Comparison of Dynamic Software Updating Methods for IEC 61499 with Erlang

Ainara Matey Benito

TRABAJO FIN DE MÁSTER
Máster Habilitante de Ingeniería Industrial
Especialidad Automática y Electrónica

Tutores del Trabajo Fin de Máster:
Laurin Prenzel
Julien Provost
Technische Universität München
Manuel Ferre Perez
Universidad Politécnica de Madrid

Enero 2020
ACKNOWLEDGEMENTS

Having finished this project, I would like to thank the people who has helped me out through its completion and who made it possible.

First of all I would like to express my gratitude Laurin Prenzel, for being such a attentive advisor, for being always ready to help me with anything and for guiding me throughout the entire project. It was a pleasure working together.

Secondly, I would like to thank the entire Professorship of Safe Embedded Systems, and especially Julien Provost, whose courses and assistance made me grow my interest on the field and gave me the opportunity of finding a suitable project for me in their team.

I would also like to thank Manuel Ferre for his support and help as my supervisor from Madrid.

I would like to extend my gratitude to both the universities UPM and TUM. They gave me the golden opportunity to complete my M.Sc. degree in Munich and enrich my studies with different insights.

Last, but not least, I would like to thank my family and friends, who gave me the emotional support and encouragement I needed, not only during the course of this project, but in every aspect of my life.

Thank you for everything.

Ainara
ABSTRACT

Having to update the Control Software in a system is inevitable, due to bug fixes, upgrades, security patches, etc. Dynamic Software Updating (DSU) methods provide with ways of carrying out such reconfigurations reducing the downtime to zero, this way saving a lot in time and costs, and growing more and more crucial due to constant technology and market variations.

This project aims to explore the possibilities the programming language Erlang offers for DSU, since it provides with Hot Code Loading methods, as well as with functions that alter processes on runtime.

All the project is performed based on the system architecture suggested by the Industrial Automation standard IEC 61499, which although does not have a strong acceptance in industry yet, it proves a powerful tool for Control Automation.

For this purpose, a set of Reconfiguration Services are implemented in Erlang according to the IEC 61499 and based on the compiler FBBeam. With those Services, different Reconfiguration Applications are designed and run in various case studies, in order to test the advantages they bring and the limitations to overcome.

Finally, a comparative assessment is performed between this solution and the current DSU solution provided by FBBeam, this way trying to bring together the advantages of both methods.

**Key words:** Distributed Automation Systems, Erlang Runtime System, Dynamic Reconfiguration, Flexible and reconfigurable manufacturing systems.

**UNESCO Codes:** 120305, 331101, 331102
RESUMEN

Motivación y objetivos

El mundo está cambiando más rápido que nunca y la tecnología avanza a un ritmo frenético. Esto provoca a su vez un cambio en el mercado, que demanda más rapidez, calidad y más personalización de sus productos.

Es por esto que la Industria 4.0 pretende disponer de nuevos métodos y técnicas de producción que mejoren la flexibilidad y adaptabilidad, como Agile Manufacturing y Just In Time.

La aplicación de estas nuevas técnicas y el aumento en la flexibilidad suele acarrear incrementos en costes y tiempos. Por ejemplo, el tener que parar una línea de producción para efectuar algún cambio puede suponer grandes sobrecostes, o puede ser sencillamente inviable debido a largos tiempos de arranque.

Este conflicto entre adaptabilidad a cambio y la minimización de tiempos y costes afecta asimismo al campo de la Automatización Industrial. Existe una necesidad clara de poder actualizar la lógica de control sin detener la planta.

Una solución a este problema es la actualización dinámica de software o Dynamic Software Update (DSU). DSU permite reducir el tiempo de parada a cero durante una actualización de software.

Entre los diferentes enfoques de este tema, el uso del lenguaje de programación Erlang, desarrollado por Ericsson, resulta bastante prometedor. Erlang ofrece funcionalidades que permiten actualizar código sin detener el proceso, así como alterar un proceso en tiempo de ejecución.

El objetivo principal de este proyecto realizado en la Universidad Tecnológica de Múnich (TUM) es estudiar cómo pueden ser utilizadas diversas funcionalidades de Erlang con el fin de generar Aplicaciones de Reconfiguración de Control para Automatización Industrial.

Con este fin se sigue el estándar IEC 61499, que ofrece encapsulación de código, y por lo
tanto modularidad y reusabilidad. Para la ejecución del proyecto se estudia la arquitectura de dicho estándar, haciendo un especial hincapié en su enfoque con respecto a las Aplicaciones de Reconfiguración y en su aplicación para DSU.

Más adelante se realiza un estudio de las diversas funcionalidades de Erlang, y cuáles podrían ser encapsuladas en módulos basándose en la arquitectura de IEC 61499.

Una vez se tiene una visión completa del problema y de las diferentes herramientas disponibles, se implementan los algoritmos necesarios para crear un sistema de DSU genérico y reutilizable.

Tras la implementación, se realizan diversos tests para poner a prueba la solución propuesta y otras soluciones existentes basadas en Erlang. Finalmente, los resultados de dichos tests son empleados en para realizar un análisis comparativo de las diferentes soluciones.

**Estado del Arte**

**Dynamic Software Update**

Los métodos de DSU son aquellas técnicas que permiten actualizar piezas de software mientras están siendo ejecutadas. Son aplicados en diversos campos, ya que resulta inevitable tener que actualizar el software, ya sea por introducción de nuevas actualizaciones, cambios o por depuración de código.

El DSU se puede aplicar o bien por la inclusión de hardware redundante o utilizando software. Por ejemplo, mediante diferentes lenguajes de programación que incluyen herramientas para DSU, se pueden implementar Aplicaciones de Reconfiguración, que pueden ser utilizadas para actualizar el software de control sin necesidad de parar la producción por completo.

Sin embargo estos métodos todavía presentan limitaciones, ya que aún es complicado determinar un punto de actualización seguro y no son capaces de determinar exactamente los cambios de código que son necesarios en una determinada actualización.

**IEC 61499**

Los diferentes planteamientos para sistemas de control industrial, han ido convergiendo en un enfoque más distribuido, alejándose de códigos monolíticos. De este modo se puede contar con un sistema de control en cada dispositivo, controlados por un sistema central [4].
La Comisión Internacional de Electrotrcnia (IEC) publicó en 2005 el estándar IEC 61499 [5], cuyos sistemas siguen una arquitectura basada en la encapsulación de código en Bloques de Funciones (FB) interconectados.

Dichos FB disponen de una interfaz como la representada en la Figura 1, con eventos y datos de entrada y de salida. Su ejecución está desencadenada por eventos que activan los diferentes FB. Estos reciben datos al ser activadas, y envían datos y eventos a sus FB consecutivas tras ejecutar sus algoritmos internos [7] [8].

Los diferentes FBs se agrupan en aplicaciones y subaplicaciones, y pueden ser de diferentes tipos. Pueden ser FB Básicos (BFB), Compuestos (CFB) si son una agrupación de BFB, y de Interfaz de Servicio (SIFB), o también denominados Servicios, que cubren las funcionalidades que están más allá de la especificación, como por ejemplo los Servicios de Reconfiguración, que pueden ser empleados para DSU.

El estándar especifica que los datos deben ser intercambiados en lenguaje textual o en XML. Actualmente hay diferentes implementaciones del estándar, ya que únicamente ofrece descripciones a alto nivel.

Existe una herramienta para el uso de IEC 61499 denominada 4diac IDE, un entorno de desarrollo de código abierto que aporta una interfaz para crear sistemas de acuerdo al estándar, y que exporta los datos de dicho sistema en XML [13].

**Erlang**

Una de las implementaciones de IEC 61499 es en Erlang, lenguaje de programación funcional creado por Ericsson. Erlang es un lenguaje basado en módulos instanciables que se pueden comunicar entre ellos mediante mensajes, lo que lo hace perfecto para implementar IEC 61499. Es altamente escalable y puede ser ejecutado en cualquier lugar en su máquina virtual [14].

Erlang ofrece diversos comportamientos contenidos en la denominada Open Telecom Plat-
form (OTP). Cada comportamiento encierra funcionalidades relacionadas con un uso específico. La OTP provee a los desarrolladores de una buena base para estandarización.

Una característica fundamental de Erlang para DSU es Hot Code Loading, que permite actualizar una aplicación a su nueva versión mediante la ejecución de de un archivo appup que contiene las instrucciones necesarias para la actualización [15].

Estas actualizaciones pueden ser ejecutadas en tiempo en paralelo al sistema a actualizar. Si se requiere la actualización del estado actual de un módulo, es necesario suspenderlo, pero no pararlo. De este modo recibirá mensajes pero no reaccionará hasta que sea reanudado.

Existen también diversas funciones en la OTP que permiten alterar procesos en tiempo de ejecución, que pueden ser empleadas del mismo modo para DSU.

**FBBBeam**

Una de las implementaciones de IEC 61499 en Erlang fue creada por Prenzel y Provost en la Universidad Tecnológica de Múnich [17]. Dicha implementación emplea el comportamiento OTP gen_statem, que corresponde a una máquina de estados genérica.

Esta implementación consiste en un compilador en Python que toma como entrada los documentos XML de acuerdo con IEC 61499 y los compila traduciendolos a código Erlang. De este modo, modela cada tipo de FB como un módulo con diferentes instancias, cada una representando cada FB incluida en aplicaciones. Estos módulos se mandan mensajes entre ellos, que corresponden a las conexiones de eventos y datos.

Las aplicaciones de IEC 61499 son modeladas como aplicaciones de Erlang, así como las CFB son modeladas como subaplicaciones de Erlang. Cada aplicación dispone de un supervisor a cargo de arrancar y controlar las diferentes instancias de FBs que contiene.

Las SIFBs deben ser modeladas aparte en el compilador en forma de plantillas en Erlang, que FBBBeam completará con las correspondientes instancias.

Los archivos generados por FBBBeam deben ser compilados y ejecutados por el usuario.

Actualmente FBBBeam ofrece la posibilidad de comparar una versión de código con su actualización y generar automáticamente el archivo appup necesario para su actualización. Sin embargo no cuenta con Servicios de Reconfiguración como los propuestos por IEC 61499.
**Metodología**

El proyecto se ha completado siguiendo las especificaciones de IEC61499.

Se ha empleando Visual Studio Code para la manipulación de código en los diferentes lenguajes empleados, así como el IDE 4diac para crear sistemas y diferentes Servicios de acuerdo con el estándar.

Los pasos seguidos en el proyecto son los siguientes:

1. Identificación y clasificación de los Servicios de Reconfiguración a implementar
2. Implementación de dichos Servicios utilizando funciones de OTP
3. Para cada clase determinada, se crea una Aplicación de Reconfiguración a pequeña escala para comprobar que todas las FB de dicha clase funcionen correctamente durante la implementación, de este modo identificando posibles errores y pudiendo corregirlos.
4. Tras la implementación se crean casos de estudio a una escala más grande para poner a prueba la solución propuesta, así como las appups generadas por FBBeam.
5. Se realiza un análisis comparativo de los diferentes métodos probados, teniendo en cuenta diferentes criterios.

**Implementación**

La implementación de la nueva solución para DSU con Erlang, consiste en implementar los diferentes Servicios de Reconfiguración prouestos por IEC 61499 como plantillas en Erlang, que puedan ser añadidas al compilador FBBeam, e instanciadas en Aplicaciones de Reconfiguración.

Este compilador es extendido para que reconozca estos Servicios en los archivos XML generados por 4diac siguiendo el estándar IEC 61499. A partir de dichos archivos FBBeam genera el código Erlang correspondiente a las Aplicaciones de Reconfiguración diseñadas.

Lo primero que hay que realizar es estudiar qué Servicios son necesarios para poder efectuar cualquier reconfiguración. Cada Servicio estará encargado de realizar una tarea que efectue un pequeño cambio en la aplicación. Algunos ejemplos de estos cambios son añadir o eliminar un FB, o leer o cambiar el valor de un dato de entrada.

Las tareas básicas que propone el estándar son:
 CREATE: Introducir un nuevo elemento en el sistema

 DELETE: Borrar un elemento del sistema

 START: Arrancar un elemento

 STOP: Suspender un elemento

 KILL: Parar un elemento

 QUERY: Solicitar información del sistema

 READ: Solicitar información actual de un elemento del sistema

 WRITE: Cambiar un valor de una instancia de FB

 RESET: Resetear una instancia

Basándose en el estado del arte actual y teniendo en cuenta las funcionalidades y cualidades de Erlang, se determina la clasificación presentada en la Tabla 1. Todos los Servicios representados en dicha tabla son los Servicios que es necesario implementar en Erlang para dar con la solución deseada.

Una vez identificados los Servicios a implementar, y teniendo ya una clasificación adecuada de los mismos en cuatro categorías según su función, se procede a la implementación de todos ellos.

Esta implementación se realiza en bloques, correspondiendo cada uno a una de las clases especificadas. Esto se debe a que dentro de cada clase, algunas de las funciones de Erlang o las estructuras de código empleadas se pueden reusar o pueden ser utilizadas como base para implementar procesos similares en otros Servicios de su clase.

Para cada Servicio implementado se sigue el mismo procedimiento:

1. Se valoran qué eventos y datos de entrada y salida son necesarios. En el caso de los eventos, todos los servicios implementados disponen de un evento de entrada llamado $REQ$ que es activado cuando se requiere la ejecución del servicio. También disponen todos de un evento de salida denominado $CNF$ que envía un evento cuando el servicio ha finalizado su ejecución. En el caso de las entradas y salidas de datos, varían de un servicio a otro.

2. Se crea la interfaz del mismo en 4diac, para incluir el elemento necesario en esta herramienta, y para comprobar su funcionamiento en consiguientes tests. Un ejemplo de interfaz de un Servicio creado se muestra en la Figura 2.
<table>
<thead>
<tr>
<th>Clasificación</th>
<th>Nombre del Servicio</th>
<th>Descripción</th>
</tr>
</thead>
<tbody>
<tr>
<td>Servicios de consulta</td>
<td>QUERY_FBs</td>
<td>Solicitar una lista de los FBs en la aplicación correspondiente</td>
</tr>
<tr>
<td></td>
<td>QUERY_FB_STATE</td>
<td>Solicitar el estado actual de un FB</td>
</tr>
<tr>
<td></td>
<td>QUERY_CONs</td>
<td>Solicitar una lista con las conexiones de un FB</td>
</tr>
<tr>
<td></td>
<td>QUERY_TYPE</td>
<td>Solicitar el tipo del elemento correspondiente</td>
</tr>
<tr>
<td></td>
<td>QUERY_TYPE_LIST</td>
<td>Solicitar la lista de los tipos existentes en una aplicación</td>
</tr>
<tr>
<td>Servicios de Control de</td>
<td>START</td>
<td>Arrancar una instancia de FB</td>
</tr>
<tr>
<td>Ejecución</td>
<td>STOP</td>
<td>Suspender una instancia de FB</td>
</tr>
<tr>
<td></td>
<td>KILL</td>
<td>Parar una instancia de FB</td>
</tr>
<tr>
<td>Servicios de Interacción</td>
<td>READ</td>
<td>Leer el valor de una entrada, salida o variable interna de un FB</td>
</tr>
<tr>
<td>de Estados</td>
<td>WRITE</td>
<td>Cambiar el valor de una entrada, salida o variable interna de un FB</td>
</tr>
<tr>
<td></td>
<td>WRITE_FB_STATE</td>
<td>Cambiar el estado interno de un FB</td>
</tr>
<tr>
<td>Servicios Estructurales</td>
<td>CREATE_SUBAPP</td>
<td>Crear una nueva Subaplicación</td>
</tr>
<tr>
<td></td>
<td>CREATE_FB</td>
<td>Crear una nueva instancia de FB</td>
</tr>
<tr>
<td></td>
<td>CREATE_CON</td>
<td>Crear una conexión entre instancias de FB</td>
</tr>
<tr>
<td></td>
<td>DELETE_SUBAPP</td>
<td>Eliminar una Subaplicación</td>
</tr>
<tr>
<td></td>
<td>DELETE_FB</td>
<td>Eliminar una instancia de FB</td>
</tr>
<tr>
<td></td>
<td>DELETE_CON</td>
<td>Eliminar una conexión entre FB</td>
</tr>
</tbody>
</table>

**Table 1:** Servicios de Reconfiguración implementados
3. Se implementa el código correspondiente en Erlang, en forma de un módulo que siga el comportamiento OTP gen_statem y que cumpla las características y las funciones requeridas por el servicio. Las entradas y salidas de datos se modelan con el envío de mensajes entre instancias, y para modelar la funcionalidad se emplean funciones de la librería OTP.

4. Se comprueba su comportamiento en solitario y se depura el código a base de ejecutar el código implementado e interactuar con el módulo a través del envío de mensajes al mismo que simulen entradas de eventos y datos.

Todos los servicios implementados tienen en cuenta posibles fallos en el sistema, por ejemplo, entradas de datos incorrectas. En caso de que se produzca un error, detienen la ejecución e informan del error encontrado. Esto es muy útil a la hora de generar Aplicaciones de Reconfiguración, para identificar posibles fallos en pruebas previas a la actualización real.

Una vez generados los archivos correspondientes para cada clase, se genera en 4diac un test a pequeña escala para probar el funcionamiento de los servicios de la clase, y así continuar con la detección de errores y depuración de código.

Para esto se cuenta con un pequeño sistema que representa un contador, como se aprecia en la Figura 3. Para cada clase se genera una Aplicación de Reconfiguración que emplee todos los servicios de una determinada clase y los pruebe sobre el sistema, para así efectuar sobre el mismo pequeñas reconfiguraciones.
Casos de Estudio

Con el fin de poner a prueba el funcionamiento de los servicios creados en sistemas más cercanos a la realidad, se procede a crear sistemas de escala más grande con 4diac a los que se le aplican diferentes reconfiguraciones, utilizando tanto aplicaciones que contengan los Servicios de Reconfiguración implementados, como los archivos appup generados automáticamente con FBBeam.

Para cada caso de estudio se generan en 4diac la versión inicial del sistema, el sistema con los cambios sufridos tras la actualización y las aplicaciones de reconfiguración correspondientes. Con FBBeam se compilan todos los archivos, y se genera el archivo appup a partir de las diferencias entre dos versiones del sistema. De este modo se pueden poner a prueba ambas soluciones sobre el sistema.

Los tres sistemas considerados como casos de estudio son los tratados en los siguientes apartados. Las Aplicaciones de Reconfiguración son brevemente mostradas, encontrándose su explicación en detalle en la documentación del proyecto.

Sistema de tanques interconectados

El primer modelo es un sistema de tanques interconectados. La versión inicial del sistema consiste en dos tanques interconectados, estando en el primero la entrada y en el segundo la salida de fluido. Ambos tanques tienen asociado un control PID con un nivel del tanque de referencia, como muestra la Figura 4, y su implementación en 4diac en la Figura 5.

![Figure 4: Diagrama del sistema de tanques antes de la reconfiguración](image)

A dicho sistema se le aplican dos reconfiguraciones diferentes, con diferentes objetivos. La primera altera el modelo añadiendo un tercer tanque con su PID, como ilustra la Figura 6. Esta reconfiguración tiene como objetivo emplear servicios muy variados, para
así comprobar el funcionamiento de un gran número de servicios diferentes actuando juntos en la misma aplicación, como se muestra en el diagrama de la Figura 7.

Figure 5: Modelo de tanques con IEC 61499 antes de la Reconfiguración

Figure 6: Modelo del tanque tras la primera Reconfiguración
En la Figura 8 se aprecia la inclusión del tercer tanque en el segundo 20 de ejecución. La segunda reconfiguración, representada en la Figura 9, consiste en cambiar los valores de las ganancias de los PID para comprobar su funcionamiento en una situación más cercana a una reconfiguración real.
En la Figura 10 se puede apreciar la reconfiguración en el segundo 20, llegando a los valores objetivo de 30 y 50 litros.

Ambas soluciones consiguen realizar la reconfiguración de manera correcta, ejecutando la nueva versión del sistema tras la actualización.

**Tambor de vapor**

El segundo sistema probado consiste en un modelo de una caldera con tambor de vapor con un control PD para el nivel del tambor, como muestra la Figura 11. En este caso, la caldera es modelada haciendo uso de subaplicaciones para así poder probar diferentes servicios e instrucciones del appup, como representa el modelo de la Figura 12.
En este caso también se efectúan dos reconfiguraciones. La primera actualiza el modelo de la caldera por uno más complejo que considera la entrada de aire, como muestra la Figura 13. De este modo, se hace uso de los servicios que trabajan con subaplicaciones, como se aprecia en la Aplicación de Reconfiguración representada en la Figura 14.
La segunda, representada en el diagrama de la Figura 15 al igual que en el caso anterior, actualiza el control de la caldera, en este caso, cambiando el PD por un PID, consiguiendo así el nivel objetivo que es 40 litros, como se observa en la Figura 16.
TambiÉn en este caso de estudio, resultan los dos mÉtodos efectivos para realizar la actualizaciÓn.

**EstaciÓn de taladrado**

Finalmente, se modela una estaciÓn de mecanizado de una lÍnea de producciÓn, correspondiente a una estaciÓn de taladrado mostrada en la Figura 17, y modelada en 4diac como muestra la Figura 18.
En este caso sólo se modela una reconfiguración, representada en la Figura 19, con la que se actualiza el control de la cinta transportadora, pasando de aceptar las piezas de una en una, a aceptar varias piezas en la estación.
El objetivo de esta reconfiguración es diferente al de las anteriores. Esta reconfiguración prueba la capacidad de los servicios de reconfiguración de usar datos actuales de la ejecución del sistema para la propia reconfiguración.

Durante la actualización, se utiliza un Servicio de Reconfiguración para leer el estado de la primera versión de la cinta transportadora, y así poner la nueva versión en el mismo estado. Del mismo modo, se lee el estado de otro FB y no se continúa la ejecución de la actualización hasta que no se encuentra en el estado deseado. De este modo se puede asegurar que la actualización, o parte de ella, se lleva a cabo en un punto seguro de la ejecución.

En este caso también realizan la actualización de manera correcta ambos métodos, pero sólo siendo los Servicios capaces de incluir información actual del sistema en ejecución en la reconfiguración.

**Análisis comparativo**

Una vez han sido generados y probados todos los casos de estudio, pueden ser evaluados los diferentes métodos para DSU. En este análisis son considerados los siguientes métodos:

- Aplicación de Reconfiguración usando Servicios de Reconfiguración
- Reconfiguración usando un archivo appup generado automáticamente por FBBeam
- Reconfiguración usando un archivo appup generado de forma manual

El objetivo de incluir el appup generado de manera manual es para incluir todo lo que puede ofrecer Erlang en cuanto a DSU, ya que el appup generado por FBBeam es aún una solución subóptima.

Los criterios empleados para el análisis comparativo son: Exactitud, duración, usabilidad y extensibilidad.

**Exactitud de la actualización**

Este criterio evalúa la capacidad de cada método de realizar todas las actualizaciones posibles de manera correcta, es decir llevando a cabo todos los cambios esperados en el sistema, y que las realice sin dar lugar a errores durante la actualización.

En este caso todos los métodos cumplen con lo esperado, salvo en el caso de las aplicaciones de reconfiguración, que pueden dar lugar a paradas en la reconfiguración si se suspen de algún FB en el momento de ser actualizado. Un resumen de los resultados se presenta en la Tabla 2.

<table>
<thead>
<tr>
<th></th>
<th>Archivo appup FBBeam</th>
<th>Archivo appup manual</th>
<th>Aplicación de Reconfiguración</th>
</tr>
</thead>
<tbody>
<tr>
<td>Es posible realizar TODOS los pasos de la actualización</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>No de lugar a errores durante la actualización</td>
<td>✓</td>
<td>✓</td>
<td>Puede dar errores con procesos suspendidos</td>
</tr>
</tbody>
</table>

**Table 2: Resultados de Exactitud**

Los tres métodos son capaces de llevar a cabo todas las reconfiguraciones, pero una Aplicación de Reconfiguración puede dar lugar a errores si los procesos suspendidos no son manejados debidamente.

**Duración de la actualización**

Este criterio evalúa las diferencias en duración que puede experimentar una actualización al ser realizada con diferentes métodos. Para ello se emplean los casos de estudio de los
tanques y de la caldera, ya que en el de la estación de mecanizado el tiempo depende de la información ofrecida por el sistema y no de la eficiencia del método empleado.

Ambos sistemas probados muestran un comportamiento cíclico, y disponen de un delay entre ciclos, por lo que se probará cada sistema con diferentes tiempos de ciclo, ya que un tiempo muy pequeño puede provocar una sobrecarga de memoria o de procesador e influir en los tiempos de actualización.

Cada Aplicación de Reconfiguración con cada tiempo de ciclo (0ms, 1ms, 5ms y 10ms) es ejecutada 100 veces, para contar con resultados fiables.

A partir de los resultados se puede concluir que cuando el delay es mayor que 0, los tiempos de las Aplicaciones de Reconfiguración son cortos (media de 1-5ms) y menores que los de appups (5-12ms).

Sin embargo, cuando se cuenta con tiempo de ciclo nulo, las distribuciones de tiempo tienen una media similar con ambos métodos. Sin embargo, las Aplicaciones de Reconfiguración dan lugar a distribuciones de tiempo con una desviación típica más amplia, por lo que se podría decir que las appups (bien manual o generada automáticamente) ofrecen una mayor fiabilidad en este caso.

Un resumen de los resultados con respecto al tiempo se presenta en la Tabla 3.

<table>
<thead>
<tr>
<th>Tiempo de ciclo</th>
<th>Método Recomendado</th>
<th>Motivo</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;0</td>
<td>Aplicación de Reconfiguración</td>
<td>Los tiempos son generalmente menores.</td>
</tr>
<tr>
<td>0</td>
<td>Appup FBBeam/Manual</td>
<td>La dispersión de los datos es menor, y es difícil determinar qué método es más rápido</td>
</tr>
</tbody>
</table>

Table 3: Resultados de Tiempos

Los tiempos de actualización son considerablemente mejores usando una Aplicación de Reconfiguración si el delay es mayor que cero. Con un delay nulo, los tiempos dependen del caso, pero el appup ofrece menor incertidumbre.

**Extensibilidad**

La extensibilidad hace referencia a la facilidad para añadir nuevas funcionalidades al método.

En el caso de las Aplicaciones de Reconfiguración, añaden mucha flexibilidad, ya que su
estructura modular y la posibilidad de incluir cualquier FB permite muchas funcionalidades como por ejemplo alterar el orden de los pasos de reconfiguración o usar datos del propio sistema en la reconfiguración.

Por otra parte, los archivos appup de FBBeam ofrecen poca flexibilidad. Sin embargo, siendo aún una solución subóptima, ofrecen un appup mínimamente funcional que conforma una buena base sobre la que poder alterar el orden de los comandos o añadir otros nuevos de manera manual.

<table>
<thead>
<tr>
<th>Cualidades para extensibilidad</th>
<th>Applicación de Reconfiguración</th>
<th>Appup de FBBeam</th>
<th>Appup manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Modularidad</td>
<td>-Ofrece una appup funcional básica</td>
<td>-Cambio de orden de los pasos de reconf.</td>
<td></td>
</tr>
<tr>
<td>-Interacción con cualquier FB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Cambio de orden de los pasos de reconf.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 4: Resultados de extensibilidad**

La modularidad de las Aplicaciones de Reconfiguración ofrece mucha más libertad, flexibilidad y reutilización de código, con el único requisito de conocer Erlang.

**Usabilidad**

La usabilidad se corresponde con cómo de fácil es para el usuario utilizar el método. Esto incluye las habilidades que son necesarias para utilizarlo, lo simple que es, su tolerancia a fallos, y las funcionalidades que ofrece.

Todas las cualidades mencionadas se encuentran recogidas en la Tabla 5, con respecto a los diferentes métodos.
Habilidades Requeridas

- Comandos básicos de Erlang
- 4diac e IEC 64199
- FBBeam

Simplicidad

- Solo requiere algunos comandos básicos de Erlang

Manipulación de errores humanos

- Sólo identifica errores durante la compilación de la nueva versión
  - Sólo al compilar el appup
  - La mayoría de errores humanos son manejados y reportados

Funcionalidades extra

- También ofrece volver a la versión original

<table>
<thead>
<tr>
<th>Appup de FBBeam</th>
<th>Appup manual</th>
<th>Aplicación de Reconf.</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Comandos básicos de Erlang</td>
<td>- Erlang</td>
<td>- Comandos básicos de Erlang</td>
</tr>
<tr>
<td>- 4diac e IEC 64199</td>
<td>- IEC 64199</td>
<td>- 4diac e IEC 64199</td>
</tr>
<tr>
<td>- FBBeam</td>
<td></td>
<td>- FBBeam</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Servicios de Reconfiguración</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Simplicidad</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Solo requiere algunos comandos básicos de Erlang</td>
<td>Requiere crear un archivo appup funcional completo en Erlang</td>
<td>Requiere preparar una Aplicación de Reconfiguración completa, que puede resultar complejo</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Manipulación de errores humanos</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Sólo identifica errores durante la compilación de la nueva versión</td>
<td>Sólo al compilar el appup</td>
<td>La mayoría de errores humanos son manejados y reportados</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Funcionalidades extra</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>También ofrece volver a la versión original</td>
<td>También ofrece volver a la versión original</td>
<td>Ofrece mucha flexibilidad en la actualización</td>
</tr>
</tbody>
</table>

Table 5: Resultados de usabilidad

Otros factores

Es importante considerar que al realizar una actualización con un archivo appup, el sistema es suspendido. A pesar de que tras la actualización es reanudado en el punto original, se pierde cierto tiempo de ejecución.

Esto no sucede con una Aplicación de Reconfiguración. Sin embargo, estas aplicaciones actualizan procesos, pero el código sigue siendo el original, por lo que una vez parado el proceso por completo sería necesario ejecutar el appup para actualizar el código.

Se recoge un resumen de estos factores en la Tabla 6.

Table 6: Otros factores

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>No es necesario suspender los procesos</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>
La conclusión principal es que en el caso de infraestructuras críticas, en las que algunos milisegundos marcan una gran diferencia, o para casos donde se necesita más flexibilidad durante la actualización, una Aplicación de Reconfiguración es la solución ideal. Sin embargo, si el sistema necesita ser parado, se precisa de un appup para preservar los cambios.

Discusión de los resultados y conclusiones

El objetivo de este proyecto es implementar las herramientas necesarias para crear Aplicaciones de Reconfiguración usando Erlang y siguiendo la norma IEC 61499, y comparar su funcionamiento con otros métodos de DSU basados en Erlang, con el objetivo de encontrar qué funcionalidades de Erlang son mejores para DSU.

Tras implementar todos los Servicios necesarios, y realizar diversos tests, se ha probado que Erlang puede actualizar cualquier sistema, bien utilizando appups como las generadas por FBBeam, o utilizando Servicios para no necesitar suspender procesos. Pero resulta importante destacar en qué resulta mejor un método que otro.

Los beneficios que aportan las Aplicaciones de Reconfiguración con los Servicios de Reconfiguración respecto a las appups son:

- Las aplicaciones no deben ser suspendidas para ser actualizadas.
- Es una solución modular y flexible.
- Los Servicios pueden ser combinados con cualquier FB para crear nuevas funcionalidades, como utilizar información del sistema en la actualización o crear bucles.

En cambio, acarrea otras desventajas con respecto a los archivos appup generados por FBBeam:

- Las Aplicaciones de Reconfiguración pueden resultar muy complejas y laboriosas, lo que puede llevar a errores.
- Son los procesos los que son actualizados, pero el código permanece inalterado.
- Todavía es una solución no óptima, y puede dar lugar a errores cuando se necesita suspender procesos.

Como conclusión general, lo ideal es combinar ambos métodos. Siempre que se precise de una actualización sin suspender los procesos, o se necesite más flexibilidad o utilizar más
funciones que una appup no ofrece, se debe usar una Aplicación de Reconfiguración. Una vez que se tenga que parar por completo el proceso, se podrá ejecutar el archivo appup para actualizar también el código subyacente.

Utilizando estas técnicas, se puede mejorar mucho la eficiencia de las actualizaciones, permitiendo así realizarlas más a menudo. Esto puede ayudar a reducir costes, tiempo y energía, así como a tener siempre software actualizado y con las últimas actualizaciones, incluidas las de seguridad. Esto puede ayudar a tener puestos de trabajo con mayor seguridad y estabilidad, y ahorrando tiempo a los trabajadores. Del mismo modo, las técnicas de DSU pueden generar puestos de trabajo relacionados con implantar estas actualizaciones.

**Líneas futuras**

Se han demostrado las posibilidades que ofrece Erlang para DSU, pero las soluciones propuestas no son completamente óptimas, por lo que todavía queda trabajo por realizar.

Para empezar, los Servicios de Reconfiguración pueden ser más intuitivos y accesibles. Esto se puede conseguir, solucionando los problemas que pueden acarrear los procesos suspendidos e introduciendo más posibles errores humanos a evitar.

Por otro lado, se pueden generar soluciones para integrar los archivos appup y los Servicios de Reconfiguración. Para ello, se puede extender el compilador FBBeam para que a partir de una Aplicación de Reconfiguración genere el código de la nueva versión y el correspondiente appup.

Del mismo modo, se podría generar la solución opuesta, es decir, a partir de dos versiones de un sistema generar una Aplicación de Reconfiguración mínimamente funcional, que pueda ser más tarde editada para conseguir la flexibilidad deseada.

Con estos avances, se conseguiría un método de DSU mucho más completo e intuitivo.

**Planificación del proyecto**

**Planificación temporal**

En este proyecto se han tenido en cuenta diferentes estadios y tareas. De este modo se ha dividido el trabajo en diferentes paquetes como muestra la Estructura de Descomposición del Proyecto de la Figura 20.
Todos estos paquetes de trabajo han sido evaluados y se les asignó un marco temporal como se muestra en el diagrama de Gantt de la Figura 21.

**Presupuesto**

En el presupuesto del trabajo, resumido en la Tabla 7 se tiene en cuenta las horas de ingeniería, al precio pagado en la TUM a los estudiante y la depreciación del ordenador empleado, ya que el software empleado es de licencia libre o de licencia de estudiante, y la energía eléctrica empleada se considera despreciable.

<table>
<thead>
<tr>
<th>Coste de ingeniería</th>
<th>Unidades (h)</th>
<th>Precio(€)</th>
<th>Coste(€)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Planificación e Investigación</td>
<td>280</td>
<td>12</td>
<td>3360</td>
</tr>
<tr>
<td>Implementación y Pruebas</td>
<td>512</td>
<td>12</td>
<td>6144</td>
</tr>
<tr>
<td>Documento y Presentación</td>
<td>248</td>
<td>12</td>
<td>2976</td>
</tr>
<tr>
<td>Costes de depreciación</td>
<td>Tiempo</td>
<td>Precio(€)</td>
<td>Coste(€)</td>
</tr>
<tr>
<td>Ordenador</td>
<td>6.25 %</td>
<td>700</td>
<td>43.75</td>
</tr>
<tr>
<td></td>
<td><strong>Total</strong></td>
<td></td>
<td><strong>12523.75</strong></td>
</tr>
</tbody>
</table>

Table 7: Presupuesto del proyecto
<table>
<thead>
<tr>
<th>Nombre de tarea</th>
<th>Duración</th>
<th>Comienzo</th>
<th>Fin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Definition and Planning</td>
<td>15 días</td>
<td>lun 15/04/19</td>
<td>v 03/05/19</td>
</tr>
<tr>
<td>First Approach and Research</td>
<td>8 días</td>
<td>lun 15/04/19</td>
<td>milé 24/04/19</td>
</tr>
<tr>
<td>Scope Definition</td>
<td>4 días</td>
<td>jue 25/04/19</td>
<td>mar 30/04/19</td>
</tr>
<tr>
<td>Project Exposé</td>
<td>3 días</td>
<td>milé 01/05/19</td>
<td>v 03/05/19</td>
</tr>
<tr>
<td>Research and Formation</td>
<td>20 días</td>
<td>lun 06/05/19</td>
<td>v 31/05/19</td>
</tr>
<tr>
<td>Deep Literature review</td>
<td>10 días</td>
<td>lun 06/05/19</td>
<td>v 17/05/19</td>
</tr>
<tr>
<td>Erlang Tutorial</td>
<td>10 días</td>
<td>lun 20/05/19</td>
<td>v 31/05/19</td>
</tr>
<tr>
<td>Implementation</td>
<td>36 días</td>
<td>lun 03/06/19</td>
<td>jun 22/07/19</td>
</tr>
<tr>
<td>Execution Control services</td>
<td>10 días</td>
<td>lun 03/06/19</td>
<td>v 14/06/19</td>
</tr>
<tr>
<td>Query Services</td>
<td>8 días</td>
<td>lun 17/06/19</td>
<td>milé 26/06/19</td>
</tr>
<tr>
<td>Structural Services</td>
<td>10 días</td>
<td>jue 27/06/19</td>
<td>milé 10/07/19</td>
</tr>
<tr>
<td>State interaction Services</td>
<td>8 días</td>
<td>milé 11/07/19</td>
<td>jun 22/07/19</td>
</tr>
<tr>
<td>Case studies</td>
<td>13 días</td>
<td>mar 21/07/19</td>
<td>jue 08/08/19</td>
</tr>
<tr>
<td>Selecting Case studies</td>
<td>3 días</td>
<td>mar 23/07/19</td>
<td>jue 25/07/19</td>
</tr>
<tr>
<td>Model systems</td>
<td>10 días</td>
<td>milé 26/07/19</td>
<td>jue 08/08/19</td>
</tr>
<tr>
<td>Model Reconf Applications</td>
<td>10 días</td>
<td>milé 29/07/19</td>
<td>jue 09/08/19</td>
</tr>
<tr>
<td>Comparative Assessment</td>
<td>15 días</td>
<td>lun 12/08/19</td>
<td>milé 30/08/19</td>
</tr>
<tr>
<td>Update Time</td>
<td>8 días</td>
<td>lun 12/08/19</td>
<td>milé 21/08/19</td>
</tr>
<tr>
<td>Update Accuracy</td>
<td>7 días</td>
<td>jue 22/08/19</td>
<td>milé 30/08/19</td>
</tr>
<tr>
<td>Usability</td>
<td>7 días</td>
<td>jue 22/08/19</td>
<td>milé 30/08/19</td>
</tr>
<tr>
<td>Extendability</td>
<td>7 días</td>
<td>jue 22/08/19</td>
<td>milé 30/08/19</td>
</tr>
<tr>
<td>Writing document</td>
<td>26 días</td>
<td>lun 02/09/19</td>
<td>jue 07/10/19</td>
</tr>
<tr>
<td>Document Structure</td>
<td>1 día</td>
<td>lun 02/09/19</td>
<td>milé 02/09/19</td>
</tr>
<tr>
<td>Writing</td>
<td>12 días</td>
<td>mar 03/09/19</td>
<td>milé 18/09/19</td>
</tr>
<tr>
<td>Formatting</td>
<td>15 días</td>
<td>mar 03/09/19</td>
<td>lun 23/09/19</td>
</tr>
<tr>
<td>Revision+Corrections</td>
<td>13 días</td>
<td>jue 19/09/19</td>
<td>milé 07/10/19</td>
</tr>
<tr>
<td>Preparing presentation</td>
<td>5 días</td>
<td>mar 08/10/19</td>
<td>lun 14/10/19</td>
</tr>
</tbody>
</table>
# CONTENTS

1. INTRODUCTION 1
   1.1. Motivation .............................................. 1
   1.2. Objectives ............................................. 2
   1.3. Document structure ..................................... 3

2. BACKGROUND 5
   2.1. Dynamic Software Update ............................... 5
   2.2. IEC 61499 .................................................. 6
       2.2.1. Architecture ......................................... 7
       2.2.2. IEC 61499 as a base for DSU ....................... 8
       2.2.3. 4diac IDE ........................................... 9
   2.3. Erlang .................................................... 9
       2.3.1. Use of Erlang for DSU ............................. 10
   2.4. FBBeam .................................................. 11
       2.4.1. FBBeam application updates ...................... 11
   2.5. Summary ................................................ 12

3. METHODOLOGY 13
   3.1. Resources ................................................. 13
3.2. Specifications ......................................................... 14
3.3. Steps followed ....................................................... 14

4. IMPLEMENTATION OF RECONFIGURATION SERVICES IN ER- LANG ............................................ 17

4.1. Query Services ...................................................... 21
    4.1.1. Query a List with all the FB instances ..................... 21
    4.1.2. Query the process status of a FB ......................... 23
    4.1.3. Query the output connections of a FB .................... 24
    4.1.4. Query a List with all the used FB types ................. 26
    4.1.5. Query the type of a FB ................................... 27
    4.1.6. Testing of Query Services ................................. 29

4.2. Execution Control Services ...................................... 30
    4.2.1. Stop a FB ............................................... 32
    4.2.2. Restart a FB .......................................... 34
    4.2.3. Kill a FB ........................................... 35
    4.2.4. Testing of Execution Control Services .................. 36

4.3. Structural Services .............................................. 37
    4.3.1. Create a new FB ...................................... 38
    4.3.2. Delete an existing FB .................................. 40
    4.3.3. Create a new connection between FBs ................. 42
    4.3.4. Delete a connection between FBs ...................... 45
    4.3.5. Create a new Subapplication ............................ 47
    4.3.6. Delete a Subapplication ............................... 50
    4.3.7. Testing of Structural Services .......................... 51
4.4. State Interaction Services ............................................. 55
  4.4.1. Write a FB Parameter ............................................. 55
  4.4.2. Read a FB Parameter ............................................. 57
  4.4.3. Change the State of a FB ........................................ 59
  4.4.4. Read the State of a FB ............................................. 61
  4.4.5. Testing of State Interaction Services ......................... 62

5. CASE STUDIES .............................................. 65
  5.1. Interconnected tanks .............................................. 65
    5.1.1. System modeling ................................................. 66
    5.1.2. Reconfiguration Applications .................................. 68
  5.2. Boiler Steam Drum .............................................. 75
    5.2.1. System modeling ................................................. 75
    5.2.2. Reconfiguration Applications .................................. 78
  5.3. Machining station .............................................. 87
    5.3.1. System modeling ................................................. 88
  5.4. Reconfiguration Application ...................................... 90

6. COMPARATIVE ASSESSMENT ..................................... 95
  6.1. Update Accuracy ................................................. 96
  6.2. Update Time .................................................. 98
  6.3. Extensibility .................................................. 106
  6.4. Usability ................................................... 108
  6.5. Other reconfiguration issues .................................. 109

7. DISCUSSION .................................................. 111
Chapter 1

INTRODUCTION

1.1. Motivation

“It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is most adaptable to change.”

(Charles Darwin)

The world is changing faster than ever before. Technology is improving at increasing rates. We are getting used to the fact that what today is brand new, might become obsolete within months. Markets are developing in the same direction. Consumers are growing more and more demanding, asking for an always improving supply, and for products and services that exactly fit their needs, if not surpassing them.

Here is where the Industry 4.0 comes into picture, bringing new production techniques and opening a wide range of possibilities that make it possible to adapt to this new complicated market. In order to address the current situation, flexibility is a crucial skill to be acquired by manufacturers.

Production techniques such as Agile Manufacturing and Just In Time are moving the Sales/Production relationship away from Make To Stock strategy and closer to Make To Order, giving the customer a increased choice power in the final product. However, these delays in decision making with regards to production normally carry enlarged production times and costs, thus reducing competitiveness.

In the view of these circumstances, and with lowering entry barriers in many industrial sectors that lead to an increasing competition, just the companies that prove efficiently adaptable to change will survive. However, it is not always easy to adapt the production
1. INTRODUCTION

to a changing market. An example for this are large production plants where stopping the process means incurring in high costs, or where due to especially long process ramp up times it is not possible to stop the process in order to make variations.

This conflict between adaptability to change and the need for reduced production costs and times remains an important concern for Industrial Automation. There is a need to make plants flexible without stopping them, and that also means being able to change the control logic behind while keeping the processes running.

A solution for this specific problem is the implementation of Dynamic Software Update (DSU), i.e. modifying the control software while it is running, reaching a zero-downtime. This would make it possible to change how a plant works without having to interrupt production, and therefore making software updates, downgrades or changes without significantly affecting production times, what could save a lot in costs.

There have already been made a number of approaches to this issue, based on different programming languages. One of these languages is Erlang, developed by Ericsson, proves really promising, since it offers a broad extent of functionalities that help changing and updating processes on run time. There has already been made implementations in Erlang for Control Automation, using some Erlang functionalities for DSU. However, these implementations still lack flexibility, being not possible to schedule the update point or to automatically set the reconfiguration steps order.

This project studies which different Erlang functionalities can be applied to DSU in order to generate control reconfiguration applications which overcome those drawbacks. This will be performed following the Industrial Automation standard IEC 61499, that provides software encapsulation, and therefore modularity and reusability. This feature helps generating tools for creating control reconfiguration applications that can be widely used to make process updates on run time.

Finally, a comparative analysis of the different DSU possibilities based in Erlang is to be performed, studying how Erlang can be further used for this purpose, based on the solutions presented. In this analysis, the proposed solution is compared to the current solutions implemented in Erlang.

1.2. Objectives

The main goal of the project is to study how Erlang functionalities can be used to generate control reconfiguration applications for the Industrial Automation field following the standard IEC 61499, by implementing Reconfiguration Services in Erlang, and comparing their performance with other Erlang methods.
In order to achieve this goal, a deep study of the IEC 61499 is made, in order to understand its modular structure and how it can help when it comes to DSU. An special focus on its approach to reconfiguration applications is applied, considering it as a base for the generation of reconfiguration application.

Afterwards, the programming language Erlang is studied, having in mind the current implementations, aiming to find new functionalities that can be encapsulated based on IEC 61499.

Once the entire situation is fully understood and evaluated, the necessary tools and algorithms for DSU are generated, so that they are generic and reusable. This will be done using Erlang and based on IEC 61499.

After the implementation of all the necessary tools, their performance is tested using different test cases, which other Erlang solutions also undergo.

Finally, the test results are evaluated through a comparative analysis of the different tools.

1.3. Document structure

The following chapters of the document are structured as follows:

Chapter 2: Background

In this chapter the current state of the art related to the topic is reviewed. It copes with the concept of DSU, the standard IEC 61499, and Erlang as a base for DSU, finishing with an explanation of Erlang implementations.

Chapter 3: Methodology

In this chapter, the steps followed to the completion of the project, as well as the tools used are discussed.

Chapter 4: Implementation of Reconfiguration Services in Erlang

This chapter explains the implementation of the Reconfiguration Systems in Erlang, exposing their interface, how they are used, and which Erlang functions are mainly used.

Chapter 5: Case Studies

This chapter presents the different systems used to test the different Erlang implementation approaches. That includes the solution implemented in this project, as well as the already implemented solution using Erlang appup files generated by the compiler FBBeam.
1. INTRODUCTION

Chapter 6: Comparative assessment

In this chapter, a comparison of the tested solutions is performed according to different criteria.

Chapter 7: Discussion

This chapter sums up and discuss the results obtained in the project.

Chapter 8: Conclusions

In this chapter, the conclusions of the given results are presented.

Chapter 9: Outlook and future work

This chapter outlines the steps to make following the conclusion of this project, addressing the things to improve in the current solutions and possible different uses of them.

Chapter 10: Project Planning

This final chapter presents the temporal planning and the budget of the project.
Chapter 2

BACKGROUND

In this chapter a revision of the current state of the art is performed, setting the project into its context. For that purpose, the situation of Dynamic Software Updating Methods is outlined: In addition, the standard and main methods to be used will be explained in detail.

2.1. Dynamic Software Update

DSU methods, are those means of updating software while running, without the need to stop its execution. It has been applied in many fields, since the need to update software is inevitable, due to the need of introducing software upgrades, changes or bug fixing.

There have been developed different approaches to this problem. One possible way of doing DSU is to have redundant hardware, and change from running a machine, to run a new one where the new version is installed. However, that would carry many problems, as the state of the first machine is lost when starting the second [1]. A better solution for this would be a software approach. There using different programming language and approaches. Some programming languages already include features for DSU [2]. Those languages are:

- Erlang
- Common LISP
- Smalltalk
- UpgradeJ
2. BACKGROUND

They provide a base for DSU, but the applications to be reconfigured must be entirely implemented in the used language.

DSU methods can be classified into software Development DSU and Production DSU, depending on if they are applied merely to software or to a production plant [3]. One of the fields where applying DSU improves the current production is Control Automation. It allows developing changes in the control applications for industrial processes offline, and then updating the processes without the need to completely stop them, this way reducing the downtime to almost zero.

The current approaches to Production DSU have a lot of limitations. They still affect the application performance, it is difficult to determine a safe point to perform the update, and they have limited abilities to determine the changes to be made during update. Some examples for current Production DSU solutions are Rubah or Pymoult [3].

2.2. IEC 61499

Industrial control systems have always been divided in two types: Programmable Logic Controllers (PLC) and Distributed Control Systems (DCS). A PLC based system has always consisted of a number of interconnected PLCs, whose state is displayed in a Human-Machine Interface. On the other hand, a DCS [5] consists of a central station which is in charge of controlling and supervising the different distributed instruments of the system [4].

Both approaches have always been linked to long monolithic code, that can be hardly updated or reused. However, over the past years, both concepts have converged in a more distributed approach in which each device has its own control system, controlled from a central station. As a result, an improvement in flexibility and solution reusability is reached [4].

In order to provide a norm to facilitate the implementation of distributed automation systems, the International Electrotechnical Commission (IEC) published in 2005 the standard IEC 61499 [5], based on the algorithm encapsulation in inter-connectable Function Blocks (FB), which are modules that represent functional units of software attached to a hardware device or resource from a control system [6]. The architecture proposed by the standard is detailed in the following section.
2.2.1. Architecture

The main element of the IEC 61499 architecture is the FB. It encapsulates and defines a piece of code at a higher level. It has a defined interface, consisting of input data provided to the FB and input events in charge of activating the FB. When activated, the code it contains is executed and as a result, output variables and events are delivered. If connected to other FB instances, they can pass them their output values or activate them via sending them events. Each FB can also include internal variables, which are completely protected, since they can not be accessed from the outside [7] [8].

Every FB has a type and an instance name. The type defines its structure, its I/O, internal algorithms and variables. [4] There can exist many instances of the same type, making the code reusable. As a result, each instance must have a unique name. Both the instance name and the type must be displayed in the FB interface [8].

The interface of a FB is outlined in Figure 2.1. The events are displayed in the upper part, the “Execution Control” part, while the data I/O are in the lower part of the FB, which contains the FB functionality. Each data I/O is associated to one or more event I/O by means of a WITH connection. Whenever an input event is received, their associated input data values are read and stored in the FB before the internal algorithm is executed. After being executed, the data outputs are updated. If the algorithm sends an event output, the data outputs connected to it are also sent to the following FB instances. [4]

There are different types of FB [10]:

- **Basic FB (BFB)**: It contains a state machine that defines the states and transitions the FB contains. These transitions are triggered by events, and in each state, an algorithm can be executed, with the possibility to trigger an output event. These states and transitions are mapped in a Execution Control Chart (ECC).

- **Composite FB (CFB)**: It is defined by a network of interconnected FB instances. The I/O of the CFB are passed to the internal FB instances and viceversa.
2. BACKGROUND

- Service Interface FB (SIFB): They are FB whose functionalities lay beyond the scope of IEC 61499.

FB instances are event-driven. They are just executed when receiving an input event [7]. They are combined and interconnected inside Applications and Subapplications, as depicted by the scheme in Figure 2.2. These applications are not necessarily associated to a single device, they can cover more than one device.

![Figure 2.2: Application Model [4]](image)

In order to work ad exchange system data between different providers and systems, two different ways are provided by the standard: a textual syntax, and using eXtensible Markup Language (XML) [8] [11].

There are currently various execution implementations in different programming languages for IEC 61499, given that it is just a high level description and offers flexibility for its implementation. These implementations can be sequential, parallel or cyclic. A sequential implementation is based on a purely event-driven conceptualization of the FBs. A cyclic approach is based on PLCs execution, so it would execute each FB on a cyclic manner. A parallel execution would involve the possibility of running different FBs at the same time, which is currently outdated [5].

2.2.2. IEC 61499 as a base for DSU

The fact that the IEC 61499 is based on FBs that are completely decoupled inside an application, without using global variables, makes it easier to reconfigure to a system, since every FB is completely independent from the others, and its update does not affect the rest. In addition, its event-driven approach is also a key feature for reconfiguration, making it easier to identify a safe state when to perform an update. [9].

Apart from the structural advantages IEC 61499 provides, it also outlines a series of possible SIFBs in charge of application reconfiguration. Each of these SIFBs is in charge
of performing a reconfiguration step on the system (e.g. creating a new FB instance) [12]. These Services could be combined in entire Reconfiguration Applications in charge of performing deeper updates. This application runs in parallel to the system to be updated performing the reconfiguration. For that purpose these special Reconfiguration Services are used to update processes on runtime.

2.2.3. 4diac IDE

There are many software approaches to IEC 61499. One of the most widely used ones when it comes to sequential approaches is 4diac Integrated Development Environment (IDE). It is an open source tool which allows to create systems according to IEC 64199. It is possible to create new FB types and instances, work with applications and subapplications, and includes a runtime environment called FORTE [13].

The runtime environment of 4diac is based on C++, but the algorithms of the FB can be implemented in other programming languages. Then, the system structure can be exported as XML files and used for further implementations.

2.3. Erlang

One of the current implementations of IEC 61499 is in the programming language Erlang. Erlang (Ericsson Language) is a programming language created by Ericsson, which was primarily used for communication systems. It is a functional language that is highly scalable, given that it works with lightweight processes. It can run anywhere in its virtual machine [14].

The different processes implemented with Erlang are stored in modules, which can run algorithms and communicate with each other with messages. This modularity makes this language a suitable option for an IEC 61499 FB implementation.

Another characteristic of Erlang is its Open Telecom Platform, which is a library that contains different common applications of Erlang, classified in behaviours. These behaviours store functionalities associated with a certain application. Then, callback modules must be created, which use the functionalities provided by the behaviour and add functionalities to it [15]. Most of Erlang developers use these behaviours, so it provides a way of code standardization that is also convenient for an IEC 61499 implementation.
2.3.1. Use of Erlang for DSU

An important functionality of Erlang is the possibility of Dynamic Updating, or as named in Erlang environment, Hot Code Loading. It allows updating code on runtime, without the need to stop the code. In order to do so, it is necessary to have some OTP applications, combined in a release. Then, a new version of the release is created, whose updated modules must contain the callback function `code_change` for changing the internal state of the module instances [16]. For updating the release code, create an appup file for each updated application. An appup file is a set of Erlang instructions to be performed during reconfiguration. It has to be written according to the instructions given by the Erlang documentation [15].

Once the new application versions and the corresponding appup files are created, the new release containing them is set up, from which a relup file is generated (it contains the instructions for the release upgrade). Finally the new application can be installed in a running Erlang Shell [14].

However, the use of releases is not strictly necessary to use Hot Code Loading. An appup file can be directly executed to update an specific application. Nevertheless, when performing bigger updates, it is safer to use releases.

When updating an application, if the internal state of a process must be modified, i.e. if its `code_change` function must be called, it is necessary to suspend the process. When suspended, it can still receive messages, but can not react to them until it is resumed. Suspending it avoids errors during reconfiguration due to a state change. In order to update a process with these characteristics, these steps are followed [16]:

1. Suspend the process.
2. Load the new module, change its internal state and upgrade to the new version.
3. Remove the old module.
4. Resume the suspended process

In case the internal state does not have to be updated, e.g. in a code extension, the new version can be compiled and updated without having to suspend it.

Apart from all functionalities Erlang offers for code updating, some OTP behaviours also provide with ways of updating processes on the runtime, e.g. adding or removing processes, suspending or resuming them, updating how the processes behave without the need to suspend anything. However, these functions modify processes but can not change the underlying code.
2.4. FBBeam

There are currently many IEC 61499 implementations. One of them is the implementation in Erlang by Prenzel and Provost [17]. This implementation takes advantage of the Erlang modularity and scalability. This approach uses the OTP behaviour `gen_statem` for a generic state machine. It models each FB type as an Erlang module following that behaviour. Each module contains some functions which are general for every FB, based in the behaviour functions, and others whose content depends on the internal data and algorithms of the FB ECC, which must be written in Erlang. Each of the modules contains as well a set of process instances, each of them corresponding to a FB instance of the corresponding type.

In order to generate the files according to this implementation, they created the compiler FBBeam. It is a compiler implemented in Python 3 that takes the XML files describing a system according to IEC 61499 and generates the corresponding Erlang files, based on templates. That set of files includes an Erlang `gen_statem` module for each FB type included in the system. Moreover, for each application, an Erlang application file is generated, as well as a supervisor, in charge of starting and controlling all the FB instances within the applications [18].

It also includes the possibility of using subapplications, from which their corresponding supervisors are also generated, as well as the use of SIFBs, for which their corresponding module should be written apart and included in the collection of the FBBeam SIFBs, so that it can be used by the compiler as a template. This way, any new functionality beyond the scope of IEC 61499 can be included.

Once the files are generated, everything must be compiled and run by the user, using Erlang commands.

2.4.1. FBBeam application updates

FBBeam offers currently also the possibility of automatically generating appup files. In order to do so, the new version of the system must be firstly created. The, its Erlang files must be generated using the compiler. Once both versions of the system have been generated in Erlang, they are taken as inputs for FBBeam, which compares both systems and generates an appup file from the differences between them. This functionality is not yet fully implemented, so it must suspend all the updated processes during the update [18].
2.5. Summary

As a summary of all the tools used and how they relate to each other, the explanatory Figure 2.3 shows in a visual manner the information gathered in this chapter.
Chapter 3

METHODOLOGY

In order to accomplish the project Objectives, it was necessary to set a methodology. It is important to take into account which resources are needed and out of them, which are available to use.

Moreover, choosing which norms and specifications are going to be followed is crucial to set a base for the methodology to be followed, and in order to make the work consistent with the current state of the art.

Finally, it proves also important to follow some guidelines and steps in order to have a path to follow and not getting away from the project goals.

3.1. Resources

For the completion of this project, the only resources needed were software-related, and are either free software, or already provided by the Technical University of Munich:

- Erlang shell V10.3, with Erlang OTP 21.
- Python 3.7, compatible with FBBBeam.
- A Code Editor for Erlang, Python and XML. In this case Visual Studio Code was used to work with all languages, due to its versatility and simplicity, and because it is easily available.
- IDE 4diac, in order to create the FB interfaces, as well as to create new systems to try the implemented code. This IDE will generate the necessary input files for FBBBeam.
3. METHODOLOGY

- Lynux OS in order to run the test cases, since it offers a more accurate internal clock.

3.2. Specifications

For the completion of this project, the standard IEC 61499 explained in Section 2.2 architecture has been followed in the implementation of systems and models, as well as for creating the interface of the implemented Reconfiguration Services.

It serves as a good basis for implementing Distributed Systems consistent with the current state of the art, and helps as a starting point to model modular automation systems with Erlang.

3.3. Steps followed

The steps to be followed are the ones represented in Figure 3.2.

1. Identification and classification of the Control Reconfiguration Services to be implemented. For that purpose the current norm and state of the art is revised, and adapted to the needs of a system implemented with FBBeam.

2. Implementation of the Control Reconfiguration Services of each specified class, making use of Erlang OTP functions.

3. For each class, a simple Reconfiguration Application including all its Services is created and tested in the system represented in Figure 3.1. The purpose of these small-scale test are intended to test if the implemented Services perform their tasks, and to test error handling at a small-scale, so that mistakes can be easily identified and solved. It consists of the following FBs:

![Diagram of FBs]

Figure 3.1: Testing System
Comparison of Dynamic Software updating methods for IEC 61499 with Erlang

- **start:** It is called to start the execution of the system.
- **cycle:** It is a CFB (treated by FBBeam as a subapplication) that sends its output event 'REQ' every period of time set by the input variable 'CT', given in milliseconds.
- **counter:** It counts the number of times it has received a 'REQ' input event. The current count is sent through the output variable 'CNT'.
- **print_any:** It prints in the console the current value of the counter, received in its input value 'IN'.

4. Once all the Services are implemented, greater test cases are designed and more complex reconfigurations are executed both using a Reconfiguration Application, and an Appup file generated by FBBeam. The goal of these test cases is to test the implemented solution at a larger scale, as well as checking other features the proposed Reconfiguration Applications offer. These test cases are also updated using the FBBeam appup files, for a later comparison.

5. Both methods are subjected to a comparative analysis in order to identify strengths and weaknesses of both methods. With that purpose, both methods are assessed based on different software characteristics:

- **Update Accuracy:** How correct is a certain method, i.e. how close to the expected result can its output be. In this case, the expected result is the new version of the system, therefore being accurate if the resulting system coincides with the planned new version of it.
- **Update Time:** How long does the update take. This time is considered from the time the Reconfiguration Application or the appup file are started, until the last step of the reconfiguration is performed.
- **Extendability:** How easy and simple is to add new functionalities to the assessed method.
- **Usability:** How simple and user-friendly is the solution.
- **Other reconfiguration issues:** They deal with other issues and features which are beyond the previously stated methods.
3. METHODOLOGY

Figure 3.2: Testing System
Chapter 4

IMPLEMENTATION OF RECONFIGURATION SERVICES IN ERLANG

In this chapter, an analysis and classification of the possible Reconfiguration Services is performed taking the state of the art as a base in order to determine which are necessary for this implementation. Afterwards, all the Reconfiguration Services are exposed, explaining their interface, how they work, and outlining the main Erlang functionalities they use to perform their reconfiguration tasks. Then, some small tests are performed to validate that the implemented services work as expected.

A reconfiguration process can be divided into different steps. Each of these steps performs a specific change in the running process, e.g. adding/deleting a FB or changing an input value. According to IEC 61499, different SIFBs must be provided in order to perform the different reconfiguration tasks. These SIFBs are then combined to build up a complete application that carries out the reconfiguration in parallel to the updated process. Subsequently, the FBBeam compiler transforms the XML files generated by 4diac into Erlang code, creating for every FB type a module following the Erlang behavior \texttt{gen_statem}.

Being the Reconfiguration Services SIFBs, the compiler does not generate all the code for them, but adds to a template just the code for the declaration of its instances. Therefore, the code for these Services must be written beforehand and stored as templates in FBBeam.

In this chapter, all the implementation of these Reconfiguration Services is explained, giving all the necessary information about its use and about how they work. First of all it is necessary to define which Services are needed. In the Appendix B of IEC 61499-4 [12] some examples for Reconfiguration Services are proposed, divided into nine different type of reconfiguration actions:
4. IMPLEMENTATION OF RECONFIGURATION SERVICES IN ERLANG

- **CREATE**: Introduce a new item in the system
- **DELETE**: Remove an item from the system
- **START**: Start an item
- **STOP**: Suspend an item
- **KILL**: Terminate an item
- **QUERY**: Request information about the system
- **READ**: Request current information about a FB instance
- **WRITE**: Change a value in a FB instance
- **RESET**: Set an item to its initial state and values

Based on the given examples for Reconfiguration Services, a new list was developed [9], with a different classification approach, dividing the given Services into categories related to which extent do they affect the system, and which kind of items do they address.

**Query Services**: Reconfiguration Services that request information about the control application.

**Execution Control Services**: Reconfiguration Services that change the execution state of a certain item.

**State Interaction Services**: Reconfiguration Services that allow interaction with the state of a control application, i.e. with the ECC state of a FB or its data I/O and Internal Variables.

**Structural Services**: Reconfiguration Services that change the structure of the application, adding, removing or connecting items.

**Library Services**: Reconfiguration Services that change the Type Library of the application

According to this classification and taking into account the different possible control actions proposed by IEC 61499, a Control Reconfiguration Services list was proposed by Zoitl [9]. This list is outlined in Table 4.1, along with its classification and a description of their functions.
### Table 4.1: Control Reconfiguration Services [9]

<table>
<thead>
<tr>
<th>Classification</th>
<th>Service Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query Services</td>
<td>QUERY_RESs</td>
<td>Request a list of the given resources and their types</td>
</tr>
<tr>
<td></td>
<td>QUERY_FBs</td>
<td>Request a list of the existing FBs in a given destination (application/resource)</td>
</tr>
<tr>
<td></td>
<td>QUERY_FB_STATE</td>
<td>Request the current state of a certain FB instance</td>
</tr>
<tr>
<td></td>
<td>QUERY_CONs</td>
<td>Request a list of all the connections of a FB</td>
</tr>
<tr>
<td></td>
<td>QUERY_TYPE</td>
<td>Request the type of a certain FB or resource instance</td>
</tr>
<tr>
<td></td>
<td>QUERY_TYPE_LIST</td>
<td>Request a list of the existing types in an application/resource</td>
</tr>
<tr>
<td>Execution Control Services</td>
<td>START</td>
<td>Start a FB instance</td>
</tr>
<tr>
<td></td>
<td>STOP</td>
<td>Suspend a FB instance</td>
</tr>
<tr>
<td></td>
<td>KILL</td>
<td>Terminate a FB instance</td>
</tr>
<tr>
<td></td>
<td>RESET</td>
<td>Reset a FB instance to its initial state</td>
</tr>
<tr>
<td>State Interaction Services</td>
<td>READ</td>
<td>Read the value of a data I/O or an Internal Variable</td>
</tr>
<tr>
<td></td>
<td>WRITE</td>
<td>Write a new value in a data I/O or an Internal Variable</td>
</tr>
<tr>
<td>Structural Services</td>
<td>CREATE_RES</td>
<td>Create a new resource of a certain type</td>
</tr>
<tr>
<td></td>
<td>CREATE_FB</td>
<td>Create a new FB instance</td>
</tr>
<tr>
<td></td>
<td>CREATE_CON</td>
<td>Create a new connection between FBs (data and event connections)</td>
</tr>
<tr>
<td></td>
<td>DELETE_RES</td>
<td>Delete a given resource</td>
</tr>
<tr>
<td></td>
<td>DELETE_FB</td>
<td>Delete a FB instance</td>
</tr>
<tr>
<td></td>
<td>DELETE_CON</td>
<td>Delete a connection between FBs</td>
</tr>
<tr>
<td></td>
<td>WRITE</td>
<td>Write a new value in a data I/O or an Internal Variable</td>
</tr>
<tr>
<td>Library Services</td>
<td>CREATE_TYPE</td>
<td>Create a new type in the type library</td>
</tr>
<tr>
<td></td>
<td>DELETE_TYPE</td>
<td>Remove a type from the type library</td>
</tr>
</tbody>
</table>
Having in mind the previously outlined Reconfiguration Services list when implementing them as Erlang Generic State Machines, it is found that some of them are not implementable, or that some are just not necessary due to how FBBeam generates the code for representing control applications. As a result, some of the mentioned services were left out of this implementation, outlined in Table 4.2, outlining as well the reasons for not considering them.

<table>
<thead>
<tr>
<th>Classification</th>
<th>Service Name</th>
<th>Reason for not implementing it</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Control Services</td>
<td>RESET</td>
<td>In the systems generated by FBBeam, when killing a process it is automatically restarted, acting like a reset</td>
</tr>
<tr>
<td>Structural Services</td>
<td>CREATE_RES</td>
<td>Resources are not considered in the implementation done by FBBeam</td>
</tr>
<tr>
<td></td>
<td>DELETE_RES</td>
<td></td>
</tr>
<tr>
<td>Query Services</td>
<td>QUERY_RESs</td>
<td></td>
</tr>
<tr>
<td>Library Services</td>
<td>CREATE_TYPE</td>
<td>Types are not loaded anywhere when using Erlang State Machines, so it would be related with having the corresponding file compiled or not, so they are not necessary</td>
</tr>
<tr>
<td></td>
<td>DELETE_TYPE</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Not Implemented Control Reconfiguration Services

On the other hand, the FBBeam implementation in Erlang results as well in a need of new Control Reconfiguration Services not considered before. First of all, the fact that all the CFB are considered Subapplications for FBBeam makes it necessary to have special Services to deal with them. In addition, the WRITE service is supposed to be able to change parameter values, as well as the ECC state of a FB. However, these two actions are performed differently, and therefore a separate Service is needed for changing the ECC state of a FB. A summary of the new Services included in this implementation and the reason of their inclusion are shown in Table 4.3.

Finally, as the Reconfiguration Service WRITE_STATE is considered as a State Interaction Service, because it interacts directly with the ECC of the FB, the Service QUERY_FB_STATE is also considered the same way, hence changing its name for READ_STATE.

Note: All the Reconfiguration Control Services implemented in this project have been named with the prefix “rec_.”
### Table 4.3: New introduced Control Reconfiguration Services

<table>
<thead>
<tr>
<th>Classification</th>
<th>Service Name</th>
<th>Reason for including it</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural Services</td>
<td>CREATE_SUBAPP</td>
<td>CFB are treated as Subapplications by FBBeam, so the commands to create/delete an instance are different as for BFB</td>
</tr>
<tr>
<td></td>
<td>DELETE_SUBAPP</td>
<td></td>
</tr>
<tr>
<td>State Interaction Services</td>
<td>WRITE_FB_STATE</td>
<td>Changing an ECC state of a FB requires specific commands, different from those used to write a parameter value</td>
</tr>
</tbody>
</table>

#### 4.1. Query Services

The Query Services are Control Reconfiguration Services that request information about the control application, e.g. which FB does it include, or if a certain FB is running or suspended. This information can be needed in order to trigger some parts of the Reconfiguration Application, or to use that data in the reconfiguration.

According to the definition of Query Services, and the way they are implemented in Erlang, five Control Reconfiguration Services are implemented, as discussed at the beginning of Section 4:

- **rec_QUERY_FBs**: It gives a list of all the current FB instances in an application.
- **rec_QUERY_FB_STATUS**: It gives the current Execution Status of a FB instance.
- **rec_QUERY_CON**: It gives a list of all the output connections of a FB instance.
- **rec_QUERY_TYPE_LIST**: It gives a list of all the FB types used in the given application.
- **rec_QUERY_TYPE**: It gives the type of a certain FB instance.

#### 4.1.1. Query a List with all the FB instances

**INTERFACE**

Figure 4.1 represents the interface of the Reconfiguration Service rec_QUERY_FBs. Its I/O are explained in Table 4.4.
4. IMPLEMENTATION OF RECONFIGURATION SERVICES IN ERLANG

![Diagram](image)

**Figure 4.1:** Interface of rec_QUERY_FBs

<table>
<thead>
<tr>
<th>INPUT EVENTS</th>
<th>OUTPUT EVENTS</th>
<th>INPUT VARIABLES</th>
<th>OUTPUT VARIABLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ</td>
<td>Event</td>
<td>Event to require a list with all the FB instances in an application</td>
<td></td>
</tr>
<tr>
<td>CNF</td>
<td>Event</td>
<td>Event to confirm that the list was provided</td>
<td></td>
</tr>
<tr>
<td>APP_NAME STRING</td>
<td>Name of the Application (atom)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STATUS STRING</td>
<td>Service Status: RDY, NO_SUCH_OBJECT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LIST STRING</td>
<td>List with all the FB instances in the app</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 4.4:** I/O in rec_QUERY_FBs

**ERLANG FUNCTIONALITIES USED**

In order to provide with a list of all the FB instances, this Reconfiguration Service uses the functions `supervisor:count_children/1`, and `supervisor:which_children/1`, creating with that information a list with all the names of the FB instances belonging to the given application.

If it can not be read because of a wrong application name given, the rec_QUERY_FBs catches the exception thrown and prints what the problem is.

**FUNCTIONING**

The Reconfiguration Service rec_QUERY_FBs provides with a list with all the FB instances in an application.

When this Service receives an input event ‘REQ’, it reads its input value ‘APP_NAME’. It must be provided as an atom (either enclosed with ‘ ’ or starting with lowercase).
After reading the input values, it starts an algorithm that gets a list of all the current FB instances in the provided application. If it is successfully read, the message “Number of FB in ‘APP_NAME’ = # : ‘LIST’ ” is printed, the Output Variable ‘STATUS’ is set to “RDY”, and the Output Variable ‘LIST’ sends a list with all the names of the current FB instances belonging to the application. If it is not possible to get the list, the Output Variable ‘STATUS’ is set to “NO SUCH OBJECT”. This can happen because the application given does not exist in the system. In this case the message “‘APP_NAME’ is not a valid application” is printed.

4.1.2. Query the process status of a FB

INTERFACE

Figure 4.2 represents the interface of the Reconfiguration Service rec_QUERY_FB_STATUS. Its I/O are explained in Table 4.5

![Interface of rec_QUERY_FB_STATUS](image)

**Figure 4.2: Interface of rec_QUERY_FB_STATUS**

<table>
<thead>
<tr>
<th>INPUT EVENTS</th>
<th>OUTPUT EVENTS</th>
<th>INPUT VARIABLES</th>
<th>OUTPUT VARIABLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ Event</td>
<td>CNF Event</td>
<td>APP_NAME STRING</td>
<td>STATUS STRING</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FB_NAME STRING</td>
<td>FB_STATUS STRING</td>
</tr>
</tbody>
</table>

Ainara Matey Benito
4. IMPLEMENTATION OF RECONFIGURATION SERVICES IN ERLANG

ERLANG FUNCTIONALITIES USED

In order to obtain the execution status of the FB, this Reconfiguration Service uses the function `sys:get_status/1`, and takes the wished value using pattern matching.

If it can not be read because of a wrong application or FB name given, the `rec_QUERY_FB_STATUS` catches the exception thrown and prints what the problem is.

FUNCTIONING

The Reconfiguration Service `rec_QUERY_FB_STATUS` reads the current ECC state of a FB instance.

When this Service receives an input event ‘REQ’, it reads the input values. All the input variables must be provided as atoms (either enclosed with ‘ ‘ or starting with lowercase). It is important to take into account that if the FB is inside a Subapplication, its name must be provided as “Subapplication Name.FB Name” in the input variable ‘FB_NAME’.

After reading the input values, it starts an algorithm that reads the current Execution Status. If it is successfully read, the message “FB ‘APP_NAME.FB_NAME’ is currently ‘FB_STATUS’” is printed, the Output Variable ‘STATUS’ is set to “RDY”, and ‘FB_STATUS’ to its corresponding status. The status can be “suspended” or “running”. A terminated FB is not considered, since it is automatically restarted by the supervisor.

If it is not possible to get its status, the Output Variable ‘STATUS’ is set to “NO_SUCH_OBJECT”. This can happen because the application or FB given does not exist in the system. In this case the message “APP_NAME.FB_NAME’ is not a valid FB” is printed.

4.1.3. Query the output connections of a FB

INTERFACE

Figure 4.3 represents the interface of the Reconfiguration Service `rec_QUERY_CON`. Its I/O are explained in Table 4.6

![Figure 4.3: Interface of rec_QUERY_CON](image)
Comparison of Dynamic Software updating methods for IEC 61499 with Erlang

<table>
<thead>
<tr>
<th>INPUT EVENTS</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ</td>
<td>Event</td>
<td>Event to require the output connections of a FB instance</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUTPUT EVENTS</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CNF</td>
<td>Event</td>
<td>Event to confirm that the connections were provided</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INPUT VARIABLES</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>APP_NAME</td>
<td>STRING</td>
<td>Name of the Application containing the FB (atom)</td>
</tr>
<tr>
<td>FB_NAME</td>
<td>STRING</td>
<td>Name of the FB instance (atom)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUTPUT VARIABLES</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS</td>
<td>STRING</td>
<td>Service Status: RDY, NO_SUCH_OBJECT</td>
</tr>
<tr>
<td>CON</td>
<td>STRING</td>
<td>Current Output connections of the FB</td>
</tr>
</tbody>
</table>

Table 4.6: I/O in rec_QUERY_CON

**ERLANG FUNCTIONALITIES USED**

In order to obtain the execution status of the FB, this Reconfiguration Service uses the function `sys:get_state/1`, and takes the wished value using pattern matching.

If it can not be read because of a wrong application or FB name given, the rec_QUERY_CON catches the exception thrown and prints what the problem is.

**FUNCTIONING**

The Reconfiguration Service rec_QUERY_CON reads the current output connections of a FB instance.

When this Service receives an input event ‘REQ’, it reads the input values. All the input variables must be provided as atoms (either enclosed with `' ' or starting with lowercase). It is important to take into account that if the FB is inside a Subapplication, its name must be provided as “Subapplication Name.FB Name” in the input variable ‘FB_NAME’.

After reading the input values, it starts an algorithm that reads the current output connections of the FB instance. If they are successfully read, the message “FB ‘APP_NAME.FB_NAME’ connected to: EVENT CONNECTIONS: ‘EO’, DATA CONNECTIONS: ‘DO’” is printed, the Output Variable ‘STATUS’ is set to “RDY”, and ‘CON’ to a tuple including both ‘EO’ and ‘DO’. ‘EO’ stands for Event Outputs, and ‘DO’ for Data Outputs. Only the output connections are considered, since they are the only ones the module itself stores, while the input connections are only stored in the instance of the sender FB.

If it is not possible to get its connections, the Output Variable ‘STATUS’ is set to
“NO_SUCH_OBJECT”. This can happen because the application or FB given does not exist in the system. In this case the message “‘APP_NAME.FB_NAME’ is not a valid FB” is printed.

4.1.4. Query a List with all the used FB types

INTERFACE

Figure 4.4 represents the interface of the Reconfiguration Service rec_QUERY_TYPE_LIST. Its I/O are explained in Table 4.7

![Diagram](image)

**Figure 4.4: Interface of rec_QUERY_TYPE_LIST**

<table>
<thead>
<tr>
<th>INPUT EVENTS</th>
<th></th>
<th>OUTPUT EVENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ</td>
<td>Event</td>
<td>Event to require a list with all the types in an application</td>
</tr>
<tr>
<td>CNF</td>
<td>Event</td>
<td>Event to confirm that the list was provided</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INPUT VARIABLES</th>
<th>OUTPUT VARIABLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>APP_NAME STRING</td>
<td>STATUS STRING</td>
</tr>
<tr>
<td>Name of the Application (atom)</td>
<td>Service Status: RDY, NO_SUCH_OBJECT</td>
</tr>
</tbody>
</table>

| LIST STRING | List with all the FB types in the app |

**Table 4.7: I/O in rec_QUERY_TYPE_LIST**

ERLANG FUNCTIONALITIES USED

In order to obtain a list of all the used types, this Reconfiguration Service uses the functions `supervisor:count_children/1`, and `supervisor:which_children/1`, creating with that information a list with all the type names of the FB instances belonging to the given application.

If it can not be read because of a wrong application name given, the rec_QUERY_TYPE_LIST
catches the exception thrown and prints what the problem is.

**FUNCTIONING**

The Reconfiguration Service `rec_QUERY_TYPE_LIST` provides with a list with all the FB types in an application.

When this Service receives an input event ‘REQ’, it reads its input value ‘APP_NAME’. It must be provided as an atom (either enclosed with ‘ ’ or starting with lowercase).

After reading the input values, it starts an algorithm that gets a list of all the current FB types in the provided application. If it is successfully done, the message “FB types in ‘APP_NAME’ = ‘LIST’” is printed, the Output Variable ‘STATUS’ is set to “RDY”, and the Output Variable ‘LIST’ sends a list with all the names of the current FB types belonging to the application. It must be considered, that those types generated by FBBeam have the prefix “mod_”.

If it is not possible to get the list, the Output Variable ‘STATUS’ is set to “NO_SUCH_OBJECT”. This can happen because the application given does not exist in the system. In this case the message “‘APP_NAME’ is not a valid application” is printed.

**4.1.5. Query the type of a FB**

**INTERFACE**

Figure 4.5 represents the interface of the Reconfiguration Service `rec_QUERY_TYPE`. Its I/O are explained in Table 4.8

![Figure 4.5: Interface of rec_QUERY_TYPE](image-url)
4. IMPLEMENTATION OF RECONFIGURATION SERVICES IN ERLANG

### INPUT EVENTS

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ</td>
<td>Event to require the type of a FB instance</td>
</tr>
</tbody>
</table>

### OUTPUT EVENTS

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNF</td>
<td>Event to confirm that the type has been provided</td>
</tr>
</tbody>
</table>

### INPUT VARIABLES

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APP_NAME</td>
<td>STRING</td>
<td>Name of the Application containing the FB (atom)</td>
</tr>
<tr>
<td>FB_NAME</td>
<td>STRING</td>
<td>Name of the FB instance (atom)</td>
</tr>
</tbody>
</table>

### OUTPUT VARIABLES

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS</td>
<td>STRING</td>
<td>Service Status: RDY, NO_SUCH_OBJECT</td>
</tr>
<tr>
<td>TYPE</td>
<td>STRING</td>
<td>FB Type</td>
</tr>
</tbody>
</table>

Table 4.8: I/O in rec_QUERY_TYPE

### ERLANG FUNCTIONALITIES USED

In order to obtain the type of the FB instance, this Reconfiguration Service uses the function `sys:get_status/1`, and extracts the necessary data using pattern matching and the function `lists:keyfind/3`, since the information regarding the type is not always located in the same position.

If it can not be read because of a wrong application or FB name given, the rec_QUERY_TYPE catches the exception thrown and prints what the problem is.

### FUNCTIONING

The Reconfiguration Service rec_QUERY_TYPE gets the type of a certain FB instance.

When this Service receives an input event 'REQ', it reads its input values. They must be provided as atoms (either enclosed with ' ' or starting with lowercase).

After reading the input values, it starts an algorithm that gets the current type of a certain FB instance of the provided application. If it is successfully done, the message “Type of ‘APP_NAME.FB_NAME’ = ‘TYPE’” is printed, the Output Variable ‘STATUS’ is set to “RDY”, and the Output Variable ‘TYPE’ sends the type of the corresponding FB instance. It must be considered, that those types generated by FBBeam have the prefix “mod_”.

If it is not possible to get the type, the Output Variable ‘STATUS’ is set to “NO_SUCH_OBJECT”. This can happen because the application or FB name given does not exist in the system.
In this case the message “‘APP_NAME.FB_NAME’ is not a valid FB” is printed.

4.1.6. Testing of Query Services

In order to test the Query Services, the system represented in Figure 4.6 was reconfigured. A simple system was used, since the purpose of this test is to make sure that the Query Services work as they are expected to.

![Figure 4.6: Testing System](image)

The Reconfiguration Application tested is the one depicted in Figure 4.7. It includes all the Query Services implemented.

![Figure 4.7: Reconfiguration Application for Query Services](image)

In this case, the reconfiguration performed does not make any change in the system, since it is tests just the Query Services, which request information about the system, without performing any change. The first three Query Services request information about a certain FB instance (it was tested with the counter FB instance, and with a FB instance inside the subapplication cycle). The requested information consists of its output connections, its type and its execution status, respectively. Then, the last two FBs request a list of the FB instances in the application and a list with all the types used in the application.
This application is started in parallel to the execution of the reconfigured system by sending a ‘START’ event to the FB `start`. Then, the different Query Services were applied to the system application and to one of its FB instances, requesting all the possible information made available by the Query Services.

**RESULT**

The test system and test reconfiguration application were able to demonstrate the ability of implementing the IEC 61499 Query Services in Erlang. This Reconfiguration Application was also tested giving incorrect values to their inputs, and all the Query Services responded as expected, informing about the error and blocking the execution flow of the reconfiguration.

The three first services, which address a particular FB instance were tested setting as an input value the name of the BFB `counter` as well as with the instance `delay`, which is part of the Subapplication `cycle`, also resulting in a correct performance. It was tested with both to prove that they also work when accessing the information inside a Subapplication.

### 4.2. Execution Control Services

The Execution Control Services are those Reconfiguration Services in charge of modifying the normal execution flow of a certain FB instance. According to the norm IEC 61499, the state machine of a FB comprises four execution states, as discussed at the beginning of Section 4:

- **IDLE**: Initial state reached once the FB instance has been created. In this state all the variables are initialized with their initial values and initial ECC state.
- **RUNNING**: After starting the FB it gets to this state, in which the corresponding ECC is executed.
- **STOPPED**: In this state the FB execution is suspended, but it remains in the same state it was at the moment it was stopped. This way, it can be restarted at the same execution point, preserving the same values for its I/O and Internal Variables.
- **KILLED**: While the FB is running, it can be terminated and set to this state, in which the process is killed and must be reset before being restarted. Once it has been killed, the FB can be deleted.

When implementing a FB with Erlang using the `gen_statem` behavior, its execution flow changes due to the control action taken by the corresponding supervisor, whose restart strategy is “permanent”. This means that whenever a FB is killed it will be automatically
Figure 4.8: Execution of a Function Block State Machine according to IEC 61499 [8]

restarted. Considering the same states mentioned before, the following variations can be remarked:

- As long as its supervisor is running, when creating a new FB it will be automatically initialized and started by the supervisor. This way it goes straight to the state “RUNNING” without waiting to be started in “IDLE”.

- When killing a FB it will be terminated and automatically restarted by the supervisor with its initial values.

- A FB can be deleted while in “RUNNING”, or “SUSPENDED”. When its deletion is required, it is first terminated and then deleted.

- The action “Reset” is not needed, since a FB is always initialized and restarted after terminated. Therefore, killing a FB leads to the same result as restarting it.

A scheme of the explained execution flow of the FB created by FBBeam is depicted in 4.9.

According to the stated Execution Flow, three Control Reconfiguration Services are implemented:
4. IMPLEMENTATION OF RECONFIGURATION SERVICES IN ERLANG

**Figure 4.9:** Execution of a Function Block State Machine according to FBBeam implementation

- **rec_STOP:** It suspends a running FB.
- **rec_START:** It starts again a suspended FB.
- **rec_KILL:** It terminates a running or suspended FB.

As a terminated FB will be automatically initialized and restarted, it works exactly as resetting it, so a “RESET” FB is not needed. The control actions “CREATE” and “DELETE” are considered Structural Services, so their implementation is addressed in the section 4.3.

4.2.1. **Stop a FB**

**INTERFACE**

Figure 4.10 represents the interface of the Reconfiguration Service rec_STOP. Its I/O are explained in Table 4.9
**Comparison of Dynamic Software updating methods for IEC 61499 with Erlang**

Figure 4.10: Interface of rec_STOP

<table>
<thead>
<tr>
<th>INPUT EVENTS</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ</td>
<td>Event</td>
<td>Event to require suspending a FB instance</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUTPUT EVENTS</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CNF</td>
<td>Event</td>
<td>Event to confirm that the FB has been suspended</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INPUT VARIABLES</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>APP_NAME</td>
<td>STRING</td>
<td>Name of the Application containing the FB (atom)</td>
</tr>
<tr>
<td>FB_NAME</td>
<td>STRING</td>
<td>Name of the FB instance (atom)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUTPUT VARIABLES</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS</td>
<td>STRING</td>
<td>Service Status: RDY, NO_SUCH_OBJECT</td>
</tr>
</tbody>
</table>

Table 4.9: I/O in rec_STOP

**ERLANG FUNCTIONALITIES USED**

The algorithm in charge of suspending the FB, calls the Erlang function `sys:suspend/1`, whose only argument is the id of the process, in this case `APP_NAME.FB_NAME`. It returns 'ok' if its execution was successful. If the called FB does not exist, an exit exception is caught by rec_STOP.

**FUNCTIONING**

The Reconfiguration Service rec_STOP suspends the execution of a FB instance that was running, preserving the status and all its current values.

When this Service receives an input event ‘REQ’, it reads the input values. Both the ‘APP_NAME’ and the ‘FB_NAME’ must be provided as atoms (either enclosed with ‘ ’ or starting with lowercase). It is important to take into account that if the destination FB is inside a Subapplication, its name must be provided as “Subapplication Name.FB Name” in the input ‘FB_NAME’. Then, it starts an algorithm which suspends the corresponding FB. If it is successfully suspended, the message “APP_NAME.FB_NAME was STOPPED” is printed and the Output Variable ‘STATUS’ is set to “RDY”. If the FB instance was already suspended, it returns the same output.
If an Input Value is wrong, e.g. a non-existent application name is provided or it is not correctly written, the message "APP_NAME.FB_NAME is not a valid FB" is printed and the Output Variable 'STATUS' is set to "NO_SUCH_OBJECT".

4.2.2. Restart a FB

INTERFACE

Figure 4.11 represents the interface of the Reconfiguration Service rec_START. Its I/O are explained in Table 4.10

![Interface of rec_START](image)

**Figure 4.11: Interface of rec_START**

<table>
<thead>
<tr>
<th>INPUT EVENTS</th>
<th>OUTPUT EVENTS</th>
<th>INPUT VARIABLES</th>
<th>OUTPUT VARIABLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ</td>
<td>CNF</td>
<td>APP_NAME</td>
<td>STATUS</td>
</tr>
<tr>
<td>Event</td>
<td>Event</td>
<td>STRING</td>
<td>STRING</td>
</tr>
<tr>
<td>Event to require resuming a FB instance</td>
<td>Event to confirm that the FB has been resumed</td>
<td>Name of the Application containing the FB (atom)</td>
<td>Service Status: RDY, NO_SUCH_OBJECT</td>
</tr>
</tbody>
</table>

**Table 4.10: I/O in rec_START**

ERLANG FUNCTIONALITIES USED

The algorithm in charge of suspending the FB, calls the Erlang function `sys:resume/1`, whose only argument is the id of the process, in this case ‘APP_NAME.FB_NAME’. It returns 'ok' if its execution was successful. If the called FB does not exist, an exit exception is caught by rec_START.
FUNCTIONING

The Reconfiguration Service rec_START resumes the execution of a FB instance that was previously suspended, preserving the status and all the values it had as it was suspended.

When this Service receives an input event ‘REQ’, it reads the input values. Both the ‘APP_NAME’ and the ‘FB_NAME’ must be provided as atoms (either enclosed with ‘ ’ or starting with lowercase). It is important to take into account that if the destination FB is inside a Subapplication, its name must be provided as “Subapplication Name.FB_Name” in the input ‘FB_NAME’. Then, it starts an algorithm which restarts the corresponding FB instance. If it is successfully restarted, the message “APP_NAME.FB_NAME was STARTED” is printed and the Output Variable ‘STATUS’ is set to “RDY”. If the FB instance was already running, it returns the same output.

If an Input Value is wrong, e.g. a non-existent application name is provided or it is not correctly written, the message “APP_NAME.FB_NAME is not a valid FB” is printed and the Output Variable ‘STATUS’ is set to “NO_SUCH_OBJECT”.

4.2.3. Kill a FB

INTERFACE

Figure 4.12 represents the interface of the Reconfiguration Service rec_KILL. Its I/O are explained in Table 4.11

![Interface of rec_KILL](image)

Figure 4.12: Interface of rec_KILL

ERLANG FUNCTIONALITIES USED

The algorithm in charge of suspending the FB, calls the Erlang function `sys:terminate/1`, being the first argument the id of the process, in this case ‘APP_NAME.FB_NAME’. The second argument corresponds to the reason for the termination, in this case set as ‘shutdown’. It returns ‘ok’ if its execution was successful. If the called FB does not exist, an exit exception is caught by rec_KILL.
4. IMPLEMENTATION OF RECONFIGURATION SERVICES IN ERLANG

Table 4.11: I/O in rec_KILL

<table>
<thead>
<tr>
<th>INPUT EVENTS</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ</td>
<td>Event</td>
<td>Event to require terminating a FB instance</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUTPUT EVENTS</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CNF</td>
<td>Event</td>
<td>Event to confirm that the FB has been terminated</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INPUT VARIABLES</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>APP_NAME</td>
<td>STRING</td>
<td>Name of the Application containing the FB (atom)</td>
</tr>
<tr>
<td>FB_NAME</td>
<td>STRING</td>
<td>Name of the FB instance (atom)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUTPUT VARIABLES</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS</td>
<td>STRING</td>
<td>Service Status: RDY, NO_SUCH_OBJECT</td>
</tr>
</tbody>
</table>

FUNCTIONING

The Reconfiguration Service rec_KILL terminates the execution of a FB instance that was previously running or suspended. Given that the restarting strategy of the supervisor is “permanent”, after being terminated, it is automatically initialized and restarted by the supervisor, so it works as if it was reset.

When this Service receives an input event ‘REQ’, it reads the input values. Both the ‘APP_NAME’ and the ‘FB_NAME’ must be provided as atoms (either enclosed with ‘ ’ or starting with lowercase). It is important to take into account that if the destination FB is inside a Subapplication, its name must be provided as “Subapplication Name.FB Name” in the input ‘FB_NAME’. Then, it starts an algorithm which terminates the corresponding FB instance.

If it is successfully terminated, the message “APP_NAME.FB_NAME was KILLED” is printed and the Output Variable ‘STATUS’ is set to “RDY”. If an Input Value is wrong, e.g. a non-existent application name is provided or it is not correctly written, the message “APP_NAME.FB_NAME is not a valid FB” is printed and the Output Variable ‘STATUS’ is set to “NO_SUCH_OBJECT”.

4.2.4. Testing of Execution Control Services

In order to test the Execution Control Services, the system represented in Figure 4.13 was reconfigured. A simple system was used, since the purpose of this test is to make sure that the Execution Control Services work as they are expected to. The Reconfiguration Application tested is the one depicted in Figure 4.14. It includes all the Execution Control Services implemented.
This application is started in parallel to the execution of the reconfigured system by sending a ‘REQ’ event to the FB `start`. Then, the counter in the test system is suspended, and a delay of 4 seconds is applied. During this time, the whole system should be on hold, since the suspended counter blocks the flow of execution.

Afterwards, the counter is resumed, preserving its previous count value. After 4 seconds, the counter is terminated, and then restarted by its supervisor. Once restarted, the count resumes, but starting from 0 again.

**RESULT**

The test system and test reconfiguration application were able to demonstrate the ability of implementing the IEC 61499 Execution Control Services in Erlang. This Reconfiguration Application was also tested giving for the values ‘APP_NAME’ and ‘FB_NAME’ incorrect values, and all the Execution Control Services responded as expected, informing about the error and blocking the execution flow of the reconfiguration.

### 4.3. Structural Services

The Structural Services are those Reconfiguration Services that allow changing the structure of the control system. The structure refers to which FBs and Subapplications is the control system composed of, and how are they connected to each other.

Therefore, the Structural Services make it possible to modify this structure by adding
and removing new FBs and Subapplications. They also allow creating and deleting both data and event connections between those items.

According to the definition of Structural Services and the way systems are generated in Erlang, six Control Reconfiguration Services are implemented, as discussed at the beginning of Section 4:

- **rec_CREATE_FB**: It creates a new FB instance.
- **rec_DELETE_FB**: It deletes an existing FB instance.
- **rec_CREATE_CON**: It creates a new connection among two FBs.
- **rec_DELETE_CON**: It deletes an existing connection among two FBs.
- **rec_CREATE_SUBAPP**: It creates a new Subapplication.
- **rec_DELETE_SUBAPP**: It deletes an existing Subapplication.

### 4.3.1. Create a new FB

**INTERFACE**

Figure 4.15 represents the interface of the Reconfiguration Service recCREATE_FB. Its I/O are explained in Table 4.12.
Comparison of Dynamic Software updating methods for IEC 61499 with Erlang

**INPUT EVENTS**

| REQ | Event | Event to require creating a FB instance |

**OUTPUT EVENTS**

| CNF | Event | Event to confirm that the FB has been created |

**INPUT VARIABLES**

| FB_TYPE | STRING | Type of the FB instance (atom) |
| FB_NAME | STRING | Name of the FB instance (atom) |
| APP_NAME | STRING | Name of the Application containing the FB (atom) |

**OUTPUT VARIABLES**

| STATUS | STRING | Service Status: RDY, NO_SUCH_OBJECT |

| Table 4.12: I/O in recCREATE_FB |

**ERLANG FUNCTIONALITIES USED**

In every Erlang module corresponding to each type, an instance called ‘aux_instance’ is always created by FBBEam. It is an instance of that type without connections, and whose initial values are all ‘0’.

The algorithm in charge of creating the FB instance, calls the Erlang function **MODULE: instance_args/1**, being its only argument the id of the auxiliary instance of that module (‘aux_instance’). It returns the initial values needed to start a new instance.

Afterwards, the Specification for the new child is created, by storing in a tuple the following values:

- **Instance Name**: Generated from the input values, ‘APP_NAME.FB_NAME’.
- **Start Function**: a tuple composed by the module name corresponding to such FB type, the start function name ‘start_link’, and its argument, which is a list formed by the instance name and the initial values from ‘aux_instance’.
- **Restarting mode**: ‘permanent’.
- **Shutdown time**: time to wait if it can not be created, in this case 5000 ms.
- **Child type**: As it is a BFB, it is set as ‘worker’.
- **Modules where it is located**: module name corresponding to such FB type.

Finally, the function **supervisor:start_child/2** is called, being the first argument the supervisor name `sup_APP_NAME` and the second argument the specifications tuple.

Ainara Matey Benito
If it can not be created because the type or the application name given are wrong, or because the FB already exists, the rec_CREATE_FB catches the corresponding exception and prints what the problem is.

**FUNCTIONING**

The Reconfiguration Service rec_CREATE_FB creates a new FB instance as a child of a supervisor.

When this Service receives an input event ‘REQ’, it reads the input values. Both the ‘APP_NAME’ and the ‘FB_NAME’ must be provided as atoms (either enclosed with ‘ ’ or starting with lowercase). It is important to take into account that if the destination FB is inside a subapplication, the ‘APP_NAME’ must be the name of that subapplication. Moreover, if there is no FB of that type in the system, i.e. if the files for the corresponding type have not been created and compiled, an instance of that type should be included in the Reconfiguration Application, so that it can be read and generated by FBBeam. This FB instance must not be renamed or connected to any other FB.

After reading the input values, it starts an algorithm which creates the corresponding FB instance. If it is successfully created, the message “APP_NAME.FB_NAME WAS SUCCESSFULLY CREATED” is printed and the Output Variable ‘STATUS’ is set to “RDY”. If it is not possible to create it, Output Variable ‘STATUS’ is set to “NO_SUCH_OBJECT”. This can happen due to one of the following reasons:

- The type does not exist: in this case the message “The type FB_TYPE’ is not valid, ‘APP_NAME.FB_NAME’ can’t be created” is printed.
- The FB instance already exists in the system: in this case the message “The FB ‘APP_NAME. FB_NAME’ is already started” is printed.
- The application given does not exist in the system: in this case the message “The application ‘APP_NAME’ is not valid, ‘APP_NAME. FB_NAME’ can’t be created” is printed.

### 4.3.2. Delete an existing FB

**INTERFACE**

Figure 4.16 represents the interface of the Reconfiguration Service rec_DELETE_FB. Its I/O are explained in Table 4.13
Comparison of Dynamic Software updating methods for IEC 61499 with Erlang

Figure 4.16: Interface of rec_DELETE_FB

<table>
<thead>
<tr>
<th>INPUT EVENTS</th>
<th></th>
<th>OUTPUT EVENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ</td>
<td>Event</td>
<td>CNF</td>
<td>Event</td>
</tr>
<tr>
<td></td>
<td>Event to require deleting a FB instance</td>
<td>Event to confirm that the FB has been deleted</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INPUT VARIABLES</th>
<th>OUTPUT VARIABLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>FB_NAME STRING</td>
<td>STATUS STRING</td>
</tr>
<tr>
<td>APP_NAME STRING</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.13: I/O in rec_DELETE_FB

ERLANG FUNCTIONALITIES USED

The function `supervisor:terminate_child/2` is called, being the first argument the supervisor name `sup:’APP_NAME’` and the second argument the id of the child `APP_NAME’.FB_NAME’`. This function terminates the FB instance in its application.

Finally, the function `supervisor:delete_child/2` is called, with the same arguments as the ones given to supervisor:terminate_child/2. This function deletes the FB instance from its supervisor, and therefore from the app.

If it can not be deleted because the called FB or application do not exist, the rec_DELETE_FB catches the corresponding exception and prints what the problem is.

FUNCTIONING

The Reconfiguration Service rec_DELETE_FB deletes an existing FB from its parent application.

When this Service receives an input event ‘REQ’, it reads the input values. Both the
4. IMPLEMENTATION OF RECONFIGURATION SERVICES IN ERLANG

‘APP_NAME’ and the ‘FB_NAME’ must be provided as atoms (either enclosed with ‘ ’ or starting with lowercase). It is important to take into account that if the destination FB is inside a subapplication, the ‘APP_NAME’ must be the name of that subapplication.

After reading the input values, it starts an algorithm which deletes the corresponding FB instance. If it is successfully deleted, the message “APP_NAME.FB_NAME WAS SUCCESSFULLY DELETED” is printed and the Output Variable ‘STATUS’ is set to “RDY”. If it is not possible to delete it, Output Variable ‘STATUS’ is set to “NO_SUCH_OBJECT”. This can happen due to one of the following reasons:

- The FB instance does not exist in the provided application: in this case the message “The FB ‘APP_NAME.FB_NAME’ is not valid, it can’t be deleted” is printed.

- The application given does not exist in the system: in this case the message “The application ‘APP_NAME’ is not valid, ‘APP_NAME.FB_NAME’ can’t be deleted” is printed.

4.3.3. Create a new connection between FBs

INTERFACE

Figure 4.17 represents the interface of the Reconfiguration Service rec_CREATE_CON. Its I/O are explained in Table 4.14

![Figure 4.17: Interface of rec_CREATE_CON](image)

ERLANG FUNCTIONALITIES USED

In each FB instance, its output connections are stored in two records, one for event connections and one for data connections. Inside of each record, the connections set for each output are stored as records of type conx. They are kept in the correct order so that every output has its position in the list. If there is no connection defined for an output
Comparison of Dynamic Software updating methods for IEC 61499 with Erlang

### Table 4.14: I/O in rec.CREATE_CON

<table>
<thead>
<tr>
<th>INPUT EVENTS</th>
<th>OUTPUT EVENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ Event</td>
<td>CNF Event</td>
</tr>
<tr>
<td>Event to require creating a connection</td>
<td>Event to confirm that the connection has been created</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INPUT VARIABLES</th>
<th>OUTPUT VARIABLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE_CON STRING</td>
<td>STATUS STRING</td>
</tr>
<tr>
<td>Type of connection: data/event</td>
<td>Service Status: RDY, NO_SUCH_OBJECT</td>
</tr>
<tr>
<td>SRC_APP_NAME STRING</td>
<td></td>
</tr>
<tr>
<td>Name of the application that contains the Source FB (atom)</td>
<td></td>
</tr>
<tr>
<td>SRC_FB_NAME STRING</td>
<td></td>
</tr>
<tr>
<td>Name of the Source FB instance (atom)</td>
<td></td>
</tr>
<tr>
<td>SRC_FB_PARAM STRING</td>
<td></td>
</tr>
<tr>
<td>Name of the connected output in the Source FB (atom)</td>
<td></td>
</tr>
<tr>
<td>DST_APP_NAME STRING</td>
<td></td>
</tr>
<tr>
<td>Name of the application that contains the Destination FB (atom)</td>
<td></td>
</tr>
<tr>
<td>DST_FB_NAME STRING</td>
<td></td>
</tr>
<tr>
<td>Name of the Destination FB instance (atom)</td>
<td></td>
</tr>
<tr>
<td>DST_FB_PARAM STRING</td>
<td></td>
</tr>
<tr>
<td>Name of the connected input in the Destination FB (atom)</td>
<td></td>
</tr>
</tbody>
</table>

value, the atom ‘undefined’ substitutes the corresponding record. If there is more than one connection coming out of an output, they are stored as a list of records.

As a result of how connections are stored as internal data of the FB instance, and being its record types specific for FBBBeam implementation, there is no Erlang function to directly change the connection list. That is why it should be done from the module itself, with a specific function that is called from the Reconfiguration Service.

For this reason, the compiler FBBBeam has been extended, so that it adds a declaration of the function `Module:handle_event/4` in charge of changing the internal data of a FB instance. The function is triggered when receiving a call with a suitable message form another process. Given that it handles calls, it gives always an answer to the calling FB, thus communicating if the change was successful. For the implementation this function, a function clause is added for every data to be changed (not only connections, but also variable values). Every clause for this function has the following arguments:

- Event Type: a tuple `{call, From}`, receiving in the variable `From` the pid of the process sending the call.
Message: a record of the type msg, which stores the necessary information for the needed change. This message is used both for passing the values needed and for pattern matching, i.e. or accessing the correct clause depending on the arguments given. This way, different changes can be performed depending on the arguments of the sent message.

State: The current ECC state.

Data: The current internal data of the FB instance, containing the Internal Data and the Output Memory

An extra clause is also added, which receives every kind of message, in order to make exception handling easier. If this clause is called, it answers with an error message.

In this case, for adding a connection, a function clause is added for each output connection. In each clause, the corresponding connection record is updated using functions from the STDLIB module lists.

The needed function clause is called from the Reconfiguration Service rec_CREATE_CON by making use of the function gen_statem:call/4. The message sent in this case is of the form \{create_con, event/data, Value\}. In the variable Value, the data related to the connection is stored, including the corresponding output event/data name, so that its matching clause is called, and the correct connection is added.

If it can not be created because either the source application, FB, or the parameter do not exist, the rec_CREATE_CON catches the exception thrown and prints what the problem is.

FUNCTIONING

The Reconfiguration Service rec_CREATE_CON creates a new connection between two FBs. It can be an event or a data connection.

When this Service receives an input event ‘REQ’, it reads the input values. All the input variables must be provided as atoms (either enclosed with ‘ ’ or starting with lowercase). It is important to take into account that if the source/destination FB is inside a Subapplication, its name must be provided as “Subapplication Name.FB Name” in the input variable ‘FB_NAME’.

After reading the input values, it starts an algorithm which creates the corresponding connection. If it is successfully created, the message “Connection created: ‘SRC_APP_NAME. SRC_FB_NAME’: ‘SRC_PARAM’ —> ‘DST_APP_NAME.DST_FB_NAME’: ‘DST_PARAM’” is printed and the Output Variable ‘STATUS’ is set to “RDY”. If it is not possible to create it, Output Variable ‘STATUS’ is set to “NO_SUCH_OBJECT”. This can happen
Comparison of Dynamic Software updating methods for IEC 61499 with Erlang

due to one of the following reasons:

- The parameter given for the source FB does not exist: in this case the message "Parameter ‘SRC_PARAM’ of ‘APP_NAME.FB_NAME’ not valid" is printed.

- The source application or FB given does not exist in the system: in this case the message "‘APP_NAME.FB_NAME’ is not a valid FB" is printed.

If an already existing connection is added, it will not be added, because if it was, that connection would send the event/data twice each time.

**ENCOUNTERED PROBLEMS**

Since this implementation only accesses the FB instance which is the source of the connection, there is no way to know whether the data regarding the Destination of the connection is correct, i.e. if the destination FB instance or its given parameter exist. For this reason, giving incorrect data for the destination do not give an error immediately, but can lead to future errors when making use to that connection.

For the same reason, type check is not performed, since it needs to access the data type of the destination parameter, and it currently does not have access to that information.

Taking this into account, it is important to be careful when using this Reconfiguration Service, and to make sure that the destination parameters are correctly given.

### 4.3.4. Delete a connection between FBs

**INTERFACE**

Figure 4.18 represents the interface of the Reconfiguration Service rec_DELETE_CON. Its I/O are explained in Table 4.15

![Figure 4.18: Interface of rec_DELETE_CON](image_url)
4. IMPLEMENTATION OF RECONFIGURATION SERVICES IN ERLANG

**INPUT EVENTS**

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ</td>
<td>Event to require deleting a connection</td>
</tr>
</tbody>
</table>

**OUTPUT EVENTS**

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNF</td>
<td>Event to confirm that the connection has been deleted</td>
</tr>
</tbody>
</table>

**INPUT VARIABLES**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE_CON</td>
<td>STRING</td>
<td>Type of connection: data/event</td>
</tr>
<tr>
<td>SRC_APP_NAME</td>
<td>STRING</td>
<td>Name of the application that contains the Source FB (atom)</td>
</tr>
<tr>
<td>SRC_FB_NAME</td>
<td>STRING</td>
<td>Name of the Source FB instance (atom)</td>
</tr>
<tr>
<td>SRC_FB_PARAM</td>
<td>STRING</td>
<td>Name of the connected output in the Source FB (atom)</td>
</tr>
<tr>
<td>DST_APP_NAME</td>
<td>STRING</td>
<td>Name of the application that contains the Destination FB (atom)</td>
</tr>
<tr>
<td>DST_FB_NAME</td>
<td>STRING</td>
<td>Name of the Destination FB instance (atom)</td>
</tr>
<tr>
<td>DST_FB_PARAM</td>
<td>STRING</td>
<td>Name of the connected input in the Destination FB (atom)</td>
</tr>
</tbody>
</table>

**OUTPUT VARIABLES**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS</td>
<td>STRING</td>
<td>Service Status: RDY, NO_SUCH_OBJECT</td>
</tr>
</tbody>
</table>

**Table 4.15: I/O in rec_DELETE_CON**

**ERLANG FUNCTIONALITIES USED**

This Reconfiguration Service also makes use of the call handling explained in 4.3.3.

In this case, for deleting a connection, a function clause is added for each output connection. In each clause, the corresponding connection record is updated using functions from the STDLIB module **lists**.

The needed function clause is called from the Reconfiguration Service rec_DELETE_CON by making use of the function **gen_statem:call/4**. The message sent in this case is of the form `{delete_con, event/data, Value}`. In the variable **Value**, the data related to the connection is stored, including the corresponding output event/data name, so that its matching clause is called, and the correct connection can be deleted.

If it can not be deleted because either the source application, FB, or the parameter do not exist, the rec_DELETE_CON catches the exception thrown and prints what the problem is.
**FUNCTIONING**

The Reconfiguration Service `rec_DELETE_CON` deletes an existing connection between two FBs. It can be an event or a data connection.

When this Service receives an input event `REQ`, it reads the input values. All the input variables must be provided as atoms (either enclosed with ‘ ’ or starting with lowercase). It is important to take into account that if the source/destination FB is inside a Subapplication, its name must be provided as “Subapplication Name.FB Name” in the input variable ‘FB_NAME’.

After reading the input values, it starts an algorithm which deletes the corresponding connection. If it is successfully deleted, the message “Connection deleted: ‘SRC_APP_NAME.SRC_FB_NAME’: ‘SRC_PARAM’ —/> ‘DST_APP_NAME.DST_FB_NAME’: ‘DST_PARAM’” is printed and the Output Variable ‘STATUS’ is set to “RDY”. If it is not possible to delete it, Output Variable ‘STATUS’ is set to “NO_SUCH_OBJECT”. This can happen due to one of the following reasons:

- The parameter given for the source FB does not exist: in this case the message “Parameter ‘SRC_PARAM’ of ‘APP_NAME.FB_NAME’ not valid” is printed.
- The source application or FB given does not exist in the system: in this case the message “‘APP_NAME.FB_NAME’ is not a valid FB” is printed.

If there is no connection to the destination parameters sent, it will also send a success message as if it had deleted it.

4.3.5. Create a new Subapplication

**INTERFACE**

Figure 4.19 represents the interface of the Reconfiguration Service `rec_CREATE_SUBAPP`. Its I/O are explained in Table 4.16

![Figure 4.19: Interface of rec_CREATE_SUBAPP](image-url)
4. IMPLEMENTATION OF RECONFIGURATION SERVICES IN ERLANG

<table>
<thead>
<tr>
<th>INPUT EVENTS</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ</td>
<td>Event</td>
<td>Event to require the creation of a Subapplication</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUTPUT EVENTS</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CNF</td>
<td>Event</td>
<td>Event to confirm that the subapplication has been created</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INPUT VARIABLES</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBAPP_NAME STRING</td>
<td>STRING</td>
<td>Name of the new Subapplication (atom)</td>
</tr>
<tr>
<td>APP_NAME STRING</td>
<td>STRING</td>
<td>Application where the Subapplication must be created (atom)</td>
</tr>
<tr>
<td>TYPE STRING</td>
<td>STRING</td>
<td>Subapplication type name (atom)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUTPUT VARIABLES</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS STRING</td>
<td>STRING</td>
<td>Service Status: RDY, NO_SUCH_OBJECT</td>
</tr>
</tbody>
</table>

Table 4.16: I/O in rec_CREATE_SUBAPP

ERLANG FUNCTIONALITIES USED

Creating a Subapplication based on an existing one is completely different as creating a FB instance, since for each Subapplication there is a unique supervisor in charge of controlling all the FB instances that it contains. For this reason, an instance of the Subapplication to be created must be added to the Reconfiguration Application. This way, the Reconfiguration Service rec_CREATE_SUBAPP can access all the information related to its structure and “copy it into the correct application.

For this purpose, this Reconfiguration Service runs an algorithm that performs the following actions:

1. It gets the number of children the Subapplication supervisor has using supervisor: count_children/1, and stores a list with all its children using supervisor: which_children/1.

2. It starts the Subapplication supervisor as a child of the supervisor of the application where the Subapplication is to be created. For this purpose, the function supervisor:start_child/2 is used.

3. It starts each of the needed FBs as children of the Subapplication supervisor by following these steps for each element of the children list previously created:
   a) Store the initial values of the FB using Module:instance_args/1 in a Variable.
b) Change the instance name for the new one.

c) Create the specifications of the FB using the new name and the initial values and type of the copied one.

d) Start the FB as a child of the Subapplication supervisor using `supervisor:start_child/2`

If it cannot be created because the given type or application name do not exist, or because the Subapplication already exists in the system, the Service `rec_CREATE_SUBAPP` catches the exception thrown and prints what the problem is.

**FUNCTIONING**

The Reconfiguration Service `rec_CREATE_SUBAPP` creates a new Subapplication of a certain created type.

When this Service receives an input event ‘REQ’, it reads the input values. All the input variables must be provided as atoms (either enclosed with ‘ ’ or starting with lowercase). It is important to take into account that an instance of the Subapplication to be created must be added to the Reconfiguration Application. This instance must not be connected to any other FB and its name should not be changed, i.e. its name must be identical to its type name. The initial values of the inputs of the Subapplication can be added if wished, so that they are copied in the new one. If left empty, they must be set afterwards making use of the Reconfiguration Service `rec_WRITE`, explained in Section 4.4.1.

After reading the input values, it starts an algorithm which creates a new Subapplication as a copy of the one in the Reconfiguration Application. If it is successfully created, the message “‘APP_NAME.SUBAPP_NAME’ WAS SUCCESSFULLY CREATED” is printed and the Output Variable ‘STATUS’ is set to “RDY”. If it is not possible to create it, Output Variable ‘STATUS’ is set to “NO_SUCH_OBJECT”. This can happen due to one of the following reasons:

- The type does not exist: in this case the message “The type ‘SUBAPP_TYPE’ is not valid” is printed.

- The Subapplication instance already exists in the system: in this case the message “The subapp ‘APP_NAME.SUBAPP_NAME’ is already started” is printed.

- The application given does not exist in the system: in this case the message “The application ‘APP_NAME’ is not valid” is printed.
4. IMPLEMENTATION OF RECONFIGURATION SERVICES IN ERLANG

4.3.6. Delete a Subapplication

INTERFACE

Figure 4.20 represents the interface of the Reconfiguration Service rec_DELETE_SUBAPP. Its I/O are explained in Table 4.17.

![Figure 4.20: Interface of rec_DELETE_SUBAPP](image)

<table>
<thead>
<tr>
<th>INPUT EVENTS</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ</td>
<td></td>
<td>Event to require the deletion of a Subapplication</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUTPUT EVENTS</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CNF</td>
<td></td>
<td>Event to confirm that the Subapplication has been deleted</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INPUT VARIABLES</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBAPP_NAME</td>
<td>STRING</td>
<td>Name of the new Subapplication (atom)</td>
</tr>
<tr>
<td>APP_NAME</td>
<td>STRING</td>
<td>Application where the Subapplication must be deleted (atom)</td>
</tr>
<tr>
<td>TYPE</td>
<td>STRING</td>
<td>Subapplication type name (atom)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUTPUT VARIABLES</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS</td>
<td>STRING</td>
<td>Service Status: RDY, NO_SUCH_OBJECT</td>
</tr>
</tbody>
</table>

Table 4.17: I/O in rec_DELETE_SUBAPP

ERLANG FUNCTIONALITIES USED

The function supervisor:terminate_child/2 is called, being the first argument the supervisor name sup_'APP_NAME' and the second argument the id of the child sup_'APP_NAME.FB_NAME'. This function terminates the Subapplication in its application.

Finally, the function supervisor:delete_child/2 is called, with the same arguments as the ones given to supervisor:terminate_child/2. This function deletes the Subapplication from its supervisor, and therefore from the app.
If it can not be deleted because of the Subapplication or the application names are wrong, the rec\_DELETE\_SUBAPP catches the corresponding exception and prints what the problem is.

**FUNCTIONING**

The Reconfiguration Service rec\_DELETE\_SUBAPP deletes an existing Subapplication from its parent application.

When this Service receives an input event ‘REQ’, it reads the input values. Both the ‘APP\_NAME’ and the ‘FB\_NAME’ must be provided as atoms (either enclosed with ‘ ’ or starting with lowercase). It is important to take into account that if the destination Subapplication is a child of another Subapplication, the ‘APP\_NAME’ must be the name of the parent Subapplication.

After reading the input values, it starts an algorithm which deletes the corresponding Subapplication. If it is successfully deleted, the message “‘APP\_NAME.FB\_NAME’: DELETED” is printed and the Output Variable ‘STATUS’ is set to “RDY”. If it is not possible to create it, Output Variable ‘STATUS’ is set to “NO\_SUCH\_OBJECT”. This can happen due to one of the following reasons:

- The Subapplication does not exist in the provided application: in this case the message “The Subapp ‘APP\_NAME.FB\_NAME’ is not valid, it can’t be deleted” is printed.
- The application given does not exist in the system: in this case the message “The application ‘APP\_NAME’ is not valid, ‘APP\_NAME.FB\_NAME’ can’t be deleted” is printed.

**4.3.7. Testing of Structural Services**

In order to test the Structural Services, the system represented in Figure 4.21 was reconfigured. A simple system was used, since the purpose of this test is to make sure that the Execution Control Services work as they are expected to. In this case, two Reconfiguration Applications were tested, one for creating and deleting FBs and another for Subapplications.

**Test for updating FBs**

The Reconfiguration Application used is the one depicted in Figure 4.22 The objective of
this Reconfiguration is to substitute the print\_any FB for a new version called print\_any2, and update all its connections.

This Reconfiguration Application is started in parallel to the execution of the reconfigured system by sending a ‘REQ’ event to the FB start. Then, the new FB print\_any2 is created. Then the data connection with print\_any is deleted. A delay of 7 seconds is applied, so that it can be checked that it prints the same number all the time, since it is not updated anymore.

Then, the event connection is also deleted, and a new delay is applied to prove that it is not printing anything, since it is not receiving any input event requesting it.

Once the old version is completely disconnected, the new event connection is established with the new FB print\_any2. A delay is again applied to check that it prints “0.\text{a}ll the time, given that it has not received an input value yet.

Afterwards, the data connection is established, and print\_any2 starts printing the actual current count. Finally the old version print\_any is deleted.

**RESULT**

The test system and test Reconfiguration Application were able to demonstrate the ability of implementing the Structural Services in Erlang when it comes to BFBs. This Reconfiguration Application was also tested giving incorrect input values to the different Reconfiguration Services and all the Structural Services responded as expected, informing about the error and blocking the execution flow of the reconfiguration.

There is only the exception of the Service rec\_CREATE\_CON, that is unable to handle the errors provoked by incorrect input parameters for the destination of the connection. However, this behavior was expected from this Service.

Another issue that can be noted, is that both rec\_CREATE\_CON and rec\_DELETE\_CON just work if the source FB of the connection is not suspended, since they are calling a function in their module.

It is important to remark that execution time is not a value tested in these reconfiguration.
Figure 4.22: Reconfiguration Application for Structural Services-1
For this reason, delays were applied after each reconfiguration step so that it could be easily identified whether it was working properly or not.

**Test for updating Subapplications**

The Reconfiguration Application used is the one depicted in Figure 4.23.

![Diagram of Reconfiguration Application for Structural Services-2](image)

**Figure 4.23:** Reconfiguration Application for Structural Services-2

The objective of this Reconfiguration is to substitute the `cycle` Subapplication for a new version called `newCycle`, and update all its connections.

This Reconfiguration Application is started in parallel to the execution of the reconfigured system by sending a `REQ` event to the FB `start`. Then, the new Subapplication `newCycle` is created, by copying it from the `cycle` added to the Reconfiguration Application. Then all the connections with `cycle` are deleted.
Once the old version is completely disconnected, the new connections are established with the new Subapplication \texttt{newCycle}. Finally the old version \texttt{cycle} is deleted.

\section*{RESULT}

The test system and test reconfiguration application were able to demonstrate the ability of implementing the Structural Services in Erlang with Subapplications. This Reconfiguration Application was also tested giving incorrect input values to the different Reconfiguration Services and all the Structural Services responded as expected, informing about the error and blocking the execution flow of the reconfiguration. There is only the exception of the Service \texttt{rec\_CREATE\_CON}, that is unable to handle the errors provoked by incorrect input parameters for the destination of the connection. However, this behavior was expected from that Service.

\section*{4.4. State Interaction Services}

The State Interaction Services are those Reconfiguration Services that allow interaction with the state of a control application, i.e. with the the ECC state of a FB or its data I/O and Internal Variables.

According to the definition of State Interaction Services, and the way they are implemented in Erlang, four Control Reconfiguration Services are implemented, as discussed at the beginning of Section 4:

\begin{itemize}
  \item \texttt{rec\_WRITE}: It writes a new value in a data I/O or an Internal Variable.
  \item \texttt{rec\_READ}: It reads the value stored in a data I/O or an Internal Variable.
  \item \texttt{rec\_WRITE\_STATE}: It changes the ECC state of a FB.
  \item \texttt{rec\_READ\_STATE}: It reads the ECC state of a FB.
\end{itemize}

\subsection*{4.4.1. Write a FB Parameter}

\section*{INTERFACE}

Figure 4.24 represents the interface of the Reconfiguration Service \texttt{rec\_WRITE}. Its I/O are explained in Table 4.18

Ainara Matey Benito
4. IMPLEMENTATION OF RECONFIGURATION SERVICES IN ERLANG

Figure 4.24: Interface of rec_WRITE

<table>
<thead>
<tr>
<th>INPUT EVENTS</th>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ</td>
<td>Event</td>
<td>Event to require to write the value of an Input, an Output, or an Internal Variable from a FB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUTPUT EVENTS</th>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNF</td>
<td>Event</td>
<td>Event to confirm that the parameter has been written</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INPUT VARIABLES</th>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>APP_NAME</td>
<td>STRING</td>
<td>Name of the Application containing the FB (atom)</td>
</tr>
<tr>
<td>FB_NAME</td>
<td>STRING</td>
<td>Name of the FB instance (atom)</td>
</tr>
<tr>
<td>PARAMETER</td>
<td>STRING</td>
<td>Name of the parameter to be changed (atom)</td>
</tr>
<tr>
<td>VALUE</td>
<td>STRING</td>
<td>New value of the parameter</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUTPUT VARIABLES</th>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS</td>
<td>STRING</td>
<td>Service Status: RDY, NO_SUCH_OBJECT</td>
</tr>
</tbody>
</table>

Table 4.18: I/O in rec_WRITE

ERLANG FUNCTIONALITIES USED

This Reconfiguration Service also makes use of the call handling explained in 4.3.3.

In this case, for writing, a function clause is added for each input, output and internal variable. In each clause, the corresponding data value is updated in the internal data of the FB instance. Each clause has a guard in charge of type checking, in order to avoid writing a value of an incorrect type.

The needed function clause is called from the Reconfiguration Service rec_WRITE by making use of the function `gen_statem:call/4`. The message sent in this case is of the form `{data_write, 'PARAMETER', Value}`. In the variable `Value`, the new value for the parameter is stored.

When the value to be updated is an Output variable, after changing its value, it is sent to the following FB to which this output is connected. If it was not sent, changing an output
would be of no use, since its value would be sent just after running its algorithm, and at that point the value would be already updated.

When the value to be updated is an Input variable, after changing its value, the buffer for this variable is also updated, so that if it is already running, it can work with the new value.

If it can not be written because of a wrong input value, the rec_WRITE catches the exception thrown and prints what the problem is.

**FUNCTIONING**

The Reconfiguration Service rec_WRITE writes a new value in an Input, Output or Internal Variable.

When this Service receives an input event ‘REQ’, it reads the input values. All the input variables must be provided as atoms (either enclosed with ‘ ’ or starting with lowercase), except for the input ‘VALUE’, which must be of the same data type of the parameter receiving it. It is important to take into account that if the FB is inside a Subapplication, its name must be provided as “Subapplication Name.FB Name” in the input variable ‘FB_NAME’.

After reading the input values, it starts an algorithm that writes the new value. If it is successfully written, the message “The parameter ‘PARAMETER’ of ‘APP_NAME.FB_NAME’ has been rewritten to ‘VALUE’” is printed and the Output Variable ‘STATUS’ is set to “RDY”. If it is not possible to delete it, Output Variable ‘STATUS’ is set to “NO_SUCH_OBJECT”. This can happen due to one of the following reasons:

- The parameter given for the source FB does not exist, or the data type of the value given is incorrect: in this case the message “The parameter ‘PARAMETER’ or its value ‘VALUE’ are not valid” is printed.

- The application or FB given does not exist in the system: in this case the message “‘APP_NAME.FB_NAME’ is not a valid FB” is printed.

**4.4.2. Read a FB Parameter**

**INTERFACE**

Figure 4.25 represents the interface of the Reconfiguration Service rec_READ. Its I/O are explained in Table 4.19
4. IMPLEMENTATION OF RECONFIGURATION SERVICES IN ERLANG

![Figure 4.25: Interface of rec_READ](image)

<table>
<thead>
<tr>
<th>INPUT EVENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ</td>
<td>Event</td>
</tr>
<tr>
<td></td>
<td>Event to require to read the value of an Input, an Output, or an Internal Variable from a FB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUTPUT EVENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CNF</td>
<td>Event</td>
</tr>
<tr>
<td></td>
<td>Event to confirm that the parameter has been read</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INPUT VARIABLES</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>APP_NAME</td>
<td>STRING</td>
</tr>
<tr>
<td>FB_NAME</td>
<td>STRING</td>
</tr>
<tr>
<td>PARAMETER</td>
<td>STRING</td>
</tr>
<tr>
<td></td>
<td>Name of the Application containing the FB (atom)</td>
</tr>
<tr>
<td></td>
<td>Name of the FB instance (atom)</td>
</tr>
<tr>
<td></td>
<td>Name of the parameter to be read (atom)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUTPUT VARIABLES</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS</td>
<td>STRING</td>
</tr>
<tr>
<td></td>
<td>Service Status: RDY, NO_SUCH_OBJECT</td>
</tr>
</tbody>
</table>

Table 4.19: I/O in rec_READ

ERLANG FUNCTIONALITIES USED

This Reconfiguration Service also makes use of the call handling explained in 4.3.3.

In this case, for reading, a function clause is added for each input, output and internal variable. In each clause, the corresponding data value is read from the internal data of the FB instance.

The needed function clause is called from the Reconfiguration Service rec_READ by making use of the function `gen_statem:call/4`. The message sent in this case is of the form `{data_read, ‘PARAM’, undefined}`. The function clause sends an answer of the type `{ok, Parameter Value}`

If it can not be read because of a wrong input value, the rec_READ catches the exception thrown and prints what the problem is.
FUNCTIONING

The Reconfiguration Service rec_READ reads the current value stored in an Input, Output or Internal Variable.

When this Service receives an input event ‘REQ’, it reads the input values. All the input variables must be provided as atoms (either enclosed with ‘ ’ or starting with lowercase). It is important to take into account that if the FB is inside a Subapplication, its name must be provided as “Subapplication Name.FB Name” in the input variable ‘FB_NAME’.

After reading the input values, it starts an algorithm which reads the value. If it is successfully read, the message “‘APP_NAME.FB_NAME’→‘PARAMETER’:‘VALUE’” is printed and the Output Variable ‘STATUS’ is set to “RDY”. If it is not possible to delete it, Output Variable ‘STATUS’ is set to “NO_SUCH_OBJECT”. This can happen due to one of the following reasons:

- The parameter given for the source FB does not exist: in this case the message “The parameter ‘PARAMETER’ is not valid” is printed.
- The application or FB given does not exist in the system: in this case the message “‘APP_NAME.FB_NAME’ is not a valid FB” is printed.

4.4.3. Change the State of a FB

INTERFACE

Figure 4.26 represents the interface of the Reconfiguration Service rec_WRITE_STATE. Its I/O are explained in Table 4.20

![Interface of rec_WRITE_STATE](image)

Figure 4.26: Interface of rec_WRITE_STATE
4. IMPLEMENTATION OF RECONFIGURATION SERVICES IN ERLANG

<table>
<thead>
<tr>
<th>INPUT EVENTS</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ</td>
<td>Event</td>
<td>Event to require to change the state of a FB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUTPUT EVENTS</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CNF</td>
<td>Event</td>
<td>Event to confirm that the FB state was updated</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INPUT VARIABLES</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE</td>
<td>STRING</td>
<td>Name of the new FB ECC state (atom)</td>
</tr>
<tr>
<td>FB_NAME</td>
<td>STRING</td>
<td>Name of the FB instance (atom)</td>
</tr>
<tr>
<td>APP_NAME</td>
<td>STRING</td>
<td>Name of the Application containing the FB (atom)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUTPUT VARIABLES</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS</td>
<td>STRING</td>
<td>Service Status: RDY, NO_SUCH_OBJECT</td>
</tr>
</tbody>
</table>

Table 4.20: I/O in rec_WRITE STATE

ERLANG FUNCTIONALITIES USED

In order to change the ECC state of the FB, this Reconfiguration Service uses the function `sys:get_state/1` to first get the current state. Then it calls the function `sys:replace_state/3` to replace the current state for the given one.

If it can not be changed because of a wrong application or FB name given, the rec_WRITE_STATE catches the exception thrown and prints what the problem is. However, if a wrong state is given, it will not crash because it continues operating in a nonexistent state, leading to an incorrect operation.

FUNCTIONING

The Reconfiguration Service rec_WRITE_STATE changes the current ECC state of a FB instance.

When this Service receives an input event 'REQ', it reads the input values. All the input variables must be provided as atoms (either enclosed with ‘ ’ or starting with lowercase). It is important to take into account that if the FB is inside a Subapplication, its name must be provided as “Subapplication Name.FB Name” in the input variable ‘FB_NAME’.

After reading the input values, it starts an algorithm that changes the current ECC state. If it is successfully changed, the message “FB ’APP_NAME.FB_NAME is currently in the state ‘STATE’ ” is printed and the Output Variable ‘STATUS’ is set to “RDY”. If it is not possible to change it, the Output Variable ‘STATUS’ is set to “NO_SUCH_OBJECT”. This can happen because the application or FB given does not exist in the system. In this case the message “ ’APP_NAME.FB_NAME’ is not a valid FB” is printed.
4.4.4. Read the State of a FB

INTERFACE

Figure 4.27 represents the interface of the Reconfiguration Service rec_READ_STATE. Its I/O are explained in Table 4.21.

![Figure 4.27: Interface of rec_READ_STATE](image)

<table>
<thead>
<tr>
<th>INPUT EVENTS</th>
<th></th>
<th>OUTPUT EVENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ</td>
<td>Event</td>
<td>Event to require to read the state of a FB</td>
<td></td>
</tr>
<tr>
<td>CNF</td>
<td>Event</td>
<td>Event to confirm that the FB state was read</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INPUT VARIABLES</th>
<th></th>
<th>OUTPUT VARIABLES</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FB_NAME STRING</td>
<td></td>
<td>NAME</td>
<td></td>
</tr>
<tr>
<td>APP_NAME STRING</td>
<td></td>
<td>STRING</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Service Status: RDY, NO_SUCH_OBJECT</td>
<td></td>
</tr>
<tr>
<td>STATUS STRING</td>
<td></td>
<td>STRING</td>
<td></td>
</tr>
<tr>
<td>STATE STRING</td>
<td></td>
<td>Current ECC state of the FB</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.21: I/O in rec_READ_STATE

ERLANG FUNCTIONALITIES USED

In order to read the ECC state of the FB, this Reconfiguration Service uses the function `sys:get_state/1`.

If it can not be read because of a wrong application or FB name given, the rec_READ_STATE catches the exception thrown and prints what the problem is.

FUNCTIONING

The Reconfiguration Service rec_READ_STATE reads the current ECC state of a FB instance.
When this Service receives an input event ‘REQ’, it reads the input values. All the input variables must be provided as atoms (either enclosed with ‘ ’ or starting with lowercase). It is important to take into account that if the FB is inside a Subapplication, its name must be provided as “Subapplication Name.FB_Name” in the input variable ‘FB_NAME’.

After reading the input values, it starts an algorithm that reads the current ECC state. If it is successfully read, the message “FB ‘APP_NAME.FB_NAME is currently in the state ‘STATE’” is printed and the Output Variable ‘STATUS’ is set to “RDY”. If it is not possible to read it, the Output Variable ‘STATUS’ is set to “NO_SUCH_OBJECT”. This can happen because the application or FB given does not exist in the system. In this case the message “‘APP_NAME.FB_NAME’ is not a valid FB” is printed.

4.4.5. Testing of State Interaction Services

In order to test the State Interaction Services, the system represented by Figure 4.28 was reconfigured. A simple system was used, since the purpose of this test is to make sure that the State Interaction Services work as they are expected to. The Reconfiguration Application tested is the one depicted in Figure 4.29, which includes all the implemented State Interaction Services.

This application is started in parallel to the execution of the reconfigured system by sending a ‘REQ’ event to the FB `start`. Then, the counter in the test system is suspended, and a delay of 4 seconds is applied. During this time, the whole system should be on hold, since the suspended counter blocks the flow of execution. Afterwards, the counter is resumed, preserving its previous count value. After 4 seconds, the counter is terminated, and therefore restarted by its supervisor. Once restarted, the count resumes, but starting from 0 again.

RESULT

The test system and test reconfiguration application were able to demonstrate the ability of implementing the IEC 61499 State Interaction Services in Erlang. This Reconfiguration Application was also tested giving incorrect values to their inputs, and all the State
Interaction Services responded as expected, informing about the error and blocking the execution flow of the reconfiguration.

However, the Service `rec_WRITE_STATE` accepts nonexistent states as inputs, making the system to crash as it tries to execute an FB in an incorrect state. Nevertheless, this behavior was expected from this Service, so it is necessary to be careful when using this Service.
4. IMPLEMENTATION OF RECONFIGURATION SERVICES IN ERLANG
Chapter 5

CASE STUDIES

In this chapter, three different case studies are modeled in order to try and test the correct functioning of the implemented Reconfiguration Services. For each test case, a system was modeled following IEC 61499 architecture for, and then it was updated using the implemented Reconfiguration Services.

Some of the reconfigurations performed, update the control action upon the system and study how the system would evolve in such a situation. However, as the scope of this project do not involve considerably large models, the control systems are not complex and updating them does not involve a large Reconfiguration Application that includes many of the implemented Services. For that reason, some other Reconfiguration Applications were tested, which update some parts of the model, not necessarily updating the main part of the control system, but implying a deeper update that needs more reconfiguration steps and hence a more complex Reconfiguration Application.

5.1. Interconnected tanks

The first system modeled for testing purposes, consists of a system of interconnected tanks, taking as a base the tank model proposed by Prenzel and Provost [18]. A case with liquid tanks is a typical example used for automatic control purposes.
5. CASE STUDIES

Figure 5.1: Diagram of the Tank Model before Reconfiguration

5.1.1. System modeling

Physical model

For this test case, the system of two interconnected liquid tank depicted in Figure 5.1 was considered.

The system consists of two tanks. The Tank 1 receives a random input liquid flow from another part of the plant. Such input flow is not controlled by this system, and there is no valve or gate in charge of stopping the input flow. It has also an output pipe with a valve V1, connecting Tank 2. This second tank has another output pipe with the valve V2 leading to the following part of the plant, from which no extra information is known.

Each tank has a different capacity, and a target level is set for each of them, as shown in Table 5.1.

<table>
<thead>
<tr>
<th>Capacity (l)</th>
<th>Target level (l)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tank 1</td>
<td>80</td>
</tr>
<tr>
<td>Tank 2</td>
<td>50</td>
</tr>
</tbody>
</table>

Table 5.1: Tank levels

In order to maintain the level of the tanks close to the target level, no matter what the flow input in the system is, a control action is set. Each of the tanks was provided with a PID control which regulates its outflow depending on the current level error.
IEC 61499 model

The previously explained system is modeled with IEC 61499 architecture using 4DIAC, as shown in Figure 5.2.

![Tank Model with IEC 61499 before Reconfiguration](image)

Figure 5.2: Tank Model with IEC 61499 before Reconfiguration

The execution begins when the FB `start` receives an event 'START'. It sends an event to `rand_inflow`, of the type `rand_norm`. It generates a random number following a normal distribution with average 5 and variance 0.2. This generated number represents the input flow coming into Tank 1 in each cycle.

An instance `max_inflow`, of the type `max`, limits the inflow to 7, hence simulating a limit caused by piping geometries.

Afterwards, the same scheme is repeated twice, once for representing each tank. This scheme consists of the following FB instances:

- **tank_sim_1**: Calculates the tank level, considering its capacity, as well as the input and output flows of the current cycle.

- **write1** (type `write`): Writes on a text file the tank level, along with the current system time.

- **print_level_1** (type `print_any`): Prints the current tank level in the console.

- **pid_control_1** (type `pid_control`): Calculates the PID control action considered the given gains and the level error.

Ainara Matey Benito
5. CASE STUDIES

- **max_1** (type \textit{max}): It limits the tank outflow, thus simulating limits due to the geometries of the piping and valves.

This scheme is repeated again for modeling the Tank 2. Finally a delay is included for having a control of the cycle time, this time set to 100 ms.

All the FBs in this model are BFB and their algorithms are written in Erlang, in order to generate the entire working system with FBBeam.

5.1.2. Reconfiguration Applications

Two reconfiguration cases are presented. The first one updates the model adding a third tank, in order to test how is the performance of a bigger Reconfiguration Application involving just FB.

Afterwards, another Reconfiguration Application will be tested the system described in Section 5.1.1, this time just updating the control action upon the system.

Model Update

The goal of this update is to prove how the different implemented Services work together in a larger Reconfiguration Application, when updating an IEC 61499 system that needs various changes. It helps also testing how complex is a reconfiguration on a larger scale and which problems or errors can it lead to.

For such update the two tank system outlined in Section 5.1.1 is extended to a third tank, including its corresponding control action performed again by a PID.

The new system version is depicted by the diagram in Figure 5.3, showing the variations with respect to the previous version in yellow. The system created with 4diac is outlined in Figure 5.4.
This model is the same one as the previous with only two tanks, but with the scheme corresponding to the tank replicated a third time in order to represent Tank 3.

The control action to be performed can be divided in the following steps:
1. Create the new FB instances.

2. Set the input variables of the new FB instances to their desired initial values.

3. Create the event and data connections between the new instances.

4. Connect the new part to the rest of the system and change the connections leading to the delay.

All these steps are performed by the Reconfiguration Application depicted in Figure 5.5.

Figure 5.5: Reconfiguration application for adding a third tank

The Reconfiguration is triggered by a **start** FB instance, which starts the execution of the first reconfiguration steps. Both the instances **New_Tank** and **New_PID** are Subapplications in charge of creating a new instance of the type `tank_sim` and `pid_control` respectively, and of writing their corresponding input values. Both Subapplication schemes are depicted in Figure 5.8 and Figure 5.7.

The Execution continues by creating and writing the initial values of the new FB instances for the types `print_any`, `max` and `write`.
It can be noted that the only needed Reconfiguration Services so far were `rec_CREATE_FB` and `rec_WRITE`, for respectively creating new FB instances and writing their input values.

![Diagram](image1.png)

**Figure 5.6:** New_Tank Subapplication for adding a third tank

![Diagram](image2.png)

**Figure 5.7:** New_Con Subapplication for adding a third PID control

Once all the new FB instances have been created and their initial values are written, all the necessary connections can be established. For this purpose the Subapplication **New_Connections** is included, which uses the Structural Service `rec_CREATE_CON` to establish all the event and data connections with the new instances, and `rec_DELETE_CON` to delete the connection among the second tank system and the delay.
5. CASE STUDIES

Figure 5.8: New_Con Subapplication for adding a third tank

RESULT

Performing this reconfiguration was successful, since it added the third tank without affecting the others. In the Figure 5.9, a diagram with the level of the three tanks versus time is plotted. It shows how the Tank 1 starts filling up to approximately its target level.

Figure 5.9: Tank Levels when adding a third tank
(it has a deviation due to a not optimal control action that is discussed in the Section 5.1.2) before leaving any outflow to Tank 2. Both tanks continue operating for 20 seconds, time after which the Reconfiguration takes place and the third tank is added to the system.

The noise in the level signal happens due to the random input applied.

Control Update

In this section, a second update is performed. The main goal of this update is to test how a reconfiguration could affect a system in a more realistic scenario, in which just the control action is updated. In this case, the updated system is again the system with just two tanks, changing the values for their PID gains.

As previously shown in Figure 5.9, all of the tank levels follow a certain level, always around 5 liters above the target set for each of them.

The reason for that is that the PID applied has no Integration Gain, thus eliminating overshoot but resulting in a steady state error.

For that reason, an Integration Gain was applied to both, and the Derivative gain was also reduced in both, and the Proportional Gain was also augmented in both cases.

Table 5.2 shows the PID gains before Reconfiguration (both tanks have the same PID gains), and after Reconfiguration. The values for I and D gains are different after the update, so that the outcomes can be compared.

<table>
<thead>
<tr>
<th></th>
<th>Before</th>
<th>After (Tank 1)</th>
<th>After (Tank 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P Gain</td>
<td>1.5</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>I Gain</td>
<td>0</td>
<td>0.1</td>
<td>1.5</td>
</tr>
<tr>
<td>D Gain</td>
<td>15</td>
<td>10</td>
<td>5</td>
</tr>
</tbody>
</table>

**Table 5.2:** Tank PID values

In order to change the control gain for both tanks, the State Interaction Service rec.WRITE is used to overwrite the PID gains, as depicted in Figure 5.10. The execution time is recorded in a file making use of the FB write.
RESULT

The graph in Figure 5.11 shows the levels of both tanks before and after the PID Reconfiguration performed after 20 seconds of normal execution.

Some changes can be noted thanks to reconfiguration:

- The steady state error is eliminated thanks to the inclusion of the Integral action and an increase in the Proportional Gain.
- The oscillations of the level increase in both tanks due to the Integral Action, being higher in Tank 2. The main reason for that is that in Tank 2 a higher Integral Gain
is applied, which increases the oscillations. Moreover, its Derivative Gain is lower, so it does not smooth enough the output.

- The level of the Tank 2 shows an initial overshoot that is not present in Tank 1. That is due to a higher Integral Gain and because it must handle the increased inflow coming from Tank 1.

All in all, the system response to the new Control action fits its expected behavior, so we can consider the reconfiguration successful.

5.2. Boiler Steam Drum

The second system to update consists of a drum boiler. When working with industrial boilers, ramp-up times are usually long, therefore making DSU suitable and helpful to save a lot in time and costs.

5.2.1. System modeling

Physical model

The goal of this update is to prove how the different implemented Services work together in a larger Reconfiguration Application with Subapplications involved, when updating an IEC 61499 system that needs various changes. It helps also testing if the reconfiguration gets more complex when using updating Subapplications and which problems or errors can it lead to. The system considered in this test case is the Boiler Steam Drum depicted in Figure 5.12.

The system consists of a feed water pipe that leads to a steam drum. In this drum the water is transformed into steam using the heat coming from a combustion process.

In order to transfer the heat to the liquid, the water sinks through a downcomer going down to a heat exchanger inside the combustion chamber, where the water receives the heat coming from the combustion process, thus evaporating and ascending through the riser back to the drum. The drum has a steam outlet for the generated steam to flow out of the drum.

In the combustion chamber, the incoming fuel flow is regulated by a valve, which depends of a controller, that allows a certain fuel inflow depending on the water level of the drum.

The incoming fuel undergoes a combustion process that was simplified for this test. This
5. CASE STUDIES

![Diagram of the Boiler Model before Reconfiguration](image)

**Figure 5.12:** Diagram of the Boiler Model before Reconfiguration

process follow the equation in Eq.5.1 [19].

\[
Q_{out} = \rho_{fuel} \Delta q_{fuel} (LHV_{fuel} + c_{p,fuel} \Delta \theta_{fuel}) \eta_{comb}
\] (5.1)

Each of the unknown terms included in the equation means the following:

- \(Q_{out}\): Output heat from the combustion process (kJ).
- \(q_{fuel}\): air entering the combustion chamber per cycle (m³).
- \(\Delta \theta_{fuel}\): Temperature difference experienced by the fuel, i.e. the difference among its combustion temperature and the ambient temperature.

The selected fuel for this case was natural gas (NG), and the values selected for each variable are the presented in 5.3 [19].

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\rho_{NG})</td>
<td>Fuel density</td>
<td>0.84 kg/m³</td>
</tr>
<tr>
<td>(LHV_{NG})</td>
<td>Lower Heating value of the fuel</td>
<td>26.63 MJ/kg</td>
</tr>
<tr>
<td>(c_{p,NG})</td>
<td>Specific heat of the fuel</td>
<td>2.34 kJ/kgK</td>
</tr>
<tr>
<td>(\theta_{ig,NG})</td>
<td>Ignition Temperature of the fuel</td>
<td>873 – 953 K</td>
</tr>
<tr>
<td>(\theta_{ambient})</td>
<td>Ambient Temperature</td>
<td>N(298,0.2) K</td>
</tr>
<tr>
<td>(\eta_{comb})</td>
<td>Efficiency of the combustion process</td>
<td>97%</td>
</tr>
</tbody>
</table>

**Table 5.3:** Selected Variable Values

The value for the ignition \(\theta_{ig,NG}\) is given as a range, because it is considered that it can vary among those values depending on the environment. The ambient temperature was
considered as a normal distribution with a mean temperature of 298 K, given that it is considered the *normal temperature* in chemical processes.

With regards to the steam produced, the drum behavior is modeled following Eq. 5.2.

\[ q_s = \frac{Q_{in}\eta_{thermal}}{(h_s - h_w)} \]  

Each of the unknown terms included in Eq.5.2. means the following:

- \( q_s \): Steam flow generated (\( m^3 \)).
- \( Q_{in} \): Heat transferred to the liquid (kJ).

The \( Q_{in} \) considered is the output heat from the combustion process. In order to get the values for the enthalpies, the pressure inside the drum is considered 20 bar, the feed water temperature is considered 323 K and the steam temperature 823 K. Considering this information, the input values in Eq.5.2 are those presented in 5.4.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( h_s )</td>
<td>Enthalpy of the steam</td>
<td>211 kJ/kg</td>
</tr>
<tr>
<td>( h_w )</td>
<td>Enthalpy of the water</td>
<td>3579 kJ/kg</td>
</tr>
<tr>
<td>( \eta_{thermal} )</td>
<td>Thermal Efficiency of the heat transfer</td>
<td>97%</td>
</tr>
</tbody>
</table>

**Table 5.4: Selected Variable Values**

**IEC 61499 model**

Once determined how the physical system is modeled, it is implemented in 4diac according to IEC 61499 architecture, as shown in Figure 5.13.

The application execution begins when the FB instance `start` is triggered by an input event 'START'. Then it sends an event to the FB instance `T_AMB.K`, which provides a value for the ambient temperature, which takes randomly from a normal distribution with mean in 298 K.

This temperature is sent as an input for the combustion process, represented by the subapplication `combustion_sim`, depicted by 5.14. In this combustion simulation, all the data provided in Section 5.2.1 are given as input for the combustion process simulation, which follows Eq.5.1. The ignition temperature is given by the instance `T_ignition_Kelvin`, which generates a random number between 873 and 953 K.
5. CASE STUDIES

Figure 5.13: Boiler Model with IEC 61499 before Reconfiguration

The computed output heat is sent to the Subapplication Drum_sim, in charge of calculating the current Drum level, using the schema in Figure 5.15. The evaporation FB instance computes the steam outflow using the Eq. 5.2. Then the calculation_new_level, takes this value, along with the feed water inflow and the drum level in the previous cycle and calculates the drum level in the current cycle.

The FB limit_tank, limits the tank level to its capacity. Finally, the FB instance error computes the error between the current and the target level of the drum.

The current level provided, is printed in the console by print_level and written in a file by write.

The computed error is sent to the pd_control, in charge of calculating the control action, which represents the difference in fuel necessary to achieve the desired drum level. Therefore, this control action is added by add_1 to the fuel input that was used in the current cycle, and limited to a maximum fuel input of 3000 by max.

Finally a delay in ms was applied to set how long the cycle must be.

5.2.2. Reconfiguration Applications

Two reconfiguration cases are presented. The first one updates the model changing the combustion simulation model, in order to test how is the performance of a bigger Recon-
figuration Application which deals with Subapplications.

Afterwards, another Reconfiguration Application will be tested the system described in Section 5.2.1, this time just updating the control action upon the system.

**Model Update**

This Reconfiguration aims to show how do updates involving Subapplications work.

With that purpose, a change on how the combustion is computed is performed. This time, the air input must also be considered. Depending on the fuel used, a certain proportional quantity of air is needed, given by 5.3 [19].

\[ q_{air} = q_{NG} A_{ir_{min}} AFR \]  

(5.3)

Each of the unknown terms included in Eq.5.2. means the following:

- \( q_{air} \): Air entering the combustion chamber per cycle (\( m^3 \)).
5. CASE STUDIES

- $q_{NG}$: Fuel entering the combustion chamber per cycle ($m^3$).

Considering that the fuel used is still the same natural gas, the corresponding input values in Eq.5.3 are those presented in Table 5.5 [19].

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Air_{min}$</td>
<td>Minimum amount of air needed for a complete combustion</td>
<td>$8.38 \text{ m}^3/\text{m}^3\text{NG}$</td>
</tr>
<tr>
<td>$AFR$</td>
<td>Air/fuel Ratio, which represents how much excess air is needed to avoid an incomplete combustion</td>
<td>1.3</td>
</tr>
</tbody>
</table>

Table 5.5: Selected Variable Values

Taking these values into account, the air input is controlled by opening or closing an input valve, taking into account the natural gas input, and an upper limit for the input flow, thus simulating limiting piping geometries.

The resulting updated system is the one depicted by 5.16, showing the variations with respect to the previous version in yellow. In order to perform this Reconfiguration, the previous Subapplication `combustion_sim` is substituted by the one represented in 4diac as shown in Figure 5.17. For the inclusion of the air inflow control, three new FB instances were added:

- `air_inflow`: Computes the necessary air volume for a complete combustion of the input fuel, according to Eq.5.3.
- **max**: Limits the air inflow to a max of 3300.

- **combusted fuel_real**: It recomputes the combusted fuel, since a limited air inflow can result in an insufficient air input.

![Combustion Model with IEC 61499 after Reconfiguration](image)

**Figure 5.17**: Combustion Model with IEC 61499 after Reconfiguration

For this test case, the current cycle was suspended by stopping one FB instance. The reason for that is to show how the reconfiguration would work if a part of the system must be suspended for a correct reconfiguration. The control action to be performed can be thus divided in the following steps:

1. Suspend the current cycle.
2. Delete the old version of the updated Subapplication.
3. Create the new Subapplication instance.
4. Set the input variables of the new FB instances to their desired initial values.
5. Update the event and data connections with the new instances, by deleting the ones with the old instances and creating the new ones.
6. Resume the cycle execution delay

All these steps are performed by the Reconfiguration Application depicted in Figure 5.18.

The Reconfiguration is triggered by a **start** FB instance, which starts the execution of the first reconfiguration steps. The instance **SUSPEND_FBS_1** is a Subapplication in charge of making sure the FB instance **pd_control** is suspended, as depicted in 5.19. In order to do so, it makes use of the Query Service **rec_QUERY_FB_STATUS** to check if its status is “suspended.” and if its not, it is suspended with the Execution Control Service **rec_STOP**.

Ainara Matey Benito
5. CASE STUDIES

Figure 5.18: Reconfiguration Application for updating the combustion process

Figure 5.19: Subapplication Suspend_FBs

The Execution continues by reading the current input value of one of the combustion inputs, with the State Interaction Service rec_READ, in order to later rewrite it in the new Subapplication.

Afterwards, the old Subapplication is deleted and the new one is created based on the combustion_sim2 instance included in the reconfiguration application. These actions are performed making use of the Structural Services rec_CREATE_SUBAPP and rec_DELETE_SUBAPP.

Then, the Subapplication Write_Values, whose scheme is outlined in 5.20, makes use of the value read before and the State Interaction Service rec_WRITE to write all the initial values of the new instance.

It must be remarked that those values could have been added in the combustion_sim2 instance, but one of the aims of this reconfiguration is to test how the Reconfiguration
Services work together in a larger scale.

Later on, the connections are updated using the Subapplications **Delete_Con** and **Create_Con**, respectively presented in Figure 5.21 and Figure 5.22. They make use of the Structural Services `rec_CREATE_CON` and `rec_DELETE_CON`.

Finally, the suspended instance is restarted using the Execution Control Service `rec_START`, and the system time is recorded in a file using `write`.

Figure 5.20: Subapplication Write_Values

Figure 5.21: Subapplication Delete_Con
In this section, a second update is performed. The main goal of this update is to test how a reconfiguration could affect a system in a more realistic scenario, in which just the control action is updated. In this case, we update again the system with the old combustion chamber version, changing the current PD controller for a PID, in order to eliminate the steady state error of the system, and this way test how a controller update would work on a system.

Table 5.6 shows the control gains before Reconfiguration, and after Reconfiguration.

<table>
<thead>
<tr>
<th></th>
<th>Before</th>
<th>After (Boiler 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P Gain</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>I Gain</td>
<td>0</td>
<td>0.1</td>
</tr>
<tr>
<td>D Gain</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 5.6: Boiler PID values

In order to change the control FB instance, the Reconfiguration Application of Figure 5.23 was executed. After triggering the FB start, the Subapplication New_PID depicted...
Comparison of Dynamic Software updating methods for IEC 61499 with Erlang

**Figure 5.23:** Reconfiguration Application for updating the controller

in 5.24 creates an instance of the new type *pid_control* included in the Reconfiguration Application and writes its initial values. For that purpose it uses the Reconfiguration Services *rec_CREATE_FB* and *rec_WRITE*. Afterwards, it uses *rec_CREATE.CON* to create the new event and data connections for the new FB instance, while in parallel it deletes the old connections with the Subapplication *deleteCon* of Figure 5.25.

Finally, the old PD control FB is deleted using *rec.DELETE_FB* and the system time is recorded in a text file using *write*.

**RESULT**

The graph in Figure 5.26 and its augmented view in Figure 5.27 show how the Drum level was very stable but with a considerable steady state error. This error has been corrected by adding the Integral control action after approximately 40s of execution. This integral action completely eliminates the steady state error, but carries some initial oscillations in the level of the drum until the new equilibrium is reached.
5. CASE STUDIES

Figure 5.24: New_PID Subapplication

Figure 5.25: deleteCon Subapplication
5.3. Machining station

This system depicts a case that is more related to an industrial manufacturing plant, and helps picturing a case closer to a production plant.
5. CASE STUDIES

5.3.1. System modeling

Physical model

The system considered in this test case is the Machining Station depicted in Figure 5.28. It consists of a conveyor that transports a work piece through a machining station. There are three position sensors in the conveyor. The entrance to the station is detected by sensor S1. When the workpiece is in the right position to be machined it is detected by sensor S2. Finally, when leaving the station, it is detected by sensor S3.

The machining considered is drilling. When a piece arrives to the station the sensor S1 is activated and the conveyor starts to move.

When it reaches the sensor S2, the conveyor stops and the drilling tool makes an approximation movement.

Then, it starts the drilling while penetrating into the piece. After the machining is finished, the drill starts to move away from the piece. Once the drill is in a position in which it is no longer in contact with the piece, the sensor SD is activated and the conveyor starts moving again. However, the drill continues moving up until it gets to its upper position.

When the piece leaves the station, i.e., when activating S3, the conveyor stops moving and waits for the next piece to come.

IEC 61499 model

Once determined how the physical system is modeled, it is implemented in 4diac according to IEC 61499 architecture, as shown in Figure 5.29.
Its execution starts when triggering **start**. It sends an event to **rand_norm_int**, which generates a random integer following a normal distribution with mean 15000 and variance 5000. This number represents the time in ms the next piece takes to arrive to the station.

Then the FB instance **conveyor** receives the data from the sensors, triggered by the FB instances **S1**, **S2**, **S3** and **S4**.

As it receives those sensor events, the conveyor changes its ECC state following the scheme in 5.30. Each of the algorithms in this ECC are in charge of printing in the console which sensor was activated and if the conveyor is moving or it is stopped.
The FBs **delay2** and **delay3** simulate the time the piece takes to move from S1 to S2 and from S2 to S3.

Finally, the drill is modeled following the ECC in Figure 5.31.

![Drill ECC](image)

**Figure 5.31:** Drill ECC

Each of the movements performed by the drill is represented by one of the States, in charge of printing the current drilling action and sent the corresponding time each state takes to **delay**. Once the state "moving_away" is reached, the sensor SD is triggered.

According to this implementation, the system works as expected, but just letting one piece in the system at a time.

### 5.4. Reconfiguration Application

This Reconfiguration aims to test how different Reconfiguration Services can be used in order to use information from the system in the Update process. It helps proving the flexibility these Services offer when planning a reconfiguration, allowing to use the information of the current state of the system in order to fix the moment of the update, or to use system data in the reconfiguration.

The update consists in changing the current conveyor state model for a new one that considers the possibility of having more than one piece in the station at a time. The main requirement it must meet is that it must be stopped while the drill is mechanizing the piece.

The new system is depicted in Figure 5.32, showing the variations with respect to the previous version in yellow.
For this reconfiguration, there are two extra requirements:

- The new conveyor should be started on the same state it had before the update.
- The disconnection of the old conveyor model and connection of the new one must start just when the drill is starting its operation, i.e. when its state “move_down” is active.

The ECC of the new conveyor model is more complex than in the previous version, as shown in Figure 5.33. It considers all the possibilities for sensor activation from each state, and includes all the states considered in the previous version, so that it can be restarted in the same state.
5. CASE STUDIES

In order to update the system, the Reconfiguration Application of Figure 5.34 is used.

![Reconfiguration Application with IEC 61499](image)

Figure 5.34: Reconfiguration Application with IEC 61499

It is started when \texttt{start} is triggered. Then it creates the new instance of the new type \texttt{conveyor2} using \texttt{rec\_CREATE\_FB}. Then it stops the old conveyor instance and connect the system to the new conveyor using the Subapplication \texttt{Create\_Con} outlined in Figure 5.35.

![Reconfiguration Application with IEC 61499](image)

Figure 5.35: Reconfiguration Application with IEC 61499
After creating the connections, it uses the Service \texttt{rec\_READ\_STATE} to get the current value of the suspended old conveyor model, which is written in the new one using \texttt{rec\_WRITE\_STATE}.

Then, the Service \texttt{rec\_READ\_STATE} is used again to get the ECC state from the drill until its value coincides with the desired state. Only then, the reconfiguration continues, starting the new conveyor instance, deleting the connections with the old one using the Subapplication \textbf{Delete\_Con} outlined by DFigure 5.36.

![Diagram](image)

\textbf{Figure 5.36:} Reconfiguration Application with IEC 61499

Finally, the old conveyor instance is deleted and the execution time written in a file.

**RESULT**

It worked as expected, setting the new FB to the desired state, and scheduling the last part of the reconfiguration for the specified situation, thus proving that it is possible to use current data from the system in the reconfiguration.
5. CASE STUDIES
Chapter 6

COMPARATIVE ASSESSMENT

Once the test cases have been performed, both using a Reconfiguration Application with Erlang Reconfiguration Services and with an Erlang appup file generated with FBBeam, the different methods can be compared and assessed taking into account diverse characteristics to evaluate. The reconfiguration methods considered for this assessment are the following:

1. Reconfiguration Application using Reconfiguration Services

The first method taken into account is based on the Reconfiguration Services implemented in Erlang explained in Section 4 and their use in the creation of Reconfiguration Applications with 4diac, such as the ones presented in Section 5.

As seen in those chapters, these Reconfiguration Applications include Services performing each required step of the update. They are implemented and run in parallel to the processes to be updated.

2. Reconfiguration by means of an appup file generated by FB-Beam

This method consists in generating the new version of the system with 4diac. Afterwards, this new version is generated in Erlang using FBBeam, and then it is compiled.

Once all the Erlang files for the new system version have been generated, FBBeam is again executed, but this time it is called to generate an appup file. This is performed by comparing the code of the old version to the new code, and from the differences found, it writes in the appup files all the instructions to be followed during reconfiguration.

In order to execute the appup file, the process to be updated must be suspended. Then, the appup is executed using the Erlang function `release_handler:upgrade_app/2` (or
6. COMPARATIVE ASSESSMENT

release_handler:downgrade_app/2 for carrying out downgrades). Once the reconfiguration is finished, the system is resumed at the same execution point it was before reconfiguration.

3. Reconfiguration by means of an appup file generated manually

Given that the appup files automatically generated by FBBeam are still a suboptimal solution and they can get more complex and offer more flexibility and options that what has been accomplished so far, it is important to remark how far the Erlang appup files can get when it comes to DSU.

This method is performed the same way as with an appup file generated by FBBeam, but this time the file must be created or modified manually.

For the assessment of both methods, they are subjected to different criteria:

- Update Accuracy
- Update Time
- Usability
- Extensibility

6.1. Update Accuracy

The Update Accuracy refers to the correctness of the performed update. A Reconfiguration can be considered as correct if the resulting updated system is perfectly consistent with what it was expected to be, i.e. that all the needed changes were successfully performed without any errors.

Therefore, the requirements to fulfill to be considered accurate methods are the following:

1. All the possible updates can be performed.

2. It does not lead to errors during update.

In all the considered case studies, all the reconfiguration actions needed are possible with the tested reconfiguration methods. The FBBeam compiler compares both versions and makes an appup based on all the changes introduced, so it catches every variation that needs to be done.
In the case of an appup manually written, it also works as expected, since if an automatically generated appup can perform every update without errors, this option cannot fail, given that it also permits corrections and inclusion of more instructions.

On the other hand, the Reconfiguration Applications can include all the possibly needed Reconfiguration Services, and all of them were successfully implemented in Erlang. Therefore, all possible updates can be carried out.

Regarding the possibility of errors, the appup (both generated by FBBeam or manually) does not lead to errors, because the appup transforms the code into the new version of code, so the only errors it can lead to are those the new code might include. However, that does not mean a failed update, but a wrong system design.

In Reconfiguration Application, if the values given to the Reconfiguration Services are wrong, the Reconfiguration execution will not execute the Service with wrong data, and the update will be stopped there. But this is also not a error caused by the reconfiguration method, but for a wrong use of it, since in this case the reconfiguration tried leads to a system with mistakes, and the Reconfiguration Services avoid that to happen.

However, there are cases in which the Reconfiguration Application will not work. As there are some Reconfiguration Services that require certain data sent by the processes in the system to be updated, if a process is suspended using `rec_STOP` it can not provide that information anymore.

For that reason, it must be taken into account that if a connection or some data of a FB must be updated, that FB instance can not be suspended before. If it suspended the Reconfiguration will not be successfully finished.

This problem can also happen when a there is in an algorithm of a FB instance that suspends that instance for a while. This can happen, for example, when implementing delays using the function `timer:sleep/1`, that suspends the process during a given amount of time.

In this case, the Reconfiguration will wait until the FB instance is not suspended anymore, leading to longer reconfiguration times. If that instance is never restarted, the Reconfiguration Application will not continue its execution.
6. COMPARATIVE ASSESSMENT

<table>
<thead>
<tr>
<th></th>
<th>FBBeam appup file</th>
<th>Manual appup file</th>
<th>Reconfiguration Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>It is possible to perform ALL update steps</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>It does not lead to errors during update</td>
<td>✓</td>
<td>✓</td>
<td>It can lead to errors with suspended applications</td>
</tr>
</tbody>
</table>

Table 6.1: Update Accuracy Results

The three methods can successfully perform every reconfiguration, but a Reconfiguration Application can lead to errors if suspended applications are not properly handled.

6.2. Update Time

The update time refers to how long does it take to perform the Reconfiguration. For time measurements, the same files will be used for both the manually and the automatically generated appup files, since for this comparison, a complex appup file is not needed, and the one generated by FBBeam is also representative of what can be generated manually.

In the case of the use of an appup file, the update time is the time it takes to perform all the instructions in the appup. In order to measure it, an Erlang file was created to measure the time before calling the update and again once the update is done, and then write the time difference (in nanoseconds) in a file, as shown in Listing 6.1.

```erlang
1 % Start the system application
2 io:format("-p\n",[application:start('TestCase_Tank_App')]),
3 timer:sleep(50),
4 % Time measurements and update
5 T1=erlang:monotonic_time(nanosecond),
6 release_handler:upgrade_app('TestCase_Tank_App', "TestCase_Tank_App-2.0/"),
7 T2=erlang:monotonic_time(nanosecond),
8 % Time calculation and writing in file
9 Tdiff=T2-T1,
10 file:write_file("TANK_Times_Appup_1-2.txt", io_lib:fwrite("-p\n", [Tdiff]),[append])
```

Listing 6.1: Algorithm to measure the time for FBBeam appup files

When using a Reconfiguration Application, the Update Time is the time it takes to start and execute the Reconfiguration Application. In order to measure it, an Erlang file was created that at first measures the time and writes it on a text file. Then, the
Reconfiguration Application is started and a triggering 'START' event is sent to execute the reconfiguration. All this process is performed by the piece of Erlang code in Listing 6.2 (in this case applied to the interconnected tanks test case).

```
% Start the system application
io:format("p
",[application:start('TestCase_Tank_App')]),
send_event('TestCase_Tank_App.start','START'),
timer:sleep(Time),
% Record time in a file
file:write_file("TANK_Times_RecApp.txt", io_lib:fwrite("T1,”p
", [erlang:

monotonic_time(nanosecond)]),[append]),
% Start the Reconfiguration Application
io:format("p
",[application:start('TestCase_Tank_ReconfApp')]),
send_event('TestCase_Tank_ReconfApp.start','START'),
timer:sleep(200),
% Stop both applications
io:format("p
",[application:stop('TestCase_Tank_App')]),
io:format("p
",[application:stop('TestCase_Tank_ReconfApp')]).
```

Listing 6.2: Algorithm for starting the test and measuring the starting time

At the end of each Reconfiguration Application, a FB instance write_time is added. This instance writes the end time in the same file where the starting time was written (given as the input value ‘FILE’), using the Erlang command in Listing 6.3. The update time is computed as the difference of both time values.

```
file:write_file("IM#im.'FILE', io_lib:fwrite("T2,”p
", [erlang:

monotonic_time(nanosecond)]),[append]),
```

Listing 6.3: Algorithm for time measurement in write_time

In order to measure the time, the Erlang function `erlang:monotonic_time/1` was used in both cases, setting the time unit in nanoseconds. This function measures a monotonically increasing time starting at a non-specified point in time.

The above explained time measurements were applied to the test cases of the Interconnected tanks (Section 5.1.1) and the Drum Boiler (Section 5.2.1), both using an appup file and a Reconfiguration Application. The model of the Machining Station (Section 5.3.1) was not tested, since in this case the part of the reconfiguration is triggered by the information coming from the system, so it does not depend just on the efficiency of the reconfiguration method.

As the models where it was tested are cyclic and a delay set the time between cycles, the reconfiguration time can depend on the delay time, since the lower the delay, the more memory, I/O and scheduler is using, and thus, the slower the process can be.

Each time test was run 100 times, in order to have a wide enough sample size, since depending on which state the application is, it would take a different amount of time to reconfigure it.

Ainara Matey Benito
6. COMPARATIVE ASSESSMENT

In the Figure 6.1 and the Figure 6.2 the memory, I/O and Scheduler utilization for a delay of 0ms and 1ms respectively, applied to the Test Case of the Drum Boiler.

There is a big difference between the system with no delay and the one with a delay, even being just 1 ms. When the delay is suppressed, the memory usage is around 100 times higher and the I/O usage around 10 times higher in the case without delay.

The same procedure was done with a delay of 5ms and 10 ms, and the results are very close to the ones with a delay of 1 ms.

For the case of the appup, the delay time does not have any influence on the execution time, since the update is performed with the system suspended.

For all these reasons, the time tests were performed for the appup case, and for the different Reconfiguration Applications with delays of 0ms, 1ms, 5ms and 10 ms. For each of the test cases (The case of the interconnected Tanks and the one of the Drum Boiler), the 2 considered updates described in Section 5 are tested:

- v1.0-v2.0 refers to the Model Update
- v1.0-v3.0 refers to the Control Update

The time results are presented as histograms for each case, displaying the time distribution for each case. They are presented in the tables 6.2, 6.3, 6.4, 6.5, and present all the values in milliseconds.

Figure 6.1: Boiler Test Case Observer for Delay=0
Comparison of Dynamic Software updating methods for IEC 61499 with Erlang

Figure 6.2: Boiler Test Case Observer for Delay=1
6. COMPARATIVE ASSESSMENT

INTERCONNECTED TANKS TEST CASE - MODEL UPDATE

<table>
<thead>
<tr>
<th>Tank Level</th>
<th>µ</th>
<th>σ</th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1.0 v2.0</td>
<td>23.47</td>
<td>15.83</td>
<td>6.52</td>
<td>83.04</td>
</tr>
<tr>
<td>v1.0 v2.0</td>
<td>3.63</td>
<td>0.89</td>
<td>2.58</td>
<td>8.61</td>
</tr>
<tr>
<td>v1.0 v2.0</td>
<td>3.24</td>
<td>1.06</td>
<td>2.28</td>
<td>8.67</td>
</tr>
<tr>
<td>v1.0 v2.0</td>
<td>3.27</td>
<td>0.89</td>
<td>2.28</td>
<td>7.37</td>
</tr>
<tr>
<td>v1.0 v2.0</td>
<td>9.72</td>
<td>1.79</td>
<td>8.81</td>
<td>24.49</td>
</tr>
</tbody>
</table>

Table 6.2: Tank levels
Table 6.3: Reconfiguration Times for the Interconnected tanks case. Control Update
### 6. COMPARATIVE ASSESSMENT

#### DRUM BOILER TEST CASE - MODEL UPDATE

<table>
<thead>
<tr>
<th>Model Update</th>
<th>Mean (μ)</th>
<th>Standard Deviation (σ)</th>
<th>Minimum (min)</th>
<th>Maximum (max)</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1.0.v2.0</td>
<td>6.58</td>
<td>3.43</td>
<td>2.99</td>
<td>25.33</td>
</tr>
<tr>
<td>v1.0.v2.0</td>
<td>3.57</td>
<td>1.28</td>
<td>2.74</td>
<td>14.72</td>
</tr>
<tr>
<td>v1.0.v2.0</td>
<td>6.60</td>
<td>1.26</td>
<td>2.71</td>
<td>13.93</td>
</tr>
<tr>
<td>v1.0.v2.0</td>
<td>3.65</td>
<td>1.35</td>
<td>2.75</td>
<td>14.33</td>
</tr>
<tr>
<td>v1.0.v2.0</td>
<td>11.12</td>
<td>0.98</td>
<td>10.29</td>
<td>19.38</td>
</tr>
</tbody>
</table>

*Table 6.4: Reconfiguration Times for the Drum Boiler case. Model Update*
The Update times are lower in the Control Updates than in the Model Updates. That happens because the Reconfiguration Applications used in these test cases for Control Updates are simpler, and therefore, take less time.

When using a Reconfiguration Application, the Reconfiguration times for a cycle delay > 0 is always very similar and does not depend on the cycle time. Therefore, as long as there is some cycle delay, its duration does not affect reconfiguration time in Reconfiguration Applications.

When using a Reconfiguration Application, with cycle delay > 0, the update time is small (mean of 1-5 ms), and lower than the time taken by the appup (5-12 ms).
6. COMPARATIVE ASSESSMENT

- When using a Reconfiguration Application, with cycle delay = 0, the update times are highly disperse, always bigger than with a non-zero delay. That happens due to the high memory and I/O it requires, since it makes it more difficult to execute in parallel the updating processes, and this time depends more on the process point in which the update takes place.

- There is not a clear preference in time of an appup file (from FBBeam or manually generated) and the Reconfiguration Application without delay, since it depends of the case, and on if there is a nonzero cycle delay or not.

A summary of the results is presented in Table 6.6.

<table>
<thead>
<tr>
<th>Cycle delay</th>
<th>Recommended method</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;0</td>
<td>Reconfiguration Application</td>
<td>The times are generally lower</td>
</tr>
<tr>
<td>0</td>
<td>FBBeam/Manual appup file</td>
<td>The dispersion of the data is lower, and it is difficult to determine if one method is faster than the other</td>
</tr>
</tbody>
</table>

Table 6.6: Time Results

The Update times are considerably better using a Reconfiguration Application if there is a non-zero delay. With a cycle delay = 0, the time values depend on the case, but the appup offers a tighter time uncertainty.

6.3. Extensibility

The extensibility of the method corresponds to how easy is to add new functionalities to the reconfiguration method.

The Reconfiguration Applications are more flexible than appup files generated by FBBeam, because of two reasons. The first one is that as it consists of SIFBs, it is modular, and makes it easier to implement new Reconfiguration Services if needed. The only requirements to do so is to be familiar with Erlang and know how the IEC 61499 systems are implemented in Erlang.

The second reason is that it allows many more functionalities, since it can be used together with any other FB that can be implemented by anyone, and easily shared. Some examples of the functionalities that this feature can allow are the following:
- Decide the order in which the update steps are executed.
- Perform some part of the update just if a I/O value or an Internal Value of a FB has a certain value.
- Start the update (or a part of it) just when a FB instance is in a specific state.
- Write the I/O or Internal variables of the FB instances depending on the current values of other instances.

These are just some examples of how the updating functionalities can be extended with their combination with other regular FBs.

In the case of FBBeam, it does not provide that flexibility when generating automatically the appup file, cause it makes the entire update based on all the code changes while the code is suspended, so it does not let any extra functionality.

Moreover, in order to change how the resulting appup is structured, or what does it include, being familiar with Erlang and how the FBs are implemented in Erlang is not enough. You should also be familiar with Python and with the FBBeam compiler, since a code change of the compiler would be required. However, it proves a good resource as a basis for an appup, that can later be modified or extended by the user to create more suitable appups.

However, the appup files generated by FBBeam are not optimal yet, they are just a minimal functioning example of an appup. In a manually created or modified appup file, the order of the steps can be chosen, and more steps can be included if needed.

A summary of the extensibility features of each method is outlined in Table 6.7

<table>
<thead>
<tr>
<th>Features for extensibility</th>
<th>Reconfiguration Application</th>
<th>FBBeam appup file</th>
<th>Manual Appup file</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Modularity</td>
<td>-Provides with a basic functioning appup file</td>
<td>-Change of the order of the recon. steps</td>
<td></td>
</tr>
<tr>
<td>-Interaction with any FB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-Change of the order of the recon. steps</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 6.7: Extensibility Results**

The modularity of the Reconfiguration Apps offer much more freedom, flexibility and code reusability, with just the requirement of being familiar with Erlang.
6.4. Usability

The usability of each method deals with how easy is to use it, and with how skillful should the user be.

Regarding the needed skills, updating with an appup file just requires to be familiar with basic Erlang commands, 4diac and the related standard IEC 61499, as well as with the commands needed to use FBBeam, while for creating an appup file manually, it is necessary to be familiar with Erlang. However, when using Reconfiguration Applications, it is also important to be familiar with the Reconfiguration Services and know how to use them.

When it comes to simplicity, using a Reconfiguration Application involves designing it, while for generating an appup file with FBBeam, a new version of the system must be designed. Building up a Reconfiguration Application can be much more difficult and complex, since for every change, even for a simple one, at least one Service should be added, and in most of the cases, a number of Service instances is required. Moreover, if more flexibility is needed, the Reconfiguration Application can grow more complex. It is also clearer to make changes in the new version of the system, since it is easier to visualize where the changes are.

On the other side, manually building an appup can also be complicated for complex updates, since it is not so user-friendly to manually determine which modules have to be changed and exactly how.

It is also important to consider how both methods handle human errors. When using Reconfiguration Applications it is more probable to make mistakes, since there are a lot of values that should be typed in the data inputs of the Reconfiguration Services. However, most of these possible errors are correctly detected, making the reconfiguration stop and informing of what the problem was.

Nevertheless, when making a mistake in the new version of the system, the only errors that will be those detected during the compilation. With regards to manually generated appup files it is also easy to forget some steps of the reconfiguration, since the update it is not so visually and clearly presented.
Comparison of Dynamic Software updating methods for IEC 61499 with Erlang

<table>
<thead>
<tr>
<th>Required Skills</th>
<th>FBBeam appup file</th>
<th>Manual appup file</th>
<th>Reconfiguration Application</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-Erlang basic commands</td>
<td>-Erlang</td>
<td>-Erlang basic commands</td>
</tr>
<tr>
<td></td>
<td>-4diac and IEC 64199</td>
<td>-IEC 64199</td>
<td>-4diac and IEC 64199</td>
</tr>
<tr>
<td></td>
<td>-FBBeam</td>
<td></td>
<td>-FBBeam</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-Reconfiguration Services</td>
</tr>
<tr>
<td>Simplicity</td>
<td>It just requires a couple of FBBeam commands</td>
<td>It requires to write a complete functioning Erlang appup file</td>
<td>It requires to prepare an entire Reconfiguration Application, that can be sometimes complex</td>
</tr>
<tr>
<td>Human Errors handling</td>
<td>Just notices if there are errors during compiling the new version</td>
<td>Just while compiling the appup</td>
<td>Most of the human errors are handled and informed</td>
</tr>
<tr>
<td>Extra functionalities</td>
<td>It also allows downgrading to the original version</td>
<td>It also allows downgrading to the original version</td>
<td>It offers a lot of flexibility during the update</td>
</tr>
</tbody>
</table>

Table 6.8: Usability Results

6.5. Other reconfiguration issues

A crucial point to remark, that makes a great difference between both methods, is how the reconfiguration is performed.

On the one hand, when using an appup, both generated by FBBeam or manually, some parts of the code are replaced with the new code, making the change permanent (if wished, if not a downgrade can be performed).

However, in order to do this all the application must be suspended before, and it will not be resumed until the update is complete. Although it starts again at the same point where it was left before reconfiguration, the control action has to be suspended during some milliseconds. Nevertheless, it is important to consider that this only happens in the current implementation of FBBeam. When writing an appup by hand, it can be done so that just the needed modules are suspended, and not the entire application.

On the other hand, when using a Reconfiguration Application, the processes are modified, but the code remains the same. The main consequence of this is that if the application is
terminated, all the changes made would be lost. Nevertheless, the advantage it provides is that the system does not need to be suspended at any time, or if necessary, just a part of it can be suspended without affecting the rest of the system.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Permanent changes</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>No need to suspend</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 6.9: Other Reconfiguration issues

The main conclusion out of this is that for critical infrastructures, where some milliseconds can make a great difference, or for cases when more flexibility during update is needed, a Reconfiguration Application would be ideal. However, if the system needs to be terminated at some point, the appup is needed to preserve the changes.
Chapter 7

DISCUSSION

After implementing all the Reconfiguration Services that were considered necessary for making every update possible, it was proven that Erlang provides with many functionalities that can successfully update processes, without the need to terminate or even suspend the application execution.

With the implementation of different Reconfiguration Applications for various test cases, the option to combine all the Reconfiguration Services has been proven successful, since they had really good results when working together for carrying out more complex updates, which are performed in really little time. However, this is also the case of a reconfiguration made with an appup file generated by FBBeam. Therefore, it is necessary to set in which ways is the appup file better and what can the Reconfiguration Applications offer that it is not already performed by the appup files.

One of the advantages the Reconfiguration Applications provide, is that they can be executed in parallel to the reconfigured system, while for executing the appup file generated by FBBeam, the system applications must be suspended during the time the reconfiguration takes. That could be really beneficial for those cases where the control action can not be suspended, even for some milliseconds, because it can lead to critical situations.

Another benefit the use of Reconfiguration Applications brings, is that it offers a lot of flexibility when updating compared to the appup files. FBBeam will always generate the same appup file when two versions are given as inputs. The only possibility to alter how the update is performed is by changing the appup file code by hand.

However, when creating a Reconfiguration Application the implemented Reconfiguration Services can be combined with each other and with any other FB types in order to provide the update with new functionalities. Some examples for functionalities provided by this combination of FB instances are the following:
7. DISCUSSION

- The order of the reconfiguration steps can be chosen depending on how it better fits each case.

- The parameters of the FB instances can be set as a function of the current values of other instances.

- Some parts of the Reconfiguration Application are executed only if the system is in a certain situation (e.g. a FB instance is in a certain state or if a parameter value has reached the desired value).

- Loops can be created so that the reconfiguration can just continue when some situation has been reached.

Nevertheless, the Reconfiguration Applications also present some drawbacks compared to the appup files generated by FBBeam. The flexibility it offers brings along more complexity when creating the Reconfiguration applications. One Reconfiguration Service must be added for every change the update requires. Therefore, some Reconfiguration Applications can turn out huge and complicated.

In such cases, it is easy to make a mistake forgetting one of the necessary steps, or typing some input value wrong. Most of these mistakes are correctly handled by the Reconfiguration Services, and the Reconfiguration will not continue from that point. However, the reconfiguration would remain uncompleted until corrected. That is why it is always crucial to perform a correct verification and validation before applying a reconfiguration to a real system, this way avoiding problems derived from such mistakes.

This is less probable to happen when using an automatically generated appup. The only mistakes that can be generated are those related with a bad implementation of the new version, and that were not identified while compiling.

Another drawback the Reconfiguration Applications present, is that they call Erlang functions that change the running process, but the code remains the same. As a result, if the application is terminated, all the changes will be lost. For changing the underlying code it is necessary to use the appup.
Chapter 8

CONCLUSIONS

The aim of this project was to implement the necessary tools to create Reconfiguration Applications using Erlang and following the standard IEC 61499, and to compare its performance to other Erlang-based DSU methods, in order to find out which Erlang functionalities can work better for DSU purposes.

In order to do so, the compiler FBBeam was extended with a number of selected Reconfiguration Services, which were chosen based on the current state of the art, the standard IEC 61499, and adapting them to how the compiler FBBeam implements IEC 61499 systems in Erlang.

All the considered Reconfiguration Services were implemented in Erlang, using functions from different OTP libraries, and making sure they were correct and could not only perform the desired tasks, but also handle errors properly, by performing little reconfiguration tests during implementation phase.

Once every Service was implemented, various larger case studies were modeled in order to test different aspects from the Reconfiguration Services created, trying them working combined in Reconfiguration Applications. These case studies were also updated using another tool of FBBeam based in Erlang functionalities, the appup files it can automatically generate, providing with the instructions to follow during an upgrade.

Taking the results of these tests as a base, a comparative analysis was performed between both methods. It was concluded that both methods correctly perform all the needed reconfiguration tasks without errors, if they have been correctly validated and do not contain designing human mistakes.

The main advantages the Erlang Reconfiguration Services present is that it is not necessary to suspend any process to perform an upgrade, while when using appup files it is required to suspend at least the modules that suffer some sort of change. Moreover, the
Reconfiguration Services provide with a lot of flexibility when designing a reconfiguration, since their modularity allow to combine them in different orders and together with normal FBs. It is also possible to retrieve information from the running system that can be used for the reconfiguration.

However, they have as well some limitations, that the appup files generated by FBBeam can overcome. First of all, there is the fact that when using these services, the code itself is not being changed, it is just the running processes that are altered, while the appup file changes the code, and the variations are made permanent also if the system is terminated.

Moreover, it is important to consider that designing a Reconfiguration Application is more complex and time consuming than making changes on the system to create the new version. That can lead to more mistakes while its creation, which should be avoided with verification and validation.

Finally, there are some features in the Reconfiguration Application that are at the moment suboptimal, like the fact that some Services must call functions inside a FB in the system, and that can lead to errors if they have to be suspended. This and other problems are something to solve on in future work.

As a general conclusion of the correct use of both methods, it would be ideal to use both. The Reconfiguration Application would be used to update the system without the need to suspend its execution, taking advantage of all the flexibility it offers. On the other hand, if the system must be suspended, the appup can be then executed in order to load the new code to the system.

Following these guidelines, the best of each method would be put to use.

Regarding environmental and ethical implications of this project, it is important to consider how it helps making updates easier, which avoids having to stop an industrial process whenever an update needs to be made.

For this reason, more updates can be performed, and this way, for every improvement found a reconfiguration can take place without having to wait to more features to be updated. Therefore, its easier to have the system and its security software up to date, without a significant cost in money, time and energy. That leads to a better efficiency and cybersecurity for Industrial Plants, and therefore providing workers with better job stability and to less working hours to achieve the same result.

DSU can also generate new jobs, related to the preparation and execution of this updates, and in charge of making sure that the update is fault-tolerant and effective.
Chapter 9

OUTLOOK AND FUTURE WORK

It has been concluded that the Reconfiguration Applications offer a lot of advantages, but there are still options for improvement.

First of all, there are some Reconfiguration Services that only work if the FB they call is not suspended (e.g, `rec_CREATE_CON`). That can lead to blocking situations during reconfiguration. Finding a way to change the internal data of a FB instance without having to call internal functions from its module would solve this problem.

Currently, most of the Reconfiguration Services recognize mistaken input variables, and stop the reconfiguration when that happens. However, there are still two types of mistakes that it still do accept. One case is `rec_CREATE_CON`, that accepts incorrect values for the destination of the connection. The other case is `rec_WRITE_STATE`, that accepts non-existent states.

That problem can be solved adding new clauses for `Module:handle_event/4` to provide the necessary information to check whether the input values are valid or not.

However, this solution carries a consequence. If it is implemented that way, those FB instances from which the information is retrieved must be running in order for the reconfiguration to work. Therefore, trying to solve one problem that only deals with human errors, could lead to another, that can result more limiting when designing Reconfiguration Applications.

Another feature to make it more user-friendly is to make it accept strings as data inputs when atoms are required, and convert them to atoms.

Regarding the combined use of Reconfiguration Applications and appup files, there are solutions that can be implemented in order to make it easier to work with both methods together and make the most of both.
One way of making that possible is to extend the FBBeam compiler so that when it recognizes the Reconfiguration Services, makes a copy of the XML file with the information of the original system, but introducing the variations corresponding to the included Reconfiguration Services. Afterwards, the new system with all the modifications would be compiled into Erlang by FBBeam and the appup file would be generated.

With this implementation, both methods could be easily integrated. The new generated system could be also opened in 4diac in order to check that the Reconfiguration Application makes what it is desired.

Another way of combining both methods could be to make just the opposite: generate a Reconfiguration Application from the comparison between the two versions. Once the new updated system has be created in 4diac and FBBeam is executed to make the appup file, it could also generate an XML system file including instances of all the Reconfiguration Services that are needed to perform all the changes, according to the variations found during the comparison of both versions. Then, this system could be modified in 4diac to change the order of execution, or to add new FB, but always making sure that all the Reconfiguration Services have been included with the correct input data.

Performing all these improvements, an optimal use of both methods could be accomplished, making DSU much easier and useful.
Chapter 10

PROJECT PLANNING

10.1. Temporal planning

In order to perform the temporal planning of the project, a Work Breakdown Structure (WBS) was performed, dividing the project into relevant work packages, each of them corresponding to a project phase:

1. **Project Definition and Planning:** Initial approach to the project, including an initial research, enough to define the project scope and general lines. All this information is gathered in the Project Exposê, which is prepared and presented at the end of this phase.

2. **Research and Formation:** Deep research into the topic, collecting different sources and taking the relevant information. Moreover, this phase includes the formation of the topics and tools that are not mastered enough.

3. **Implementation:** Development of the proposed solution.

4. **Case studies:** Performance of various tests to proof the validity of the results. For this purpose, a number of test systems are modeled.

5. **Comparative Assessment:** Comparative analysis of the implemented solution together with existing solutions, according to different factors.

6. **Document:** Elaboration and correction of the Project document.

7. **Presentation:** Elaboration and preparation of the project presentation.

Taking into account these project packages, the Work Breaking Structure on Figure 10.1 is created, including the most relevant pieces of work inside of each package. This way, a
clear and visual project decomposition is set, making it easier to manage the work to be performed.

![Project WBS Diagram]

**Figure 10.1: Project WBS**

According to this project structure, the Gantt Diagram on Figure 10.2 is created. In order to develop it, each project task was assigned a number of days to complete it, always considering an internal buffer for every task.

This planning proves crucial to the completion of projects of this kind, since the time to complete a Master thesis is of 6 months, being of special importance the accomplishment of the deadline.

In this case, the project was started and signed on April 15th, being therefore the due date of the project the October 15th.
Figure 10.2: Gantt Diagram
10.2. Project Budget

For the completion of the project, only free software and software with free student licenses was used, so there is no cost for Software license.

No new computer or facility had to be used, so the only material cost is the depreciation of the laptop used. Considering the total life of the laptop 8 years and a estimated total cost of 700€, and considering the time it was used for other purposes as negligible during the 6 months of duration of the project, the depreciation cost of the laptop is the one computed in Equation 10.1.

\[
C_{\text{deprec}} = 700\€ \frac{0.5 \text{years}}{8 \text{years}} = 700\€ \times 6.25\% = 43.75\€
\]

(10.1)

The engineering work costs are estimated from the cost per hour and the number of hours. For this project, just one student was needed. Research students in TUM university are paid 12 €/hour worked. It is considered that every day worked, an average of 8 hours were worked.

<table>
<thead>
<tr>
<th>Engineering costs</th>
<th>Units (h)</th>
<th>Price(€)</th>
<th>Cost(€)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Planning and Research</td>
<td>280</td>
<td>12</td>
<td>3360</td>
</tr>
<tr>
<td>Project Implementation and Testing</td>
<td>512</td>
<td>12</td>
<td>6144</td>
</tr>
<tr>
<td>Document and Presentation</td>
<td>248</td>
<td>12</td>
<td>2976</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Depreciation costs</th>
<th>Time</th>
<th>Price(€)</th>
<th>Cost(€)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Laptop</td>
<td>6.25%</td>
<td>700</td>
<td>43.75</td>
</tr>
</tbody>
</table>

| Total               |       |          | 12523.75|

Table 10.1: Project Budget
REFERENCES


[18] Redacted for Peer Review. Fbbeam: Yet another iec 61499 implementation.

LIST OF FIGURES

1. Interfaz de un FB [9] ........................................ VII
2. Interfaz de rec_QUERY_FB_STATUS ........................ XII
3. Sistema de test ................................................ XII
4. Diagrama del sistema de tanques antes de la reconfiguración ........ XIII
5. Modelo de tanques con IEC 61499 antes de la Reconfiguración .... XIV
6. Modelo del tanque tras la primera Reconfiguración ................ XIV
7. Aplicación de Reconfiguración para añadir un tercer tanque ........ XV
8. Niveles de los tres tanques durante la primera reconfiguración ........ XV
9. Aplicación de Reconfiguración para el PID ........................ XVI
10. Niveles de los tanques durante la segunda reconfiguración .......... XVI
11. Diagrama del sistema de la caldera antes de la reconfiguración ...... XVII
12. Modelo de caldera con IEC 61499 antes de la Reconfiguración ...... XVII
13. Diagrama del sistema de la caldera tras la primera reconfiguración ... XVIII
14. Aplicacion de Reconfiguración de la caldera ........................ XVIII
15. Aplicación de Reconfiguración del controlador ...................... XIX
16. Nivel del tambor durante la reconfiguración ........................ XIX
17. Diagrama del sistema de mecanizado ................................ XX
18. Modelo con IEC 61499 ........................................... XX
19. Aplicación de Reconfiguración de la cinta ........................................ XXI
20. EDS del proyecto ................................................................. XXVIII
21. Diagrama de Gantt ................................................................. XXIX

2.1. Interface of a FB [9] ............................................................. 7
2.3. Background schema ............................................................ 12

3.1. Testing System ................................................................. 14
3.2. Testing System ................................................................. 16

4.1. Interface of rec_QUERY_FBs ............................................... 22
4.2. Interface of rec_QUERY_FB_STATUS .................................... 23
4.3. Interface of rec_QUERY_CON .............................................. 24
4.4. Interface of rec_QUERY_TYPE_LIST .................................... 26
4.5. Interface of rec_QUERY_TYPE ............................................. 27
4.6. Testing System ................................................................. 29
4.7. Reconfiguration Application for Query Services ...................... 29
4.8. Execution of a Function Block State Machine according to IEC 61499 [8] 31
4.9. Execution of a Function Block State Machine according to FBBeam implement- 32
4.10. Interface of rec_STOP ....................................................... 33
4.11. Interface of rec_START ..................................................... 34
4.12. Interface of rec_KILL ......................................................... 35
4.13. Testing System ................................................................. 37
4.14. Reconfiguration Application for Execution Control Services ........ 37
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.15.</td>
<td>Interface of rec_CREATE_FB</td>
<td>38</td>
</tr>
<tr>
<td>4.16.</td>
<td>Interface of rec_DELETE_FB</td>
<td>41</td>
</tr>
<tr>
<td>4.17.</td>
<td>Interface of rec_CREATE_CON</td>
<td>42</td>
</tr>
<tr>
<td>4.18.</td>
<td>Interface of rec_DELETE_CON</td>
<td>45</td>
</tr>
<tr>
<td>4.19.</td>
<td>Interface of rec_CREATE_SUBAPP</td>
<td>47</td>
</tr>
<tr>
<td>4.20.</td>
<td>Interface of rec_DELETE_SUBAPP</td>
<td>50</td>
</tr>
<tr>
<td>4.21.</td>
<td>Testing System</td>
<td>52</td>
</tr>
<tr>
<td>4.22.</td>
<td>Reconfiguration Application for Structural Services-1</td>
<td>53</td>
</tr>
<tr>
<td>4.23.</td>
<td>Reconfiguration Application for Structural Services-2</td>
<td>54</td>
</tr>
<tr>
<td>4.24.</td>
<td>Interface of rec_WRITE</td>
<td>56</td>
</tr>
<tr>
<td>4.25.</td>
<td>Interface of rec_READ</td>
<td>58</td>
</tr>
<tr>
<td>4.26.</td>
<td>Interface of rec_WRITE_STATE</td>
<td>59</td>
</tr>
<tr>
<td>4.27.</td>
<td>Interface of rec_READ_STATE</td>
<td>61</td>
</tr>
<tr>
<td>4.28.</td>
<td>Testing System</td>
<td>62</td>
</tr>
<tr>
<td>4.29.</td>
<td>Reconfiguration Application for State Interaction Services</td>
<td>63</td>
</tr>
<tr>
<td>5.1.</td>
<td>Diagram of the Tank Model before Reconfiguration</td>
<td>66</td>
</tr>
<tr>
<td>5.2.</td>
<td>Tank Model with IEC 61499 before Reconfiguration</td>
<td>67</td>
</tr>
<tr>
<td>5.3.</td>
<td>Diagram of the Tank Model after Reconfiguration</td>
<td>69</td>
</tr>
<tr>
<td>5.4.</td>
<td>Tank Model with IEC 61499 after Reconfiguration</td>
<td>69</td>
</tr>
<tr>
<td>5.5.</td>
<td>Reconfiguration application for adding a third tank</td>
<td>70</td>
</tr>
<tr>
<td>5.6.</td>
<td>New_Tank Subapplication for adding a third tank</td>
<td>71</td>
</tr>
<tr>
<td>5.7.</td>
<td>New_Con Subapplication for adding a third PID control</td>
<td>71</td>
</tr>
<tr>
<td>5.8.</td>
<td>New_Con Subapplication for adding a third tank</td>
<td>72</td>
</tr>
<tr>
<td>5.9.</td>
<td>Tank Levels when adding a third tank</td>
<td>72</td>
</tr>
</tbody>
</table>
5.10. Reconfiguration Application for changing PID parameters . . . . . . . . . . . . . 74
5.11. Tank Levels during PID reconfiguration . . . . . . . . . . . . . . . . . . . . . . . . 74
5.12. Diagram of the Boiler Model before Reconfiguration . . . . . . . . . . . . . . . . . 76
5.13. Boiler Model with IEC 61499 before Reconfiguration . . . . . . . . . . . . . . . . . 78
5.14. Combustion Model with IEC 61499 before Reconfiguration . . . . . . . . . . . . . 79
5.15. Drum Model with IEC 61499 before Reconfiguration . . . . . . . . . . . . . . . . . 79
5.16. Diagram of the Boiler Model after Reconfiguration . . . . . . . . . . . . . . . . . . 80
5.17. Combustion Model with IEC 61499 after Reconfiguration . . . . . . . . . . . . . . 81
5.18. Reconfiguration Application for updating the combustion process . . . . . . . . . 82
5.19. Subapplication Suspend_FBs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
5.20. Subapplication Write_Values . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
5.21. Subapplication Delete_Con . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83
5.22. Subapplication Create_Con . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
5.23. Reconfiguration Application for updating the controller . . . . . . . . . . . . . . . 85
5.24. New_PID Subapplication . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
5.25. deleteCon Subapplication . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
5.26. Drum Level during reconfiguration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
5.27. Drum Level during reconfiguration (Augmented) . . . . . . . . . . . . . . . . . . . . . . 87
5.28. Diagram of the Machining Model before Reconfiguration . . . . . . . . . . . . . . 88
5.29. Model with IEC 61499 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
5.30. Conveyor ECC before Reconfiguration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
5.31. Drill ECC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
5.32. Diagram of the Machining Model after Reconfiguration . . . . . . . . . . . . . . . . 91
5.33. Conveyor ECC before Reconfiguration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
5.34. Reconfiguration Application with IEC 61499 ................................. 92
5.35. Reconfiguration Application with IEC 61499 ................................. 92
5.36. Reconfiguration Application with IEC 61499 ................................. 93

6.1. Boiler Test Case Observer for Delay=0 ............................................. 100
6.2. Boiler Test Case Observer for Delay=1 ............................................. 101

10.1. Project WBS .................................................................................. 118
10.2. Gantt Diagram ............................................................................... 119
LIST OF TABLES

1. Servicios de Reconfiguración implementados  . . . . . . . . . . . . . . . . . . XI
2. Resultados de Exactitud  . . . . . . . . . . . . . . . . . . . . . . . . . . . XXII
3. Resultados de Tiempos  . . . . . . . . . . . . . . . . . . . . . . . . . . . XXIII
4. Resultados de extensibilidad  . . . . . . . . . . . . . . . . . . . . . . . . . XXIV
5. Resultados de usabilidad  . . . . . . . . . . . . . . . . . . . . . . . . . . . XXV
6. Otros factores  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . XXV
7. Presupuesto del proyecto  . . . . . . . . . . . . . . . . . . . . . . . . . . . XXVIII

4.1. Control Reconfiguration Services [9]  . . . . . . . . . . . . . . . . . . . . . . 19
4.2. Not Implemented Control Reconfiguration Services  . . . . . . . . . . . . . 20
4.3. New introduced Control Reconfiguration Services  . . . . . . . . . . . . . . 21
4.4. I/O in rec_QUERY_FBs  . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
4.5. I/O in rec_QUERY_FB_STATUS  . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.6. I/O in rec_QUERY_CON  . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
4.7. I/O in rec_QUERY_TYPE_LIST  . . . . . . . . . . . . . . . . . . . . . . . . 26
4.8. I/O in rec_QUERY_TYPE  . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
4.9. I/O in rec_STOP  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33
4.10. I/O in rec_START  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
4.11. I/O in rec_KILL  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

Ainara Matey Benito
LIST OF TABLES

4.12. I/O in rec_CREATE_FB .............................................. 39
4.13. I/O in rec_DELETE_FB ........................................... 41
4.15. I/O in rec_DELETE_CON ......................................... 46
4.16. I/O in rec_CREATE_SUBAPP ................................. 48
4.17. I/O in rec_DELETE_SUBAPP .................. 50
4.18. I/O in rec_WRITE .................................................. 56
4.19. I/O in rec_READ .................................................... 58
4.20. I/O in rec_WRITE_STATE ........................................... 60
4.21. I/O in rec_READ_STATE ........................................... 61

5.1. Tank levels ............................................................. 66
5.2. Tank PID values ...................................................... 73
5.3. Selected Variable Values ........................................... 76
5.4. Selected Variable Values ........................................... 77
5.5. Selected Variable Values ........................................... 80
5.6. Boiler PID values ...................................................... 84

6.1. Update Accuracy Results .......................................... 98
6.2. Tank levels ............................................................. 102
6.3. Reconfiguration Times for the Interconnected tanks case. Control Update . 103
6.4. Reconfiguration Times for the Drum Boiler case. Model Update ................ 104
6.5. Reconfiguration Times for the Drum Boiler case. Control Update ................ 105
6.6. Time Results ............................................................ 106
6.7. Extensibility Results .................................................. 107
6.8. Usability Results .................................................. 109
6.9. Other Reconfiguration issues ................................. 110
10.1. Project Budget .................................................... 120
LIST OF TABLES
GLOSSARY OF TERMS

4diac Open source initiative that provides a framework for Distributed Industrial Automation and Control. It offers an IEC 61499 engineering tool and a development and runtime environment [13]

Appup file Erlang file that contains all the necessary instructions to update and downgrade a piece of code to different versions

Distributed Control System Industrial control system that consists of a central station which is in charge of controlling and supervising the different distributed instruments of the system [4]

Dynamic Software Update Also referred as Hot Code Loading or Dynamic Reconfiguration

Erlang Programming language created by Ericsson. This functional language is highly scalable, given that it works with lightweight processes. It can run anywhere in its virtual machine [14]

Execution Control Chart (ECC) IEC 61499 chart that maps the state machine that describes the states and transitions of the operation of a basic Function Block

Function Block Main architectural part of the IEC 61499 standard. It encapsulates a certain algorithm and it is inter-connectable with other FBs through event and data I/O. It can be Basic (BFB), Composite (CFB) or Service Interface (SIFB)

FBBeam Compiler developed in the Technical University of Munich that receives as an input the XML system files provided by 4diac and transforms them into Erlang executable code.

IEC 61499 An Automation Standard published in 2005 by the International Electrotechnical Commission, based on algorithm encapsulation, and which main architectural unit are inter-connectable Function Blocks

OTP behaviour It stores functionalities associated with a certain Erlang application. All the behaviours are stored in libraries
**Reconfiguration Application** It is an IEC 61499 application, which provides with the necessary instructions to update a system.

**Reconfiguration Services** IEC 61499 Services which are in charge of performing the different reconfiguration steps needed to update a system.

**Service Interface Function Block (SIFB)** It refers to the IEC 61499 FB which are beyond the scope of the norm, and therefore have to be defined apart.

**Supervisor** An Erlang Supervisor is in charge of controlling his child processes, i.e. of creating, restarting, stopping, or killing them when necessary.
LIST OF ACRONYMS

BFB  Basic Function Block
CFB  Composite Function Block
DCS  Distributed Control System
DSU  Dynamic Software Update
ECC  Execution Control Chart
FB   Function Block
IDE  Integrated Development Environment
IEC  International Electrotechnical Commission
I/O  Input/Output
LHV  Lower Heating Value
OS   Operating System
OTP  Open Telecom Platform
PD   Proportional, Derivative
PID  Proportional, Integral, Derivative
PLC  Programmable Logic Controller
SIFB Service Interface Function Block
WBS  Work Breakdown Structure
XML  Extensible Markup Language
Appendix A

RECONFIGURATION FBS

Query Services

<table>
<thead>
<tr>
<th>QUERY A LIST WITH ALL THE FB INSTANCES OF AN APP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INPUT EVENTS</strong></td>
</tr>
<tr>
<td>REQ</td>
</tr>
<tr>
<td>CNF</td>
</tr>
<tr>
<td><strong>INPUT VARIABLES</strong></td>
</tr>
<tr>
<td>APP_NAME</td>
</tr>
<tr>
<td><strong>OUTPUT VARIABLES</strong></td>
</tr>
<tr>
<td>STATUS</td>
</tr>
<tr>
<td>LIST</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>QUERY THE PROCESS STATUS OF A FB INSTANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INPUT EVENTS</strong></td>
</tr>
<tr>
<td>REQ</td>
</tr>
<tr>
<td>CNF</td>
</tr>
<tr>
<td><strong>INPUT VARIABLES</strong></td>
</tr>
<tr>
<td>APP_NAME</td>
</tr>
<tr>
<td>FB_NAME</td>
</tr>
<tr>
<td><strong>OUTPUT VARIABLES</strong></td>
</tr>
<tr>
<td>STATUS</td>
</tr>
<tr>
<td>FB_STATUS</td>
</tr>
</tbody>
</table>
A. RECONFIGURATION FBS

### INPUT EVENTS
- **REQ** | Event | Event to require the output connections of a FB instance
- **CNF** | Event | Event to confirm that the connections were provided

### OUTPUT EVENTS
- **APP_NAME** | STRING | Name of the Application containing the FB (atom)
- **FB_NAME** | STRING | Name of the FB instance (atom)

### INPUT VARIABLES
- **STATUS** | STRING | Service Status: RDY, NO_SUCH_OBJECT
- **CON** | STRING | Current Output connections of the FB

### OUTPUT VARIABLES

---

### INPUT EVENTS
- **REQ** | Event | Event to require a list with all the types in an application
- **CNF** | Event | Event to confirm that the list was provided

### OUTPUT EVENTS
- **APP_NAME** | STRING | Name of the Application (atom)

### INPUT VARIABLES
- **STATUS** | STRING | Service Status: RDY, NO_SUCH_OBJECT
- **LIST** | STRING | List with all the FB types in the app

### OUTPUT VARIABLES

---

### INPUT EVENTS
- **REQ** | Event | Event to require the type of a FB instance
- **CNF** | Event | Event to confirm that the type has been provided

### OUTPUT EVENTS
- **APP_NAME** | STRING | Name of the Application containing the FB (atom)
- **FB_NAME** | STRING | Name of the FB instance (atom)

### INPUT VARIABLES
- **STATUS** | STRING | Service Status: RDY, NO_SUCH_OBJECT
- **TYPE** | STRING | FB type

### OUTPUT VARIABLES

---

### INPUT EVENTS
- **REQ** | Event | Event to require suspending a FB instance
- **CNF** | Event | Event to confirm that the FB has been suspended

### OUTPUT EVENTS
- **APP_NAME** | STRING | Name of the Application containing the FB (atom)
- **FB_NAME** | STRING | Name of the FB instance (atom)

### INPUT VARIABLES
- **STATUS** | STRING | Service Status: RDY, NO_SUCH_OBJECT

### OUTPUT VARIABLES

---

### Execution Control Services

---

Escuela Técnica Superior de Ingenieros Industriales (UPM)
Comparison of Dynamic Software updating methods for IEC 61499 with Erlang

### RESUME A FB INSTANCE

**INPUT EVENTS**
- **REQ** Event Event to require resuming a FB instance

**OUTPUT EVENTS**
- **CNF** Event Event to confirm that the FB has been resumed

**INPUT VARIABLES**
- **APP_NAME** STRING Name of the Application containing the FB (atom)
- **FB_NAME** STRING Name of the FB instance (atom)

**OUTPUT VARIABLES**
- **STATUS** STRING Service Status: RDY, NO_SUCH_OBJECT

### TERMINATE A FB INSTANCE

**INPUT EVENTS**
- **REQ** Event Event to require terminating a FB instance

**OUTPUT EVENTS**
- **CNF** Event Event to confirm that the FB has been terminated

**INPUT VARIABLES**
- **APP_NAME** STRING Name of the Application containing the FB (atom)
- **FB_NAME** STRING Name of the FB instance (atom)

**OUTPUT VARIABLES**
- **STATUS** STRING Service Status: RDY, NO_SUCH_OBJECT

### CREATE A FB INSTANCE

**INPUT EVENTS**
- **REQ** Event Event to require creating a FB instance

**OUTPUT EVENTS**
- **CNF** Event Event to confirm that the FB has been created

**INPUT VARIABLES**
- **FB_TYPE** STRING Type of the FB instance (atom)
- **FB_NAME** STRING Name of the FB instance (atom)
- **APP_NAME** STRING Name of the Application containing the FB (atom)

**OUTPUT VARIABLES**
- **STATUS** STRING Service Status: RDY, NO_SUCH_OBJECT

### DELETE A FB INSTANCE

**INPUT EVENTS**
- **REQ** Event Event to require deleting a FB instance

**OUTPUT EVENTS**
- **CNF** Event Event to confirm that the FB has been deleted

**INPUT VARIABLES**
- **FB_NAME** STRING Name of the FB instance (atom)
- **APP_NAME** STRING Name of the Application containing the FB (atom)

**OUTPUT VARIABLES**
- **STATUS** STRING Service Status: RDY, NO_SUCH_OBJECT

---

**Structural Services**
### A. RECONFIGURATION FBS

#### CREATE A DATA OR EVENT CONNECTION BETWEEN TWO FB INSTANCES

**INPUT EVENTS**

| Event | Event to require creating a connection |

**OUTPUT EVENTS**

| Event | Event to confirm that the connection has been created |

**INPUT VARIABLES**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE_CON</td>
<td>STRING</td>
<td>Type of connection: data/event</td>
</tr>
<tr>
<td>SRC_APP_NAME</td>
<td>STRING</td>
<td>Name of the application that contains the Source FB (atom)</td>
</tr>
<tr>
<td>SRC_FB_NAME</td>
<td>STRING</td>
<td>Name of the Source FB instance (atom)</td>
</tr>
<tr>
<td>SRC_FB_PARAM</td>
<td>STRING</td>
<td>Name of the connected output in the Source FB (atom)</td>
</tr>
<tr>
<td>DST_APP_NAME</td>
<td>STRING</td>
<td>Name of the application that contains the Destination FB (atom)</td>
</tr>
<tr>
<td>DST_FB_NAME</td>
<td>STRING</td>
<td>Name of the Destination FB instance (atom)</td>
</tr>
<tr>
<td>DST_FB_PARAM</td>
<td>STRING</td>
<td>Name of the connected input in the Destination FB (atom)</td>
</tr>
</tbody>
</table>

**OUTPUT VARIABLES**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS</td>
<td>STRING</td>
<td>Service Status: RDY, NO_SUCH_OBJECT</td>
</tr>
</tbody>
</table>

#### DELETE A DATA OR EVENT CONNECTION BETWEEN TWO FB INSTANCES

**INPUT EVENTS**

| Event | Event to require deleting a connection |

**OUTPUT EVENTS**

| Event | Event to confirm that the connection has been deleted |

**INPUT VARIABLES**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TYPE_CON</td>
<td>STRING</td>
<td>Type of connection: data/event</td>
</tr>
<tr>
<td>SRC_APP_NAME</td>
<td>STRING</td>
<td>Name of the application that contains the Source FB (atom)</td>
</tr>
<tr>
<td>SRC_FB_NAME</td>
<td>STRING</td>
<td>Name of the Source FB instance (atom)</td>
</tr>
<tr>
<td>SRC_FB_PARAM</td>
<td>STRING</td>
<td>Name of the connected output in the Source FB (atom)</td>
</tr>
<tr>
<td>DST_APP_NAME</td>
<td>STRING</td>
<td>Name of the application that contains the Destination FB (atom)</td>
</tr>
<tr>
<td>DST_FB_NAME</td>
<td>STRING</td>
<td>Name of the Destination FB instance (atom)</td>
</tr>
<tr>
<td>DST_FB_PARAM</td>
<td>STRING</td>
<td>Name of the connected input in the Destination FB (atom)</td>
</tr>
</tbody>
</table>

**OUTPUT VARIABLES**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS</td>
<td>STRING</td>
<td>Service Status: RDY, NO_SUCH_OBJECT</td>
</tr>
</tbody>
</table>

#### CREATE A NEW SUBAPPLICATION

**INPUT EVENTS**

| Event | Event to require the creation of a Subapplication |

**OUTPUT EVENTS**

| Event | Event to confirm that the subapplication has been created |

**INPUT VARIABLES**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBAPP_NAME</td>
<td>STRING</td>
<td>Name of the new Subapplication (atom)</td>
</tr>
<tr>
<td>APP_NAME</td>
<td>STRING</td>
<td>Application where the Subapplication must be created (atom)</td>
</tr>
<tr>
<td>TYPE</td>
<td>STRING</td>
<td>Subapplication type name (atom)</td>
</tr>
</tbody>
</table>

**OUTPUT VARIABLES**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS</td>
<td>STRING</td>
<td>Service Status: RDY, NO_SUCH_OBJECT</td>
</tr>
</tbody>
</table>
Comparison of Dynamic Software updating methods for IEC 61499 with Erlang

### DELETE A SUBAPPLICATION

<table>
<thead>
<tr>
<th><strong>INPUT EVENTS</strong></th>
<th><strong>OUTPUT EVENTS</strong></th>
<th><strong>INPUT VARIABLES</strong></th>
<th><strong>OUTPUT VARIABLES</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ Event</td>
<td>CNF Event</td>
<td>SUBAPP_NAME STRING</td>
<td>CNF Event</td>
</tr>
<tr>
<td></td>
<td></td>
<td>APP_NAME STRING</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TYPE STRING</td>
<td>STATUS STRING</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Service Status: RDY, NO_SUCH_OBJECT</td>
</tr>
</tbody>
</table>

**Input Events**
- REQ Event: Event to require the deletion of a Subapplication

**Output Events**
- CNF Event: Event to confirm that the subapplication has been deleted

**Input Variables**
- SUBAPP_NAME: Name of the new Subapplication (atom)
- APP_NAME: Application where the Subapplication must be created (atom)
- TYPE: Subapplication type name (atom)

**Output Variables**
- STATUS: Service Status: RDY, NO_SUCH_OBJECT

### State Interaction Services

#### WRITE A NEW VALUE IN AN I/O OR INTERNAL VARIABLE OF A FB INSTANCE

<table>
<thead>
<tr>
<th><strong>INPUT EVENTS</strong></th>
<th><strong>OUTPUT EVENTS</strong></th>
<th><strong>INPUT VARIABLES</strong></th>
<th><strong>OUTPUT VARIABLES</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ Event</td>
<td>CNF Event</td>
<td>APP_NAME STRING</td>
<td>CNF Event</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FB_NAME STRING</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PARAMETER STRING</td>
<td>STATUS STRING</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VALUE STRING</td>
<td>Service Status: RDY, NO_SUCH_OBJECT</td>
</tr>
</tbody>
</table>

**Input Events**
- REQ Event: Event to require to write the value of an Input, an Output, or an Internal Variable from a FB

**Output Events**
- CNF Event: Event to confirm that the parameter has been written

**Input Variables**
- APP_NAME: Name of the Application containing the FB (atom)
- FB_NAME: Name of the FB instance (atom)
- PARAMETER: Name of the parameter to be changed (atom)
- VALUE: New value of the parameter

**Output Variables**
- STATUS: Service Status: RDY, NO_SUCH_OBJECT

#### READ THE VALUE IN AN I/O OR INTERNAL VARIABLE OF A FB INSTANCE

<table>
<thead>
<tr>
<th><strong>INPUT EVENTS</strong></th>
<th><strong>OUTPUT EVENTS</strong></th>
<th><strong>INPUT VARIABLES</strong></th>
<th><strong>OUTPUT VARIABLES</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ Event</td>
<td>CNF Event</td>
<td>APP_NAME STRING</td>
<td>CNF Event</td>
</tr>
<tr>
<td></td>
<td></td>
<td>FB_NAME STRING</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>PARAMETER STRING</td>
<td>STATUS STRING</td>
</tr>
</tbody>
</table>

**Input Events**
- REQ Event: Event to require to read the value of an Input, an Output, or an Internal Variable from a FB

**Output Events**
- CNF Event: Event to confirm that the parameter has been read

**Input Variables**
- APP_NAME: Name of the Application containing the FB (atom)
- FB_NAME: Name of the FB instance (atom)
- PARAMETER: Name of the parameter to be read (atom)

**Output Variables**
- STATUS: Service Status: RDY, NO_SUCH_OBJECT
### A. RECONFIGURATION FBS

**CHANGE THE ECC STATE OF A FB INSTANCE**

<table>
<thead>
<tr>
<th>INPUT EVENTS</th>
<th></th>
<th>OUTPUT EVENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ</td>
<td>Event</td>
<td>Event to require to change the state of a FB</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CNF</td>
<td>Event to confirm that the FB state was updated</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INPUT VARIABLES</th>
<th></th>
<th>OUTPUT VARIABLES</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE</td>
<td>STRING</td>
<td>Name of the new FB ECC state (atom)</td>
<td></td>
</tr>
<tr>
<td>FB_NAME</td>
<td>STRING</td>
<td>Name of the FB instance (atom)</td>
<td></td>
</tr>
<tr>
<td>APP_NAME</td>
<td>STRING</td>
<td>Name of the Application containing the FB (atom)</td>
<td></td>
</tr>
<tr>
<td>STATUS</td>
<td>STRING</td>
<td>Service Status: RDY, NO_SUCH_OBJECT</td>
<td></td>
</tr>
</tbody>
</table>

**READ THE ECC STATE OF A FB INSTANCE**

<table>
<thead>
<tr>
<th>INPUT EVENTS</th>
<th></th>
<th>OUTPUT EVENTS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ</td>
<td>Event</td>
<td>Event to require to read the state of a FB</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CNF</td>
<td>Event to confirm that the FB state was read</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INPUT VARIABLES</th>
<th></th>
<th>OUTPUT VARIABLES</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>FB_NAME</td>
<td>STRING</td>
<td>Name of the FB instance (atom)</td>
<td></td>
</tr>
<tr>
<td>APP_NAME</td>
<td>STRING</td>
<td>Name of the Application containing the FB (atom)</td>
<td></td>
</tr>
<tr>
<td>STATUS</td>
<td>STRING</td>
<td>Service Status: RDY, NO_SUCH_OBJECT</td>
<td></td>
</tr>
</tbody>
</table>
ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES

UNIVERSIDAD POLITÉCNICA DE MADRID