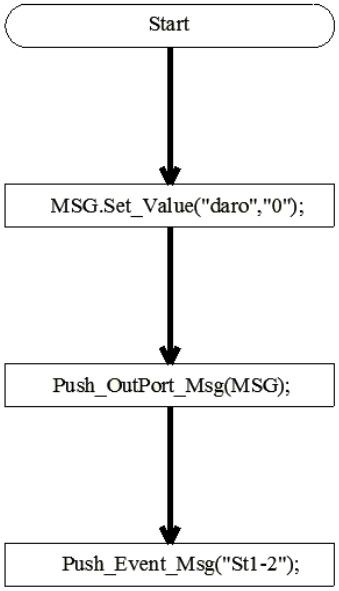

Ilustración 28: Vista algorítmica estado Start del caso de estudio.

El algoritmo es muy sencillo, esta actividad solo se ejecuta una vez, inicializa el valor a enviar a “0”, envía este valor al puerto de salida y por ultimo genera el mensaje para producir el cambio de estado.

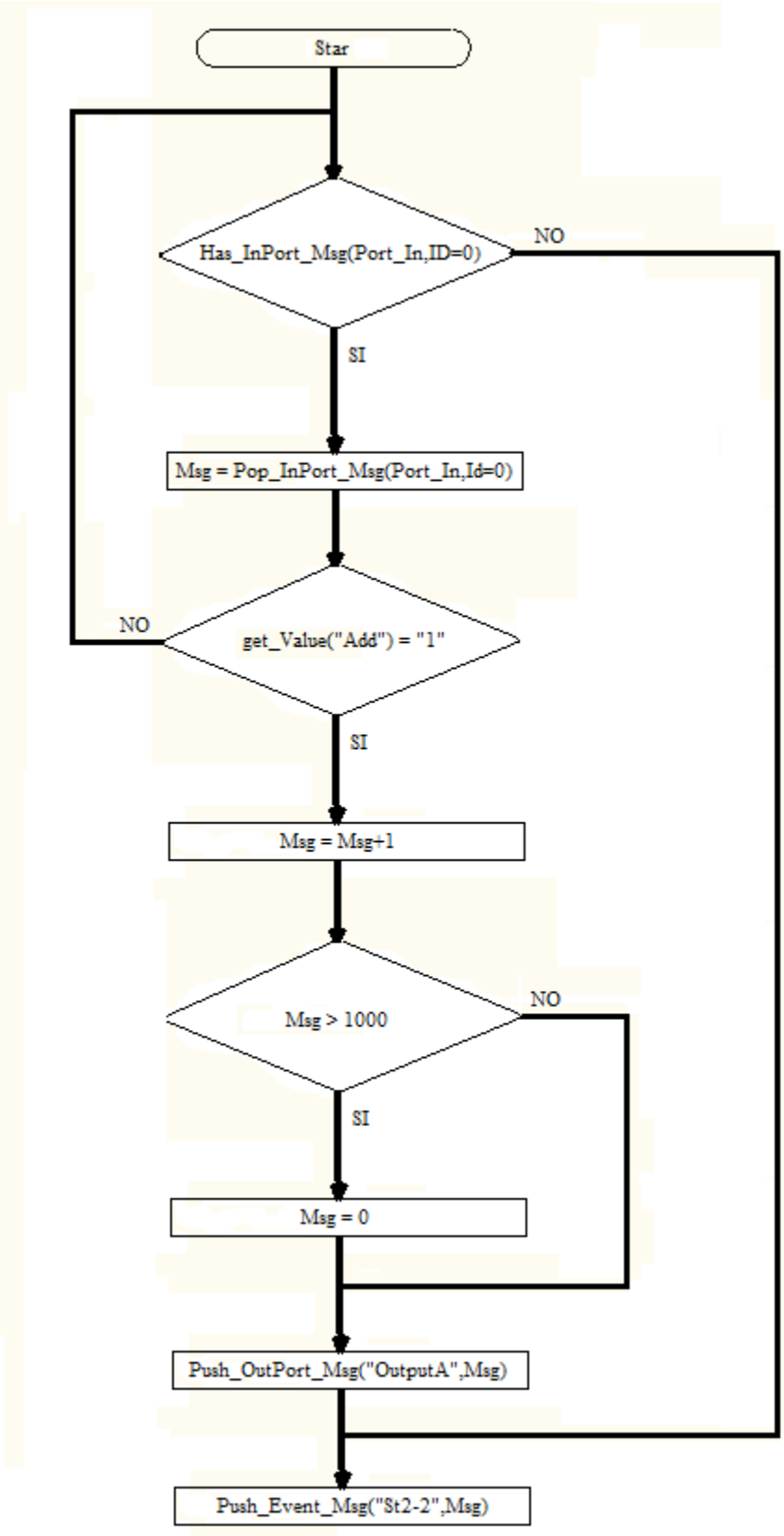

Ilustración 29: Vista algorítmica estado loop del caso de estudio.

Este algoritmo implementa los comportamientos de las dos actividades de los estados que se ejecutan en los dos componentes "Loop". Evalúa si se ha

recibido algún mensaje en el puerto de entrada destinados a ellos, en el caso de ser cierto lee dicho dato lo incrementa en uno o en cinco depende del estado, evalúa si ese dato es mayor de 1000 en caso de ser cierto, inicializa el valor a "0", envía el dato al puerto de salida y por ultimo genera el mensaje para producir el cambio de estado. En este caso el siguiente estado será este mismo estado.

5.1.5. Funcionalidad e interface de cada componente.

En el caso de estudio cada componente se comporta prácticamente de la misma forma, el componente "C1" inicia la transmisión de mensajes y una vez iniciada la transmisión se dedica a leer y renviar mensajes, en el caso del componente "C2" se dedica a leer y renviar mensajes.

C1:

- Crea dos mensajes del tipo *Minfr_Msg4.2* con un valor inicial 0 e inicia el proceso de envió de mensajes entre componente.
- Lee los mensajes que le envía el componente "C2" los modifica y los envía al otro componente.

C2:

- Lee los mensajes que le envía el componente "C1" los modifica y los envía al otro componente.

5.1.6. Interrelación entre componentes.

Esta interacción entre componentes es muy simple como se puede apreciar en la figura 30, Es un intercambio perdurable en el tiempo de mensajes.

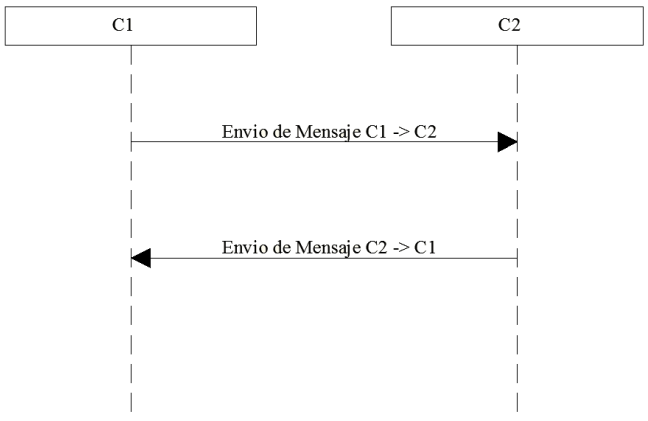

Ilustración 30: Secuencia de iteración entre componentes.

5.1.7. Aspectos de la implementación.

En esta sección se trata de definir la arquitectura de clases, que define la arquitectura de cada componente del caso de estudio.

5.1.7.1. Clases instanciadas y subclases definidas.

Se define las características y arquitectura de cada componente definiendo y describiendo los objetos creados para su definición.

5.1.7.1.1. Componente C1.

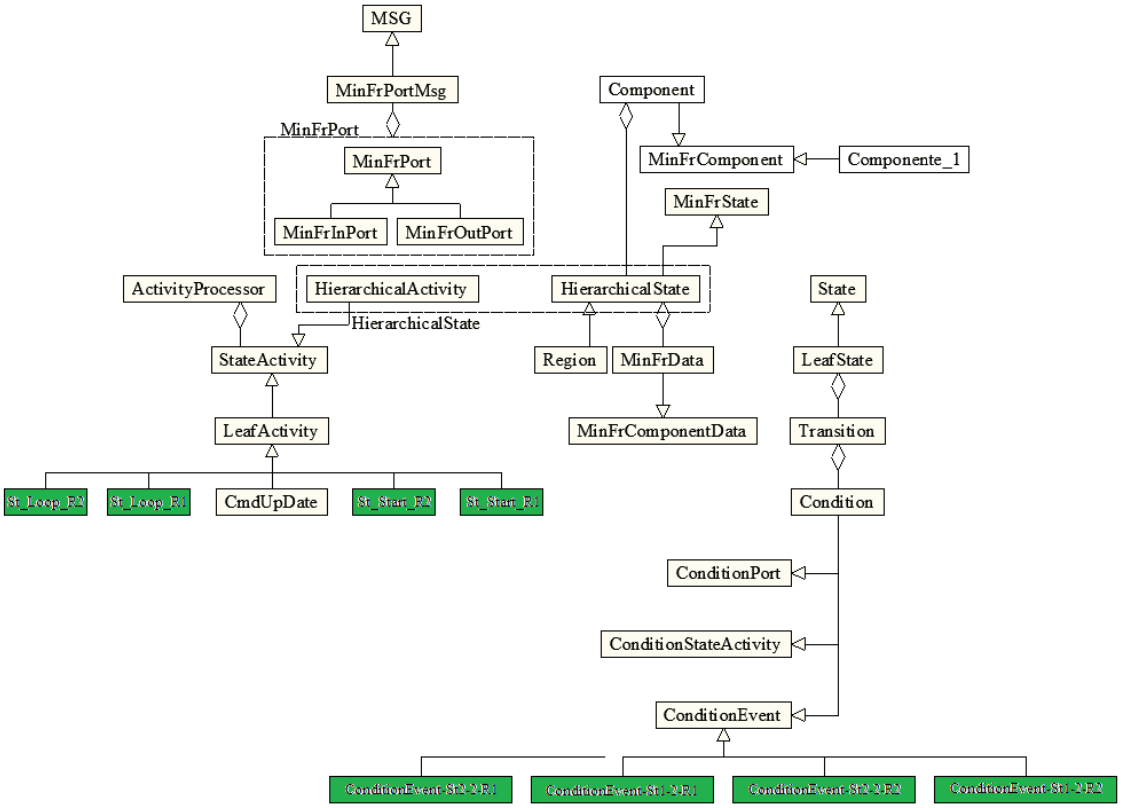

Ilustración 31: Diagrama de clases Componente C1.

Las clases y subclases que están de color verde son las más importantes que define el comportamiento del componente, en los siguientes párrafos se describen la funcionalidad de cada una de ellas.

- *ConditionEvent-St1-2-R1*.

Modela la condición para pasar del estado *St_Start_R1* al estado *St_Loop_R1*. Implementa el objeto ConditionEvent, Este objeto solo tiene dos métodos, "*Evaluate_Condition*" y "*Get_Name*". Se define como un objeto interno al componente, dado que solo se usara dentro de él.

Método	Descripción del método
Evaluate_Condition()	Evalúa la condición para la transición del estado, devuelve un valor falso o verdadero indicando si se puede realizar dicha transición.
Get_Name()	Devuelve el nombre del objeto, se utiliza para la depuración del código

- *ConditionEvent-St2-2-R1*.

Modela la condición para pasar del estado *St_Loop_R1* al mismo estado *St_Loop_R1*. Implementa el objeto ConditionEvent, Este objeto solo tiene dos métodos, "*Evaluate_Condition*" y "*Get_Name*". Se define como un objeto interno al componente, dado que solo se usara dentro de él.

Método	Descripción del método
Evaluate_Condition()	Evalúa la condición para la transición del estado, devuelve un valor falso o verdadero indicando si se puede realizar dicha transición.
Get_Name()	Devuelve el nombre del objeto, se utiliza para la depuración del código

- *ConditionEvent-St1-2-R2*.

Modela la condición para pasar del estado *St_Start_R2* al estado *St_Loop_R2*. Implementa el objeto ConditionEvent, Este objeto solo tiene dos métodos, "*Evaluate_Condition*" y "*Get_Name*". Se define como un objeto interno al componente, dado que solo se usara dentro de él.

Método	Descripción del método
Evaluate_Condition()	Evalúa la condición para la transición del estado, devuelve un valor falso o verdadero indicando si se puede realizar dicha transición.
Get_Name()	Devuelve el nombre del objeto, se utiliza para la depuración del código

- *ConditionEvent-St2-2-R2*.

Modela la condición para pasar del estado *St_Loop_R2* al mismo estado *St_Loop_R2*. Implementa el objeto ConditionEvent, Este objeto solo tiene dos métodos, "*Evaluate_Condition*" y "*Get_Name*". Se

define como un objeto interno al componente, dado que solo se usara dentro de él.

Método	**Descripción del método**
Evaluate_Condition()	Evalúa la condición para la transición del estado, devuelve un valor falso o verdadero indicando si se puede realizar dicha transición.
Get_Name()	Devuelve el nombre del objeto, se utiliza para la depuración del código

- St_Start_R1.

Modela la actividad para el estado*St_Start_R1*. Se crea un objeto que hereda del objeto "*Leaf_Activity4.16*". Este objeto solo tiene un método, "Execute_Tick".

Método	**Descripción del método**
Execute_Tick()	Ejecuta la Actividad del estado iniciando el proceso de envió de mensajes entre los componentes.

- St_Loop_R1.

Modela la actividad para el estado *St_Loop_R1*. Se crea un objeto que hereda del objeto "*Leaf_Activity4.16*". Este objeto solo tiene un método, "Execute_Tick".

Método	**Descripción del método**
Execute_Tick ()	Ejecuta la Actividad del estado, leyendo la entrada del componente y si recibe algún dato lo incrementa en uno, en el caso que el valor supere 1000, inicializa el dato a 0, y por ultimo envía el dato al puerto de salida de este componente.

- St_Start_R2.

Modela la actividad para el estado *St_Start_R2*. Se crea un objeto que hereda del objeto "*Leaf_Activity4.16*". Este objeto solo tiene un método, "Execute_Tick".

Método	Descripción del método
Execute_Tick ()	Ejecuta la Actividad del estado iniciando el proceso de envió de mensajes entre los componentes.

- St_Loop_R2.

Modela la actividad para el estado *St_Loop_R2*. Se crea un objeto que hereda del objeto "*Leaf_Activity4.16*". Este objeto solo tiene un método, "Execute_Tick".

Método	Descripción del método
Execute_Tick ()	Ejecuta la Actividad del estado, leyendo la entrada del componente y si recibe algún dato lo incrementa en cinco, en el caso que el valor supere 1000, inicializa el dato a 0, y por ultimo envía el dato al puerto de salida de este componente.

Clases y subclases instanciadas

- Relativas a la máquinas de estados.

o Región: Se define dos región que son las que contiene la máquina de estados.

o LeafState: Se define cuatro estados hoja, dos para modelar los estados*St_Start_R1, St_Start_R2* y otros dos para modelar los estados*St_Loop_R1, St_Loop_R2.*

- Relativas a las transiciones.

o Transition: Se instancian cuatro transiciones .una para ir del estado *St_Start_R1*al estado *St_Loop_R1*, otra para ir del estado *St_Loop_R1* e ir otra vez al mismo estado, otra para ir del estado *St_Start_R2*al estado *St_Loop_R2*y por ultimo otra para ir del estado *St_Loop_R2* e ir otra vez al mismo estado.

o Condition_Event: Se crean cuatro Objetos que son los que evalúan si se ha producido el cambio de estado mencionados en el apartado anterior.

- Relativas a las actividades.

o *Leaf_Activity4.16*: Se instancia las cuatro implementaciones de esta clase *St_Start_R1*,*St_Loop_R1*,*St_Start_R2* y*St_Loop_R2*.

- Relativas a las comunicaciones.

o Component: Se crea este objeto que define la arquitectura del componente, indicando el numero de regiones, numero y tipo de puertos que tiene el componerte y pizarra de dado donde se gestiona y almacenan la cola de mensajes.
o MinFr_Port: Se instancias los puertos para modelar las comunicaciones. En este componente habrá dos puertos uno de entrada y otro de salida.

o *MinFr_Data4.7*: Se crea la pizarra común al componente para almacenar y gestionar la cola de mensajes.

5.1.7.1.2. Componente C2.

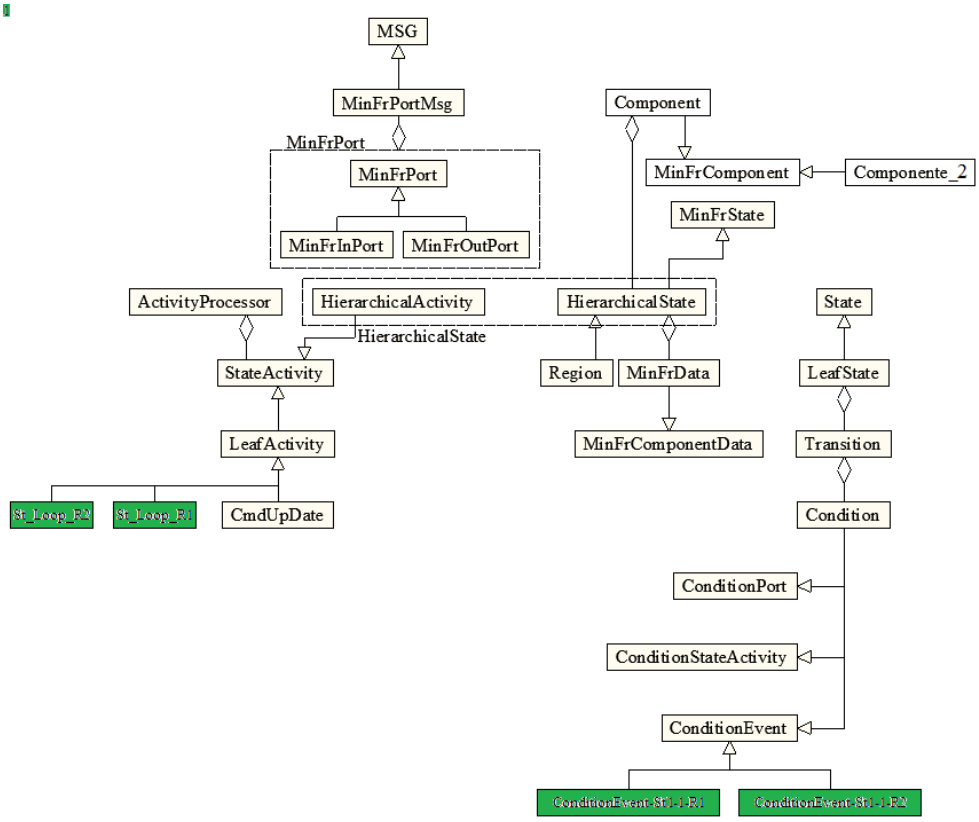

Ilustración 32: Diagrama de clases Componente C2.

Las clases y subclases que están de color verde son las más importantes que define el comportamiento del componente, en los siguientes párrafos se describen la funcionalidad de cada una de ellas.

- *ConditionEvent-St1-1-R1*.

Modela la condición para pasar del estado *St_Loop_R1* al mismo estado *St_Loop_R1*. Implementa el objeto ConditionEvent, Este objeto solo tiene dos métodos, "*Evaluate_Condition*" y "*Get_Name*". Se

define como un objeto interno al componente, dado que solo se usara dentro de él.

Método	**Descripción del método**
Evaluate_Condition()	Evalúa la condición para la transición del estado, devuelve un valor falso o verdadero indicando si se puede realizar dicha transición.
Get_Name()	Devuelve el nombre del objeto, se utiliza para la depuración del código

- *ConditionEvent-St1-1-R2*.

Modela la condición para pasar del estado *St_Loop_R2* al mismo estado *St_Loop_R2*. Implementa el objeto ConditionEvent, Este objeto solo tiene dos métodos, "*Evaluate_Condition*" y "*Get_Name*". Se define como un objeto interno al componente, dado que solo se usara dentro de él.

Método	**Descripción del método**
Evaluate_Condition()	Evalúa la condición para la transición del estado, devuelve un valor falso o verdadero indicando si se puede realizar dicha transición.
Get_Name()	Devuelve el nombre del objeto, se utiliza para la depuración del código

- St_Loop_R1.

Modela la actividad para el estado *St_Loop_R1*. Se crea un objeto que hereda del objeto "*Leaf_Activity4.16*". Este objeto solo tiene un método, "Execute_Tick".

Método	**Descripción del método**
Execute_Tick ()	Ejecuta la Actividad del estado, leyendo la entrada del componente y si recibe algún dato lo incrementa en uno, en el caso que el valor supere 1000, inicializa el dato a 0, y por ultimo envía el dato al puerto de salida de este componente.

- St_Loop_R2.

Modela la actividad para el estado *St_Loop_R2*. Se crea un objeto que hereda del objeto "*Leaf_Activity4.16*". Este objeto solo tiene un método, "Execute_Tick".

Método	**Descripción del método**
Execute_Tick ()	Ejecuta la Actividad del estado, leyendo la entrada del componente y si recibe algún dato lo incrementa en cinco, en el caso que el valor supere 1000, inicializa el dato a 0, y por ultimo envía el dato al puerto de salida de este componente.

Clases y subclases instanciadas

- Relativas a la máquinas de estados.

o Región: Se define dos regiones que son las que contiene las máquinas de estados.

o LeafState: Se define dos estados hoja, para modelar el estado *St_Loop_R1* y *St_Loop_R2*.

- Relativas a las transiciones.

o Transition: Se instancian dostransiciones, una para ir del estado *St_Loop_R1* e ir otra vez al mismo estado y otra para ir del estado *St_Loop_R2* e ir otra vez al mismo estado.

o Condition_Event: Se crean dos Objetos que son los que evalúan si se ha producido el cambio de estado mencionados en el apartado anterior.

- Relativas a las actividades.

o *Leaf_Activity4.16*: Se instancia las dos implementaciones de esta clase *St_Loop_R1* y *St_Loop_R2*.

- Relativas a las comunicaciones.

o Component: Se crea este objeto que define la arquitectura del componente, indicando el numero de regiones, numero y tipo de puertos que tiene el componerte y pizarra de dado donde se gestiona y almacenan la cola de mensajes.

o MinFr_Port: Se instancias los puertos para modelar las comunicaciones. En este componente habrá dos puertos uno de entrada y otro de salida.

o *MinFr_Data4.7*: Se crea la pizarra común al componente para almacenar y gestionar la cola de mensajes.

5.1.8. Ensamblado de código.

En este apartado se intenta explicara como se ensamblan, conectan y se asignan las tareas a las actividades de cada uno de los componentes.

En primer lugar se instancias las pizarras de datos que van a utilizar cada componte para este caso de estudio *Data1* y *Data2*, el siguiente paso es crear y definir las colas de mensajes de entrada y salida con los procedimientos "Subscribe_To_InPort_Msg" para las colas de entrada y "Create_OutPort_Buffer" para las colas de salida, se definen cuatro colas de salida dos para cada componente en las que las máquinas de estados de las regiones almacenen los mensajes de salida, se define cuatro colas de entrada para encolar los mensajes destinados a las regiones Region1-C1, Region2-C1,Region1-C2 y Region2-C2.Se definen todos los puertos que se van a utilizar en la aplicación, para este caso de estudio se define dos de salida y otros dos de entrada para cada uno de los componentes, a la hora de crear los puertos de entrada se indica cual es la dirección de la pizarra de datos en la cual tiene que leer los mensajes. Una vez definido los puertos se interconectan invocando el método "*Set_Conjugate*" de los puertos de salida indicando cual es el puerto de entrada al cual se conecta.

Instanciamos cada componente "*Component*", definiendo las características que componen el componerte.

- Al crear el objeto del componente se indica el nombre de dicho componerte y la dirección de la pizarra de datos que almacena y gestiona la cola de mensajes del componente.

- Numero de regiones, para este caso de estudio cada componente esta compuesto por tres Regiones, se agregan con el procedimiento *Add_Region.*

- Numero y tipo de puertos que esta compuesto el componente, para este caso de estudio los dos componentes tienen un puerto de entrada y otro de salida, se agregan con el procedimiento *Add_Port*.

Para esta aplicación se ha decidido realizar diferentes casos de estudio a la hora de crear actividades y la asignación de tareas a las mismas, en los siguientes apartados se explica más detalladamente cuántas se han creado y cómo se han asignado.

- Un solo proceso "*Activity_Processor*", se asignan todas las actividades a este, mediante la invocación del procedimiento "*Add_Activity*".

- Dos procesos "*Activity_Processor*", a una se le asigna las actividades del componente 1 y al otro se le asignan las actividades del componente 2, con la invocación del procedimiento "*Add_Activity*".

- Cuatro proceso "*Activity_Processor*", en este caso las actividades se asignan de la siguiente forma.

    - La actividad de la región 1 del componente 1 que es la encargada de actualizar el puerto de salida de dicho componente.

    - Las actividades de las regiones 2 y 3 del componente 1, que son las que tienen las máquinas de estados que se ejecutan en dicho componente.

    - La actividad de la región 1 del componente 2 que es la encargada de actualizar el puerto de salida de dicho componente.

    - Las actividades de las regiones 2 y 3 del componente 2, que son las que tienen las máquinas de estados que se ejecutan en dicho componente.

- Seis procesos "*Activity_Processor*", en este caso las actividades se asignan de la siguiente forma.

    - La actividad de la región 1 del componente 1 que es la encargada de actualizar el puerto de salida de dicho componente.

    - La actividad de la región2 del componente 1, que es una de las que tiene máquinas de estados que se ejecutan en dicho componente.

    - La actividad de la región 3 del componente 1, que es una de las que tiene máquinas de estados que se ejecutan en dicho componente.

    - La actividad de la región 1 del componente 2 que es la encargada de actualizar el puerto de salida de dicho componente.

    - La actividad de la región 2 del componente 2, que es una de las que tiene máquinas de estados que se ejecutan en dicho componente.

    - La actividad de la región 3 del componente 2, que es una de las que tiene máquinas de estados que se ejecutan en dicho componente.

Para acabar, se arranca los procesadores de comandos invocando método "*Start*" sobre los "*Activity_Processor*".

5.1.9. Respuesta temporal de la aplicación

En este apartado se intenta representar de forma gráfica la respuesta temporal de la aplicación.

Como se dijo en el apartado anterior, para esta aplicación se ha decidido realizar diferentes casos de estudio a la hora de crear actividades y la asignación de tareas a las mismas, en los siguientes apartados se representan las gráficas de las respuestas temporales de cada una de las actividades en función de numero de tareas.

5.1.9.1. Un Proceso Activity_Processor.

Al tener solo un proceso "*Activity_Processor*", solo tenemos una respuesta temporal de la tarea que se representa en la siguiente figura.

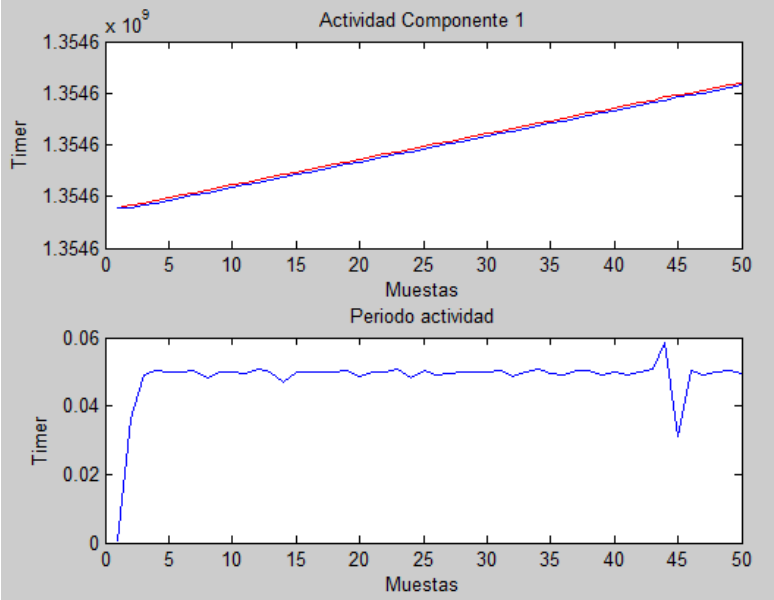

Ilustración 33: Respuesta Temporal Actividad 1, primer caso de estudio.

La gráfica superior representa la respuesta temporal de la tarea 1, la línea azul representa el tiempo en el cual comenzó la tarea y la roja cuando finalizo. La gráfica inferior es la diferencia entren el tiempo de finalización de la tarea y el de inicia, esta gráfica representa el periodo de la tarea, que como se puede observar se mantiene prácticamente constante y el periodo coincide con el máximo común divisor de los seis periodos de las actividades asignadas a la tarea, que para este caso de estudio es 50ms.

5.1.9.2. Dos Procesos Activity_Processor.

En las siguientes gráficas se representa la respuesta temporal de los dos procesos "*Activity_Processor*", de este caso de estudio.

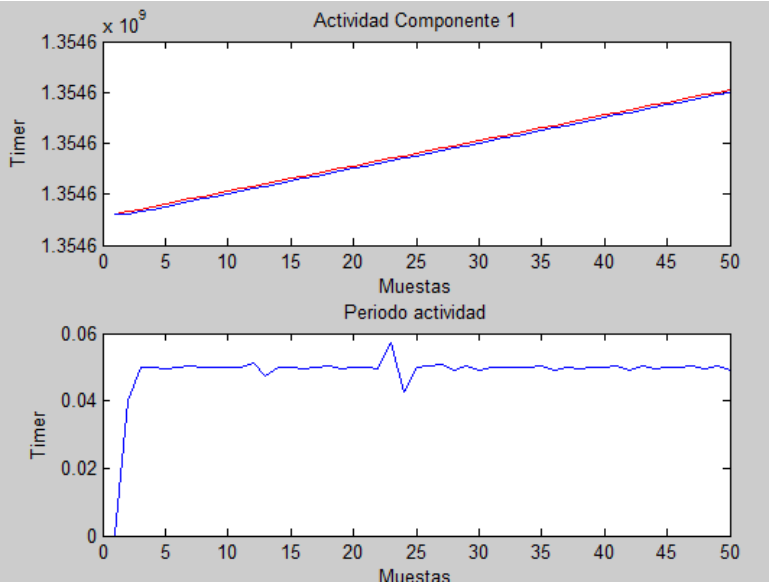

Ilustración 34: Respuesta Temporal Actividad 1, segundo caso de estudio.

La gráfica superior representa la respuesta temporal de la tarea 1, la línea azul representa el tiempo en el cual comenzó la tarea y la roja cuando finalizo. La gráfica inferior es la diferencia entren el tiempo de finalización de la tarea y el de inicia, esta gráfica representa el periodo de la tarea, que como se puede observar se mantiene prácticamente constante y el periodo coincide con el máximo común divisor de los tres periodos de las actividades asignadas a la tarea, que para este caso de estudio es 50ms.

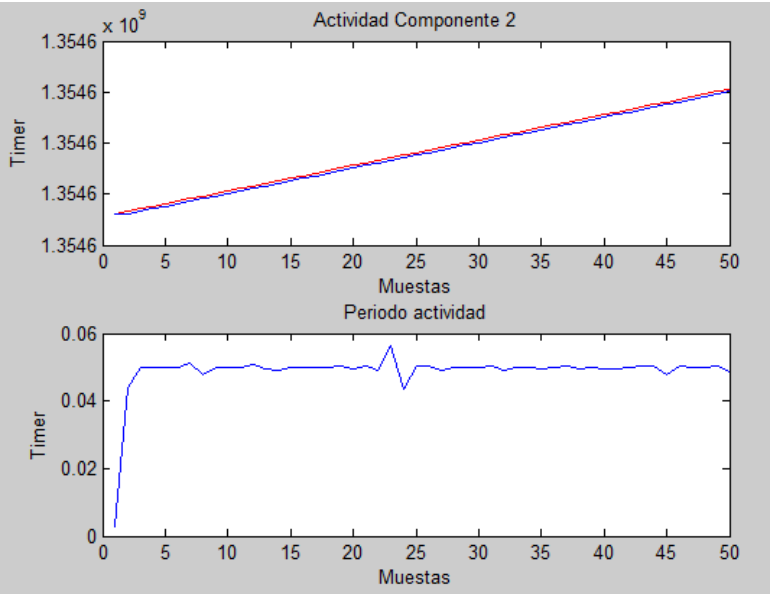

Ilustración 35: Respuesta Temporal Actividad 2, segundo caso de estudio.

La gráfica superior representa la respuesta temporal de la tarea 2, la línea azul representa el tiempo en el cual comenzó la tarea y la roja cuando finalizo. La gráfica inferior es la diferencia entren el tiempo de finalización de la tarea y el de inicia, esta gráfica representa el periodo de la tarea, que como se puede observar se mantiene

prácticamente constante y el periodo coincide con el máximo común divisor de los tres periodos de las actividades asignadas a la tarea, que para este caso de estudio es 50ms.

5.1.9.3. Cuatro Procesos Activity_Processo.

En las siguientes gráficas se representa la respuesta temporal de los cuatro procesos "*Activity_Processor*", de este caso de estudio.

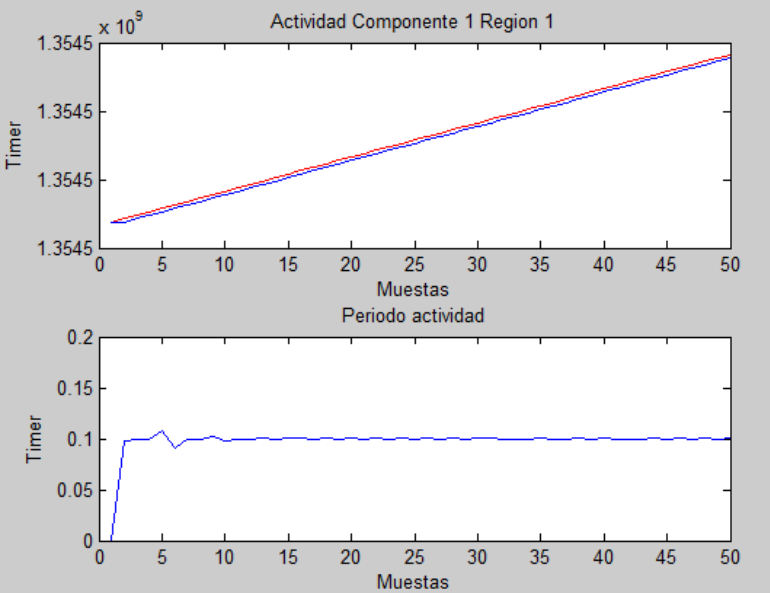

Ilustración 36: Respuesta Temporal Actividad 1, tercer caso de estudio.

La gráfica superior representa la respuesta temporal de la tarea 1, la línea azul representa el tiempo en el cual comenzó la tarea y la roja cuando finalizo. La gráfica inferior es la diferencia entren el tiempo de finalización de la tarea y el de inicia, esta gráfica representa el periodo de la tarea, que como se puede observar se mantiene prácticamente constante y el periodo coincide con el máximo común divisor de los periodos de las actividades asignadas a la tarea, que en este caso coincide con el periodo de la única actividad que se le ha asignado, que es de 100ms.

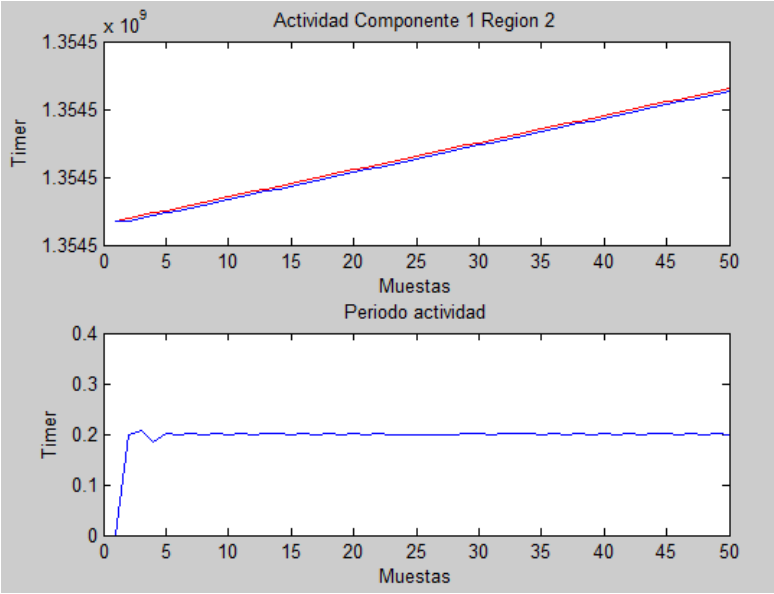

Ilustración 37: Respuesta Temporal Actividad 2, tercer caso de estudio.

La gráfica superior representa la respuesta temporal de la tarea 2, la línea azul representa el tiempo en el cual comenzó la tarea y la roja cuando finalizo. La gráfica inferior es la diferencia entren el tiempo de finalización de la tarea y el de inicia, esta gráfica representa el periodo de la tarea, que como se puede observar se mantiene prácticamente constante y el periodo coincide con el máximo común divisor de los dos periodos de las actividades asignadas a la tarea, que para este caso de estudio es 200ms.

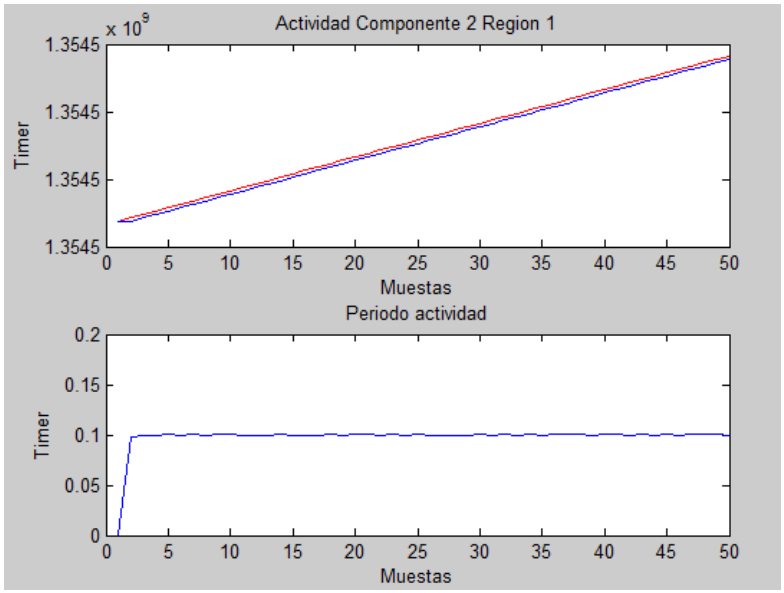

Ilustración 38: Respuesta Temporal Actividad 3, tercer caso de estudio.

La gráfica superior representa la respuesta temporal de la tarea 3, la línea azul representa el tiempo en el cual comenzó la tarea y la roja cuando finalizo. La gráfica inferior es la diferencia entren el tiempo de finalización de la tarea y el de inicia, esta gráfica representa el periodo de la tarea, que como se puede observar se mantiene

prácticamente constante y el periodo coincide con el máximo común divisor de los periodos de las actividades asignadas a la tarea, que en este caso coincide con el periodo de la única actividad que se le ha asignado, que es de 100ms.

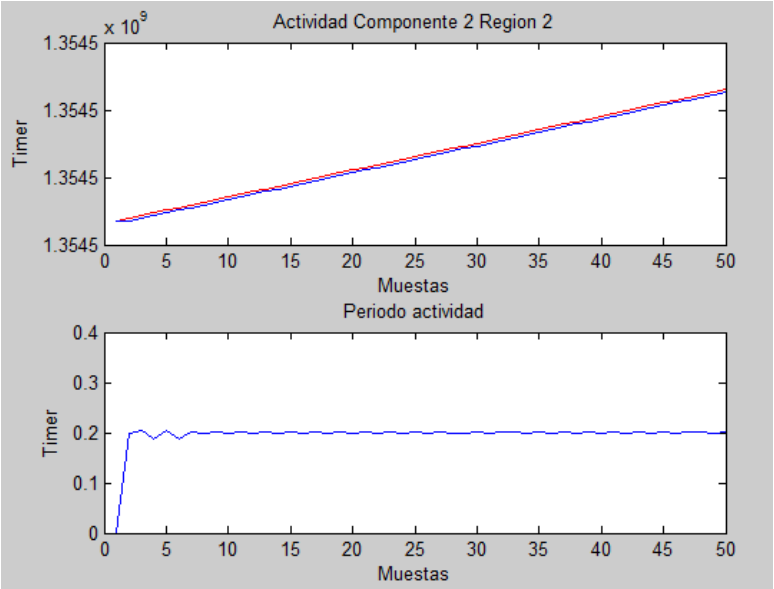

Ilustración 39: Respuesta Temporal Actividad 4, tercer caso de estudio.

La gráfica superior representa la respuesta temporal de la tarea 4, la línea azul representa el tiempo en el cual comenzó la tarea y la roja cuando finalizo. La gráfica inferior es la diferencia entren el tiempo de finalización de la tarea y el de inicia, esta gráfica representa el periodo de la tarea, que como se puede observar se mantiene prácticamente constante y el periodo coincide con el máximo común divisor de los dos periodos de las actividades asignadas a la tarea, que para este caso de estudio es 200ms.

5.1.9.4. Seis Procesos Activity_Processor.

En las siguientes gráficas se representa la respuesta temporal de los seis procesos "*Activity_Processor*", de este caso de estudio.

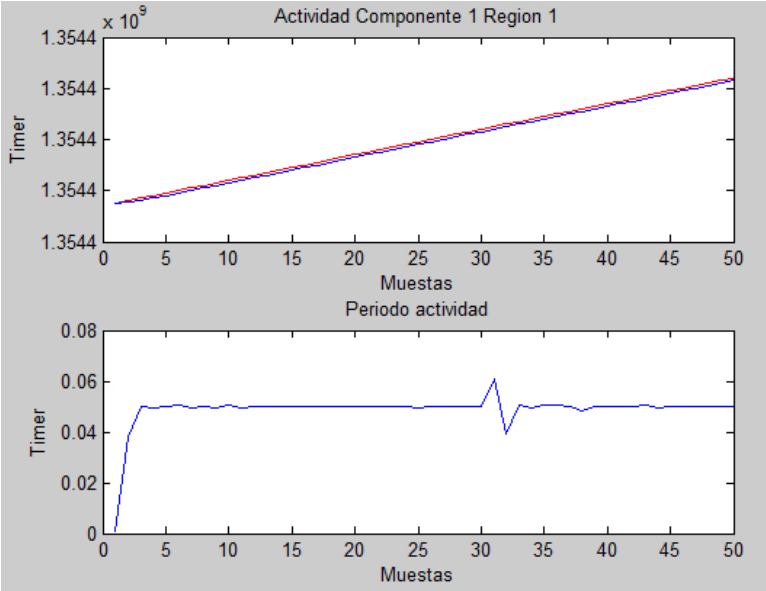

Ilustración 40: Respuesta Temporal Actividad 1, cuarto caso de estudio.

La gráfica superior representa la respuesta temporal de la tarea 1, la línea azul representa el tiempo en el cual comenzó la tarea y la roja cuando finalizo. La gráfica inferior es la diferencia entren el tiempo de finalización de la tarea y el de inicia, esta gráfica representa el periodo de la tarea, que como se puede observar se mantiene prácticamente constante y el periodo coincide con el máximo común divisor de los periodos de las actividades asignadas a la tarea, que en este caso coincide con el periodo de la única actividad que se le ha asignado, que es de 50ms.

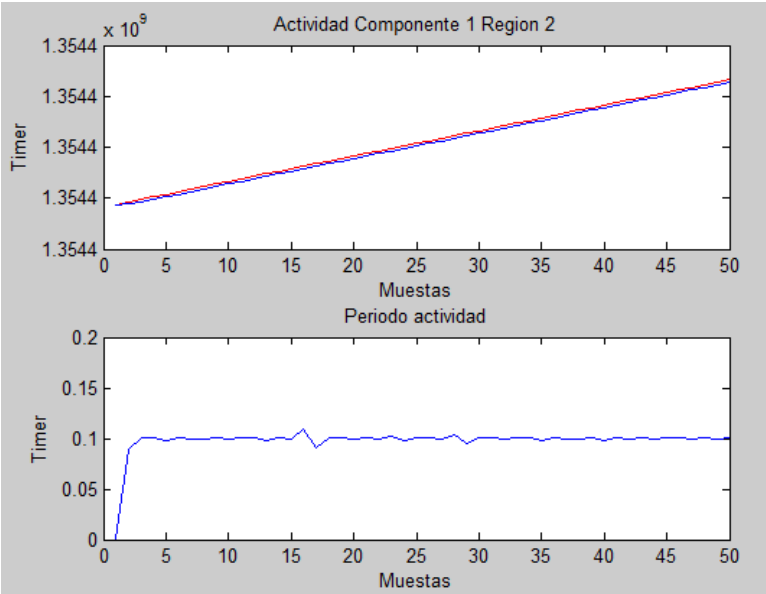

Ilustración 41: Respuesta Temporal Actividad 2, cuarto caso de estudio.

La gráfica superior representa la respuesta temporal de la tarea 2, la línea azul representa el tiempo en el cual comenzó la tarea y la roja cuando finalizo. La gráfica inferior es la diferencia entren el tiempo de finalización de la tarea y el de inicia, esta gráfica representa el

periodo de la tarea, que como se puede observar se mantiene prácticamente constante y el periodo coincide con el máximo común divisor de los periodos de las actividades asignadas a la tarea, que en este caso coincide con el periodo de la única actividad que se le ha asignado, que es de 100ms.

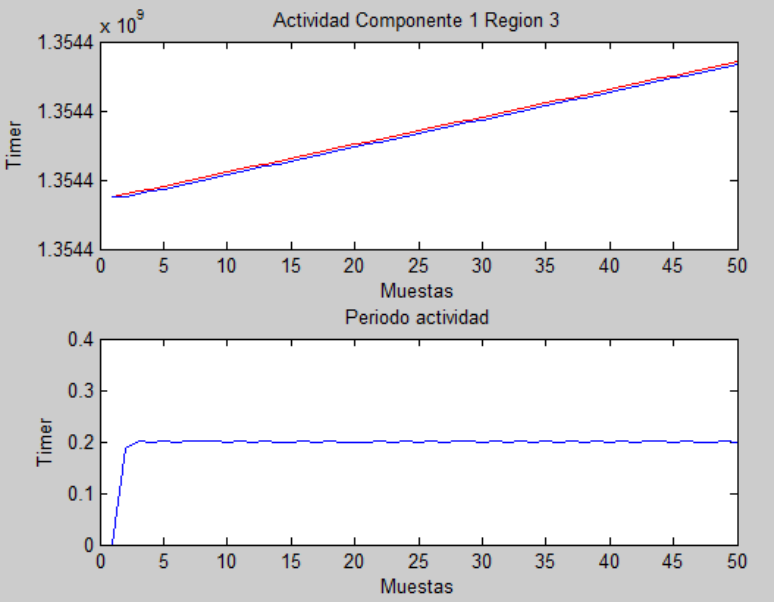

Ilustración 42: Respuesta Temporal Actividad 3, cuarto caso de estudio.

La gráfica superior representa la respuesta temporal de la tarea 3, la línea azul representa el tiempo en el cual comenzó la tarea y la roja cuando finalizo. La gráfica inferior es la diferencia entren el tiempo de finalización de la tarea y el de inicia, esta gráfica representa el periodo de la tarea, que como se puede observar se mantiene prácticamente constante y el periodo coincide con el máximo común divisor de los periodos de las actividades asignadas a la tarea, que en este caso coincide con el periodo de la única actividad que se le ha asignado, que es de 200ms.

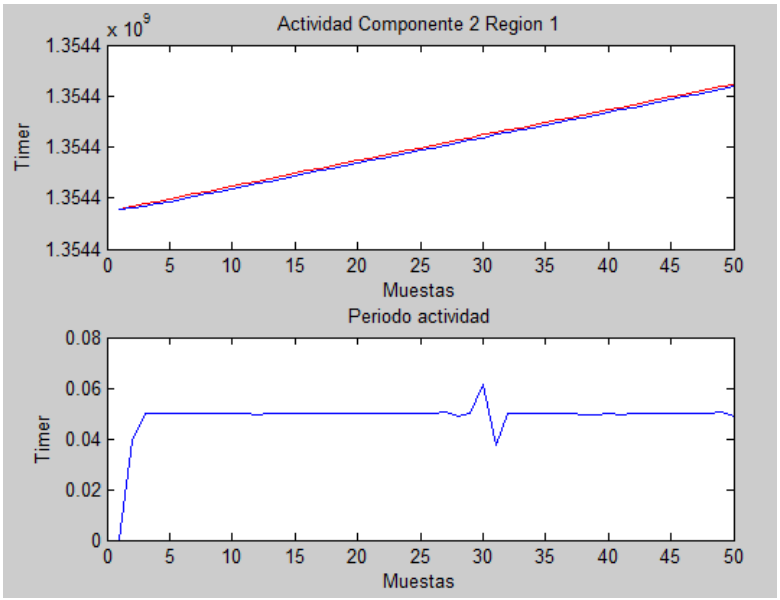

Ilustración 43: Respuesta Temporal Actividad 4, cuarto caso de estudio.

La gráfica superior representa la respuesta temporal de la tarea 4, la línea azul representa el tiempo en el cual comenzó la tarea y la roja cuando finalizo. La gráfica inferior es la diferencia entren el tiempo de finalización de la tarea y el de inicia, esta gráfica representa el periodo de la tarea, que como se puede observar se mantiene prácticamente constante y el periodo coincide con el máximo común divisor de los periodos de las actividades asignadas a la tarea, que en este caso coincide con el periodo de la única actividad que se le ha asignado, que es de 50ms.

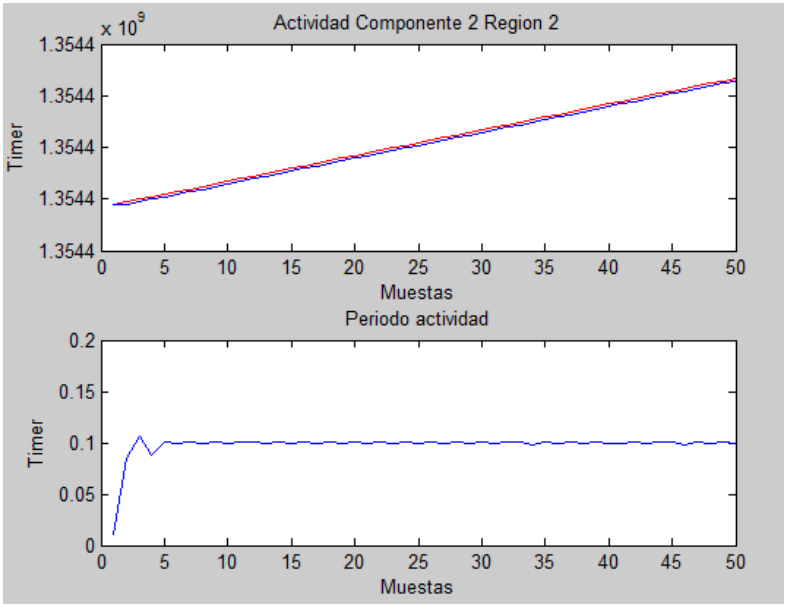

Ilustración 44: Respuesta Temporal Actividad 5, cuarto caso de estudio.

La gráfica superior representa la respuesta temporal de la tarea 5, la línea azul representa el tiempo en el cual comenzó la tarea y la roja cuando finalizó. La gráfica inferior es la diferencia entre el tiempo de

finalización de la tarea y el de inicia, esta gráfica representa el periodo de la tarea, que como se puede observar se mantiene prácticamente constante y el periodo coincide con el máximo común divisor de los periodos de las actividades asignadas a la tarea, que en este caso coincide con el periodo de la única actividad que se le ha asignado, que es de 100ms.

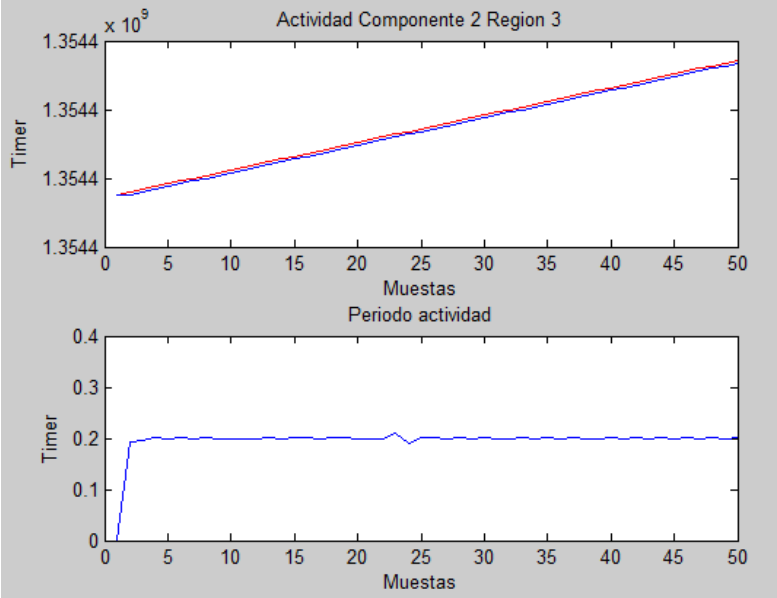

Ilustración 45: Respuesta Temporal Actividad 6, cuarto caso de estudio.

La gráfica superior representa la respuesta temporal de la tarea 6, la línea azul representa el tiempo en el cual comenzó la tarea y la roja cuando finalizo. La gráfica inferior es la diferencia entren el tiempo de finalización de la tarea y el de inicia, esta gráfica representa el periodo de la tarea, que como se puede observar se mantiene prácticamente constante y el periodo coincide con el máximo común divisor de los periodos de las actividades asignadas a la tarea, que en este caso coincide con el periodo de la única actividad que se le ha asignado, que es de 200ms.

5.1.10. Organización del código, instalación, ejecución y pruebas.

Organización del código usado en este caso de estudio.

- El código de Framework se encuentra en una carpeta llamada MinFr/src, que almacena el código del todo el Framework.

- El código relativo al caso de estudio esta localizado en el carpeta MinFr/src/Main, que contiene todos los archivosque definen y construyen todos los casos de estudio llamados:

    - "*Main_Caso_Estudio_1.adb*"
    - "*Main_Caso_Estudio_2.adb*"
    - "*Main_Caso_Estudio_3.adb*"
    - "*Main_Caso_Estudio_4.adb*"

En el siguiente diagrama se representa la organización del código

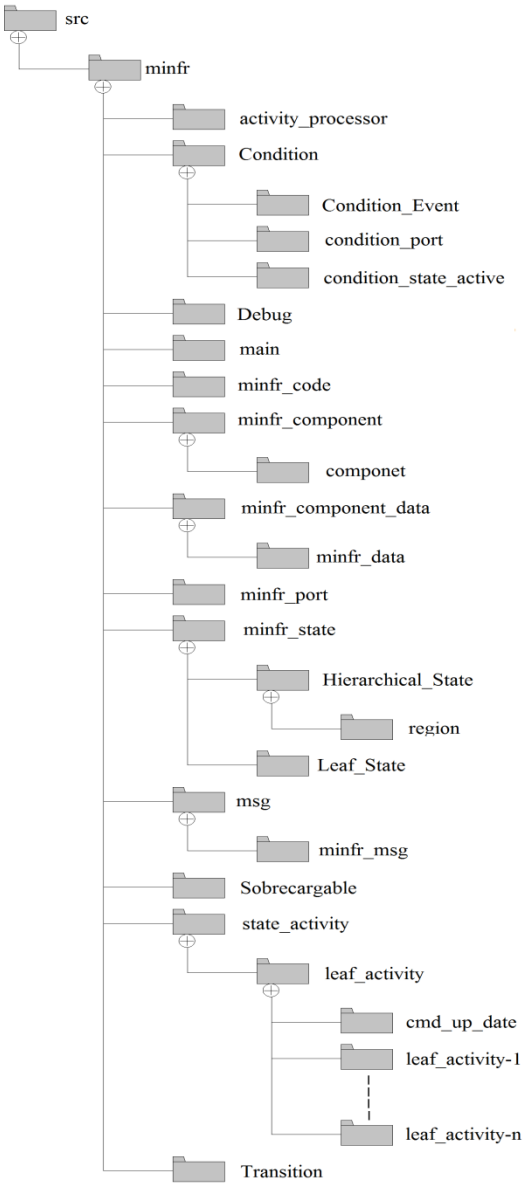

Ilustración 46: Organigrama del código.

Para la instalación y ejecución del código se enumeran los pasos necesarios a seguir:

1. Descargar la versión GNAT Programming Studio [http://libre.adacore.com/download/], la edición y compilación del código se ha realizado sobre la versión *GPS 5.1.1*.

2. Instala el programa *GPS 5.1.1*

3. Se ejecute el archivo "*minfr.gpr*" que carga el proyecto en el "GNAT Programming Studio".

4. Pulsamos la tecla F4 para compilara el código y pulsamos la tecla Run  para ejecutar la aplicación.

Como en este caso de estudio existen cuatro configuraciones distintas ha de explicarse como se ha de configurar "GNAT Programming Studio" para poderlas ejecutar.

1. Vamos a la barra de menú y seleccionamos Proyect→Edit Proyect Properties, seleccionamos la pestaña "*Main files*"

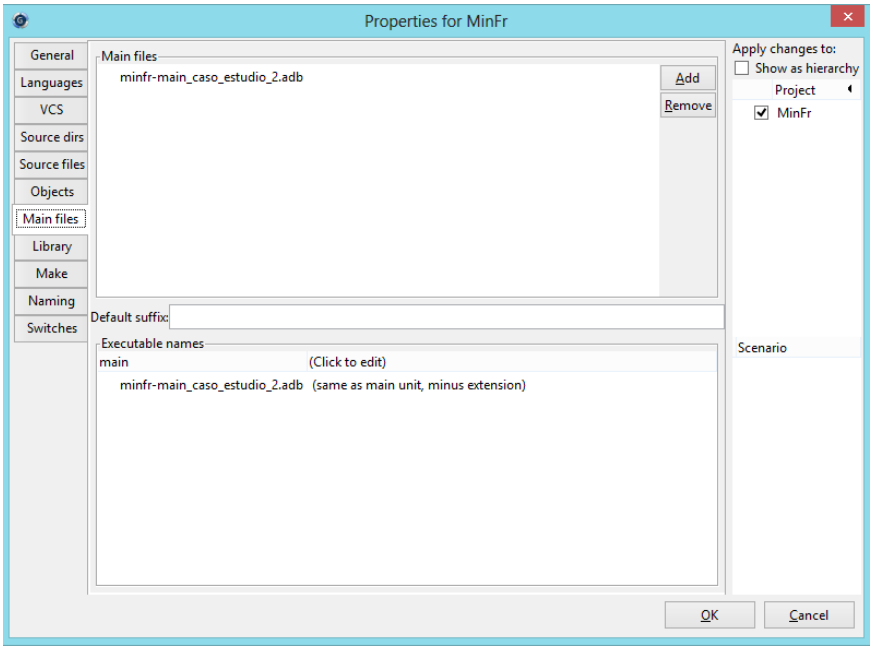

Ilustración 47: Propiedades del Proyecto.

2. Seleccionamos el texto del cuadro Main files y pulsamos a continuación el botón Remove, pulsamos el botón Add y se abrirá un menú para seleccionar el archivo que queremos que sea el principal para la ejecución.

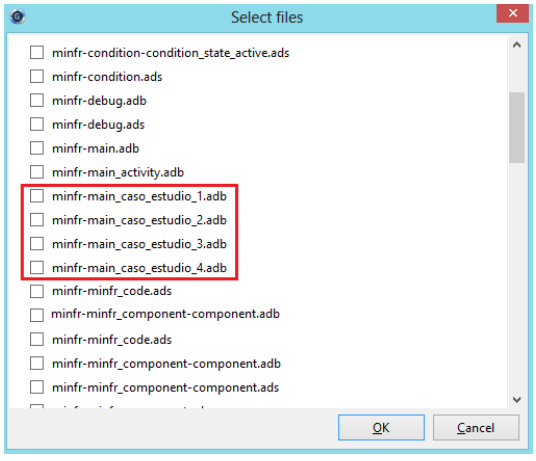

Ilustración 48: Selección de Archivos.

Los archivos que pueden ser seleccionados y definen los distintos casos de estudios son los que están rodeados por un rectángulo rojo en la figura anterior.

3. Pulsamos Ok en las dos ventanas anteriores y repetimos el punto 4 del apartado anterior para compilar y ejecutar la aplicación.

5.1.11. Conclusiones de este caso de estudio.

En primer lugar conseguimos comprobar la efectividad de este *Framework* y segundo explicamos el proceso necesario para instanciación de componentes con varias regiones con transiciones y la ejecución de actividades.

Se ha demostrado la flexibilidad del *Framework* a la hora de la asignación de tareas mediante los cuatro casos propuestos, demostrando el correcto funcionamiento de los distintos casos viendo la respuesta temporal de los mismos.

Podemos considerar por tanto, que lo que se pretendía demostrar y asegurar con el desarrollo de estos casos de estudio se ha logrado demostrar la flexibilidad y utilidad del *Framework* aquí presentado.

6. Conclusiones.

Podemos decir que se han conseguido cumplir con todos los objetivos planteados inicialmente, se ha obtenido, a niveles generales, buenos resultados en la implementación del Framework.

Se ha conseguido traducir el completo todo el Framework de C++ a Ada, manteniendo la integridad de Framework.

Se ha conseguido solucionar los problemas que aparecieron en versiones anteriores del Framework de C++ y Java. El uso excesivo de memoria y el exceso de uso de CPU.

Se ha demostrado la validez de esta integración con el estudio en profundidad de un caso de estudio, demostrando la flexibilidad de asignación de tareas. El Framework fue probado en este caso de estudio, y se analizo con detenimiento todos los resultados obtenidos por la ejecución del caso de estudio, en distintos modos. Obteniendo las gráficas de las respuestas temporales de cada uno de los modos, y así demostrando el buen funcionamiento del Framework

7. Bibliografía.

DT EXPLORE (2010). DT-EXPLORE-01.rev0 Enero 2010, Máquinas de estados y tareas en V3CMM. Un caso de estudio.

Proyecto fin de carrera (2010). Rubén Martínez Sandoval. Implementación en Java de un Framework para el desarrollo de aplicaciones con requisitos de tiempo real estricto.

Trabajo Fin de Máster(2010). Francisco Antonio Sánchez Ledesma.Implementación en C++ de un Framework de ejecución para el lenguaje de modelado V3CMM.

Guía de Programación. Bárbara Álvarez Torres, Diego Alonso Cáceres. Guía de referencia Básica ADA 2005

John Barnes. Programming in ADA 2005.

WED ADA.
http://www.adacore.com.
http://es.wikibooks.org/wiki/Programaci%C3%B3n_en_Ada/Ada_2005.
http://en.wikipedia.org/wiki/Ada_(programming_language).

DSIE WEB. http://www.dsie.upct.es