Universidad del País Vasco
Euskal Herriko Unibertsitatea

ZIENTZIA ETA TEKNOLOGIA FAKULTATEA
FACULTAD DE CIENCIA Y TECNOLOGÍA

Gradu Amaierako Lana / Trabajo Fin de Grado
Ingeniaritza Elektronikoko Gradua / Grado en Ingeniería Electrónica

Sistema de pruebas automáticas de equipos de protección y control de sistemas eléctricos.

Egilea/Autor:
Javier Vallejo García
Zuzendariak/Directores:
Josu Jugo García y Eduardo Aguirre Ayesta

Leioa, 2020ko ekainaren 17a /Leioa, 17 de Junio de 2020

Índice

Resumen

Los equipos de protección y control de protecciones de sistemas eléctricos (también llamados *IEDs* multifunción, de las siglas *intelligent electronic device*) son muy importantes en los sistemas eléctricos ya que además de protegerlos permiten saber su estado y actuar sobre ellos en tiempo real. Este trabajo de fin de grado (TFG) se centrará en la creación de un sistema automático de pruebas mediante *LabVIEW* que, a través de un banco de pruebas, valide características de estos equipos.

Comenzará diseñando el banco de pruebas. Posteriormente, se analizará el código de las pruebas para los equipos programadas en *LabVIEW* y se refactorizará para hacerlo más comprensibles.

Una vez refactorizadas las pruebas, se le añadirán mejoras y nuevas funcionalidades para optimizar y actualizar su funcionamiento. Después de esto se añadirán elementos de control que permitan que las pruebas se puedan ejecutar en paralelo sin interferir unas con otras. Después, se programará el código necesario para ejecutar y controlar el transcurso de las pruebas. Posteriormente, se creará un interfaz que permita visualizar simultáneamente el estado de todas las pruebas de forma sencilla.

Finalmente, se estudiarán las posibilidades de test que el sistema automático programado es capaz de hacer a los equipos y las pruebas desarrolladas a partir de él.

1. Introducción.

1.1. Protecciones eléctricas para un sistema electrónico de potencia.

El sistema electrónico de potencia es un conjunto de elementos que producen, distribuyen y utilizan energía eléctrica.

La parte física de este sistema se puede dividir en 4 subsistemas considerando su función [1]:

- Generación: Incluye todos los componentes que producen la energía eléctrica (centrales nucleares, eólicas...)
- Transmisión: Red eléctrica que transporta la electricidad desde donde se genera hasta las zonas de consumo.
- Distribución: Red encargada de llevar la electricidad desde el subsistema de transmisión hasta donde se va a consumir.
- Suministro: Todos los hilos de baja tensión (220V en Europa) que hay en cada domicilio y local comercial.

Aunque todos estos subsistemas estén bien diseñados, siempre puede producir un fallo en su funcionamiento un agente externo o interno. Para evitar que estos fallos puedan dañar permanentemente todo el sistema o provocar perjuicios en el suministro de electricidad, estos cuentan con elementos de protección.

Los componentes básicos para un sistema de protección de un sistema de potencia son[2]:

- Fusibles: Elementos que están conduciendo durante todo el tiempo la electricidad y se destruyen solos ante una situación de fallo, abriendo el circuito, para evitar daños mayores. A diferencia de los siguientes elementos, funcionan de forma autónoma y no necesitan ningún elemento más.
- Transformadores de corriente y tensión: Se encargan de bajar los niveles de intensidad y voltaje a unos parámetros con los que pueda trabajar el relé sin que se estropee por sobrecarga.
- Relés de protección: Reciben la señal de los transformadores y dan la orden de abrir a los interruptores si hay alguna falta.
- Interruptores: Son los elementos que abren el circuito si reciben la orden del relé.
- Sistema de alimentación: Sistema independiente del que se está protegiendo que proporciona electricidad para que funcionen los relés y los interruptores

1.1.1. Control y supervisión de sistemas eléctricos.

Para controlar y supervisar el estado del sistema eléctrico y de las protecciones, hacen falta otros elementos como:

- Medios que permitan la comunicación entre los diferentes elementos del sistema, dando acceso a los diferentes niveles de las subestaciones (centro de control, nivel de subestación y nivel de aparamenta). Esto se hace mediante comunicaciones de radio ethernet o serie.
- *HMIs* (*Human-Machine Interface*) para adquisición, monitorización y control del sistema.
- Elementos de sincronización que permitan tener una referencia de tiempo precisa en todo el sistema. Por ejemplo, Sistemas de posicionamiento global (GPS)

1.2. Descripción de los equipos a probar.

Los equipos a probar en el transcurso de este TFG son equipos de protección y control de sistemas eléctricos (también llamados *IEDs* multifunción) de la unidad de negocio *PGA* (siglas del inglés *Power Grid Automation*). Están basados en la norma *IEC 61850* (*Communication networks and systems for power utility automation*) y diseñados para realizar funciones de protección de sistemas eléctricos y para el control y supervisión de los componentes del mismo.

Estos equipos tienen integrados tanto trafos de medida de tensión y corriente como tarjetas de entradas y salidas digitales para poder supervisar y controlar el estado de la aparamenta eléctrica (interruptores, seccionadores y demás automatismos).

Los equipos de protección y control a nivel de conectividad, para comunicación con otros dispositivos, disponen de puertos serie, ethernet y *USB* (*Universal Serial Bus*). La configuración de los equipos se realiza a través de ellos. Para la comunicación por puerto Ethernet tiene protocolos basados en la norma *IEC* (*61850*, *60870-5-101* o *60870-5-104*) y otros como el Protocolo de transferencia de archivos (*FTP* por las siglas de *File Transfer Protocol*) o *SSH* (*Secure SHell*).

Dentro de la gama de equipos multifunción desarrollados por *PGA*, las pruebas se realizan sobre dos plataformas (figura 1): Ingepac EF (1a) e Ingepac DA (1b). Dentro de cada familia y dependiendo de las funcionalidades existen diferentes modelos.

(a) Equipos EF. (b) Equipo DA.

Figura 1: *Distintos equipos de protección eléctrica.*

La diferencia principal entre estos dos modelos es que el equipo Ingepac EF está diseñado específicamente como equipo de proteccion y control para soluciones en redes de transmisión de energía eléctrica. La plataforma Ingepac DA se utiliza más en redes de media tensión o para backup de algunos equipos en redes de transmisión. Ambas plataformas comparten muchas de sus funcionalidades siendo la segunda más económica y compacta.

1.2.1. Ficheros de configuración.

Dada la complejidad y versatilidad de estos equipos, son necesarios los ficheros de configuración para poder ajustar su funcionamiento y sus características. Estos ficheros están basados en la norma *IEC 61850*.

La norma dice que el formato de estos ficheros de configuración está definido mediante el lenguaje *SCL* (*Substation Configuration Language*), basado en el lenguaje *XML* (*eXtensible Markup Language*) [3].

Esta norma define varios tipos de ficheros, aunque este TFG se centrará unicamente en uno de ellos: los *ICD* (*IED Capability Description*), que describen las capacidades básicas de un equipo (modelo de datos y comunicaciones) y tienen que acabar con la extensión *.ICD*[3].

Esta norma tiene dos ediciones. La edición 1 consta de 10 partes publicadas entre el 2003 y 2012. Desde 2011, muchas de estas partes se han incluido en una edición 2 en la que se arreglan problemas técnicos y se añaden nodos lógicos y funcionalidades [4].

Los equipos ingepac EF tienen ficheros de configuración basados tanto en la edición 1 como en la edición 2 de la norma. Mientras, los archivos de configuración del equipo ingepac DA se basan únicamente en la edición 2.

1.3. Comunicación con los equipos: ethernet.

Aunque, como se ha comentado anteriormente, los equipos tienen comunicaciones basadas en la norma *IEC 61850*, para este TFG se usarán los protocolos *FTP* y *SSH*, comunicando a través de uno de los puertos Ethernet de los que dispone el equipo.

1.3.1. Protocolo de transferencia de archivos (*FTP*).

El protocolo *FTP* permite la transferencia de archivos completos entre ordenadores [5].

Se basa en la conexión entre un usuario y un servidor. El usuario se comunica a través de distintos comandos con el servidor para realizar las acciones (transferencia de datos, iniciar o finalizar conexión,...), mientras que el servidor le envía respuestas para informar cómo se han realizado las acciones [6]. Hay que tener en cuenta que esta comunicación se realiza por el puerto 21, mientras que los datos se transfieren por el puerto 20.

Un ejemplo de este protocolo se observa en la trasferencia de datos en un sistema para el control del riego con un *dataloger* basado en un *Arduino* [7].

Se usa para acceder a la mayoría de archivos de los equipos, utilizando las herramientas que nos dan los propios lenguajes de programación.

1.3.2. Protocolo *SSH*.

El protocolo *SSH* se basa en una arquitectura cliente/servidor. En ella, el cliente envía solicitudes al servidor para realizar distintas acciones (iniciar sesión, descargar un fichero, ejecutar comandos,...)[8]. Este protocolo sólo utiliza un puerto (normalmente el 22).

Este protocolo es muy seguro, ya que toda la información que se va a enviar por *SSH*, se encripta y una vez haya llegado a su destino se desencripta [8].

Un ejemplo del potencial de este protocolo de comunicación es su uso para el control de un robot de salvamento [9].

Se utiliza para acceder a archivos a los que no se puede acceder mediante *FTP* y ejecutar comandos en los equipos. Se accederá a este protocolo a través del ejecutable *plink.exe*, que es una ventana de comandos de la aplicación *PuTTY*.

1.4. Bancos de pruebas.

En el mundo del desarrollo es necesario poder probar los elementos de un sistema final para ver si funcionan correctamente, antes de lanzarlos al mercado, o para comprobar

su funcionamiento para desarrollarlos mejor. Esto se hace a través de unas plataformas llamadas bancos de pruebas o *testbeds* en inglés.

Los bancos de pruebas, sobre otros métodos de pruebas, tienen la ventaja de que sólo se estudia la parte del sistema que interesa, siendo el resto estímulos simulados. Están formados por tres componentes [10]:

- Subsistema experimental: Los componentes usados en el sistema final que se desea probar.
- Sistema de monitorización: El sistema que se encarga de recoger, gestionar y enseñar la información de las pruebas.
- Subsistema de simulación y estimulación: Encargado de que las pruebas sean lo más parecidas a la vida real, ya que genera entradas y salidas realistas.

Con estos tres componentes se prueban y simulan muchos aspectos que luego el subsistema tendrá que afrontar en la vida real.

1.5. Objetivos.

El objetivo principal de este TFG es el montaje de un banco de pruebas y la programación de una aplicación capaz de realizar de forma autónoma varias pruebas a través de este banco de pruebas, para validar las distintas funcionalidades de los equipos de protección y control de sistemas eléctricos.

Estas pruebas deben automatizar procesos largos y lentos para reducir la dedicación de personal y poder hacerse en varios equipos a la vez, para acortar los tiempos para hacerlas. Estos procesos tienes que ser capaces de certificar si:

- Los archivos de configuración son correctos.
- Los equipos pueden validar los archivos de configuración.
- Se pueden hacer conexiones *FTP* con los equipos de forma correcta.

Con este fin se piensa en utilizar los lenguajes de programación *Java, Python* o *LabVIEW*. Se puede encontrar ejemplos de automatización mediante estos lenguajes como:

- Un sistema de domótica basado en *Java*[11].
- Un sistema de domótica con monitorización para ancianos, usando *Python* (entre otras herramientas) [12].
- Un sistema de automatización y control industrial a traves de una PLC (*Programmable Logic Controlle*) y *LabVIEW*[13].

Finalmente se eligió *LabVIEW* por su gran modularidad (se puede aprovechar fácilmente el código anterior), su versatilidad y para poder usar el código de pruebas que se había programado anteriormente en este lenguaje.

Para programar el sistema de pruebas se necesita modificar el código que había anteriormente, ya que tiene muchas deficiencias que serán comentadas más adelante. El primer paso es refactorizar el código, de manera que se mantenga su funcionamiento externo pero sin que se modifique el código interno [14],facilitando los cambios futuros. Para esto, se hacen *SubVIs* de las partes más importantes del código, de manera que para cambiar el funcionamiento de estas partes baste con modificar estos *SubVIs*, y se vuelven a programar siguiendo una arquitectura de *LabVIEW*, llamada Máquina de estados (o *State Machine* en inglés), haciendo uso de estos *SubVIs*. Una vez hecho esto, es mucho más fácil arreglar las deficiencias del código y añadir nuevas funcionalidades.

Posteriormente, se crea una máquina de estados capaz de lanzar las pruebas en paralelo (más de una prueba a la vez en distintos equipos) y también secuencialmente (lanzar varias pruebas en paralelo después de que se hayan terminado de ejecutar otro grupo de pruebas en paralelo y así poder hacer varias pruebas en un mismo equipo). Para que las pruebas puedan lanzarse en paralelo se añaden elementos de control que eviten que varías pruebas intenten acceder a un mismo recurso a la vez. De esta manera las trasferencias de datos se hacen de forma correcta , es decir se hace una programación concurrente. Al conseguir que múltiples procesos se realicen de forma simultanea, se reduce el tiempo que se tarda en ejecutarse, tal y como se puede ver en la alta velocidad de un controlador de un conversor de corriente alterna a continua, que se basa en aprovechar las operaciones concurrentes de una *FPGA* (*Field-programmable gate array*) [15]

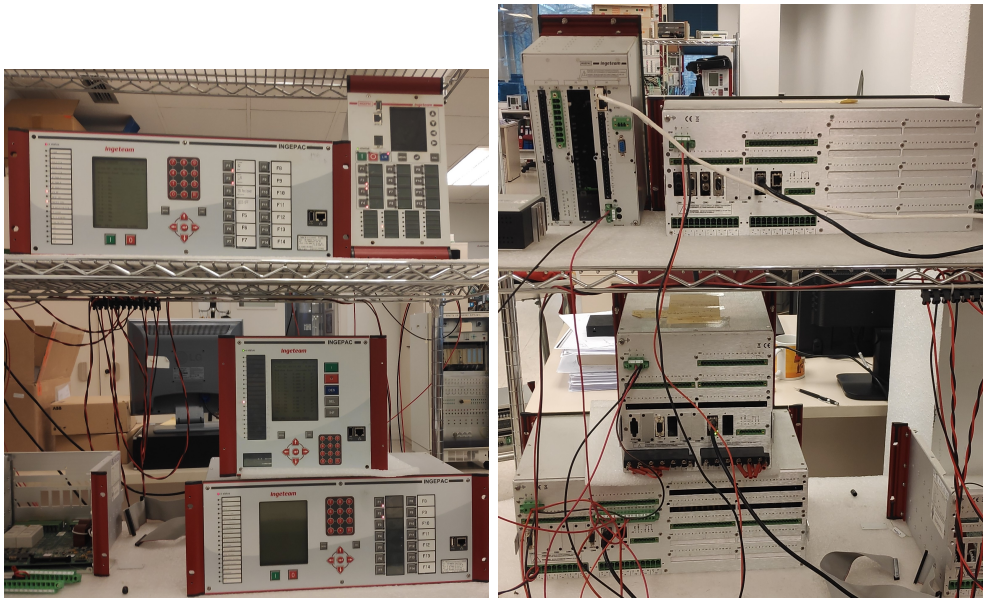
Finalmente, se programa una interfaz amigable que permita ver el transcurso de las pruebas que se estén ejecutando, añadiendo elementos que posibiliten a las pruebas comunicarse con esta interfaz.

2. Montaje del banco de pruebas.

El banco de pruebas debe cumplir los siguientes requisitos:

- Disponer de varios equipos a los que poder hacer pruebas.
- Poder conectar vía ethernet los equipos a un ordenador capaz de lanzar y controlar las pruebas.
- Poder hacer pruebas a equipos EF y DA.
- Poder reiniciar los equipos de forma independiente.

Para ello, se dispone de un total de cuatro equipos: 3 equipos EF (2 para realizar pruebas y otro auxiliar, que se usa para reiniciar los equipos) y un equipo DA. Su disposición se puede ver en la figura 2. Estos equipos están conectados mediante un cable ethernet a un *switch*, que esta a su vez conectado por otro cable ethernet a un ordenador. Así el ordenador se comunica vía ethernet con los equipos.



(a) Parte delantera. (b) Parte trasera.

Figura 2: *Banco de pruebas usado.*

En la figura 2a se pueden ver los equipos empleados: el equipo DA se encuentra arriba a la derecha, el equipo EF auxiliar es el de abajo del todo y los otros dos serían los equipos EF que se van a someter a las pruebas. Además, en la figura 2b se puede ver que el cable de alimentación de cada equipo pasa por una entrada y una salida (normalmente conectadas) del equipo auxiliar. De esta manera, para reiniciar los equipos, basta con enviar un fichero *Comandos.xml* al equipo auxiliar, que hace que se desconecte la entrada con la salida durante 5 segundos; tiempo suficiente para cortar la alimentación al equipo en pruebas y hacer que se apague y se vuelva a encender.

Este sistema se elige frente a otros, como una tarjeta de entradas y salidas digitales (como la de *National Instruments*) o a usar una fuente de alimentación (como un *Omicron*) para alimentar los equipos. Esto se debe a que, al realizarse este TFG en la unidad de negocios *PGA*, es sencillo acceder a un equipo auxiliar y si, alguna salida o entrada se estropea, es fácil remplazarla. Por otro lado, como el sistema de pruebas resultante va a ser modular, basta con cambiar algunos módulos para cambiar el método de reseteo.

Con todo lo explicado, en este banco de pruebas , el susbsitema experimental es los tres equipos a los que se van a hacer las pruebas, el sistema de monitorización es el ordenador y el subsistema de simulación y estimulación es también el ordenador junto con el equipo auxiliar.

Como herramienta para poder hacer pruebas a los equipos, se tienen varías pruebas programadas en *LabVIEW*.

2.1. Análisis de las pruebas ya programadas.

Básicamente, de las pruebas programadas, se usará en el banco de pruebas estas tres pruebas: validación de *ICDs*, envío de *ICDs* y validación de *FTP*. También hay que tener en cuenta que hay dos versiones de cada una de estas pruebas, una para equipos EF y otra para equipos DA, con lo que el sistema de pruebas estará compuesto por seis pruebas independientes en total.

2.1.1. Validación de *ICDs.*

El objetivo de esta prueba es verificar que los archivos de configuración son correctos. Para ello se comprueba si un equipo es capaz de aceptarlos y configurarse de forma correcta con ellos (es decir si es capaz de validar estos archivos). El funcionamiento de esta prueba sigue los pasos siguientes:

- Se conecta mediante *FTP* al equipo y envía el archivo de configuración a una carpeta del equipos llamada *NotValidated*.
- Se espera un tiempo y se comprueba si el archivo se encuentra en una carpeta llamada *Validated*.
    - Si no se encuentra ahí, significa que no se ha validado. Si ocurre esto se descarga el archivo *validation_traces.log* (que es donde aparece información sobre los errores en el proceso de validación), lo renombra con el nombre del *ICD* y pasa al siguiente.
    - Si se encuentra, significa que se ha validado. En este caso, comprueba unos archivos para ver si se ha validado correctamente y pasa al siguiente *ICD*. Un archivo importante es *Sucesos.xml*, que proporciona la fecha de recepción del archivo de configuración y en la que se ha validado. De esta manera el programa revisa este archivo y calcula el tiempo que se ha tardado en validar el *ICD*.

Como en esta prueba se validan *ICDs* de distintos modelos, el programa tiene que ser capaz de poder cambiar el modelo del equipo, ya que sólo acepta archivos de configuración con el mismo modelo que él. Este proceso es muy diferente para los equipos EF y DA.

En el caso de los equipos EF, el proceso que se sigue es el siguiente:

- Se envía el archivo de configuración *TestPl_Labview.ICD* a la carpeta *NotValidated*. Este es muy básico, con lo que se tarda muy poco en validar y parsear al reiniciar el equipo y es válido para cualquier modelo.
- Se cambian los tres primeros dígitos de código comercial del equipo (un código que se encuentra en un archivo de texto accesible por *FTP*) para que estos coincidan con el principio del código del modelo al que deseamos cambiar el equipo.
- A través del bloque *System Exec.vi* de *LabVIEW* se accede al ejecutable *Plink.exe* y se introduce un código (figura 3), siendo este código distinto en cada familia de modelos de *ICDs*.
- Se reinicia el equipo a través del equipo auxiliar.
- Antes de validar los *ICDs* correpondientes al modelo, se valida el archivo *TestPl.ICD*. No se mira el rastro que deja en la mayoría de los archivos, simplemente se comprueba el tiempo que ha tardado en validarse. Este es un archivo de configuración que se usa para pruebas en el departamento de producción a los equipos (por esta razón es necesario comprobar que se puede validar en todos los modelos) y válido para todos los equipos aunque más pesado y, por tanto, más lento de validar que el *TestPl_Labview.ICD*.

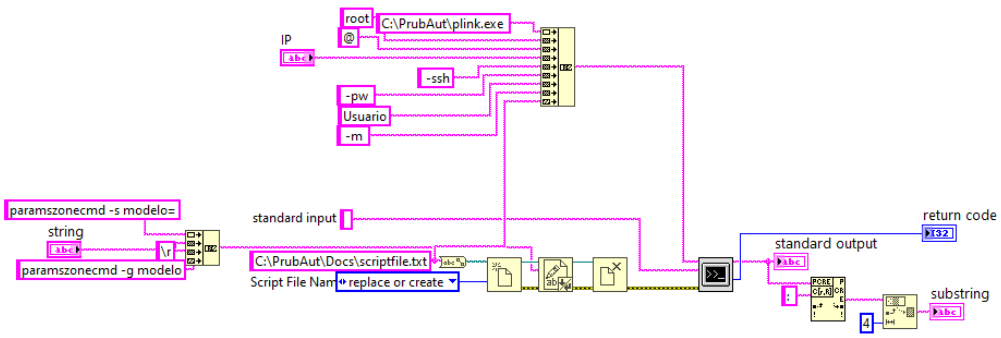


Figura 3: *Código para el cambio del código de familia.*

En el caso de los equipos DA, no se necesita ni enviar el archivo *TestPl_Labview.ICD* ni validar ningún archivo *TestPl.ICD* ni introducir ningún código de familia de modelos. Por estas razones, el cambio de modelo de estos equipos es mucho más sencillo y rápido. Los pasos que se siguen son los siguientes:

- Se cambian los primeros dígitos del código comercial para que coincida con el código del archivo de configuración.

- Se borra un fichero llamado *Settings.xml* para evitar problemas de compatibilidad entre modelos en el arranque.
- Se reinicia el equipo.
- Cada vez que se envía un *ICD*, se comprueba que unos caracteres en medio del código comercial del equipo (correspondientes a las tarjetas de entrada) coinciden con los del *ICD*. En caso de no coincidir se cambian (si no coinciden estos caracteres el equipo no sería capaz de validar el archivo de configuración). Se borra el fichero *Settings.xml* y se reinicia el equipo.

Un ejemplo de archivo de configuración de la prueba para EF se puede ver en la figura 4. El significado de las lineas de arriba a bajo son:

- La *IP* del equipo al que se desea hacer la prueba.
- La *IP* del equipo auxiliar usado para resetear el equipo.
- El número de segundos que se esperan entre envíos de *ICDs* (para esperar a que el equipo se termine de configurar).
- Las siguientes lineas son los *ICDs* a validar.

Hay que tener en cuenta que para que se valide el *TestPl.ICD*, cada vez que se añaden *ICDs* de un nuevo modelo, se tiene que poner la línea *Testpl +modelo*. Para el caso de los equipos DA, el archivo de configuración será igual que el de la figura 4, pero sin tener que poner la línea de llamada al archivo *TestPl*.

```
192.168.182.100
192.168.182.180
240
Testpl MD0A
MD0A3
MD0A3e
MD0A7
Testpl MD1A
MD1A3
```

Figura 4: *Archivo de configuración para validación de ICDs.*

Cuando se ejecuta la prueba se crea una carpeta para contener sus resultados. En ella se crean, según el transcurso de la misma, varios archivos con información del trascurso de la prueba y los ficheros *validation_traces.log*. Entre toda la información, la más importante es el fichero *.txt* (cuyo contenido se puede ver en la figura 5). Este resultado corresponde a una prueba en un equipo EF. En la imagen se puede ver cómo aparece información de los archivos de configuración validados y la comprobación de los distintos archivos del equipo. En el caso de la DA, el archivo de salida será el mismo pero sin aparecer los *TestPls*.

Analizando con detenimiento el código, se encuentra que tiene los siguientes aspectos a mejorar:

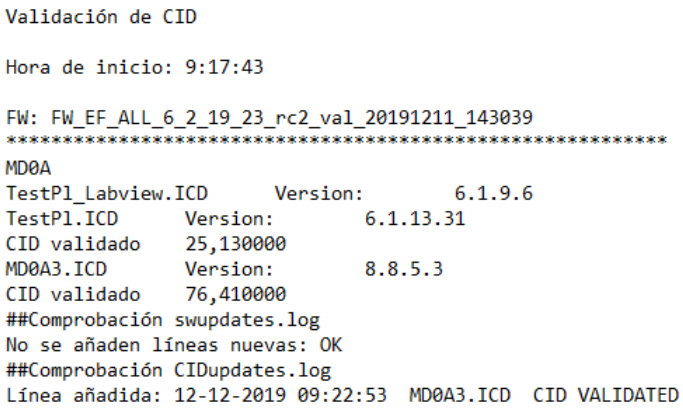


Figura 5: *Fichero de resultados de la prueba validación de ICDs.*

- La mayoría de los archivos de tipo *XML* se tratan como ficheros de texto. Al ser estos muy grandes, trae un alto tiempo computacional.
- Sólo se pueden mandar *ICDs* con un nombre determinado, esto conlleva tener que renombrar los archivos de configuración para pasarles las pruebas.
- Hay algunos modelos que se han creado nuevos y no se encuentran en la prueba, y no se pueden validar sus *ICDs*.
- Ligado al anterior, indicar que la información de los *ICDs* a validar está dentro del código en vez de obtenerla de los propios *ICDs*. Dificultando introducir nuevos modelos y archivos de configuración para validarlos.
- No se puede elegir qué archivo reseteador usa la prueba. Por ello estas pruebas sólo se pueden pasar en un equipo conectado al equipo auxiliar de una determinada forma.
- Los archivos *TestPl* tienen que ser de la misma edición que los archivos de configuración a validar en los equipos EF para que la prueba se pueda hacer de forma correcta. El problema es que esta prueba no diferencia entre ediciones y siempre coge los *TestPls* de una dirección determinada, por lo que, para que no haya cambio de edición, tienen que cambiarse a mano y es fácil tener una equivocación que cause un problema con el cambio de edición.

2.1.2. Envío de *ICDs.*

Esta prueba funciona de forma parecida a la de validación de *ICDs*. La diferencia está en que este programa, en vez de enviar una vez cada *ICD* y comprobar si se ha validado, manda varias veces cada uno y comprueba en cada envío el tiempo que se tarda en validar esos archivos de configuración. Por otro lado, si lo desea el usuario, se puede grabar una imagen del equipo por cada envío. Esto se hace en un archivo de texto donde vienen datos de este como la memoria disponible. Así se puede ver si el envío de varios archivos de configuración influye en el estado del equipo. Por último, hay que tener en cuenta que en esta prueba no se valida el *TestPl.ICD*, aunque siga haciendo falta el *TestPl_Labview.ICD* para hacer el cambio de modelo en los equipos EF.

Un ejemplo de fichero de configuración de esta prueba se ve en la figura 6. Este formato de fichero de entrada es el mismo para los equipos EF y para las DA. Cada línea de este fichero representa una serie de envíos de ficheros de configuración. De izquierda a derecha, los parámetros significan lo siguiente:

- La *IP* del equipo al que se va a hacer la prueba.
- La *IP* del equipo auxiliar usado para resetear.
- El nombre del archivo de configuración.
- Las veces que quieres que se envíe ese *ICD*.
- El tiempo (en minutos) máximo que puede estar el programa buscando el tiempo de validación.
- El tiempo (en segundos) de espera entre envíos de archivos de configuración.
- Si este último parámetro es un 1 se hace la captura del equipo en cada envío y si es un 0 no se hace.

```
192.168.182.100  192.168.182.180 MD0B7  5   6  30   1
192.168.182.100  192.168.182.180 MD0B7e 5   6  30   1
```

Figura 6: *Archivo de configuración para envío de ICDs.*

El programa genera una carpeta con ficheros con información al igual que la prueba de validación de *ICDs*. De todos los archivos de resultados, el más importante es el de la figura 7. En esta se ve cómo sólo aparecen los valores máximos, mínimos y la media de los tiempos de envío.

```
************************************************************
Hora de inicio: 12:51:57 RESULTADO DE LA PRUEBA -  CID CON NODO PTUV
FW_EF_ALL_5_17_15_4_val_20151130_143941
MD3B3.ICD 6.1.13.30 MAX:  68.540       MIN:  67.920    MEAN:  68.230
************************************************************
```

Figura 7: *Archivo de resultados de envío de ICDs.*

Aparte de los fallos comentados de la prueba anterior (que también los tiene esta prueba), esta tiene una complejidad en el archivo de configuración y falta de información del archivo de salida (estos podrían ser más parecidos a los que tiene la prueba de validación de *ICDs*). También se puede observar que, aunque su funcionamiento es muy parecido a la prueba de validación de *ICDs*, está construida de forma muy distinta. Por esta razón, se tratará de que estas pruebas tengan una construcción y unos ficheros de configuración y salida lo más parecidos posible.

2.1.3. Validación de *FTP*.

Es la prueba más simple. Su único parámetro de entrada es un archivo *.txt* con la dirección *IP* del equipo del que se quiere comprobar su conexión y no necesita resetear

ese equipo y, por tanto, no precisa de un equipo auxiliar para ejecutarse. Analiza si se puede conectar al equipo mediante *FTP* con los cuatro usuarios que tiene este para acceder. Después de esto, comprueba que no se establece conexión con esos usuarios si las contraseñas no son las correctas. Finalmente, hace un listado de los archivos a los que accede cada usuario y compara esta lista con una plantilla para ver si se puede acceder a todos los archivos que se debería.

Cabe destacar que, al ser estas pruebas tan simples, no se requiere realizar ninguna modificación, siendo las únicas a las que no se le efectuará una refactorización.

2.2. Refactorización del código.

Teniendo en cuenta el objetivo final, el primer paso será modificar las pruebas mencionadas anteriormente para ser adaptadas fácilmente a las necesidades del programa de pruebas automáticas.

Para conseguir esto, se hará una refactorización al código para hacerlo mucho más claro y facilitar la introducción de nuevos cambios.

Al revisar el código de las pruebas programadas, se encuentran los siguientes inconvenientes:

- Su programación caótica.
- Su poca modularidad.
- No hay un tratamiento de errores.
- No cumple con la buena práctica de programación en *LabVIEW* de hacer que todo el código entre en una pantalla.

Todos estos problemas dificultan introducir cambios en el código. Un ejemplo de estos fallos se puede ver en la captura de pantalla de la figura 8.

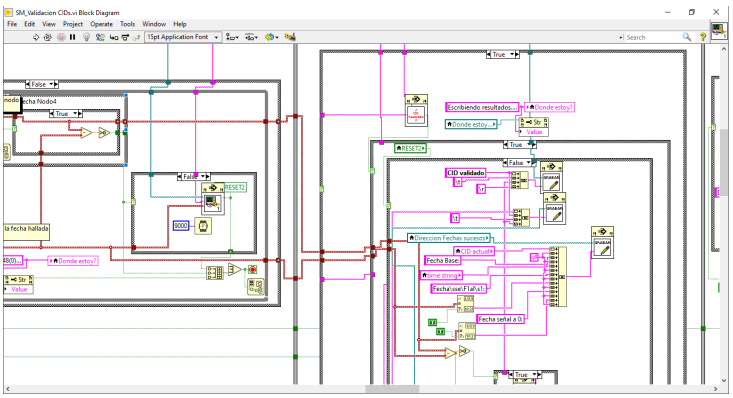


Figura 8: *Código anterior.*

Para conseguir un código mucho más manejable se han implementando *SubVIs* con las partes aprovechables, ordenándolas y añadiendo el control de errores dando como resultado un código más ordenado y la posibilidad de hacer mejoras modificando únicamente estos *SubVIs*, en vez de modificar el código de la pruebas en sí. Es decir, se ha hecho el código mucho más modular.

Una vez conseguido esto, se continua con la refactorización del código. Para ello, se usa la arquitectura de *LabVIEW* de máquina de estados.

2.2.1. Arquitectura máquina de estados en *LabVIEW*.

La arquitectura de máquina de estados (*state machine* en inglés) es muy usada en este lenguaje de programación al permitir modificar, debuggear y documentar el código fácilmente. Esta se utiliza habitualmente en sistemas de medida, test y control [16], siendo una buena opción para el objetivo del sistema de pruebas.

Un ejemplo de un estado de una máquina de estados se ve en la figura 9, que corresponde a una de las pruebas modificadas.

Una máquina de estados básica consta de los siguientes elementos [16]:

- Una constante tipo *Enum* donde se guardarán todos los estados.
- Un ciclo *While* con un registro de desplazamientos. Este registro será el encargado de pasar el estado del enumerador deseado a la siguiente iteración del ciclo.
- Una estructura de casos dentro del ciclo. Cada estado de esta corresponderá a cada caso del enumerador y en él se encuentra la información del estado al que ir después.

En el caso del sistema de pruebas automáticas se incluyen:

- Dos registros de desplazamientos conectados a *clusters* (color rosa). Se usan para pasar información entre estados.
- Una línea de error (de color amarillo y verde). Se usa para tener un control de errores en las pruebas.

Finalmente, hay que tener en cuenta que siempre hay que incluir un estado para que se encargue de finalizar el ciclo *while*, una vez se haya ejecutado todo el código.

Después de elegir la arquitectura con la que se desea refactorizar el código, se hacen cambios en las partes importantes, para ahorrar tiempo de computación de las pruebas y hacer más fáciles futuras mejoras en el comportamiento del código.

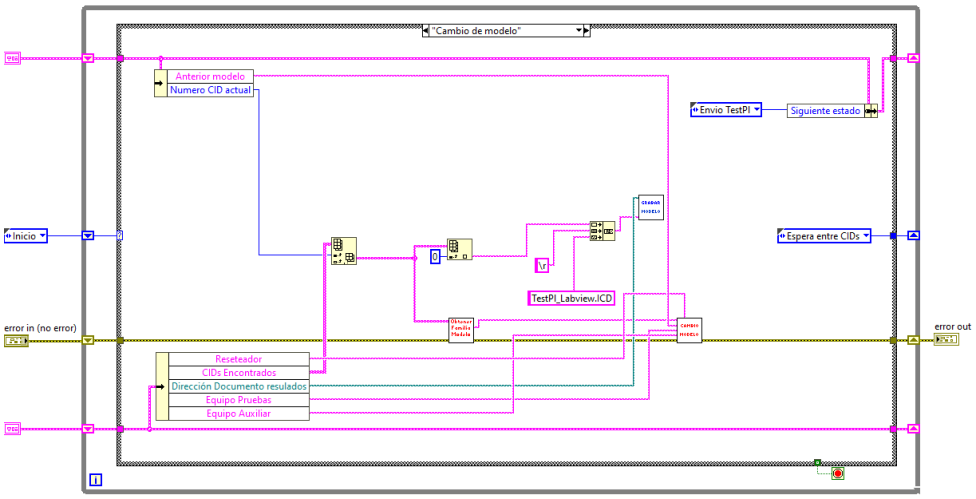


Figura 9: *Ejemplo de un estado en una máquina de estados.*

2.2.2. Cambios más importantes introducidos en la refactorización del código

Uno de los cambios de la refactorización más importantes es el cambio de la parte del código que se encarga de leer los archivos *.xml*, para usar las herramientas de parseo de estos archivos, en vez de tratarlos como archivos de texto. Un ejemplo de esto es la figura 10, este es un *SubVI* usado para la búsqueda de la fecha en que el valor de una señal ha cambiado de valor a 1 ó a 0 por última vez (depende del parámetro de entrada que se le pase). En concreto este *SubVI* se utiliza para el cálculo de tiempos de validación, usando el archivo *sucesos.xml*, y lo que hace es:

- Cargar el archivo.
- Hace una lista de todos los nodos en los que aparece el cambio de la señal buscada al valor buscado.
- Elige el último elemento (que corresponde a la última vez que ha cambiado el valor de la señal) y obtiene la fecha de este cambio.

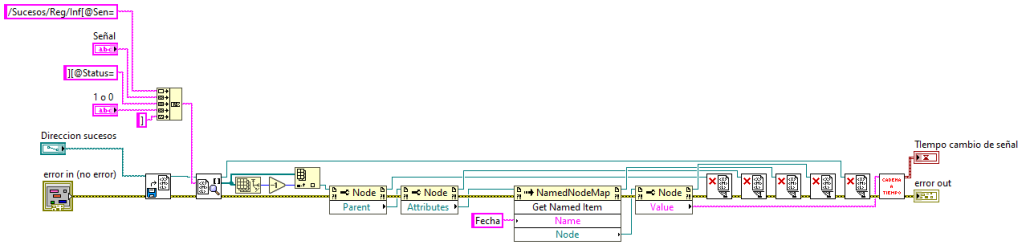


Figura 10: *SubVI para la búsqueda de la fecha en que una señal ha cambiado a valor 1 o 0 por última vez.*

Otro cambio importante añadido al código es que ahora los datos para cambiar el código comercial de los equipos para hacer el cambio de modelo, en vez de estar guardados en el código, se adquieren directamente del *ICD*. Esto se realiza aprovechando que estos

archivos tienen formato *XML*, mediante un método parecido al de la figura 10. Esto permite incluir fácilmente nuevos modelos en las pruebas, por ser el único parámetro necesario el código de la familia del modelo que está localizado en un *SubVI*, el cual simplemente consta de una estructura de casos que devuelve el código correspondiente a cada modelo.

Por último, otro cambio destacable es que el código de la prueba envío de *ICDs* se hace de manera muy semejante al código de la prueba de validación de *ICDs* y se han usado el máximo posible de *SubVIs* comunes a ambas pruebas. Así es más fácil introducir mejoras en ambas pruebas a la vez y si alguna persona tiene que trabajar con este código en un futuro va a comprender más fácilmente el funcionamiento de ambos programas.

Una vez se ha refactorizado el código va a ser más sencillo añadirle funciones y mejoras.

3. Creación del sistema de pruebas automáticas.

Para crear el sistema de pruebas automáticas, primero se deben introducir funciones y mejoras a las pruebas que se han refactorizado, para optimizar su funcionamiento y prepararlas para paralelizarlas.

3.1. Funciones añadidas y mejoras introducidas a los programas refactorizados

El primer paso es modificar los ficheros de entrada de las pruebas. Se empezará por el archivo de configuración de la prueba de validación de *ICDs*. Como siempre hay que validar los ficheros *TestPl.ICD* en los equipos EF antes de hacer un cambio de modelo. Se ha cambiado el código para que no haya que poner las líneas *testpl + modelo*, pues el programa detecta cuando hay un cambio de modelo y lo valida automáticamente. Con esto, el archivo de configuración tendrá la misma estructura para los equipos DA y EF.

Tomando el archivo de entrada de las pruebas de validación de *ICDs* como base, se ha modificado el archivo de entrada de la prueba de envío de *ICDs*, para que sea como el de la figura 11. En él, las primeras tres líneas de este archivo cumplen la misma función que para la prueba de validación de *ICDs*, las siguientes son:

- El número de veces que se envía cada *ICD*.
- El tiempo máximo (en minutos) que espera a que busque la señal.
- El carácter que nos dice si se realiza la captura o no.
- La siguientes serán los *ICDs* que se desean enviar.

```
192.168.182.100
192.168.182.180
90
10
2
1
BF0A
BF0Ae
```

Figura 11: *Nuevo archivo de configuración de envío de ICDs.*

Además, se observó que los equipos DA pueden admitir conexión con*FTP* antes de terminar de reiniciarse. Si ocurre esto en el código anterior, la prueba continúa sin haber terminado de reiniciarse el equipo, ocasionando problemas en los resultados. Para evitarlo, se añade una línea en el archivo de configuración de la figura 11 que indica el número de segundos que hay que esperar para intentar conectar por *FTP* después del reinicio del equipo. Así se evita que la prueba continúe sin haberse reiniciado el equipo y el tiempo se puede ajustar a la velocidad en la que se reinician los distintos equipos.

Por otro lado, para que no haga falta renombrar los *ICDs* para usarlos en las pruebas, se cambia la manera que tiene el programa de gestionar los archivos de entrada. El proceso que se ha implementado es el siguiente:

- Del archivo de configuración de la prueba se coge el nombre del *ICD*.
- Busca si se encuentra en la carpeta donde se guardan los *ICDs*.
    - Si lo encuentra, obtiene los datos necesarios de él, los guarda en una matriz y continúa con el siguiente archivo de configuración del equipo.
    - Si no lo encuentra, guarda el nombre del *ICD* en una línea de un archivo de salida donde aparecerán todos los *ICDs* no encontrados.

De esta manera, se sabe de un vistazo los archivos de configuración que no haya encontrado la prueba y esta los acepta con cualquier nombre.

Teniendo en cuenta que se puede obtener la versión del *ICD* directamente del contenido de este y que los que se pasan en una prueba siempre son de la misma edición, se ha añadido una función a las pruebas de equipos EF para que, mirando la edición que se ha detectado en la modificación anterior, usen los *TestPls* correspondientes a esas ediciones, ya que se han separado en dos carpetas (una para cada edición). Por otro lado, en el caso de que el fichero anteriormente validado sea de una versión distinta a la que tienen los *ICDs* con los que se realiza la prueba, se ha añadido un tiempo de espera de 2 minutos después de enviar el primer *TestPl*. Esto asegura que el equipo tenga tiempo a realizar un reseteo por cambio de modelo.

Por otro lado, como se quiere que la prueba de envío de *ICDs* tenga un archivo de salida *.txt* (donde se muestren los resultados de la prueba) parecido al archivo de la figura 5 (archivo de salida de la prueba de validación de *ICDs*), se cambia para que este sea como el de la figura 12. En ella se ve cómo se ha añadido la posibilidad de que la prueba verifique los mismos archivos del equipo que se comprueban en la prueba de validación de *ICDs*, al ser el tiempo de computación de hacer esto despreciable y da información del estado del equipo.

Anteriormente se ha comentado que, cuando una validación fallaba, se descargaba el archivo *validation_traces.log* en la prueba de validación de *ICDs*, pero estos archivos son muy grandes y es complicado buscar la parte que interesa (las líneas que se añaden entre qué se recibe el *ICD* y se valida). Por otra parte, este archivo no se consigue si el archivo de configuración se valida correctamente en la prueba de validación de *ICDs* y no se obtiene nunca en la prueba de envío de *ICDs*. Por ello se han modificado las pruebas para que en su salida aparezcan las líneas añadidas en este archivo que nos interesan en el envío de cada *ICD*, pero obviando las añadidas cuando se envían los *TestPls* (estos, al carecer de muchas características, producen errores que se reflejan en añadir muchas lineas al archivo que no interesan).

Con todo esto se obtiene un fichero como el de la figura 13, donde se encuentra el obtenido para validación de *ICDs* en una EF, aunque para DA y para la prueba de envío de *ICDs* este sería igual.

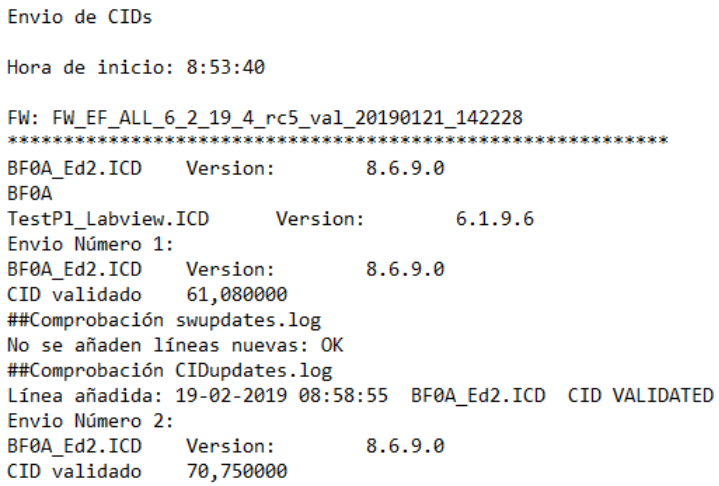


Figura 12: *Nuevo archivo .txt de salida de envío de ICDs.*

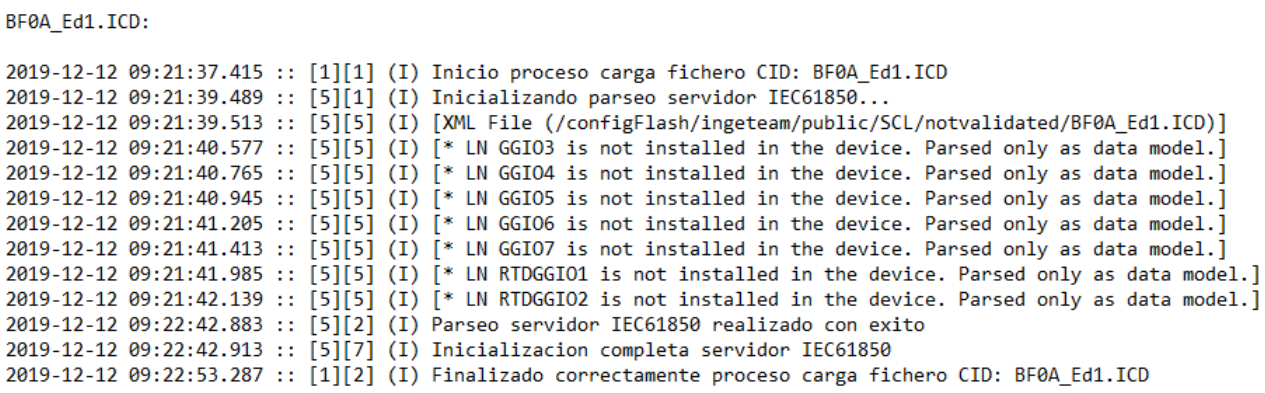


Figura 13: *Fichero .txt de salida con información de validation_traces.log .*

Para conseguir que las pruebas se lancen en paralelo, se ha añadido a estas la capacidad de poder elegir el *comandos.xml* que se envía al equipo auxiliar para resetear los equipos a los que se están realizando las pruebas. De manera que se puede conectar varios equipos al auxiliar y resetearlos en las pruebas de forma independiente. Para esto se ha añadido un controlador de tipo cadena de caracteres en el panel frontal de las pruebas, que nos permite introducir el nombre de la carpeta donde se encuentra el reseteador que deseamos usar (el programa la busca en un directorio en el que están guardados los reseteadores en distintas carpetas). Esto se controlará más tarde mediante el sistema de pruebas automáticas.

Finalmente, como se quería que las pruebas de envío y validación de *ICDs* tuvieran como salida archivos que puedan ser tratados directamente en *Excel*. Se opta porque la salida de resultados sea en archivo de formato *.csv*. Este es un archivo que se trata como un documento de texto normal, introduciendo los datos separados por el caracter *;*, que al abrirlo con un editor de *Excel*, se ve separado en columnas. Esto se puede comprobar en las figuras 14 y 15, que corresponden a ficheros de este formato de salida de la prueba de envío de *ICDs* abiertos con un editor de texto y con un editor de *Excel*, respectivamente. Este archivo será igual para las pruebas de validación de *ICDs*; salvo en el caso de los equipos EF, en los que se incluyen los tiempos de validación y la edición del fichero *TestPl.ICD*.

```
TIME;N_fichero;Edición;cod;Version;CID validado;tpo;swupdates.log;CIDupdates.log;
8:53:40;BF0A_Ed2.ICD;Ed2;BF0;8.6.9.0;OK;61,08;OK;OK;
9:01:27; ; ; ; ;OK;70,75;OK;OK;
9:06:30; ; ; ; ;OK;68,95;OK;OK;
9:11:32; ; ; ; ;OK;67,58;OK;OK;
9:16:37; ; ; ; ;OK;60,6;OK;OK;
```

Figura 14: *Fichero de resultados .csv abierto con un editor de texto.*

	A	B	C	D	E	F	G	H	I
1	TIME	N_fichero	Edición	cod	Version	CID validado	tpo	swupdates.log	CIDupdates.log
2	8:53:40	BF0A_Ed2.IC	Ed2	BF0	8.6.9.0	OK	61,08	OK	OK
3	9:01:27					OK	70,75	OK	OK
4	9:06:30					OK	68,95	OK	OK
5	9:11:32					OK	67,58	OK	OK
6	9:16:37					OK	60,6	OK	OK
7									

Figura 15: *Fichero de resultados .csv abierto con un editor de Excel.*

Una vez modificadas las pruebas, hay que conseguir que estas puedan ser lanzadas de forma paralela y el primer paso será paralelizar las pruebas programadas.

3.2. Paralelización de las pruebas.

Como se quiere que estas pruebas se ejecuten de forma concurrente y al poder estas acceder a elementos de memoria comunes, pueden aparecer problemas de sincronización que se han de solucionar. Para ello, se usa:

- Semáforos.
- Variables funcionales globales.
- Ciclos de espera para asegurar que las transferencias por *FTP* se hagan de forma correcta.

3.2.1. Semáforos.

Los semáforos son una herramienta incluida en la librería de *LabVIEW* para controlar el acceso a un recurso compartido entre varios programas [16], no dejando que dos programas accedan al recurso a la vez. Su funcionamiento se puede ver fácilmente en la figura 16.

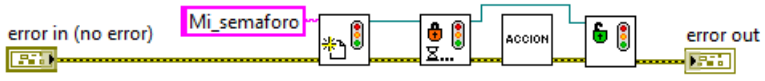


Figura 16: *Ejemplo uso semáforos.*

En ella, para sincronizar una acción (representado en la imagen por el *SubVI* con el icono de acción) que haga uso de cualquier recurso crítico, basta con usar tres bloques. La

función de cada uno, según el orden de ejecución (de izquierda a derecha en la imagen), es el siguiente.:

- *Obtain Semaphore Reference.vi*: En este caso, a través de una cadena de texto con el nombre del semáforo, si este ya existe, obtiene una referencia a él, y, si no existe, crea la referencia.
- *Acquire Semaphore.vi*: Este coge la referencia al semáforo del anterior bloque. Con ella, no deja continuar la ejecución del código (adquirir el semáforo) hasta que este se libere (si otra aplicación lo ha adquirido y no lo ha liberado todavía). Para asegurar que esto no permite a la acción crítica ejecutarse hasta que el semáforo sea liberado, se puede pasar la señal de error de salida de este bloque como señal de error de entrada de la acción (en el ejemplo la acción está representada por el *SubVI* con la palabra *ACCIÓN*).
- *Release Semaphore.vi*: Se ejecuta una vez se ha terminado la acción (conectada a ella por la línea de error). Libera el semáforo para que otra aplicación pueda adquirirlo.

Asimismo hay que tener en cuenta que, una vez se haya asegurado que ningún programa va a querer hacer uso del semáforo, hay que liberar la referencia a éste (figura 17), poniendo el bloque de *Obtain Semaphore Reference.vi* y que este pase la referencia del semáforo al bloque *Release Semaphore Reference.vi*, que es el que se encarga de liberar la referencia a un semáforo, de forma que deja libre la memoria asociada a él.

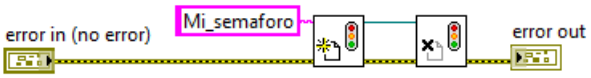


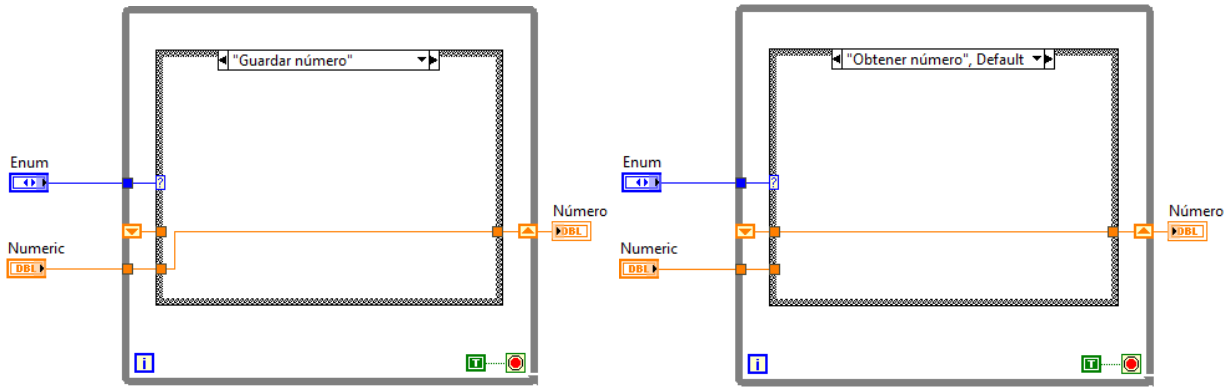
Figura 17: *Ejemplo liberar referencia semáforos.*

Los semáforos se usan para controlar los procesos de obtención del código comercial o *firmware* de los equipos y cualquier acción que necesite acceder a la aplicación *PLINK* a través de la función *System Exec.vi* de *LabVIEW*. Para el correcto funcionamiento del sistema de pruebas automáticas, se liberan las referencias a los semáforos empleados, una vez ejecutadas todas las pruebas,

3.2.2. Variables funcionales globales.

La característica fundamental de las variables funcionales globales es que consisten en un *subVI* de ejecución no re-entrante (*Non-reentrant execution* en inglés). Es decir sólo puede ser llamado por un *VI* cada vez y siempre usa el mismo espacio de memoria para guardar los parámetros con los que son llamados. Esto hace que se pueda usar este *subVI* para pasar información entre las llamadas al mismo [17]. Así se asegura pasar información entre los distintos programas sin pérdidas y que no accedan a ella dos programas a la vez.

Concretando, el *SubVI* consiste en un ciclo *while* con una estructura de casos dentro. El ciclo está configurado para solo dar una iteración cada vez que es llamado (tiene un



(a) Guardar número (b) Obtener número

Figura 18: *Ejemplo variable global funcional.*

booleano de valor *True* conectado a la condición de parada del ciclo). El ciclo posee registros de desplazamiento para guardar información entre llamadas al mismo [19]. El caso en el que se encuentre esta estructura, al ser llamada, depende de un selector que se le pasa como parámetro de entrada a la variable. La información que se desea enviar a la variable se pasa en los parámetros de entrada y se accede a la información que tenga guardada la variable a través de parámetros de salida. Un ejemplo de esto es la figura 18, donde se ve una variable global funcional que puede guardar un número (18a) y después obtenerlo (18b).

Para controlar la palarelización de las pruebas, la variable funcional global sirve para guardar un *array* con los archivos abiertos y da la información de si el archivo al que se intenta acceder está abierto o no, evitando que dos programas intenten acceder al mismo a la vez. Para ello se crea una variable funcional con tres estados distintos.

El primer estado (figura 19) sirve para inicializar la variable global funcional, creando un *array* de tipo *file path* donde se van guardando las direcciones de los archivos abiertos. Hay que considerar que, a la hora de llamar a la variable global con este estado, no hace falta pasarle ningún parámetro. Por otro lado en todos los estados se ha añadido una línea de error para controlar mejor cuándo se llama.

El siguiente estado (figura 20) es al que el programa llama cada vez que intenta acceder a un archivo; el parámetro de entrada de la llamada es de tipo *file path* y representa la dirección del archivo al que se quiere acceder. Una vez llamado, el ciclo *for* con condición recorre el *array* y compara cada elemento con el parámetro de entrada, parándose si encuentra una dirección igual a la del parámetro de entrada. Si ve que ningún elemento coincide con él (ya que el ciclo *for* recorre todo el *array* y ve que ningún elemento coincide) significa que el archivo no se encuentra abierto y lo añade al *array* devolviendo *false* (caso 20a). Si lo encuentra, significa que el archivo se encuentra abierto y devolverá *true* (caso 20b).

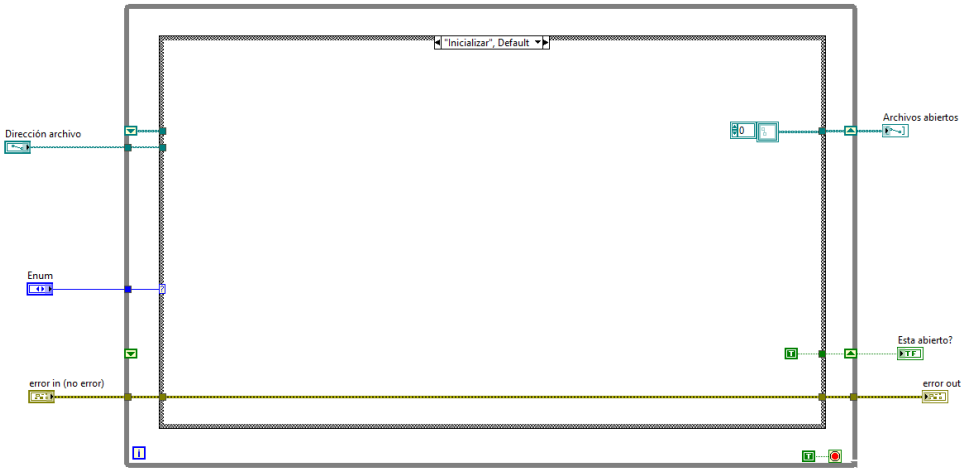
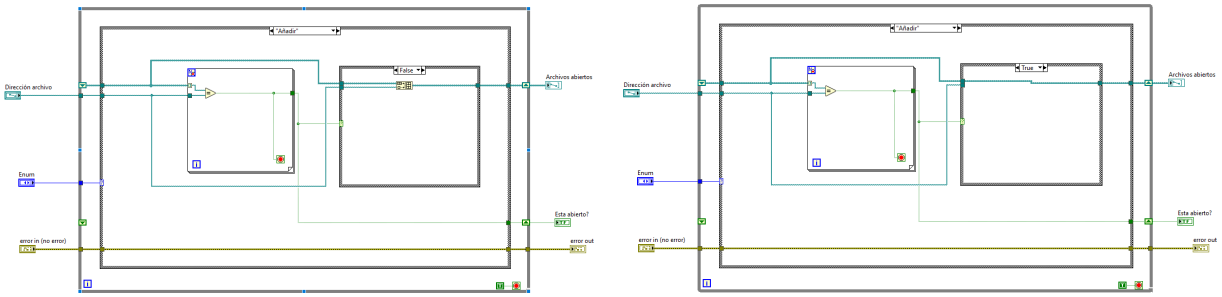


Figura 19: *Variable funcional en el estado "Inicializar".*



(a) Archivo no abierto. (b) archivo abierto

Figura 20: *Variable funcional en el estado "Añadir".*

Finalmente, el último estado al que se accede una vez el programa ha terminado de usar el archivo se puede ver en la figura 21. Este usa un ciclo *for* que busca en qué posición se encuentra el *file path* que se le pasa de parámetro de entrada a la función *Delete From Array*, que lo elimina y así otro programa puede acceder a este archivo.

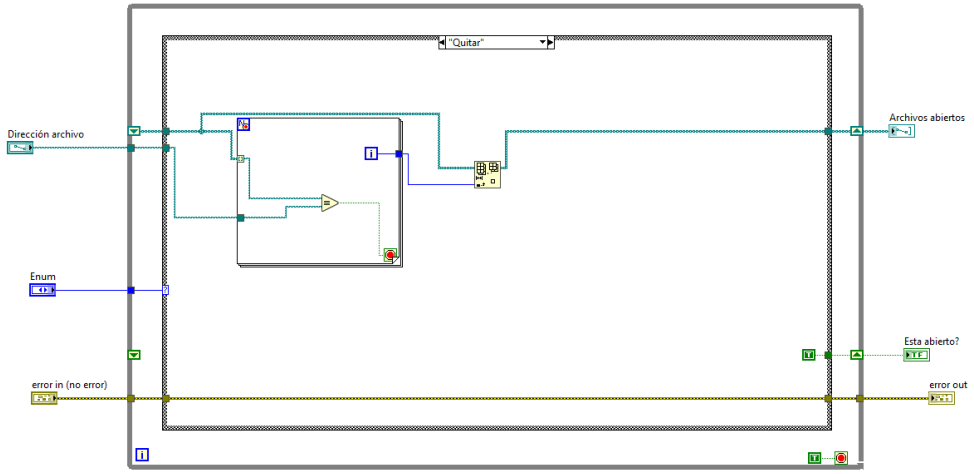


Figura 21: *Variable funcional en el estado "Quitar".*

Para emplear la variable funcional global, creada para controlar los archivos abiertos, primero hay que inicializarla (figura 22). Una vez hecho esto, para controlar los archivos abiertos se usa el código de la figura 23, donde se está utilizando un *SubVI* que escribe un texto en un archivo. Su funcionamiento es simple, dentro un ciclo *while*, se realizan las siguientes acciones:

- Se llama a la variable funcional global.
  - Si el archivo está abierto (23a).
    - Se espera un tiempo (en este caso *250 ms*).
    - Se vuelve a comprobar si el archivo sigue abierto.
  - Si el archivo no se encuentra abierto (23b):
    - Se realiza la acción pretendida (en este caso escribir).
    - Se quita el archivo de la variable funcional global para que otro programa pueda acceder a éste.
    - Se para el ciclo.

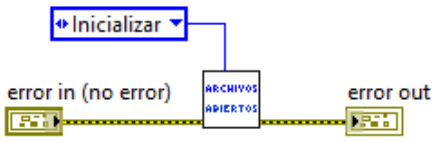
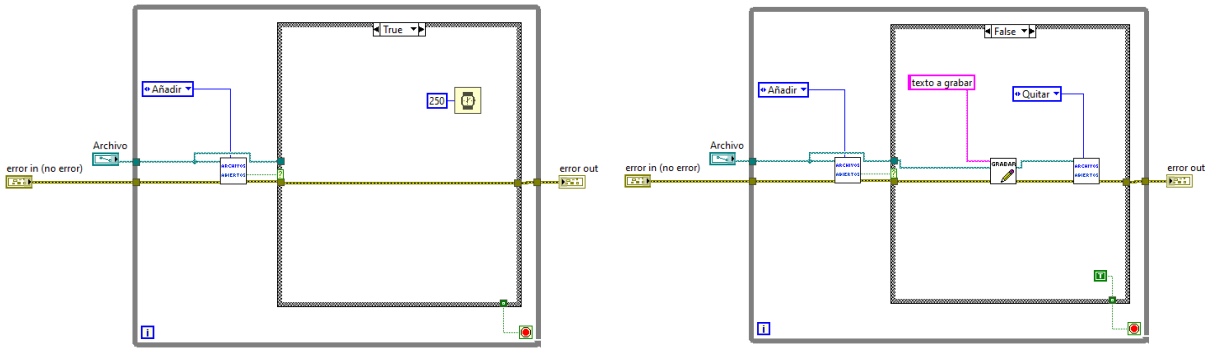


Figura 22: *Inicializar variable funcional.*

En el caso del sistema automático de pruebas, hay que añadir este código cada vez que un programa intente acceder a un archivo compartido con otros programas.



(a) Archivo no abierto.

(b) archivo abierto

Figura 23: *Uso de la variable funcional al intentar acceder a un archivo.*

3.2.3. Ciclo de espera para que la trasferencia por *FTP* se realice de forma correcta.

Las pruebas, como se ha indicado anteriormente, generalmente se basan en la transferencia de archivos *FTP* para la comunicación de los programas con los equipos y viceversa.

Al poder lanzarse varias a la vez, puede haber problemas en la conexión con este protocolo. Para asegurar que estas transferencias se hagan de forma correcta, se usará el código de la figura 24 cada vez que en un programa desee realizar cualquier operación por *FTP*. Consiste en un ciclo *while* que, si la operación que se realiza a través de *FTP* (en este caso enviar un archivo) da un error, vuelve a repetirla hasta que esta no dé ningún error, lo que quiere decir que se ha realizado de forma correcta. Hay que tener en cuenta que, para evitar que el ciclo se vuelva infinito, se limpia cualquier error en la línea de error antes de que entre en los bloques de *FTP*. También se puede ver que se ha añadido dentro del *while* un pequeño temporizador de *250 ms* para evitar que realice las iteraciones muy seguidas y consuma demasiada memoria RAM del ordenador.

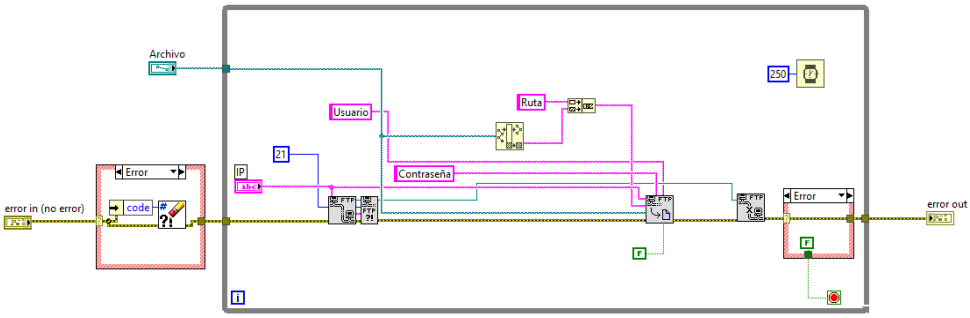


Figura 24: *Ciclo de espera para la transferencia por FTP.*

3.3. Programación del *VI* para lanzar las pruebas.

Una vez asegurado que las pruebas pueden ser ejecutadas a la vez sin que entren en conflicto entre ellas, se debe crear el código necesario para gestionar las llamadas a esas pruebas. El primer paso para este objetivo será idear una forma de gestionar las pruebas que se quieran hacer y, para ello, la mejor alternativa es un fichero de configuración.

3.3.1. Fichero de configuración.

El fichero de configuración debe reunir los siguiente requisitos:

- Dejar determinar la prueba que se desea realizar.
- Posibilitar escoger el fichero de configuración de la prueba.
- Dar la capacidad de elegir en qué carpeta se encuentra el reseteador que se desea usar en la prueba.
- Ser lo más intuitivo posible.

Considerando todo esto se obtiene el fichero de configuración ideado reflejado en la figura 25, y así el programa es capaz de lanzar el primer grupo de pruebas en paralelo, que son

las pruebas que se encuentran entre los primeros *start* y *end* (cada línea representa una prueba distinta) en el orden en el que están escritas, esperar a que estas terminen, realizar las pruebas que se encuentren en la segunda secuencia y así con todas las secuencias que se deseen lanzar.

```
start
ValCID_DA;val_DA.txt;Comandos DA
end
start
EnvioCID_DA;Env1.txt;Comandos DA
end
start
ValCID_DA;val_DA.txt;Comandos DA
EnvioCID;Env2.txt;Comandos_EF_2
ValCID;val1.txt;Comandos_EF
end
start
EnvioCID_DA;Env1.txt;Comandos DA
EnvioCID;Env2.txt;Comandos_EF_2
ValCID;val1.txt;Comandos_EF
end
```

Figura 25: *Fichero de configuración del lanzamiento de pruebas.*

La estructura de los parámetro de la línea que selecciona la prueba es *"Nombre prueba;Fichero.txt;Reseteador"*. Estos parámetros tienen los siguientes significados:

- *Nombre prueba*: Nombre de la prueba que se quiere lanzar. Los posibles nombres son:
  - *ValCID* y *ValCID_DA* para seleccionar la prueba de validación de *ICDs* en equipos DA y EF respectivamente.
  - *EnvioCID* y *EnvioCID_DA* para seleccionar la prueba de envío de *ICDs*.
  - *ValFTP* y *ValFTP_DA* para seleccionar la prueba de validación de *FTP*.
- *Fichero.txt*: Nombre del archivo de configuración de cada prueba. La aplicación accede a una carpeta distinta por cada una de las pruebas para buscar este archivo.
- *Reseteador*: Nombre de la carpeta donde se encuentra el archivo reseteador correspondiente al equipo al que se desea realizar la prueba. Este parámetro no es necesario para las pruebas de validación de *FTP*.

Una vez definido el archivo de configuración, hay que diseñar el código que interprete el archivo de configuración, ejecute las pruebas conforme al orden propuesto e inicialice los elementos de control de la paralelización de las pruebas. La solución elegida es un *VI* basado en la arquitectura de máquina de estados.

3.3.2. Máquina de estados programada.

La máquina de estados programada para interpretar el archivo explicado anteriormente y encargarse de lanzar las pruebas de manera correcta es en esta primera aproximación muy simple. La interfaz de este programa se puede ver en la figura 26. En ella sólo hay un botón para empezar a lanzar pruebas y un indicador tipo *array* de cadenas de texto que nos muestra las lineas que se encuentran en el archivo de texto de configuración usado.

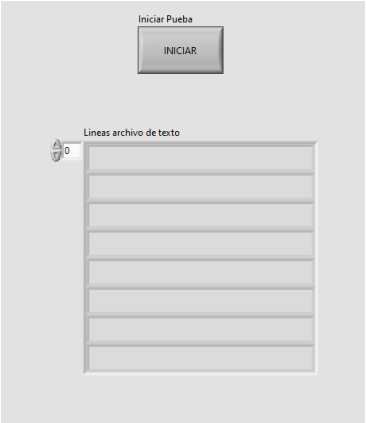


Figura 26: *Interfaz máquina de estados.*

Para que se pueda lanzar varias veces la misma prueba, hay que configurar los *VIs* de las pruebas como no reentrantes, de manera que al ser llamado se le asigna una nueva dirección de memoria y, cada vez que se llama, se ejecuta la prueba sin interferir con las otras llamadas [20]. Por otro lado, como esta máquina de estados no muestra el estado de cada prueba, se configurarán las pruebas para que, cuando se lancen, nos muestre su panel frontal (esto, al igual que lo anterior, se hace cambiando las propiedades del *VI*) y, añadiendo indicadores para la información importante, se puede ver el transcurso de las pruebas.

La máquina de estados que compone esta prueba tiene varios estados. los más importantes, que se explicarán a continuación con más detenimiento son:

- *Inicio.*
- Los encargados de lanzar las pruebas.
- *Espera a finalización de pruebas.*

Estado *Inicio.*

Cuando se empieza a ejecutar la prueba, el primer estado que entra es el de inicio.

Su principal función es permitir al usuario elegir el archivo de texto donde se encuentra la secuencia a través del bloque *File Dialog Express VI*, que hace que salte una ventana emergente que te permite elegir el archivo. Una vez se ha elegido, se hace una array con sus líneas, se representa en el panel frontal y se guarda para las siguientes iteraciones. Este estado se puede ver en la figura 27.

El siguiente estado es el que espera a que se pulse el botón de iniciar para continuar (da la posibilidad de abortar la aplicación si no has seleccionado el archivo correcto). Al pulsarlo, comprueba que la primera línea es un *start* (parando la prueba si no lo es) y continúa con los siguientes estados.

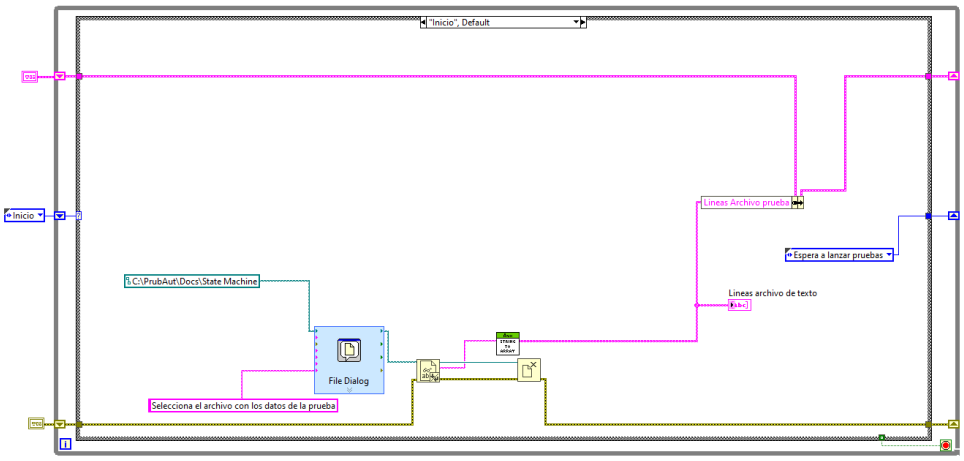


Figura 27: *Estado inicial de la máquina de estados.*

Después, la aplicación inicializa la variable funcional global y los semáforos. Luego se pasa a un estado que lee línea por línea del archivo y manda a la máquina a un estado u otro dependiendo de la prueba que se quiera ejecutar en esa línea. Los estados a los que se envía desde aquí son los que lanzan las pruebas.

Estados para lanzar las pruebas.

Hay un estado para cada prueba que se puede lanzar con la máquina de estados. Un ejemplo de un estado usado para lanzar pruebas es la figura 28 (es el estado encargado de lanzar la prueba de validación de *ICDs*, aunque los estados para lanzar el resto de pruebas serán iguales). El funcionamiento de estos estados es el siguiente:

- Se obtiene la dirección del *VI* programado para dicha prueba a través de la función *Static VI Reference*. Esta es una referencia a un *VI*.
- La referencia se pasa a un través de la función *Static VI Reference*, que devuelve la dirección de *VI*.
- La función *Open VI Reference* coge como parámetros de entrada la referencia y la dirección del *VI*. Como tiene como parámetro de entrada el numero 100 en hexadecimal, prepara la prueba para poder ser ejecutada y para que, al terminar su ejecución, se puedan recoger sus parámetros de salida y nos devuelve una referencia para lanzar el *VI*.
- La referencia que se obtiene se pasa a un *Start Asynchronous Call Node* que lanza la prueba de forma asíncrona y la aplicación puede continuar su ejecución.
- Se guarda la referencia lanzada en un *array* distinto por prueba.
- Se espera 10 segundos para continuar con el siguiente estado, para que haya un intervalo de tiempo entre lanzamientos de pruebas.

Una vez se ha terminado de ejecutar, se vuelve al estado que analiza las líneas del array. Este continúa llamando a estados para lanzar pruebas hasta que se encuentra con

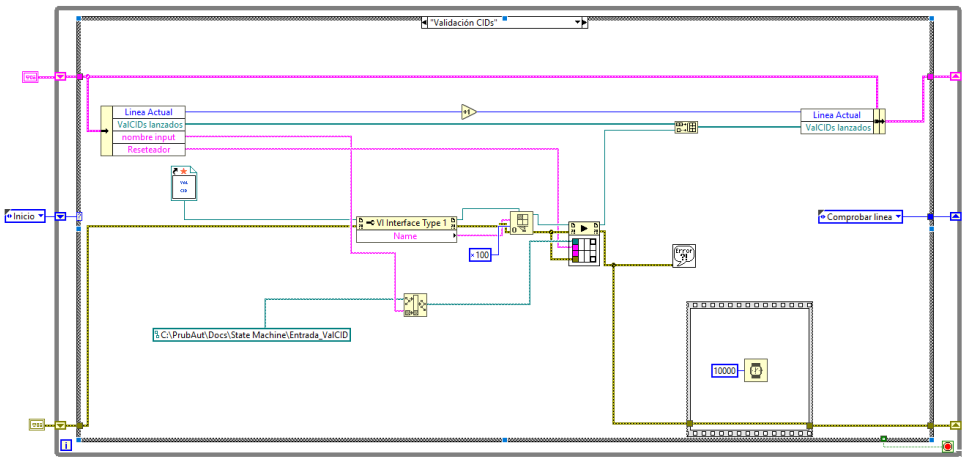


Figura 28: *Estado lanzar Validación de ICDs.*

un *end*, en cuyo caso lleva a la máquina de estados al estado *espera a finalización de pruebas*.

Estado *Espera a finalización de pruebas.*

Su finalidad es esperar a que todas las pruebas lanzadas se hayan terminado de ejecutar (figura 29). Este lleva cada *array* con las referencias de cada prueba como parámetro de entrada a un *SubVI* como el de la figura 30, comprobando si el *array* que les llega no está vacío.

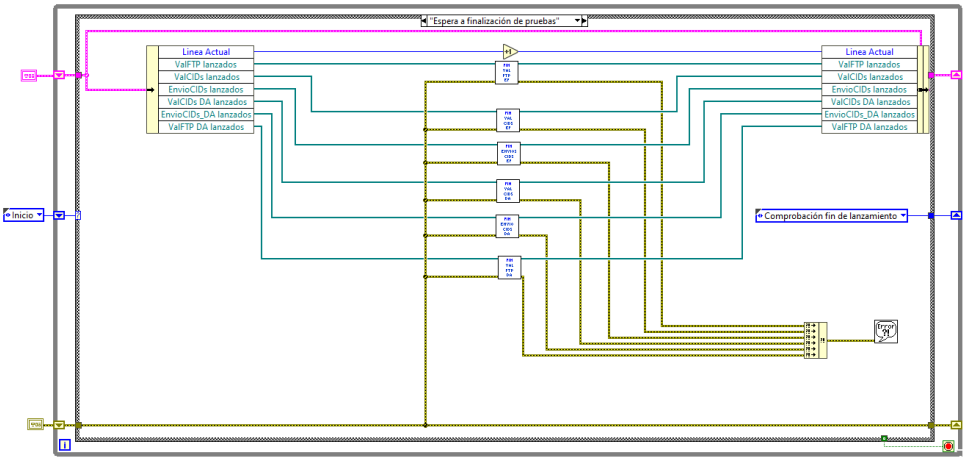


Figura 29: *Estado para esperar a que se termine la ejecución de pruebas.*

Si no está vacío, es que esa prueba se ha lanzado por lo menos una vez. Con esto recorre el *array* con un ciclo *for*. Con cada referencia espera a que se termine de ejecutar usando el bloque *Wait On Asynchronous Call Node* y una vez que se ha terminado la cierra con la función *Close Reference*. Esta libera la zona de memoria que ocupaba la referencia.

Por otro lado, este *SubVI* devuelve siempre un *array* de referencias vacío.

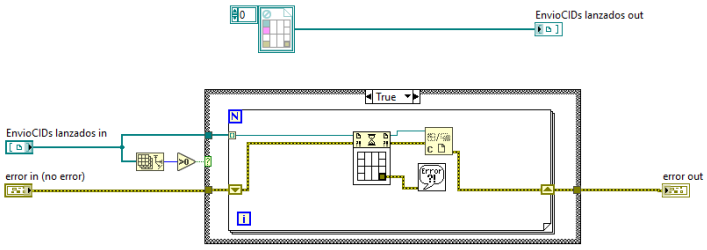


Figura 30: *SubVI para la finalización de una prueba.*

Como a estos *SubVIs* se les llama de forma paralela para cada prueba, las pruebas más cortas se pueden finalizar y liberar la memoria que tenían asociadas mientras que las más largas siguen ejecutándose.

Al finalizar todas las pruebas lanzadas, la máquina de estados pasa a un estado que comprueba la siguiente línea al *end*. Si ésta es un *start*, el programa vuelve al estado en el que se lee línea por línea y se lanzan las pruebas. Si por el contrario no es un *start*, se cierra las referencias a los semáforos usados, se lanza un mensaje por pantalla de que las pruebas han finalizado y se para el ciclo *while* de la máquina de estados.

3.4. Interfaz gráfica.

La máquina de estados que se ha creado para gestionar las ejecuciones de las pruebas es funcional pero su interfaz es muy básica y, para encontrar el estado de cada prueba, hay que ir buscando entre distintas ventanas abiertas. Por esta razón, hay que programar un *VI* que gestione y muestre, por pantalla, información del estado de cada prueba o lo que es lo mismo, sirva de interfaz del sistema automático de pruebas.

Para que esta interfaz reciba la información de las pruebas que se están lanzando, se necesitan elementos y una arquitectura que permita realizar esta operación. Para este objetivo se ha elegido una arquitectura productor/consumidor, (o *producer/consumer* en inglés) controlada por colas (o *Queue* en inglés).

3.4.1. Comunicación entre el interfaz y las pruebas: arquitectura productor/consumidor y colas.

La arquitectura productor/consumidor está basada en la arquitectura maestro/esclavo y es usada para pasar información entre procesos que utilizan esta información a distintas velocidades. Funciona cuando se pueden distinguir entre los procesos que generan datos (productores) y los que los usan (consumidores). La arquitectura productor/consumidor sincroniza la transferencia de información de forma que no se pierdan datos en el proceso [18].

La herramienta que se usará para conseguir implementarla en nuestra interfaz serán las colas que son elementos mayormente usados como FIFO (*First In, First Out*), es decir

el orden de salida de los datos es el orden de entrada de los mismos. Su uso se observa en la imagen 31 obtenida de una plantilla de *LabVIEW* para implementar esta arquitectura y modificada ligeramente para hacerla más fácil de entender a simple vista.

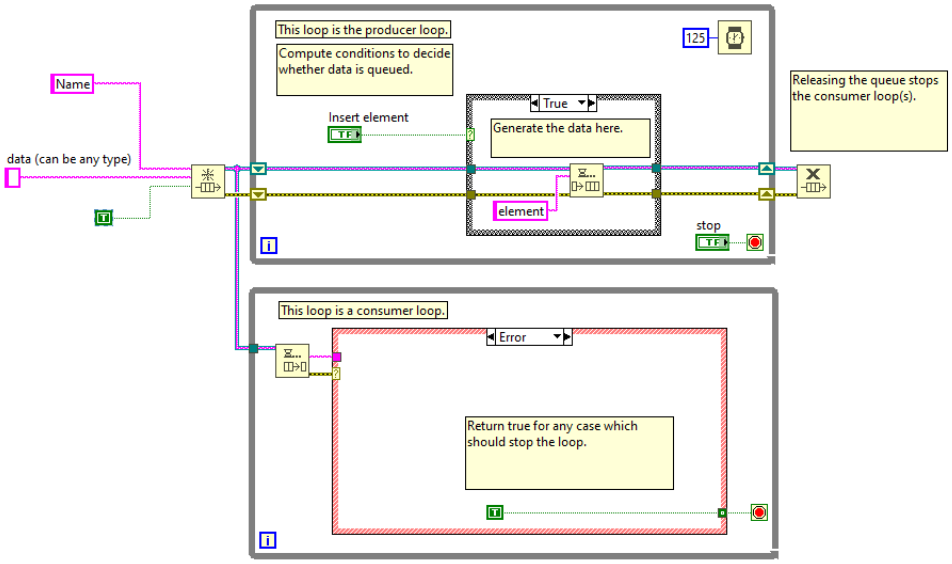


Figura 31: *Plantilla LabVIEW de una arquitectura productor/consumidor basada en colas..*

Antes de explicar la arquitectura, hace falta entender el funcionamiento de las funciones de la imagen para controlar las colas. Estas son las siguientes:

- *Obtain Queue*: Se encuentra arriba a la izquierda. Su función es devolver una referencia a la cola para ser usada después. Aunque se le pueda pasar más parámetros, los que se le han pasado en el ejemplo de la figura 31 son:
  - El tipo de datos que se desea encolar (introducir en la cola). Es el único parámetro obligatorio y en el ejemplo es una cadena de texto (se le ha conectado una constante de cadena de texto).
  - El nombre. Es una cadena de texto (en el ejemplo pone *name)* y sirve para obtener la referencia a la misma cola en varios *VIs* pasándoles el mismo nombre.
  - Un boleano. Si es verdadero, si no existe la cola con ese nombre, se crea, si es falso, si no existe la cola, devuelve un error. Su valor por defecto verdadero.
- *Enqueue Element*: Se encuentra a la derecha del bloque *Obtaien Queue*. Su función es encolar (introducir en la cola) los elementos que se pasen como parámetro de entrada (tienen que ser del mismo tipo que el que admite la cola).
- *Dequeue Queue*: Bloque que se encuentra debajo, su función es eliminar el elemento del principio de la cola y devolverlo como parámetro de salida (desencolar). Si la cola está vacía, se queda esperando a que se encole algún elemento.
- *Release Queue*: Es el bloque que se encuentra a la derecha del todo, su función es la de liberar la referencia a la cola.

Conocidas las funciones necesarias, se explica la arquitectura productor/consumidor usando la figura 31. El ciclo *While* de arriba es el ciclo productor y el de abajo el consumidor.

La función del ciclo consumidor es quedarse esperando a que el ciclo productor encole elementos (le envíe información), recoge la información y la procesa en el mismo orden en que el productor se la ha enviado (no se pierde información). El ciclo consumidor termina cuando lo hace el ciclo productor (en el ejemplo, cuando termina el ciclo productor se elimina la referencia a la cola, produce un error en el bloque encargado de desencolar y la estructura de casos hace que el ciclo termine).

Aunque este sea el funcionamiento básico de la arquitectura productor/consumidor, en la práctica puede haber varios productores o consumidores.

3.4.2. Apariencia del interfaz.

Ahora es el momento de definir el aspecto que tendrá la interfaz (su *Front Panel*). Este será como el que se puede ver en la figura 32 y constará de varios elementos:

- Un indicador de tipo *path*, arriba a la izquierda, que permite elegir el archivo en el que está guardada la secuencia.
- Tres botones a la derecha del indicador: uno para cargar la secuencia, otro que inicia la secuencia y el último que detiene la ejecución de las pruebas.
- Una tabla, en el centro, que da información de las pruebas en la secuencia y su estado de ejecución.
- Un control de pestañas, debajo del todo, que da información del archivo de entrada (en la tabla) y la ubicación de la carpeta de resultados (en el indicador de tipo *path*) de cada prueba lanzada.

Una vez definida la interfaz del sistema de pruebas automáticas, hay que establecer la forma de comportarse y mostrar los datos, es decir, su funcionamiento.

3.4.3. Funcionamiento

Lo primero que hay que tener en cuenta es que la información de las distintas columnas de la tabla central es la siguiente:

- Referencia: Una referencia única para cada prueba, será la cadena de texto con la palabra *Prueba* seguida de un espacio más el número de la prueba, se enviará a la prueba en el orden en que esté escrita en el archivo secuencia.

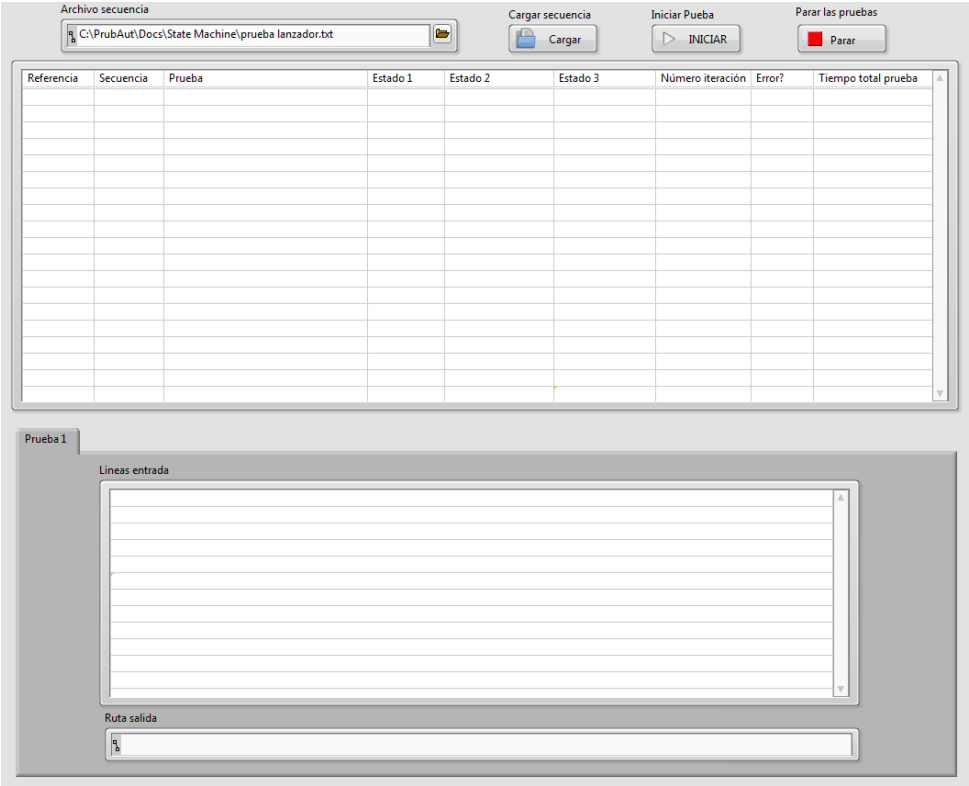


Figura 32: *Aspecto definido para la interfaz.*

- Secuencia: Número de uno en adelante que nos dirá a qué secuencia pertenece la prueba, ya que cada secuencia tendrá asociado un número en función del puesto en que será lanzada.
- Prueba: En esta columna aparece la copia de la línea del archivo de configuración en la que se especifica la prueba.
- Estado 1: Puede tener tres distintos estados en función del estado de ejecución de la prueba: *Esperando*, *En ejecución* y *Finalizado*.
- Estado 2: Dice el *ICD* que tiene cargado el equipo en cada fase de la prueba.
- Estado 3: Indica las acciones que está realizando en ese momento la prueba (espera entre *ICDs*, cambio de modelo,...).
- Numero de iteración: Columna exclusiva para la prueba de envío de *ICDs* y nos dice el número de envío del *ICD* actual.
- Error?: Si la prueba detecta cualquier error indica *Error* en ese momento. Si la prueba no detecta ningún error indica *OK* al finalizar la misma.
- Tiempo total prueba: Al finalizar la prueba muestra el tiempo total que ha durado la misma en formato *Horas:minutos:segundos*.

Según lo indicado, cuando se cargue un archivo por el controlador tipo *Path* (llamado *Archivo secuencia*), deberán escribirse en la tabla las cuatro primeras columnas de información (siendo el Estado 1 *Esperando*). Esto mismo sucede al pulsar el botón de *Cargar*

secuencia. De esta manera se pueden hacer cambios en el fichero de pruebas y ver esos cambios en la interfaz. Al pulsar el botón *Iniciar prueba* se escriben los datos en las tablas, pero además empiezan a ejecutarse las pruebas en el orden elegido. Estos botones no van a afectar al transcurso de la pruebas una vez que empiece su ejecución.

Mientras se van ejecutando las pruebas, en el control de pestañas, aparece una pestaña por prueba. En la tabla de cada pestaña se escribe el contenido del archivo de entrada de la prueba (poniendo una línea del archivo en cada fila de la tabla) y el indicador tipo *path* indica la dirección de la carpeta de resultados.

Por último, el botón de *Parar las pruebas*, detiene la ejecución de las pruebas, dejando la información que se muestra en la interfaz intacta para saber el momento en el que se han parado. Permite terminar de ejecutarse el estado en el que se encuentre para que no corte un proceso crítico que podría dar problemas al volver a ejecutar la prueba.

3.4.4. Programación del interfaz.

En la figura 33 se describe el código que define el funcionamiento de la interfaz gráfica.

Para simular el efecto de que se vayan añadiendo pestañas al control de pestañas según se van ejecutando las pruebas, como no se pueden añadir más pestañas a este control según se ejecuta el *VI*, se hará de la siguiente manera:

- Se define un control de pestañas con varías pestañas (mayor que el posible número de pruebas que se va a lanzar en una secuencia) y todas con el nombre de *Prueba* más un número.
- Al principio se ocultarán todas las pestañas menos la correspondiente a la prueba uno (ciclo *for* a la izquierda de la figura 33).
- Las pestañas se harán visibles según se van iniciando las pruebas (la forma en que se hace se explicará más adelante).

La última consideración a tener en cuenta del control de pestañas es que, al ser de muchas pestañas y cada pestaña tener dos indicadores, habrá muchos iconos en el diagrama de bloques. Estos iconos serán guardados en un estado de la estructura de casos de abajo a la derecha de la figura 33 (llamada *Contenedor de iconos*) junto al icono del control de pestañas para dar más claridad al código. La forma de modificar estos indicadores sin usar los iconos se explicará más adelante.

Para entender el funcionamiento del código de la figura 33, se explicará paso a paso su ejecución, siguiendo la línea de error de izquierda a derecha.

Al iniciar la ejecución del código, hay un ciclo *for* que recorre los elementos de *Invoke Node* tomado del control de pestañas (llamado *IN/OUT*) configurado para que devuelva las páginas. Este ciclo for deja todas las pestañas de las páginas invisibles menos la

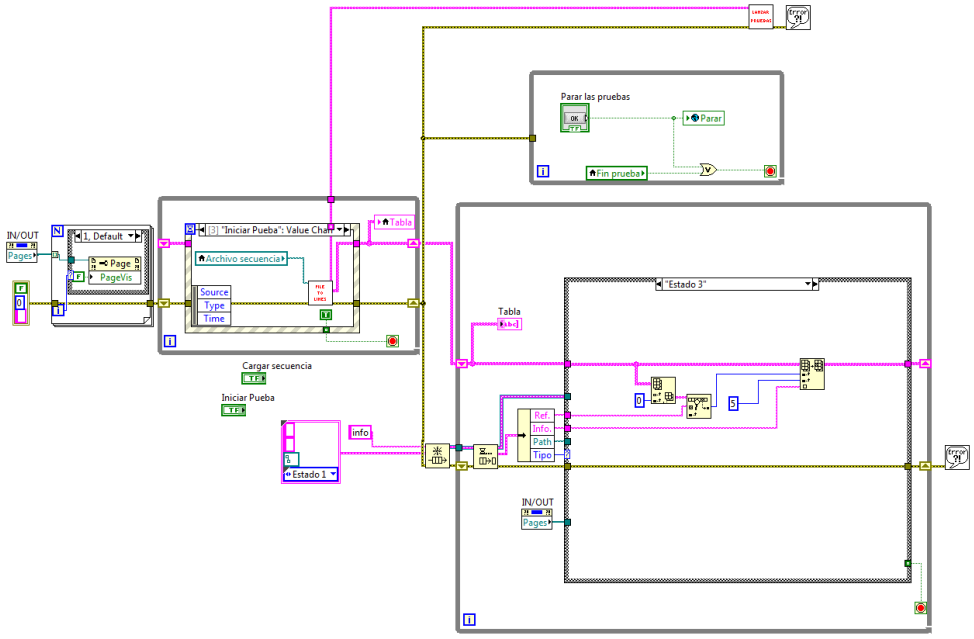


Figura 33: *Código para el funcionamiento de la interfaz.*

correspondiente a la prueba 1. Aprovechando este ciclo, también se pone a falso la variable global *Parar* y la variable local *Fin prueba* (ambas booleanas).

Una vez hecho esto, el programa entra en un ciclo *While* con una estructura de evento (o *Event Structure*) dentro. Esta estructura está configurada de tal manera que si detecta que han pulsado el botón *Cargar secuencia* o se ha hecho un cambio en el controlador *Archivo secuencia*, se entra en él *SubVI* de la figura 34. Este:

- Pasa las líneas del archivo a un *array* de cadenas de texto, introduciendo al principio de cada línea de pruebas la referencia de la prueba. Este será usado por el *SubVI* encargado de controlar la ejecución de pruebas.
- Crea la matriz de cadenas de texto que se mostrará en la tabla del centro del interfaz.

Por último, si se pulsa el botón de *Iniciar pruebas*, se hace esto mismo pero también se para el ciclo *While*, de tal manera que se sale de la estructura de eventos y así un cambio en estos controles no afectará a la ejecución de la prueba.

Una vez se ha salido de la estructura de eventos, se iniciarán tres procesos en paralelo:

- El *SubVI* que gestiona las llamadas a las pruebas.
- El ciclo *while* que lee si se ha pulsado el botón de parar las pruebas.
- El ciclo *while* que desencola y gestiona la información que dan las pruebas.

El *SubVI* que gestiona las llamadas a las pruebas es el de la zona superior de la figura 33. Este se ha programado modificando la máquina de estados del apartado 3.3.2. Las modificaciones que se le han hecho son:

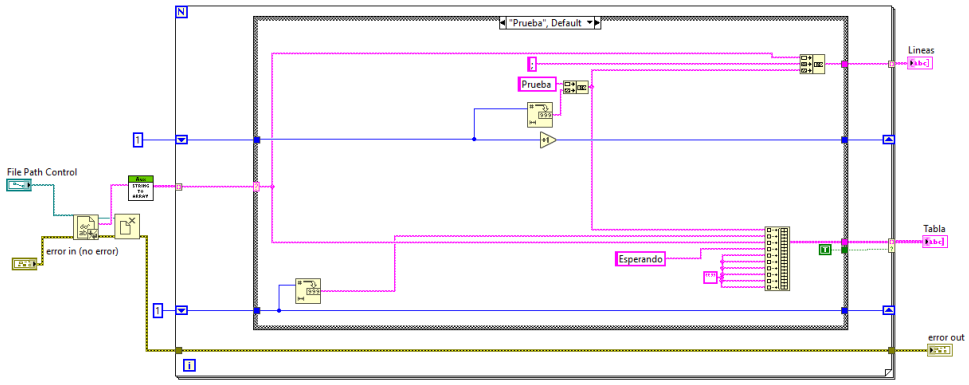


Figura 34: *SubVI para pasar el archivo a lineas.*

- En vez de pedir un archivo de texto, tiene como parámetro de entrada el *array* obtenido del *SubVI* de la figura 34.
- Se le han añadido colas para que pueda comunicarse con el interfaz a través de las referencias. También pasa estas referencias a las pruebas para que puedan comunicarse con el interfaz.
- Se le ha añadido la capacidad de notificar, a través de la cola, cuando todas las pruebas se han terminado de ejecutar.
- Se ha configurado para que cuando la variable global *Parar* sea verdadero, no se lancen más pruebas y vaya directamente a esperar a que finalicen.

Hay que tener en cuenta que la variable global *Parar* la gestiona el *while* que lee si se ha pulsado el botón de parar las pruebas (se encuentra debajo del *SubVI* que se acaba de mencionar). Este ciclo lee el estado del botón *Parar las pruebas* y cuanto lee que se pulsa pone la variable global a verdadero y se para. Por otro lado, este también se para cuando la variable local *Fin de pruebas* es verdadero.

Para que las pruebas puedan encolar información y se paren cuando lo mande la variable global, se modifica el código de las pruebas. Esto se puede ver en la figura 35, donde se muestra la porción del código de una máquina de estados modificado para poder comunicarse con la interfaz. Los cambios introducidos son:

- A través de un *Or*, si el valor de la variable *Parar* cambia a verdadero, el ciclo se para una vez se termine el estado (no pudiendo así parar en un momento crítico).
- A través de la función *Obtain Queue*, intenta obtener la referencia a la cola de nombre *info*. Esta está configurada para que si esta cola no está creada (se crea en el código del interfaz) nos devuelva un error.
  - Si la cola se encuentra creada, se ha añadido el código para que la prueba pueda encolar información.
  - Si la cola no se encuentra creada, la prueba no intentará encolar información. De esta manera se podrá usar la prueba fuera del interfaz.

- Se le ha añadido la entrada a la referencia de la prueba.

Como son las pruebas las que encolan la información, estas se convierten en las productoras de la arquitectura productor/consumidor.

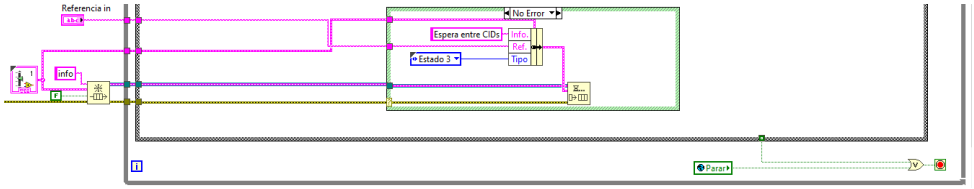


Figura 35: *Ejemplo de modificación de las pruebas.*

Finalmente, en la parte de abajo a la derecha de la figura 33 se encuentra la parte que se encarga de desencolar y gestionar la información. Para ello, primero crea, a través de la función *Obtain Queue*, una cola de nombre *info*, que utilizan las pruebas para encolar información. Como sólo se puede encolar elementos de un tipo, para poder pasar la información más fácilmente, el tipo de elementos que se han elegido para encolar son *Clusters* de cuatro elementos con los siguientes nombres:

- Ref.: Cadena de caracteres que tiene el nombre de la prueba de la que se recibe la información.
- Info.: Cadena de caracteres que se rellena si se quiere escribir algo en la tabla, en cuyo caso, esta nos dice la información a mostrar.
- Path.: *Path* que sirve para pasar la dirección del documento de entrada y de la carpeta de salida de la prueba.
- Tipo.: Enumerador que nos dice a qué tipo de información corresponde lo que se nos ha pasado. Tiene un elemento llamado *Contenedor de iconos*, cuyo uso se explicará más adelante.

Una vez se crea una referencia a la cola, se entra en un ciclo *While*. Este tiene la función *Dequeue Element*, los *clusters* que desencola se pasan a la función *Unbundle By Name*, donde se sacan los elementos de este *cluster*. El elemento *Tipo* se pasa como parámetro de una estructura de eventos. Es en esta estructura donde se gestiona la información que llega de las pruebas (consumidor), sus funciones son:

- Representa la información de la ejecución de las pruebas en la matriz del código. Para ello modifica la matriz, como se puede ver en la figura 33 (en ella añade información al estado 3).
- Se controla que cuando empiece una prueba se haga su etiqueta visible. Cuando le llega información de que se ha empezado a ejecutar una prueba llama, al *SubVI* de la figura 36. Este hace lo siguiente:
  - Busca la pestaña con el mismo nombre que la referencia a la prueba y la hace visible.

- Lee el archivo de entrada y muestra sus líneas en la tabla de la pestaña.

- Gestiona la llamada al *SubVI* (con funcionamiento parecido al del anterior punto) que escribe la dirección del archivo de resultados en el indicador *path* de la pestaña. Este no lo hace cuando se hace visible la página, ya que la carpeta se genera un tiempo después de empezar la prueba.

- Contiene el estado llamado *contenedor de iconos*, donde se guardan los iconos de los elementos de las pestañas. Nunca se entra en la ejecución del interfaz a este estado.

- Cuando el *SubVI* encargado de gestionar la ejecución de las pruebas mande información de que ha terminado, cambia la variable *Fin prueba* a verdadero, libera la referencia a la cola y para el ciclo *While* donde está contenido. De esta manera concluye la ejecución del interfaz.

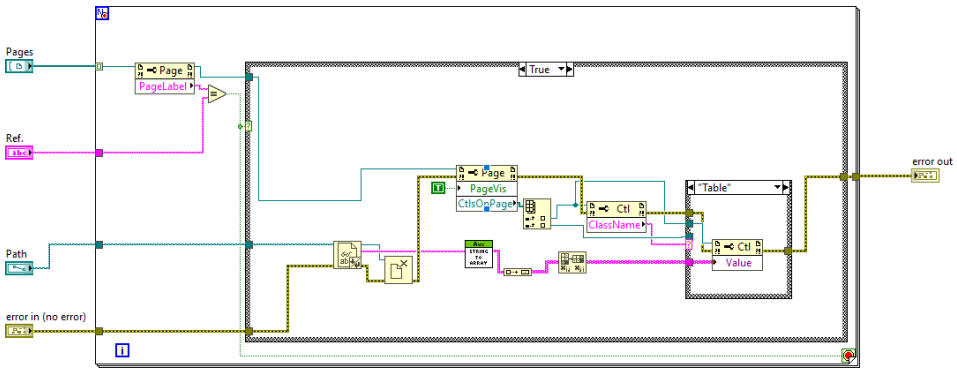


Figura 36: *SubVI que hace la etiqueta visible y añade las lineas del archivo de entrada en la tabla.*

3.4.5. Instalable.

Una vez programada la interfaz y comprobado su correcto funcionamiento, se creará un instalable para que cualquiera que desee usar la interfaz pueda ejecutar este instalable en su ordenador y se instale el software necesario paraejecutar la prueba (*LabVIEW Runtime*) y todas las carpetas y archivos necesarios.

Para crear el instalable, se han seguido los siguientes pasos:

- Incluir en el proyecto de *LabVIEW* (en el que se ha estado trabajando para crear el sistema de pruebas automático) el código de la aplicación y las carpetas y archivos necesarios para que este pueda funcionar.

- Crear un archivo ejecutable *.exe* de la aplicación e incluirlo también en el proyecto.

- Crear el instalable con todas las carpetas, archivos y el ejecutable.

Este instalable contendrá lo necesario para que, una vez instalado, se pueda usar la aplicación en cualquier equipo.

Por otro lado, si se desea pasar nuevas versiones de la aplicación a personas que ya tengan el instalable instalado y no se ha cambiado las carpetas que necesita la prueba, basta con generar un nuevo archivo ejecutable y pasarle este ejecutable.

4. Posibilidades de tests de equipos gracias a la aplicación programada

El sistema de pruebas automáticas de protecciones ha servido para poder hacer pruebas usando la aplicación y para desarrollar nuevas pruebas (fuera de este sistema) para hacer test puntuales a los equipos.

4.1. Tests usando el sistema automático de pruebas.

El sistema programado y el banco de trabajos usado proporcionaron gran versatilidad para hacer pruebas, tanto a los *ICDs* antes de hacerse oficiales para su uso en producción, como al comportamiento de los equipos al recibir estos.

Esto ha sido posible porque la máquina de estados programada daba la posibilidad de hacer una comprobación rápida de los usuarios, sistema de carpetas y permisos de la conexión *FTP* en los equipos, mediante la prueba de validación de *FTP*. Asegurándose de esta manera que las pruebas van a poder ejecutarse de forma correcta después (esta conexión es la que más problemas da).

Por otro lado, los procesos que automatizan las pruebas son muy lentos y requieren analizar un número elevado de ficheros. La automatización de estos procesos permite dejar las pruebas desatendidas, reduciendo considerablemente la dedicación de personal necesaria.

Finalmente, el poder hacer estas pruebas en varios equipos a la vez, permite probar más ficheros de configuración y equipos en menos tiempo y sin aumentar el personal.

4.2. Pruebas desarrolladas para hacer tests puntuales a los equipos.

Gracias al código programado para el sistema de pruebas automáticas, se pueden hacer tres pruebas puntuales para los equipos EF:

- Prueba *CPLD* (*Complex Programmable Logic Device*) en cámara climática.
- Comprobación de borrado del fichero *Sucesos.xml*.
- Búsqueda de señal tras reinicio.

4.2.1. Prueba *CPLD* en cámara climática.

Se programa para comprobar si una nueva tarjeta de adquisición de señales, con un *CPLD* nuevo para los equipos, funciona correctamente. En ella se dispone de dos equipos, uno con la tarjeta nueva y otro con la antigua. Las pruebas se realizan de forma independiente entre los equipos y a cada uno se le realizan las siguientes acciones:

- Se le inyecta una señal mediante un *Omicron* a través de las entradas de la tarjeta.
- Se le borra un archivo donde se guarda la información que proporciona el equipo de los estados de las entradas.
- Se reinicia el equipo.
- Se le lanza un comando al equipo para que compruebe el estado de las entradas y este vuelve a generar el archivo borrado en el segundo paso con la nueva información. Para esto hay que esperar un minuto después de lanzar el comando.
- Se comprueba a través de este archivo si el estado de todas las entradas era correcto.
    - Si no lo es, se deja de inyectar a ese equipo y se para la prueba en él.
    - Si lo es, se renombra el archivo con el número de la iteración y se guarda. Después se vuelve al paso de borrar el archivo del equipo y se repetía el proceso de nuevo.

Esta prueba esta pensada para probar si la nueva tarjeta funciona de forma correcta y compararla con la antigua, tanto a temperatura ambiente como en una cámara climática, sometida a distintas temperaturas y con distintas humedades. Para el caso de temperatura ambiente se comprueba que ambas tarjetas se comportan de forma correcta, pero no se ha podido comprobar todavía el comportamiento de las tarjetas en una cámara climática debido a la situación excepcional provocada por la pandemia de primavera de 2020.

4.2.2. Comprobación de borrado del fichero *Sucesos.xml*.

En algunos equipos, cuando se borra el archivo *Sucesos.xml* y se reinician los equipos después, hay ciertos datos de este archivo que no se borran del todo y al volver a generar el archivo, pueden aparecer datos anteriores a la generación del archivo. Por ello, se ha creado una prueba de comprobación del proceso de borrado.

Como los equipos pueden estar sometidos a cargas de trabajos muy grandes (tienen un archivo *Sucesos.xml* de gran tamaño) para intentar reproducir este problema. Esta prueba se hace en un único equipo y consiste en lo siguiente:

- Se envía un comando para generar un archivo de sucesos con unas pocas líneas y se espera un tiempo.

- Se envía un comando para borrar el archivo de sucesos y se espera un tiempo.
- Se comprueba que el archivo está borrado, parando la prueba si no lo está.
- Se envía el comando para volver a generar el archivo *Sucesos.xml*.
- Se descarga este archivo y se guardaba la primera fecha.
- Se resetea el equipo y se espera a que se termine de reeiniciar del todo.
- Se vuelve a descargar el archivo *Sucesos.xml* y se guarda la primera señal.
    - Si esta fecha no coincide con la de antes del reinicio, significa que se han colado fechas anteriores en el reinicio y se para la prueba.
    - Si ambas coinciden, indica que no ha habido ningún error y se repite la prueba desde el principio.

Gracias a esta prueba se comprueba que el problema no surge cuando el archivo *Sucesos.xml* es pequeño. Se descubre que esto pasa para archivos de sucesos largos y cuando se espera poco después de borrarlo para reiniciar el equipo, y se puede solucionar el problema.

4.2.3. Búsqueda de señal tras reinicio.

Esta prueba se crea para investigar un error con una funcionalidad en la conexión *ethernet* de los equipos que ocurre raramente cuando se reinician los equipos.

Se realiza a tres equipos de forma independiente. Estos equipos están cableados con el equipo auxiliar de tal manera que si al reiniciarse estos equipos no tienen el error en la funcionalidad, activan una entrada del equipo auxiliar. De esta manera el equipo auxiliar, a parte de servir para reiniciar los equipos, también sirve para analizar los datos.

Esta prueba hace los siguientes pasos en cada equipo:

- Se resetea el equipo y se espera a que se termine de reiniciar.
- Se descarga el archivo de sucesos del equipo auxiliar y se comprueba la fecha de la señal que se activaría si la funcionalidad funcionaba de forma correcta.
    - Si esta fecha no se encuentra o coincide con la de la anterior iteración (la primera iteración tiene como fecha la de por defecto de *LabVIEW*), significa que tiene el error en la funcionalidad tras el reinicio y se para la prueba.
    - Si esta fecha no coincide con la de la anterior iteración, significa que no tiene el error y se vuelve a repetir la prueba desde el principio.

Gracias a esta prueba se puede reproducir el problema detectado para solucionarlo y una vez solucionado, probar que el fallo no vuelve a aparecer.

5. Conclusiones

En este TFG se ha programado un sistema de pruebas automáticas que a través de un banco de pruebas certifica distintas características tanto de los equipos de protección y control de sistemas eléctricos como de sus archivos de configuración.

El banco de pruebas se ha diseñado para hacer pruebas a la vez en varios equipos disponibles. Estos equipos se pueden reiniciar, a través de un equipo auxiliar, de forma independiente.

Como herramientas para este banco de pruebas, se ha analizado un código disponible para identificar sus errores y posibles mejoras. Se le ha podido realizar una refactorización y usando una arquitectura de máquina de estados, se ha conseguido que se pueda mejorar y añadir funcionalidades de forma sencilla.

También se han descrito ciertas herramientas de *LabVIEW* para que procesos que se ejecutan en forma paralela accedan a un mismo recurso a la vez y no den problemas en su ejecución.

Por otro lado, se ha examinado cómo obtener en *LabVIEW* un programa que se encargue de lanzar las pruebas y cómo lanzarlas pruebas de forma asíncrona para que el programa pueda seguir ejecutándose a la vez que estas.

Asimismo se ha programado una interfaz para controlar el transcurso de las pruebas y se ha usado una arquitectura de productor/consumidor, basada en colas, para que este interfaz pueda comunicarse con las pruebas.

Además, caben destacar las posibilidades que ha dado la interfaz programada para testear los equipos, ya que realiza pruebas de forma autónoma, reduciendo la dedicación personal y de tiempo necesaria para ellas.

Por otra parte, a partir del sistema de pruebas automáticas, se han desarrollado las siguientes pruebas para hacer test puntuales a los equipos:

- Para probar si una nueva tarjeta de adquisición funciona correctamente.
- Comprueba si el fichero *Sucesos.xml* se borra correctamente.
- Para investigar un error en una funcionalidad de la conexión *ethernet* que ocurre raramente cuando se reinicia el equipo.

Finalmente, hay que reseñar que este proyecto ha abierto las puertas parar mejorar este sistema de pruebas automáticas. Estas mejoras serán:

- Introducir futuras pruebas como por ejemplo:
    - Validación de ajustes del equipo.

    - Prueba de las protecciones basadas en la simulación de las magnitudes del sistema eléctrico, corrientes y tensiones y entradas digitales mediante un equipo auxiliar.

- Añadir posibilidades a la interfaz para poder seleccionar y configurar pruebas y modificar o generar los ficheros de configuración.
- Crear una interfaz basada en bloques o iconos, de manera que cualquiera pueda emplear esos bloques para programar nuevas pruebas de manera sencilla.

Bibliografía

[1] A. J. Conejo and L. Baringo, Power System Operations, Springer, 2018.

[2] L. Hewitson, M. Brown and R. Balakrishnan, Practical Power System Protection, Oxford: Newnes, 2005,pp. 2-3.

[3] IEC, Comunication networks and systems for power utility automation- Part 6: Configuration description laguage for communication in electrical substations related to IEDs, 2009.

[4] "IEEE Recommended Practice for Implementing an IEC 61850-Based Substation Communications, Protection, Monitoring, and Control System," in *IEEE Std 2030*.100-2017,pp.1-67, 19 June 2017, doi: 10.1109/IEEESTD.2017.7953513.

[5] J. Davidson, An introduction to tcp/ip, New York: Springer-Verlag, 1988.

[6] J. Postel and J. Reynolds, "RFC 959 - File Transfer Protocol," Tools.ietf.org, 1895. [Online]. Available: https://tools.ietf.org/html/rfc959. [Accessed: 15- Apr- 2020].

[7] G. M. Spinelli, Z. L. Gottesman and J. Deenik, "A low-cost Arduino-based datalogger with cellular modem and FTP communication for irrigation water use monitoring to enable access to CropManage," HardwareX, vol. 6, p. e00066, 2019.

[8] D. Barrett, R. Silverman and R. Byrnes, SSH, the secure shell. Sebastopol, CA: O'Reilly Media, Inc., 2011, pp. 1-2.

[9] R. K. Megalingam, S. Tantravahi, H. S. Surya Kumar Tammana, N. Thokala, H. Sudarshan Rahul Puram and N. Samudrala, "Robot Operating System Integrated robot control through Secure Shell(SSH)," 2019 3rd International Conference on Recent Developments in Control, Automation & Power Engineering (RDCAPE), NOIDA, India, 2019, pp. 569-573.

[10] P. J. Fortier and H. E. Michel, Computer Systems Performance Evaluation and Prediction, Burlington: Elsevier Science, 2003, pp. 32-33.

[11] A. R. Al-Ali and M. Al-Rousan, "Java-based home automation system," in IEEE Transactions on Consumer Electronics, vol. 50, no. 2, pp. 498-504, May 2004.

[12] B. Vaidya, A. Patel, A. Panchal, R. Mehta, K. Mehta and P. Vaghasiya, "Smart home automation with a unique door monitoring system for old age people using Python, OpenCV, Android and Raspberry pi," 2017 International Conference on Intelligent Computing and Control Systems (ICICCS), Madurai, 2017, pp. 82-86.

[13] M. Chattal, V. Bhan, H. Madiha and S. A. Shaikh, "INDUSTRIAL AUTOMATION & CONTROL TROUGH PLC AND LABVIEW," 2019 2nd International Conference on Computing, Mathematics and Engineering Technologies (iCoMET), Sukkur, Pakistan, 2019, pp. 1-5.

[14] D. Roberts, W. Opdyke and K. Beck, Refactoring: Improving the Design of Existing Programs., Addison-Wesley Professional, 1999. pp. 8.

[15] A. de Castro, P. Zumel, O. Garcia, T. Riesgo and J. Uceda, "Concurrent and simple digital controller of an AC/DC converter with power factor correction based on an FPGA," in IEEE Transactions on Power Electronics, vol. 18, no. 1, pp. 334-343, Jan. 2003.

[16] J. R. Lajara Vizcaíno and J. Pelegrí sebastiá, LabVIEW Entorno gráfico de programación, Barcelona: Marcombo, 2007.

[17] "Diferencias entre Reentrant, Templates y Dynamic VIs - National Instruments," Knowledge.ni.com, 2020. [Online]. Available: https://knowledge.ni.com/KnowledgeArticleDetails?id=kA00Z0000019L4zSAE&l=es-ES [Accessed: 04- Apr- 2020].

[18] Sheng Zhi-yu,"Realization of the motor data acquisition and analyzation system based on the producer/consumer model of LabVIEW," 2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC), Xi'an, 2016, pp. 330-334.

[19] J. Travis, and J. Kring, LabVIEW for everyone: graphical programming made easy and fun, 5th ed. Prentice-Hall, 2007.

[20] "Reentrant and Non-Reentrant SubVIs Comparison in LabVIEW FPGA - National Instruments," Knowledge.ni.com, 2020. [Online]. Available: https://knowledge.ni.com/KnowledgeArticleDetails?id=kA00Z000000PApASAW&l=es-ES. [Accessed: 04- Apr- 2020].