Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica

Componentes Intel de Costa Rica

Definición y automatización del proceso de validación de las fallas del producto *Montvale*

Informe de Proyecto de Graduación para optar por el título de Ingeniero en Electrónica con el grado académico de Licenciatura

Alexander Guzmán Ramón

Heredia, Diciembre de 2006

INSTITUTO TECNOLOGICO DE COSTA RICA

ESCUELA DE INGENIERIA ELECTRONICA

PROYECTO DE GRADUACIÓN

TRIBUNAL EVALUADOR

Proyecto de Graduación defendido ante el presente Tribunal Evaluador como requisito para optar por el título de Ingeniero en Electrónica con el grado académico de Licenciatura, del Instituto Tecnológico de Costa Rica.

Miembros del Tribunal

Dipl. Ing. Eduardo Interiano Salguero

Profesor lector

M.Sc. Ing. Roberto Pereira Arroyo

Profesor lector

M.Sc. Ing. Carlos Badilla Corrales

Profesor asesor

Los miembros de este Tribunal dan fe de que el presente trabajo de graduación ha sido aprobado y cumple con las normas establecidas por la Escuela de Ingeniería Electrónica

Cartago, Enero de 2007

Declaro que el presente Proyecto de Graduación ha sido realizado enteramente por mi persona, utilizando y aplicando literatura referente al tema e introduciendo conocimientos propios.

En los casos en que he utilizado bibliografía, he procedido a indicar las fuentes mediante las respectivas citas bibliográficas.

En consecuencia, asumo la responsabilidad total por el trabajo de graduación realizado y por el contenido del correspondiente informe final.

Heredia, Diciembre de 2006

Alexander Guzmán Ramón

Cédula: 303870863

Resumen

Para una empresa de prestigio mundial como Intel, es muy importante garantizar a sus clientes la calidad del producto que están adquiriendo, por este motivo, se realizan pruebas funcionales y paramétricas durante el proceso de producción de un microprocesador.

Todas las unidades que presentan problemas son desechadas, lo cual, representa un desperdicio de recursos; por lo tanto, un ingeniero debe realizar un proceso de validación de las unidades defectuosas para determinar el origen de las fallas y de esta manera poder corregir alguna de las etapas del proceso de producción para minimizar el número de unidades defectuosas

Actualmente, no se cuenta con una metodología de validación para el producto *Montvale*, lo cual, aumenta considerablemente el tiempo que requiere un ingeniero para realizar la validación de las unidades

Por este motivo, este proyecto pretende definir y automatizar un procedimiento de validación para las principales fallas de: *Caché*, *SBFT* y *Scan* del producto *Montvale*.

Para lo cual, hay que realizar una investigación para definir una metodología de validación de fallas y se debe implementar un algoritmo de ejecución dinámica que permita la automatización de dicho proceso.

Es decir, se pretende diseñar una máquina virtual que sea capaz de interpretar las instrucciones del usuario y que ejecuta las acciones de manera automática en la plataforma de prueba que se utiliza para validar las fallas del producto *Montvale*.

Palabras Claves: Metodología de validación, automatización de las órdenes de trabajo, *Montvale, Scan, SBFT, Caché*

Abstract

For a world wide leader company as Intel, is very important to can guaranty to it customers the property operation of the products that they purchase, that's the main reason why is very important to make functional and structural test during the production process.

All the unit that doesn't works property are discards, that means a waste of resources for Intel, Intel, it also very interesting to reach a high level of productivity so an engineer have to make a validation process of the faulty units to detect the cause of the fail. This way he/she can identify a wrong procedure during the production process, correct it and reduce the number of faulty units that are produced.

At the moment, Intel doesn't have a validation methodology for the Montvale product, that increase considerable the time that an engineer needs to make the troubleshooting needed to detect the cause of the fail.

Those reasons by this project try to define and automate the validation process of the fails that occur at the test of: Caché, SBFT and Scan of the Montvale product.

To make that happen, it's necessary to learn basic concepts of the three tests mention above and make a research to define the validation methodology for those tests type.

Also is essential to implement a dynamic execution algorithms to automate the validation process, it means to create a virtual machine that can read, understand and execute at the CMT the instructions that the user defined.

Keywords: Validation methodology, automate of work orders, Montvale, Scan, SBFT, Caché

Los triunfos son para compartirlos con los seres más queridos; aquellas personas capaces de hacer cualquier cosa para contribuir con la realización de tus metas, sin esperar nada a cambio y que se regocijan de verte triunfar.

Por este motivo, quiero dedicar este logro a la mayor bendición que Dios pudo darme…*Mi madre*.

Sin su amor, ayuda, apoyo y guía nada de esto hubiera sido posible.

También, quiero compartir esta victoria con otras tres mujeres maravillosas que Dios puso en mi vida: *mí abuelita Lidiette, mí princesa Tricia y mí princesita Estefanía*, quienes han sido mi soporte y motivación.

Espero que ellas se sientan tan orgullosas de tenerme a mí, como yo lo estoy de ellas.

Agradecimientos

Primeramente, quiero agradecer a Dios por el regalo de la vida, por sustentarme y guiarme hasta aquí, por darme una mamá tan maravillosa, esforzada y triunfadora que ha sabido enseñar y guiarme por el camino que El ha trazado para mí.

Quiero agradecer a mi madre, por ser tan buen papá, tan buena mamá, tan buena amiga, tan buena guía, por siempre tener un sabio consejo, unas palabras de aliento y un corazón llego de amor.

Quiero agradecer a Tricia, por su cariño, por su amistad, por su compañía, por quererme por lo que soy, incluyendo mis defectos, por ayudarme a ser feliz, por compartir su vida con la mía.

Quiero agradecer a todos aquellas personas que contribuyen al desarrolla de la persona que hoy soy, a mis maestros de escuela, mis profesores de colegio y universidad que me ayudaron a desarrollar mis conocimientos, a fortalecer mi personalidad y definir una filosofía de vida.

A mis compañeros y amigos por apreciar mis virtudes y comprender mis defectos, por los buenos momentos que hemos pasado, por lo mucho que hemos reído.

Por último, quiero agradecer a las personas de Intel que de una u otra manera contribuyeron en el desarrollo del proyecto, que creyeron en mí y me dieron la oportunidad.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

Capítulo 1: Introducción

En este capítulo se realiza la descripción del problema existente y como este repercute en la empresa, se puntualiza en la importancia de su solución y se muestran al lector las primeras pinceladas de la solución que se implementó.

1.1. Problema existente e importancia de su solución

La subdivisión *Server Development and Manufacture* (SDM) del departamento *Enterprise Microprocessor Group* (EMG) se encarga de verificar el correcto funcionamiento de los microprocesadores para servidores que Intel produce. Actualmente, se cuenta con un programa de prueba[1] para el producto Montecito[2], el cual, realiza un conjunto de pruebas para determinar el correcto funcionamiento de las partes internas del microprocesador, estas pruebas no permiten identificar el origen de las fallas, sino que solo determinan si la unidad pasa o no la prueba.

Lo antes expuesto, se puede ver claramente en la Figura 1-1, en la cual, se observa una representación gráfica de un flujo de pruebas, es decir, cada caja que se observa representa una prueba en partícula. Algunas pruebas terminan de ejecutarse y salen por el puerto número 1 y otras salen por el puerto número 0.

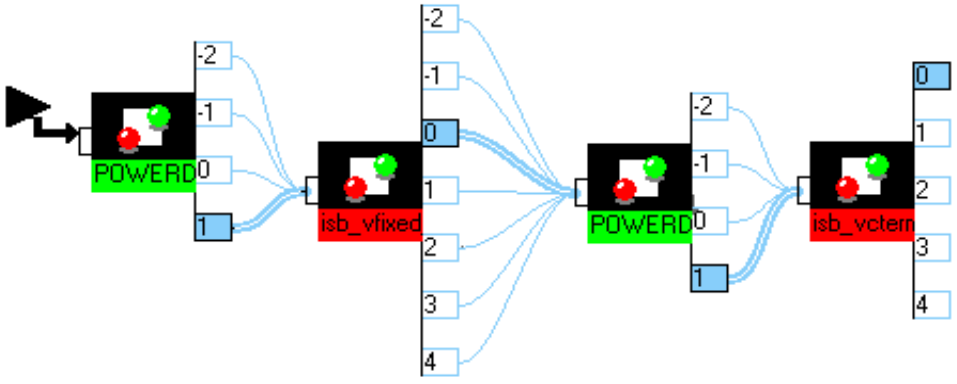

Figura 1-1 Diagrama de flujo de un conjunto de pruebas de un programa de prueba

[1] Programa de prueba: es el programa que se encarga de control las pruebas que se realizan a las unidades
[2] Montecito: el nombre utiliza Intel para referir a un microprocesador de la familia Itanium II

Todas las pruebas que terminan por el puerto 0, son pruebas en que la unidad no paso exitosamente y cualquier otra prueba que salga por un puerto diferente, implica que la unidad no presentó ningún problema en dicha prueba.

Cuando una unidad presenta una falla, el programa de prueba almacena en un archivo un código que identifica el tipo de falla que se presentó, además, se debe realizar un proceso de validación, el cual, pretende determinar si la falla se debe a un problema en la unidad, en el equipo utilizado o en el programa de prueba.

Este proceso de validación actualmente es realizado por un ingeniero, el cual, a partir del reporte de fallas que se generó durante las pruebas en producción y la utilización de varias herramientas de ingeniería disponibles en el *Tester CMT*[3], realiza una serie de pruebas adicionales y a partir de los resultados obtenidos determina el origen de la falla.

Las pruebas de validación y los experimentos que el ingeniero realiza para determinar el origen de la falla por lo general requieren de mucho tiempo de prueba, por lo tanto, es necesaria la participación de un operario que colabore en el proceso de adquisición de resultados y en el manejo de las unidades.

Por lo tanto, el problema radica en que el intervalo de tiempo entre la detección de una falla y la determinación de su causa, actualmente depende de la intervención de al menos un ingeniero y un operario que se encarguen de realizar la validación de las unidades, lo cual, repercute en el tiempo de respuesta para producir una acción correctiva en alguna de las etapas anteriores del proceso de producción.

Para el departamento, es muy importante poder establecer el origen de las fallas en el menor tiempo posible, ya que de esta manera se reduce el tiempo para realizar una acción correctiva, ya sea en el diseño o en el ensamblaje de las unidades, lo cual, implica reducir el número de unidades defectuosa producidas y por ende un aumento en la productividad y la calidad del producto.

[3] Tester CMT: la plataforma de pruebas que utiliza el departamento para probar las unidades

Además, si se logra reducir el tiempo de participación activa del ingeniero en el proceso de validación, este dispondría de mayor tiempo para dedicarse a otro tipo de labores, lo cual, implicaría un aumento en su productividad y repercutiría de forma directa en la productividad de la empresa.

Actualmente, el departamento SDM esta concentrado en el desarrollo de las pruebas para el nuevo producto Montvale[4], el cual, será la nueva versión de microprocesador de doble núcleo para servidores, por lo tanto, la herramienta de ingeniería que se desarrollará en el proyecto estará orientada a dicho producto.

Mediante la definición de un proceso de validación de fallas y la automatización del mismo, además de reducir el tiempo de respuesta de una acción correctiva, se podría obtener varios beneficios entre los que destacan:

- Como ya se mencionó, se podría reducir el tiempo de participación del ingeniero en el proceso de validación y por ende, este dispondría de tiempo adicional para hacer otras labores y aumentaría su productividad.
- El ingeniero tendría un mayor nivel de información de la falla en un menor tiempo, lo cual, reduciría el tiempo que este requiere para determinar la causa del problema.
- Con la automatización de la adquisición de los resultados de las pruebas de ingeniería, se reduce el tiempo de participación de los operarios y la dependencia de ellos, lo cual, reduce los errores de índole humana y los recursos requeridos para realizar el proceso de validación
- Se puede dar el caso que se implemente el procedimiento de validación en el proceso de producción, lo cual, mejoraría la calidad del producto y la productividad

[4] Montvale: la nueva versión del microprocesador de doble núcleo de la familia Itanium II

1.2. Solución seleccionada

1.2.1. Requerimientos

Los requerimientos del proyecto son los siguientes:

- Establecer un procedimiento de validación que permita determinar la causa de las principales fallas que se presentan en las pruebas de: Caché[5], SBFT[6] y Scan[7]del producto Montvale
- La herramienta debe ejecutar de forma automática dos tipos de órdenes de trabajo: la validación de falla y los experimentos de ingeniería
- Definir el formato de un archivo de texto, en el cual, se debe especificar el procedimiento que se debe seguir para realizar la validación de cada falla o el experimento de ingeniería
- Modificar el flujo del programa de prueba para poder realizar la validación automática de las principales fallas que presentan en las pruebas de: Caché, SBFT y Scan del producto Montvale
- Crear un archivo de texto donde se especifique los resultados más relevantes de las pruebas realizadas

1.2.2. Descripción general de la solución

Muchas de las pruebas que realiza el ingeniero para determinar la causa de la falla, por ejemplo el *Shmoo*[8], requieren de una interpretación gráfica de los resultados. Por este motivo, para poder realizar la automatización del procedimiento de validación, se requiere diseñar una herramienta que permita definir un procedimiento y que almacene en un archivo de texto los resultados más significantes de las pruebas realizadas.

[5] Caché: los arreglos de memoria internas del microprocesador
[6] SBFT: son pruebas funcionales basadas en una metodología Estructural
[7] Scan: son pruebas que determinan el estado de los flip-flops que se encuentran alrededor del procesador
[8] Shmoo: un barridos de una o más condiciones de las prueba que se realizan a una unidad

Es importante mencionar que el ingeniero será responsable de identificar el origen de la falla y la herramienta que este proyecto busca desarrollador se encargará de automatizar la ejecución del procedimiento de validación o de experimento de ingeniería y la captura de los resultados.

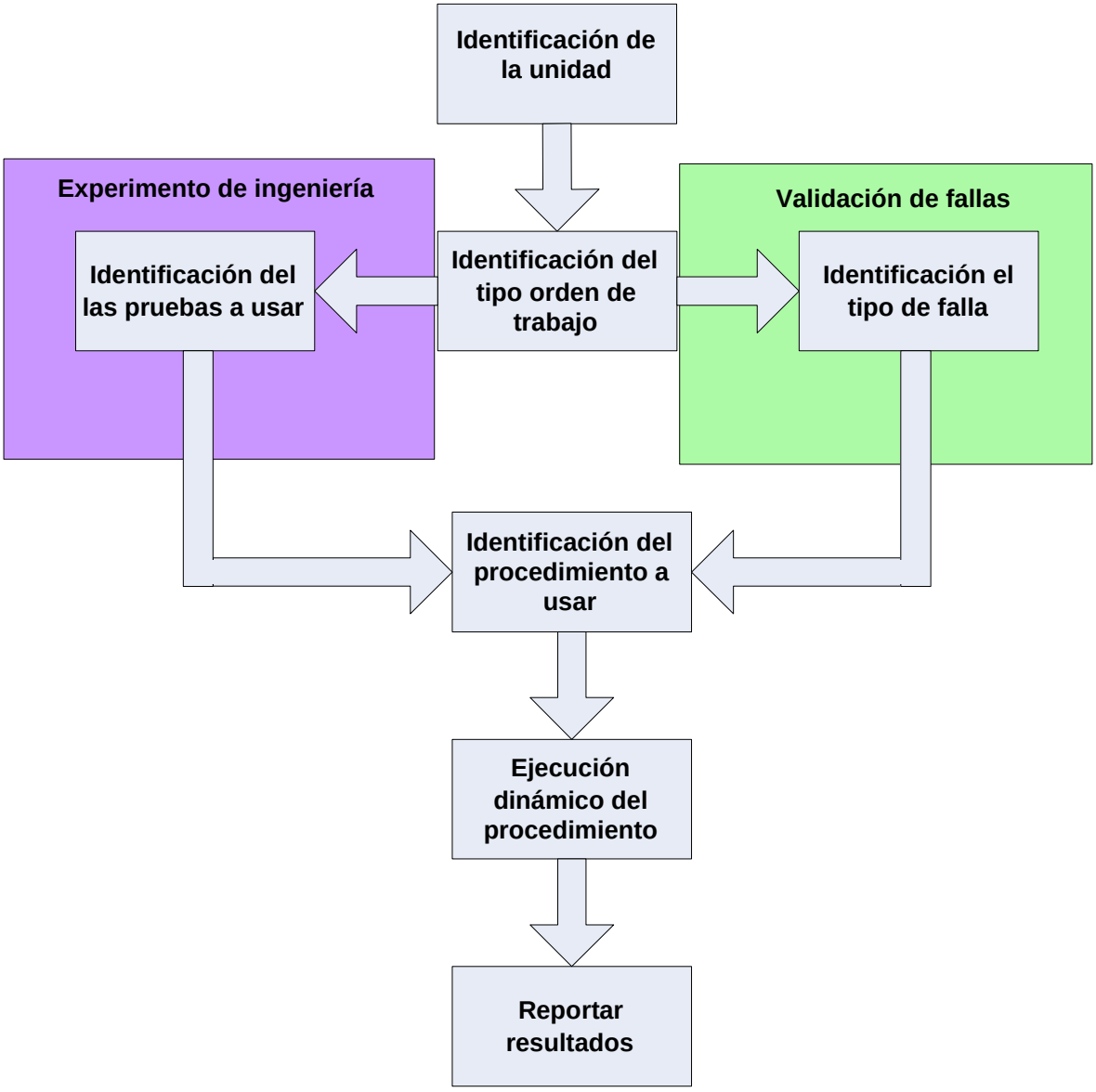

Figura 1-2 Diagrama de bloques de primer nivel de la solución

En la Figura 1-2, se muestra el diagrama de bloques de primer nivel de la solución implementada para automatizar del proceso de validación de fallas, a continuación, se realiza una breve descripción de las etapas que constituyen la solución.

- Identificación de la unidad: en esta etapa se identifica la unidad que se desea validar, es decir, se lee el *visual_ID* [9] o el *ULT*[10] de la unidad; el *visual_ID*, es un código único que se encuentra impreso en el exterior de la unidad y el *ULT*, es un código único que se encuentra grabado en los fusibles internos de la unidad.
- Identificación del tipo orden de trabajo: esta etapa utiliza el *visual_ID* o el ULT que se leyó en la etapa anterior para consultar el archivo de orden de trabajo y determinar si para la unidad que se esta probando el usuario definió un tipo de falla, de ser así, se debe realizar una validación, de lo contrario se realiza un experimento de ingeniería.
- Identificación del tipo de falla: esta etapa, se encarga de leer el tipo de falla que el usuario especificó para la unidad que actualmente se esta probando.
- Identificación de las pruebas a usar: para realizar un experimento de ingeniería el usuario debe especificar en el archivo de orden de trabajo una prueba o un conjunto de pruebas a usar, esta etapa se encarga de leer dicha información.
- Identificación del procedimiento a usar: cuando la orden de trabajo es una validación, se utiliza el tipo de falla de la unidad para identificar el procedimiento de validación que se debe ejecutar, por el contrario, si la orden de trabajo es un experimento, la herramienta lee del archivo de orden de trabajo la metodología que el usuario definió.

[9] Visual_ID: un código único impreso en el exterior de la unidad
[10] ULT: un código único grabado en los fusibles internos de la unidad

En la Figura 1-3, se muestra una representación gráfica de la modificación que se realizó del flujo del programa de prueba para introducir la automatización del proceso de validación. Esta figura, contempla un caso trivial donde solo existen dos tipos de pruebas, sin embargo, el principio que se utiliza es aplicable a n tipos de pruebas.

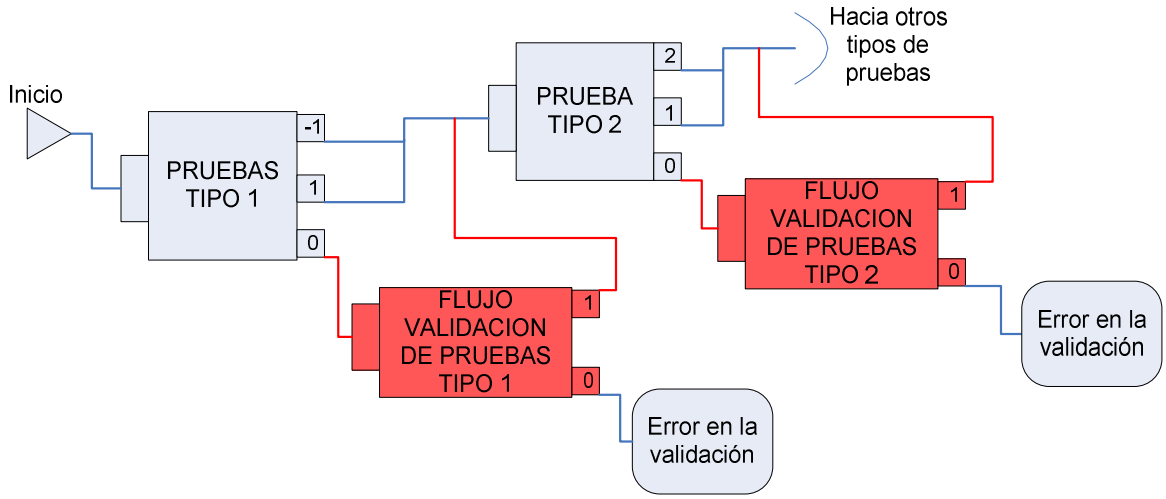

Figura 1-3 Modificación del flujo del programa de prueba para automatizar el proceso de validación

Se sabe, que cuando una prueba falla el resultado por lo generar es igual a cero, por lo que el flujo continua por el puerto cero de la prueba.

Por lo tanto, una vez que se conoce el tipo de *bin*, se puede determinar cual fue la prueba o el grupo de pruebas que fallaron, es decir, se podría visualizar como si se estuviera saliendo por el puerto cero de la prueba que falló y a partir de este punto, es que se inicia el proceso de validación de la falla, es decir, la entrada al flujo de validación de una falla esta conectada directamente con el puerto de falla de dicha prueba.

- Ejecución dinámica del procedimiento: El programa de prueba, define un flujo para todas las pruebas que se realizan a un producto, sin embargo, el usuario del *Tester CMT* puede utilizar la interfaz gráfica del sistema para redefinir el flujo y las condiciones de las pruebas que se realizan a una unidad en particular, estas modificaciones también se pueden hacer por medio de las instrucciones de *Helper Class*[11] o las instrucciones de *UserSDK Tools API*[12].

  La herramienta que se desarrollo, utiliza las instrucciones de *Helper Class* y los de *UserSDK Tools API* para definir de manera dinámica el flujo de validación de las fallas, es decir, la secuencia de las pruebas de validación que la herramienta realiza es definida a partir de la interpretación del archivo donde el usuario define la metodología de validación para cada tipo de falla.

  En la Figura 1-4, se muestra un diagrama de bloques donde se ejemplifica el comportamiento deseado de la ejecución del sistema.

  Para comprender el ejemplo de la Figura 1-4, se debe asumir que ya se ha determinado que se desea validar una falla tipo 50 y que existe una metodología para la validación de esta falla que consiste en la ejecución de las instrucciones P1, P5 y P2.

  Entonces, el primer paso del algoritmo de ejecución dinámica es leer del archivo de metodología el código que identifica a la instrucción P1, luego, se indica al *Tester CMT* que se desea ejecutar dicha prueba, se espera que la prueba se ejecute para poder leer los resultados obtenido, realizar el almacenamiento de estos en el archivo de salida y continuar con la ejecución de la siguiente instrucción.

[11] Helper Class: un conjunto de instrucciones que permiten controlar ciertas funciones del tester CMT
[12] UserSDK Tools API: un conjunto de instrucciones de bajo nivel que permiten controlar las funciones del tester CMT

Una vez, que la herramienta terminó con la ejecución y captura de los resultados de la instrucción P1, esta, lee la siguiente instrucción del archivo de metodología y repetir todo el proceso que se realizó con la prueba anterior, la herramienta realiza el mismo procedimiento con todos las instrucciones que formen parte de la metodología.

Para la ejecución de un experimento de ingeniería se utiliza el mismo procedimiento descrito anteriormente.

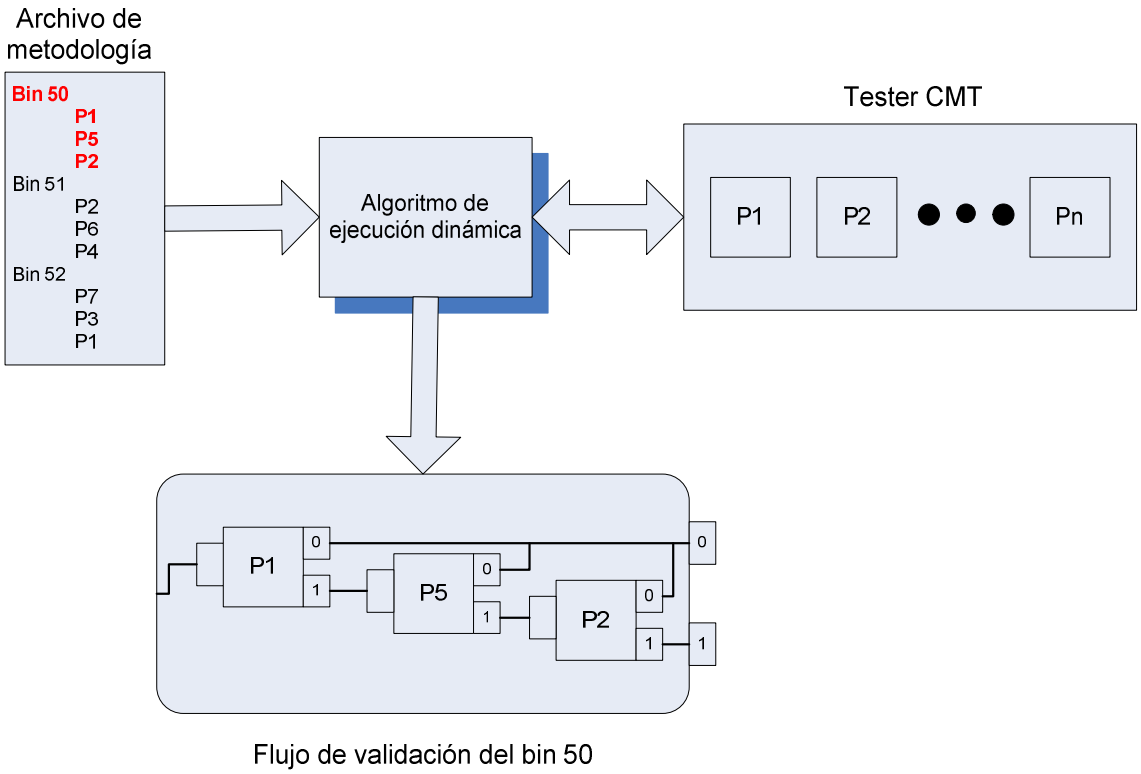

Figura 1-4 Diagrama de bloques de la ejecución dinámica del flujo de validación

- Reportar resultados: al final de la ejecución de la metodología de validación se obtenga un documento de texto donde se resumen los resultados más relevantes obtenidos con la ejecución de las pruebas de validación o los experimentos de ingeniería realizados.

Capítulo 2: Meta y objetivos

2.1. Meta

Reducir el tiempo para ejecutar una acción correctiva en el proceso de producción de Montvale con el fin de disminuir el número de unidades defectuosas que se producen.

2.2. Objetivo general

Automatizar el proceso de caracterización de las principales fallas que se presentan en las pruebas de: Caché, SBFT y Scan que se realizan al producto Montvale a partir de la definición de una metodología de validación

2.3. Objetivos específicos

2.3.1. Objetivo de Investigación

- Definir una metodología de validación para las principales fallas que se presentan en las pruebas de: Caché, SBFT y Scan del producto Montvale a partir de la metodología existente para el producto Montecito.

2.3.2. Objetivos de Software

- Desarrollar una herramienta que sea capaz de lee, interpretar y ejecutar una serie de instrucciones que el usuario defina en un archivo de texto para controlar la ejecución de un procedimiento de validación o un experimento de ingeniería
- Modificar el flujo del programa de prueba para realizar automáticamente la validación de las principales fallas que se presenten en las pruebas de: Caché, SBFT y Scan que se realizan al producto Montvale
- Almacenar en un archivo de texto los resultados más relevantes de las pruebas de validación que se realicen, esto con el fin de aumentar el nivel de información de la falla que el ingeniero pueda tener.

2.3.3. Objetivos de Documentación

- Documentar el proceso de validación que se definió a partir de la investigación realizada para determinar las principales fallas que se presentan en las pruebas de: Caché, SBFT y Scan.
- Documentar el funcionamiento de la herramienta que este proyecto desarrolló para lograr automatizar la caracterización de las principales fallas de: Caché, SBFT y Scan del producto Montvale.

Capítulo 3: Marco teórico

3.1. Descripción del proceso de producción

El proceso de producción de un microprocesador es muy extenso y complejo, por lo tanto, realizar una descripción detallada de este proceso esta fuera de los alcances de este documentos, sin embargo, se realizará una breve descripción de las etapas más relevantes que este presenta, con el fin de ubicar al lector en el contesto del proyecto.

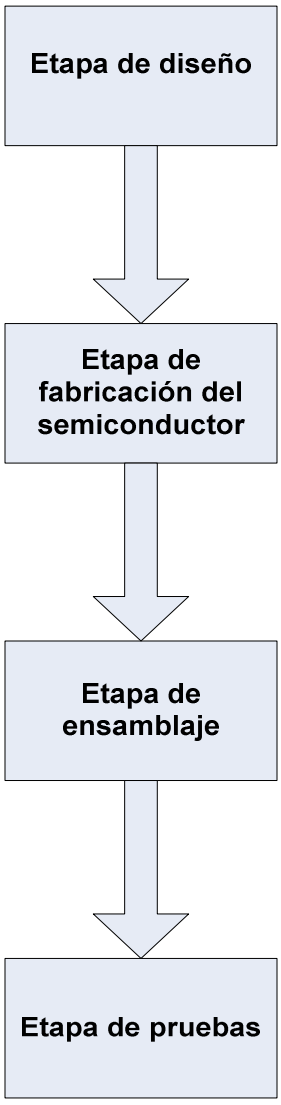

Figura 3-1 Diagrama de bloques del proceso de producción

En la Figura 3-1, se puede observar un diagrama de bloques muy general del proceso de producción de un microprocesador, seguidamente, se realiza una breve descripción de las etapas del proceso.

- Etapa de diseño: aquí es donde inicia el proceso de producción, en esta etapas se definen todas las características que el producto debe tener, es decir, se define la arquitectura del microprocesador y se obtiene un diagrama esquemático que especifique la distribución de los transistores en el semiconductor.
- Etapa de fabricación del semiconductor: en esta etapa se obtiene el *wafer*[13] de silicio, el cual, es un círculo que contiene muchos circuitos independientes, cada uno de los cuales constituye la unidad de control del microprocesador, cada uno de estos circuitos son probados para determinar su correcto funcionamiento.
- Etapa de ensamblaje: es el inicio del proceso que se realiza en Costa Rica, el cual, consiste en cortar cada uno de las unidades de control buenas que contiene un *wafer*, colocarlas en el substrato y agregarle todos los componentes necesarias para que el microprocesador pueda trabajar en un computador.
- Etapa de pruebas: si bien es cierto, se realizan pruebas para determinar el correcto funcionamiento de las unidades de control contenidas en el *wafer*, esto no garantiza que después del largo proceso de ensamblaje se obtenga un producto que funcione correctamente. Por este motivo, se realizan nuevas pruebas para garantizar la calidad del producto obtenido, en esta etapa del proceso es donde se enmarca el proyecto.

  Se realizan dos tipos de pruebas: las paramétricas y las funcionales, las pruebas paramétricas se orientan a determinar el estado de todos los pines del microprocesador y las pruebas funcionales se orientan a determinar el estado de la lógica interna del microprocesador.

[13] Wafer: la oblea de silicio que contiene las unidades de control del microprocesador

Es decir, las pruebas paramétricas, buscan establecer si el microprocesador cumple con las especificaciones de: consumo de potencia, frecuencia de operación, tensión, niveles lógicos, entre otras cosas y las pruebas funcionales, buscan determinar el correcto funcionamiento del microprocesador, para lo cual, se produce una secuencia de combinaciones lógicas a los pines de entradas y se comparan las respuestas obtenidas en los pines de salida con un patrón establecido.

3.2. Descripción del equipo que se utiliza para probar las unidades

3.2.1. El *Tester CMT*

El *Tester CMT* es la plataforma que se utiliza para realizar las pruebas funcionales y paramétricas de un microprocesador, es decir, el Tester CMT contiene todo el hardware requerido para alimentar, estimular, realizar mediciones e interpretar los resultados obtenidos en cada uno de los pines del microprocesador [7].

Por este motivo, para poder crear un programa de prueba o para modificar el flujo de ejecución de este, se requiere tener conocimiento sobre la arquitectura del *Tester CMT*, es decir, es necesario saber cuales módulos de hardware programables están disponibles en la plataforma de prueba y sobre todo, se debe conocer como se realiza la configuración y el control de estos módulos.

En la Figura 3-2, se muestra una fotografía del *Tester CMT*, en la cual, se pueden observar que el sistema se compone por tres elementos: la interfaz con el usuario, la unidad principal y la cabeza de pruebas.

Figura 3-2 Fotografía del *Tester CMT* [2] p.11

- La interfaz con el usuario: se compone de un monitor, un teclado y un ratón, es la parte del *Tester CMT* que le permite al usuario interactuar con el *System Controller*[14] del *Tester CMT*.
- La unidad principal: es donde se encuentran los módulos de hardware y el sistema de control del *Tester CMT*, es decir, la unidad principal integra todos los elementos físicos y lógicos necesarios para poder realizar las pruebas funcionales y paramétricas de los microprocesadores.

[14] System Controller: una computadora que permite la interacción entre el usuario y el Site controller

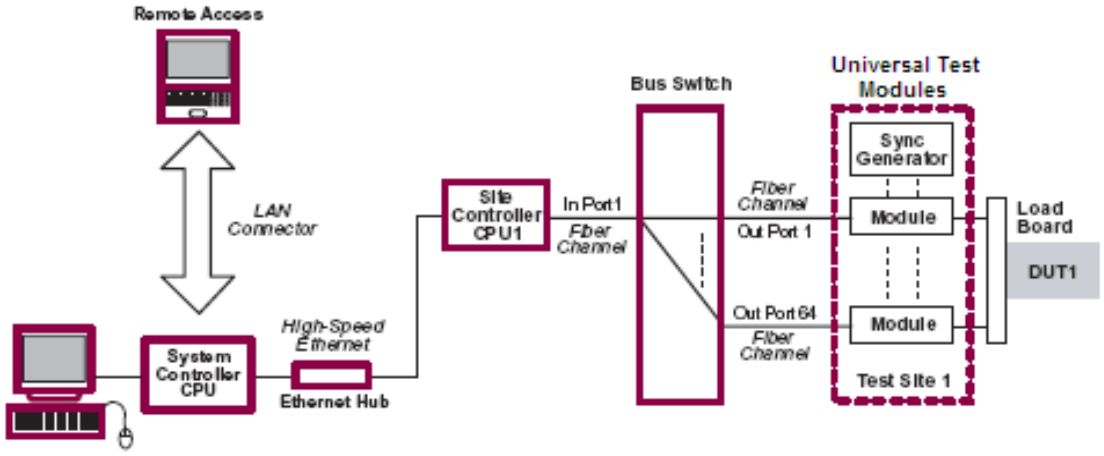

Figura 3-3 Diagrama de bloques de la unidad principal del *Tester CMT* [1] p.29

Para facilitar la comprensión del funcionamiento de la unidad principal del *Tester CMT*, a continuación se realiza una descripción de las funciones que realizan los principales bloques del diagrama de la Figura 3-3.

- *System Controller:* es una computadora que corre Windows OS y utiliza el software TSS[15] para comunicarse con el *Site controller*[16] [6], este se encarga de establecer una comunicación entre el *Site controller* y el usuario, es decir, le permite al usuario cargar el programa de prueba, correr pruebas específicas, utilizar las herramientas de ingeniería y observar los resultados obtenidos [1].

  El System Controller se encarga de interpretar las instrucciones que el usuario ejecuta, ya sea de manera gráfica o por medio de las funciones de *Helper Class*, enviar esas órdenes codificadas al *Site controller* para que este ejecute la acción, luego se encarga de leer, interpretar y desplegar los resultados en el motor para informar al usuario de los resultados obtenidos.

[15] Software TSS: el software que permite la comunicación entre el System Controller y el Site Controller
[16] Site controller: se encarga de controlar todos los módulos de hardware programables del Tester CMT

Además, como se puede observar en la Figura 3-3, el *System Controller* puede controlar más de un *Site controller* a la vez y también le permite al usuario tener control remoto del *Tester CMT*.

- *Site Controller*: es el control centralizado de todos los módulos de hardware programable que utiliza el *Tester CMT* para realizar un conjunto de pruebas a una unidad, este se encarga de interpretar las órdenes del *System Controller* y a partir de ellas configura y controla el hardware necesario para realizar las pruebas, además, se encarga de interpretar los resultados obtenidos y transferir esta información al *System Controller* [1].

  Es decir, el *Site Controller* se encarga de configurar la dirección, la temporización, los niveles de tensión de los pines de entrada salida que el *Tester CMT* utiliza para hacer las pruebas [3], además controla todas la etapas de potencia y de generación de señales necesarias para que el microprocesador pueda operar [4,5].

- *Universal test Modules:* es la colección de todos los módulos de hardware programables disponibles en el *Tester CMT*, es decir, todos los generadores de señales, las etapas de potencia, los pines de entrada salida para las pruebas y por ende contempla toda la lógica y circuitería relacionada con cada uno de estos módulos. Todos los módulos de hardware programables se encuentran conectados físicamente con la cabeza de pruebas del *Tester CMT*.

- *Load Board o Tester Interfase Unit (TIU):* es una tarjeta que realiza la interfaz entre la cabeza del Tester CMT y la unidad que se desea probar (DUT), es decir, es el medio físico que se encarga de conectar los pines de entrada salida, los pines de potencia, los pines de temporización de los módulos de hardware programables del *Tester CMT* con las patillas del microprocesador que sea probar. En la Figura 3-4, se muestra una fotografía de un TIU.

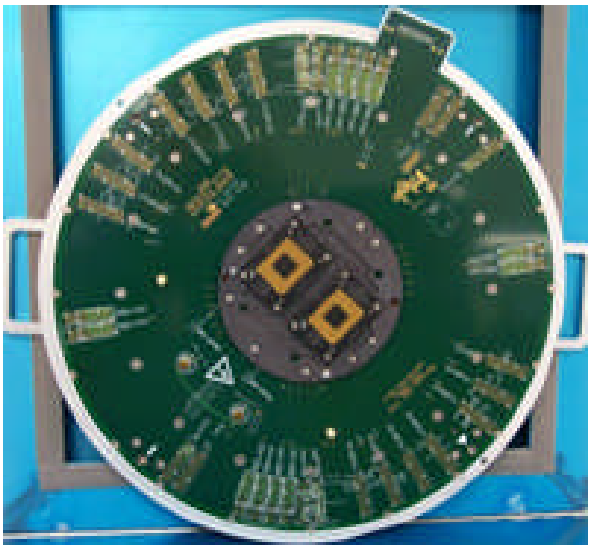

Figura 3-4 Fotografía de un TIU [4] p.3

- La cabeza de pruebas: es el conjunto de todos los conectores de los módulos de hardware programables, es decir, en la cabeza de pruebas se encuentran las señales de potencia, las señales de temporización y los pines de entrada salida que se utilizan para realizar las pruebas de las unidades, tal cual se puede observar en la Figura 3-5.

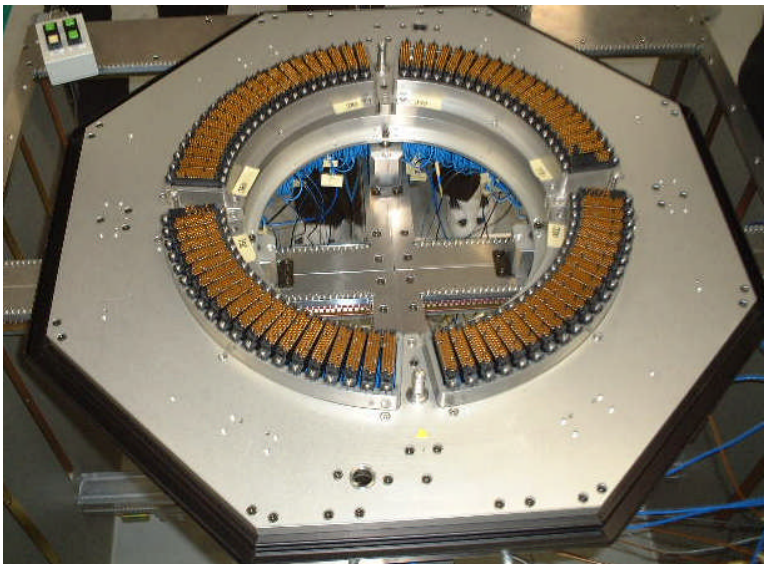

Figura 3-5 Fotografía de la cabeza de pruebas del *Tester CMT*

3.2.2. El TSS

El TSS es el software que corre en el *System Controller*, el cual, establece un medio entre el usuario y el *Site Controller*, es decir, facilita una interfaz gráfica y un medio de comunicación para que el usuario tenga control y noción de las pruebas que se realizan a la unidad en el *Site Controller*.

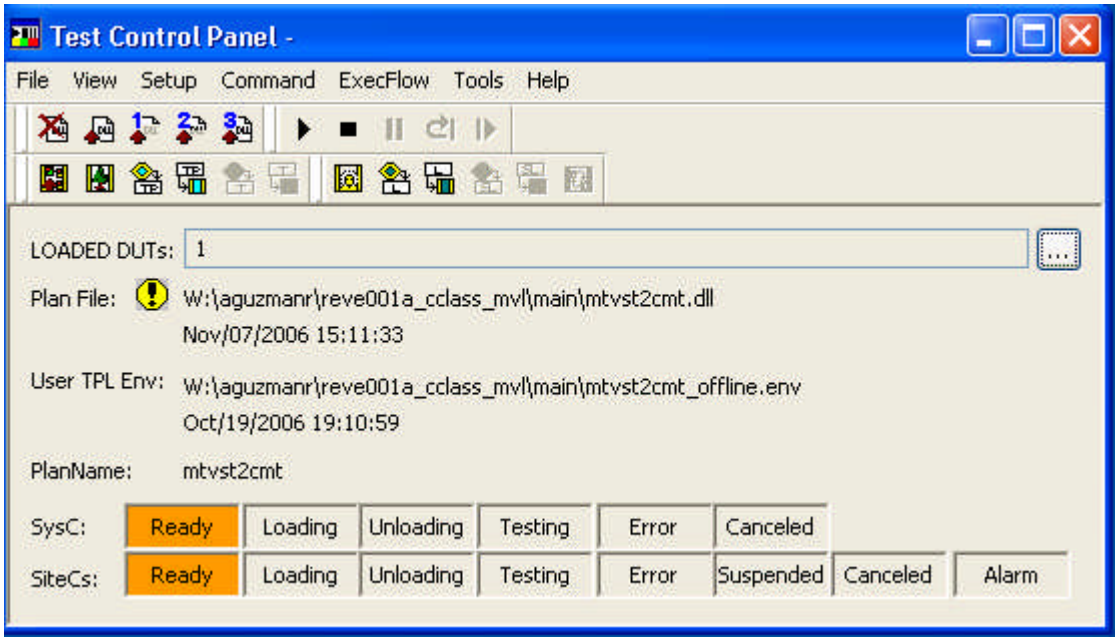

Figura 3-6 Panel de control de la interfaz gráfica del TSS

En la figura 3-6, se muestra el panel de control de la interfaz gráfica del TSS, desde el cual, el usuario tiene acceso a las herramientas de ingeniería que forman parte de la aplicación entre las cuales destacan: un editor de flujos, un editor de variables globales, la herramienta *shmoo*, entre otras, además, en el panel de control el usuario tiene acceso a botones que le permiten controlar la ejecución de las pruebas del programa de prueba e indicadores que le informan del estado de las mismas

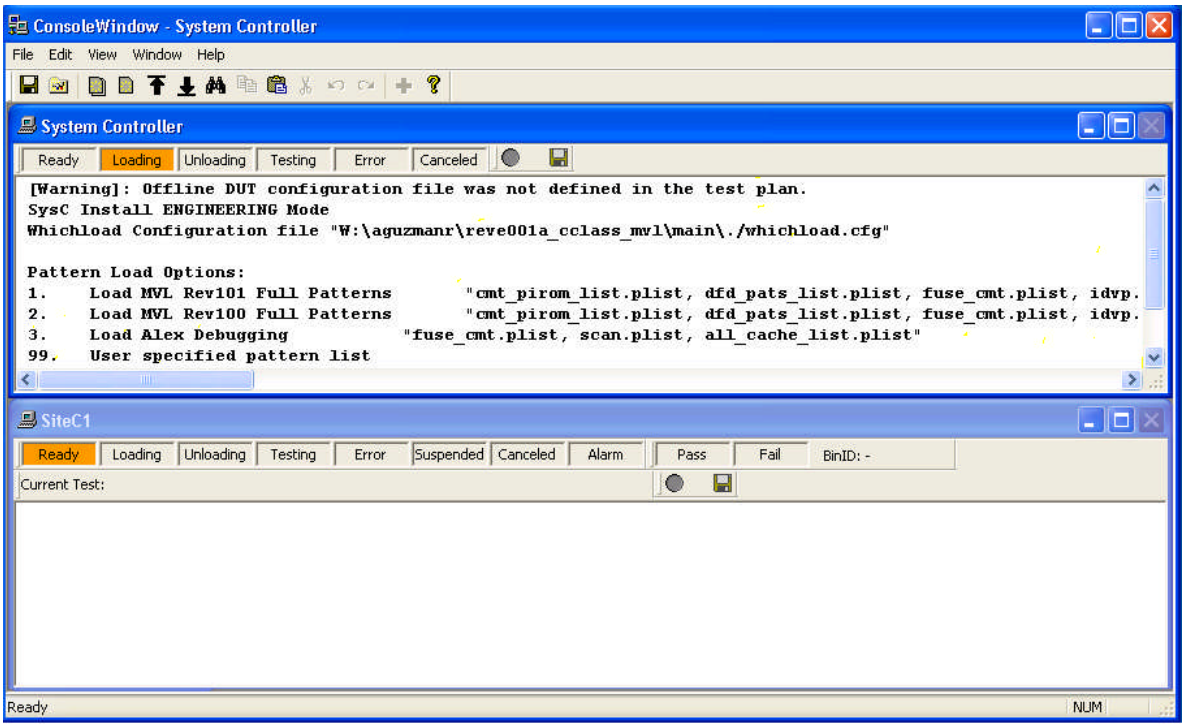

Figura 3-7 La consola donde se muestra la información del *System* y del *Site Controller*

Además del panel de control, el TSS cuenta con una consola que contiene la consola del *System Controller* y la consola del *Site Controller*, en la consola del *System Controller* el TSS despliega toda la información relacionada con el *System Controller* y en la consola del *Site Controller*, el TSS despliega toda la información del *Site Controller*, es decir, todos los resultados de las pruebas realizadas a la unidad que se esta probando.

La información que se despliega en la consola del *Site Controller* es de gran relevancia para la caracterización de las unidades y por ende para la validación de cual quiere falla que estas presentes, ya que esta contiene todos los errores que se presentan durante la ejecución de las pruebas.

3.2.3. Los *UserSDK Tools API*

Los Tools API representan el nivel más bajo de programación del Tester CMT disponible para un usuario, los cuales son implementados a nivel de clases COM (Component Object Model), estas instrucciones permiten controlar el funcionamiento de todos los módulos de hardware programables que constituyen la plataforma de prueba, por ende, permiten controlar por medio de instrucciones la gran parte de las funciones que el TSS permite a un usuario realizar de manera gráfica.

El COM es una arquitectura de programación independiente del lenguaje que se utiliza para implementar el programa, no es lenguaje de programación sino que es una especificación de programación que utiliza una librería de tiempo de ejecución para controlar la comunicación y el funcionamiento de los módulos de hardware [8].

Además, el COM permite un desarrollo orientado a objetos, lo cual implica una implementación basada en encapsulamiento, poliformismo y herencia [8], es decir, el COM encapsula los detalles de la implementación de los métodos vinculados con cada tipo de objeto, esto por medio de librerías dinámicas que se encargar de interpretar las instrucciones, esta característica, hace del COM una implementación idónea para proteger los derechos de autor, ya que le permite a un usuario utilizar una serie de instrucciones sin revelar la implementación de estos.

Al ser el COM independiente del lenguaje de programación, tiene la ventaja de poder ser utilizado en cualquier lenguaje de programación que permita una implementación orientada a objetos, tales como: VBS, C++, PERL, etc.

En la figura 3-8, se muestra el diagrama de bloques de *los UserSDK Tools API*, el cual, se aplica a cualquier COM, se puede observar que para establecer la comunicación entre el *System Controller* y el *Site Controller* del *tester CMT*, debe existir un objeto del COM en el *Site Controller* que preste el servicio y otro objeto en el *System Controller* que se actué como cliente.

Además, se utiliza un *Proxy* y un *Stub* para brindar el encapsulamiento requerido por el COM y el RPC (*Remote Procedure Calls*) para establecer y manejar la comunicación entre las dos computadoras.

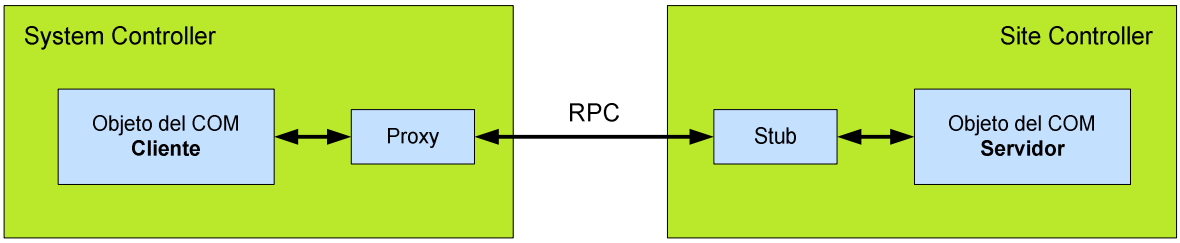

Figura 3-8 Diagrama de bloques del funcionamiento de los *UserSDK Tools API*

3.2.4. *Helper Class*

Al igual que los *Tools API, Helper Class* es un COM que permite controlar desde el *System Controller* las acciones que se realizan en el *Site Controller*, la diferencia radica en que *Helper Class* le facilita al usuario instrucciones de más alto nivel que los *UserSDK Tools API* [9], es decir, se puede visualizar a *Helper Class* como un COM que fue desarrollado por medio de la especificación de programación que facilitan los *Tools API's*.

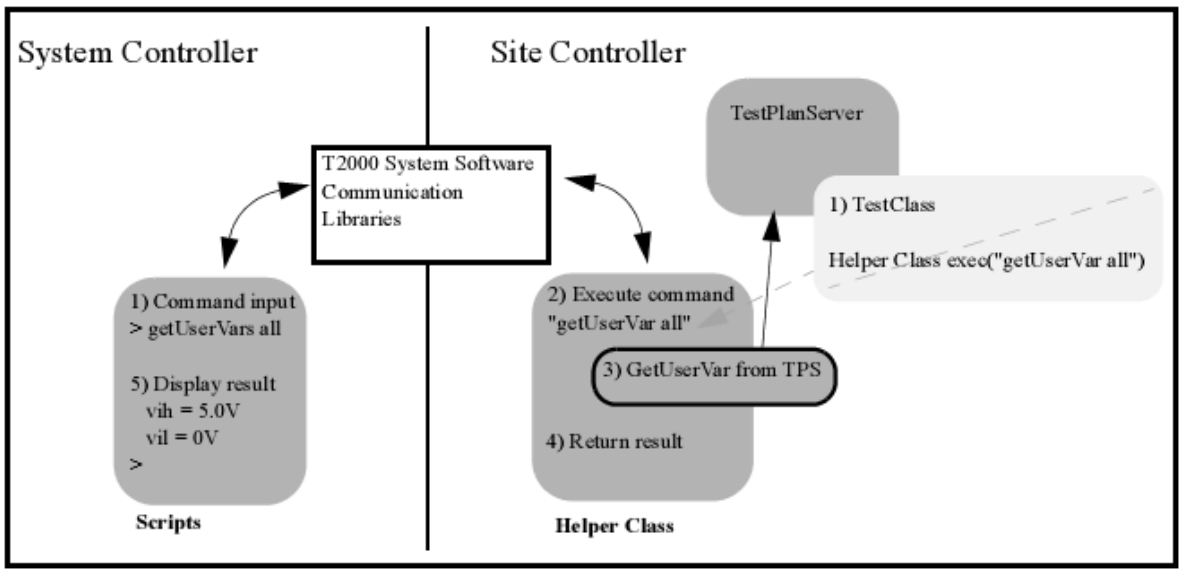

Figura 3-9 Diagrama del funcionamiento de *Helper Class* desde un *script* de ingeniería [9]

Por lo tanto, para poder utilizar las instrucciones de *Helper Class*, se requiere cargar una librería dinámica en el *Site Controller* que realiza la función de servidor y otra librería dinámica en el *Site Controller* que trabaje como el cliente. En la figura 3-9, se ejemplifica el funcionamiento de *Helper Class* desde un *scrip*t de ingeniería.

3.2.5. La herramienta *Shmoo*

Un *shmoo*, es un gráfico de dos o tres dimensiones que se utiliza para caracterizar el comportamiento de una unidad antes un conjunto específico de patrones de prueba, es decir, permite evaluar bajo cuales condiciones físicas la unidad presenta un comportamiento deseado.

En la figura 3-10, se muestra un *shmoo* de una prueba de SBFT, en la cual, se puede observar que el parámetro del eje X es *FSBper_spec*, el cual, representa la velocidad en el bus y el parámetro del eje Y es *vcore_spec*, el cual, representa la tensión en el núcleo del procesador.

Además, se puede observar que el gráfico esta constituido por símbolos y letras, los asteriscos (*) que se observan en la gráfica representa que para las condiciones de velocidad del bus y la tensión en el núcleo vinculadas con el punto todos los vectores del patrón de pruebas fueron ejecutados exitosamente, por el contrario, cada una de las letras que se observan en la gráfica presentan un vector y/o un pin específico o un conjunto de ellos del patrón de pruebas que presentó problemas bajo las condiciones de velocidad y la tensión relaciones con el punto de la gráfica.

De la gráfica se puede inferir que con forme aumenta la velocidad en el bus del procesador, se requiere mayor tensión en el núcleo para que el procesador puede ejecutar exitosamente el patrón de prueba.

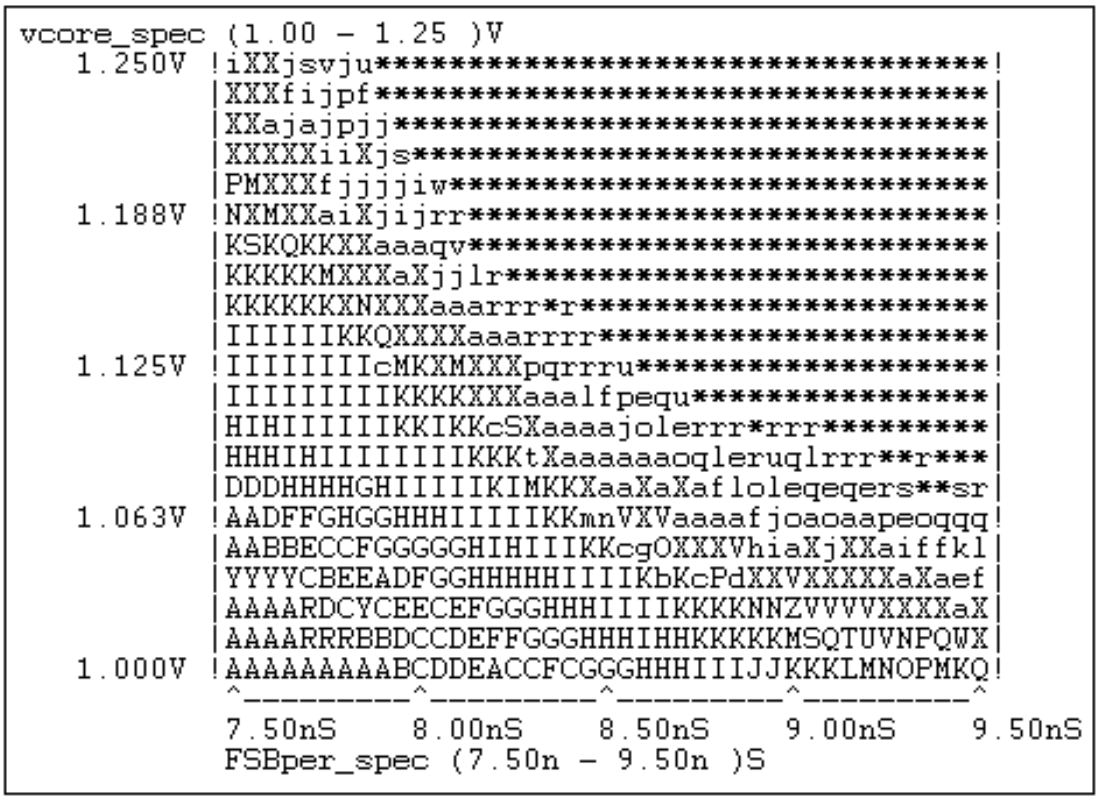

Figura 3-10 Un *shmoo* de una prueba de SBFT

3.2.6. La herramienta Full Scan

Al igual que la herramienta *shmoo*, la herramienta *FullScan*, ejecuta todos los vectores de los patrones de pruebas, la diferencia radica en que el *FullScan* ejecuta los patrones de prueba bajo las condiciones del punto de operación, es decir, utiliza un único valor de velocidad en el bus y tensión en el núcleo del procesador, sin embargo, esta herramienta presenta la ventaja que los resultados son más fáciles de interpretar ya que, en el *FullScan* se especifica el nombre del patrón de prueba presenta problemas.

En la figura 3-11, se muestra los resultados de la ejecución de un *FullScan* a la lista de patrones de prueba llamada *st2_chain_c0_narrow_list*, en la cual, se puede inferir que todos los patrones de prueba de la lista menos el *SP0_C0_rand_r1_for_conv* se ejecutaron exitosamente bajo las condiciones del punto de operación.

```
DUT ID : 1              Pattern list : st2_chain_c0_narrow_list         Burst # 255
+-------------------------------------------------------------------------------+
1 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------+
2 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------+
3 :  FAILED : SP0_C0_rand_r1_for_conv (default) Capture Count = 1  (MAX = 448)
+-------------------------------------------------------------------------------+
4 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------+
5 :  PASSED : SP1_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------+
6 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------+
7 :  PASSED : SP2_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------+
8 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------+
9 :  PASSED : SP3_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------+
10 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------+
11 :  PASSED : SP4_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------+
12 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------+
13 :  PASSED : SP5_E0_rand_r2_for_conv (default)
+-------------------------------------------------------------------------------+
14 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------+
15 :  PASSED : SP6_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------+
16 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------+
17 :  PASSED : SP7_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------+
18 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------+
19 :  PASSED : SP8_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------+
20 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------+
21 :  PASSED : SP9_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------+
22 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------+
23 :  PASSED : SP10_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------+
24 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------+
25 :  PASSED : SP11_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------+
PASSED Pattern count: 24
FAILED Pattern count: 1

================================
Total PASSED Pattern count: 24
Total FAILED Pattern count: 1
```

Figura 3-11 *FullScan* de la lista de patrones de prueba llamada *st2_chain_c0_narrow_list*

3.2.7. El *Handler*

Es una máquina que se encarga del manipular las unidades que se desean probar, es decir, se encarga de colocar las unidades en la TIU para que el *Tester CMT* pueda realizar las pruebas, también se encarga de clasificar las unidades según los resultados obtenidos de las pruebas realizadas, es decir, agrupa las unidades buenas y las unidades defectuosas.

Además, el *Handler*, se encarga de controlar la temperatura y la presión de las pruebas que se realizan, esta propiedad lo hace indispensable para poder realizar la validación de las unidades defectuosas ya que el comportamiento de un semiconductor se altera con los cambios de temperatura.

3.3. Descripción de proceso a mejorar

Actualmente, cuando una unidad no pasa todos las pruebas del programa de prueba, es necesario realizar una validación de la unidad, es decir, un ingeniero debe realizar pruebas adiciones para determinar porque la unidad no paso la prueba.

Por lo general, lo que ingeniero desea saber es si la falla que se presenta se debe a un problema en el equipo que se esta utilizando, en el programa de prueba que se esta corriendo o si es un problema propio de la unidad en prueba.

Para poder identificar el origen de la falla, el ingeniero debe consultar el archivo *Ituff* para determinar el tipo de bin por el cual la unidad que se desea validad fue clasificada con defectuosa, luego, con el bin de falla de la unidad el ingeniero puede determinar en cual prueba la unidad presentó problemas y con esto, puede iniciar el proceso de depuración de la falla.

Actualmente, el proceso de depuración de fallas para el producto Montecito esta muy orientado a la utilización de herramientas de ingeniería que se encuentran disponibles en el *Tester CMT*, ejemplo de estas herramientas son el *shmoo* o el *full scan*[17], las cuales pretenden dar al ingeniero una idea del origen de la falla, sin embargo, en algunas ocasiones los resultados obtenidos no revelan cual es el camino que se debe seguir, por lo tanto, el proceso de validación se convierte en un método de prueba y error.

Por otra parte, el proceso actual de depuración de fallas requiere de una participación activa del ingeniero, ya que, para definir la siguiente prueba que se debe realizar y las condiciones de esta, el ingeniero debe realizar una interpretación de los resultados obtenidos en las pruebas anteriores, lo cual, aumenta considerablemente el tiempo y los recursos requeridos para realizar la validación de las unidades.

Además, en la mayoría de los casos también se quiere de la participación de un operario que se encargue de ejecutar las órdenes de trabajo que el ingeniero define y del manejo de las unidades.

Esto se debe a que las órdenes de trabajo por lo general requieren de mucho tiempo de prueba y para no interrumpir con las labores cotidianas de los ingenieros ni perjudicar el tiempo de pruebas que ellos requieren, la mayoría de órdenes de trabajo se corren por la noche.

Si bien es cierto, existen varios tipos de diferentes de órdenes de trabajo, estas se pueden agrupar en los bloques principales, las órdenes de trabajo que son específicamente para la validación de unidades defectuosas y los experimentos de ingeniería, los cuales, varían según los interés y necesidades del ingeniero.

Por este motivo, se pretende que la herramienta que este proyecto pretende desarrollar, facilite al usuario un conjunto de instrucciones que él pueda utilizar para definir el procedimiento o metodología del experimento que desea realizar.

[17] Full Scan: una prueba para determinar en cuales pattern la unidad presenta problemas

3.4. Conceptos básicos de una maquina virtual

Cuando las primeras computadoras fueron desarrolladas, el hardware fue diseñado primero y software especializado fue escrito para ese hardware después. Cada sistema fue esencialmente hecho con su propio conjunto de instrucciones, y el software fue desarrollado para ese conjunto específico de instrucciones. Esto incluye el software del sistema operativo, ensambladores (luego compiladores), y programas de aplicación [10].

Con forme aumentó la cantidad de usuarios, la complejidad de los sistemas operativos y el numero de aplicación, la técnica de reescribir y distribuir software para cada nueva computadora se volvió una carga intolerable

Por lo tanto, los sistemas operativos (OS) fueron desarrollados de forma que separaran las aplicaciones de las especificaciones de hardware del sistema. Los OS son responsables de manejar los recursos de hardware, proteger las aplicaciones en ejecución y datos de usuario, proveer soporte de distribución y otras funciones útiles.

Así mismo el desarrollo de interfaces estándares para permitir que el software sea independiente del hardware ha brindado un poderoso ambiente de desarrollo que permite la expansión de complejas y extensas aplicaciones.

Las limitaciones en el software están presentes solamente porque la plataforma en la cual el software se ejecuta se asume como una entidad física con ciertas propiedades, esto es corre un OS específico y tiene cierto hardware específico, en resumen, el software corre en un tipo específico de plataforma [10].

Las máquinas virtuales eliminan esta restricción y permiten un grado más alto de portabilidad y flexibilidad. Una máquina virtual (VM) es implementada agregando software a una plataforma de ejecución para darle a esta la apariencia de una plataforma distinta, o más bien, dar la apariencia de múltiples plataformas. Una máquina virtual puede tener un OS, conjunto de instrucciones, o ambos que difieren de aquellos implementados en el hardware real [10].

3.4.1. Virtualización de una maquina virtual

Los problemas que pretende eliminar una VM, son resueltos implementando una capa de software que provee un ambiente de máquina virtual en el cual el software es ejecutado. En una máquina virtual se tiene una capa de software que hace que el hardware del sistema sea visto por la aplicación de forma distinta a como este realmente es, de forma que el hardware con el que trabaja la aplicación "parece ser" uno que en realidad es diferente al que realmente se tiene, lo que permite convertir las distintas plataformas de hardware en una sola plataforma estándar la cual será vista por la aplicación. Este proceso de virtualización requiere lo siguiente:

- El mapeo de recursos virtuales, esto es registros y memoria, a los recursos del hardware real.
- Utilizar instrucciones del hardware real para ejecutar las acciones especificadas por las instrucciones de la máquina virtual, esto es emulación del hardware.

La virtualización del software puede ser aplicada en distintas maneras para conectar y adaptar los tres componentes principales de un sistema. La emulación brinda una flexibilidad considerable permitiendo mezclar y emparejar la portabilidad de software de distintas plataformas. La virtualización del software puede realizar la emulación por medio de la optimización, tomando información específica de la implementación en consideración mientras realiza la emulación, o bien es posible realizar solamente la optimización. La virtualización del software puede proveer además replicación, por ejemplo dándole a una única plataforma de hardware la apariencia de múltiples plataformas, cada una capaz de correr un completo OS o bien un conjunto de aplicaciones. Finalmente los distintos tipos de VM pueden ser unidos para formar una amplia variedad de arquitecturas [10].

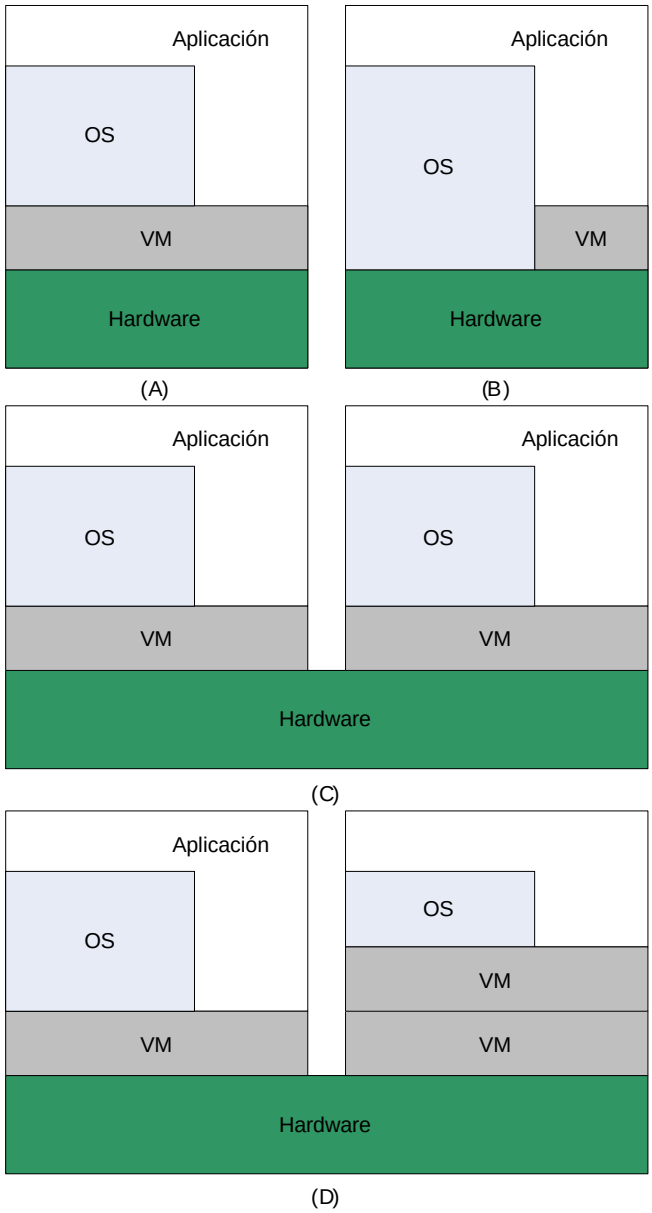

Figura 3-12 (a) Emulando un conjunto de instrucciones con otro. **(b)** Optimizando una aplicación binaria existente para el mismo conjunto de instrucciones. **(c)** Extendiendo una VM de forma que múltiples *OSs* pueden ser soportados al mismo tiempo. (d) Integrando múltiples *VMs* para crear estructuras mas complicadas.

3.4.2. Perspectivas de una maquina virtual

En los sistemas computacionales, con sus muchos niveles de abstracción, el significado de máquina es una cuestión de perspectiva. Desde la perspectiva de un proceso, la máquina consiste en un espacio de direccionamiento de memoria que ha sido asignado al proceso. El sistema de I/O, como es percibido por el proceso, es más bien abstracto. Discos y otros medios secundarios de almacenamiento aparecen como una colección de archivos a los cuales el proceso tiene permiso de acceso. En el ambiente de escritorio el proceso puede interactuar con un usuario a través de una ventana que es creada dentro de un ambiente de interfaz de usuario (GUI). La única forma en la cual el proceso puede interactuar con el sistema de I/O es por medio de llamadas al OS, ya sea directa o indirectamente a través de librería que son dedicadas al proceso.

Desde una perspectiva de más alto nivel, un sistema completo es apoyado en una máquina inferior. Un sistema es un sistema completo de ejecución que puede soportar simultáneamente un número de procesos distintos creados por distintos usuarios. Todos los procesos comparten un *filesystem* y recursos de I/O. El sistema persiste (con etapas de reinicio ocasionales) aun cuando los procesos terminan y se ejecutan otros nuevos. El sistema brinda memoria física y recursos de I/O a los procesos, y permite que los procesos interactúen con sus recursos por medio del OS que es parte del sistema.

Así como existen una perspectiva de proceso y una perspectiva de sistema para una máquina, esta misma visión puede ser extendida a las VM. Como el nombre sugiere una máquina virtual de procesos (PVM) es capaz de soportar procesos individuales. Una PVM es creada cuando el proceso se crea y termina cuando el proceso termina de ejecutar. Por ejemplo si se toma una plataforma que consiste en un sistema Linux en una maquina Intel X86, esta puede claramente soportar aplicaciones nativas para el X86 compiladas para un sistema Linux. Sin embargo el sistema puede además procesos de Java con máquinas virtuales de Java. Igualmente se pueden ejecutar otros procesos en el mismo ambiente, acceder a los recursos y compartir información entre ellos.

Un sistema de maquina virtual (SVM) provee un completo ambiente de sistema. Este ambiente puede proveer soporte para múltiples procesos, incluyendo el *filesystem*, acceso a dispositivos de I/O y si es del caso un GUI.

3.4.3. Interprete de instrucciones

Un intérprete es un tipo de programa de traducción el cual convierte lenguaje de alto nivel en código máquina o en un lenguaje de más bajo nivel. Una vez que es traducida una instrucción, entonces el código resultante es ejecutado, luego se traduce la siguiente instrucción, se ejecuta y el proceso continúa hasta terminar el programa. La característica principal es que este nunca traslada el programa de forma completa como si lo hace un compilador [11].

El tiempo que toma ejecutar un programa bajo un interprete es mayor que correr el código compilado, pero toma menos tiempo interpretar el código y ejecutarlo que el tiempo total que requiere compilarlo y luego ejecutarlo. Esto es muy importante especialmente cuando se encuentra en la fase de prototipo y prueba del código, cuando el ciclo de interpretación, edición y corrección del código puede ser más corto que el ciclo de compilación, edición y corrección.

El código interpretado es más lento que el código compilado, esto debido a que el intérprete debe de analizar cada sentencia en el programa cada vez que esta es ejecutada y entonces realizar la acción deseada mientras que el código compilado solamente realiza la acción. El acceso a las variables es también más lento por el interprete debido a que el mapeo de los identificadores de la posiciones de memoria debe ser hecho de forma repetitiva durante la ejecución del código a diferencia del código compilado [11].

3.5. Teoría de lenguajes de programación

Los lenguajes de programación entran dentro de un tema muy amplio llamado herramientas de software. Debido a que existen muchos lenguajes y tipos de lenguajes, es necesario primero estudiar que es lo que se busca en un lenguaje de programación. Existe un conjunto de características que son deseables en un lenguaje de programación [12]:

- *Simplicidad*, deberían de haber tan pocos conceptos como sea posible. Regularmente el trabajo del diseñador del lenguaje es el de descartar elementos que sean superfluos, difíciles de leer o difíciles de compilar.
- *Uniformidad*, este conecto básico debería ser aplicado consistentemente y universalmente. Deberíamos ser capaces de utilizar las características de un lenguaje en diferentes contextos sin cambiar la forma. La no uniformidad puede resultar muy molesta, por ejemplo en Pascal no es posible asignar valores devueltos por expresiones a una variable.
- *Ortogonalidad*, funciones independientes deberían de ser controladas por mecanismos independientes.
- *Abstracción*, debería haber una forma de esconder la implementación de la aplicación del lenguaje, la abstracción significa esconder los detalles de construcción.
- *Claridad*, los mecanismos deben de ser bien definidos, y la salida del código debe de ser fácilmente predecible.
- *Modularidad*, las interfaces entre las unidades de programación deberían ser declaradas explícitamente.
- *Eficiencia*, el lenguaje debe ser capaz de producir código eficiente.
- *Seguro*, los errores de semántica deben de ser fácilmente detectables, preferiblemente en el momento de la compilación.

3.5.1. Lenguajes interpretados

Lenguajes interpretados como es el caso de *Perl, Pitón, Tcl* y otros son un estilo muy diferente de programación que los lenguajes de programación de sistema. Los lenguajes interpretados asumen que ya existe una colección de componentes utilizables escritos en otros lenguajes. Los lenguajes interpretados no son concebidos con la idea de escribir aplicaciones completas, si no mas bien para juntar un conjunto de componentes mas poderosos escritos en otros lenguajes de programación [13]. Por ejemplo *Tcl* y *Visual Basic* fueron concebidos principalmente para agrupar colecciones de interfaces de usuario en la pantalla, mientras que *Unix Shell* esta dedicado a ensamblar programas de filtrado y *pipeline*. Los lenguajes interpretados son utilizados además para extender las capacidades de los componentes que deben unir, pero no para escribir algoritmos complejos o estructuras de datos.

3.6. Descripción de los principales principios de software y electrónicos relacionados con la solución del problema.

3.6.1. Principios de software

Todas las pruebas que se realizan a una unidad pueden ser programadas en C++, el cual es un lenguaje de programación orientado a objetos y se utiliza Microsoft Visual Studio.NET para realizar la compilación de esas funciones.

Sin embargo, la programación en C++ es solo una parte de un gran proceso, ya que se debe tener presente que la programación que se realiza esta orientada a controlar los módulos de hardware programables del *Tester CMT*.

Por este motivo, se deben crear archivos donde se especifique la temporización de las señales, los niveles de tensión para cada uno de los pines de entrada salida, la ubicación física de cada señal en la cabeza de pruebas, la frecuencia de la del pruebas, el valor de las salidas de potencia del *Tester CMT* entre otras cosas.

Luego, se debe traducir toda esta información al seudo lenguaje que interpreta el *Tester CMT*, para esto se utiliza *Avator* que es una herramienta que se encarga de traducir toda la información requerida para hacer las pruebas al lenguaje que el *Tester CMT* puede interpretar.

Además, en algunas ocasiones es necesario utilizar PERL, el cual, es un lenguaje de programación que facilita la interpretación, creación y modificación de archivos de texto.

Otra herramienta de programación que puede ser de gran utiliza para el proyecto son las instrucciones de *Helper Class*, las cuales permiten desde una consola tipo DOS tener control sobre las pruebas que se realizan en el *Tester CMT*, es decir, permiten modificar las condiciones y ejecutar una prueba particular, además, dan la posibilidad de ejecutar las herramientas de ingeniería disponibles en *Tester CM*T por ejemplo del *Shmoo*. También, existen las instrucciones de *UserSDK Tools API* que representan el lenguaje de más bajo nivel de programación del CMT disponible para un usuario.

3.6.2. Principios electrónicos

Si bien es cierto, todas las pruebas que se realizan a una unidad son programadas, para poder realizar la programación de las pruebas se debe conocer las características eléctricas y funcionales de la unidad.

Se debe conocer cuales son los parámetros de operación del dispositivo, específicamente: los niveles de tensión de operación de los pines de entrada salida [3], es decir, cuales valores son interpretados como un 1 o un 0 lógico, además es importante conocer la temporización necesaria para que el dispositivos pueda comprender una secuencia de instrucciones y sobre todo se debe conocer cual debe ser la respuesta del dispositivo ante una secuencia de instrucciones en particular [5].

Es muy importante tener conocimiento de cómo se ven afectados los parámetros eléctricos de un dispositivo ante diferentes condiciones, específicamente si se aumenta o disminuye la temperatura, la tensión de alimentación o la frecuencia de operación, ya que esto es de suma importancia para realizar la caracterización de las unidades y por ende para la validación de las fallas en estas.

Además, se debe tener un alto conocimiento de arquitectura de microprocesadores para poder comprende la metodología de las pruebas de: Caché, Scan y SBFT y la lógica involucrada en la validación de dichas pruebas.

Capítulo 4: Procedimiento metodológico

4.1. Reconocimiento y definición del problema

A continuación, se presentan las principales actividades realizadas para reconocer, comprender y definir el problema:

- Se realizó una reunión inicial en la cual, el asesor de la empresa se refirió a los aspectos generales del proyecto, es decir, realizó una breve descripción del procedimiento de validación actual, además, el asesor se refirió a la importancia de la validación y las desventajas que el procedimiento presenta
- Se llegó el curso en línea de Digital Test Fundamentals, en el cual, se especifican los principales conceptos necesarios para comprender la metodología de prueba y el funcionamiento de una plataforma de prueba
- Se realizaron reuniones semanales con el asesor de la empresa para obtener más detalles sobre los alcances y características que se desea de la herramienta que se pretende desarrollar
- Se observó el procedimiento de validación que realiza un ingeniero, con el fin de facilitar la comprensión del problema que se busca solucionar.
- Una entrevista con Daniel Viquez para tener más conocimiento sobre la metodología de validación existente para el producto Montecito

4.2. Obtención y análisis de información

Como el proyecto esta constituido por una investigación para definir una metodología de validación y por la de implementación de esta, se realizaron las siguientes actividades para obtener y analizar la información necesaria:

- Se tuvo una semana de capacitación en el departamento para obtener todos los conocimientos técnicos y de programación básicos necesarios para comprender el funcionamiento del *Tester CMT* y del *Handler*, los cuales, constituyen el equipo necesario para realizar el proyecto
- Se realizó un estudio de la metodología de prueba definida por Intel para el producto Montecito, ya que este fue el punto de partida para el desarrollo de la metodología para Montvale
- Se realizaron entrevistas a los ingenieros encargados de realizar el proceso actual de validación de las fallas de: Caché, SBFT y Scan, para comprender conceptos básicos de las pruebas y la metodología de validación de las fallas en ellas
- Se realizaron pruebas básicas de depuración en el test CMT para familiarizarse con el funcionamiento del equipo y poder tener una noción de las herramientas disponibles
- Se realizó una investigación sobre las instrucciones de *Helper Class*
- Se reviso la documentación sobre los lenguajes de programación de PERL y C++
- Se realizaron reuniones de manera periódica con el Ingeniero Gilbert Figueroa, el cual, me enseño a utilizar las instrucciones de *UserSDK Tools API* y conceptos fundamentales para desarrollar el proyecto
- Se realizó una reunión semanal con el asesor en la empresa para determinar el camino a seguir a partir de la interpretación de la información.

4.3. Evaluación de las alternativas y síntesis de una solución

Se realizaron pruebas con *Helper Class* para familiarizarse con las instrucciones disponibles y sobre todo para determinar con es que estos se ejecutan en el CMT, lo cual, implicó interpretar el *script* de Visual Basic que se encargaba brindar una interfaz de consola entre el usuario y *Helper Class*.

Una vez que se comprendió, el funcionamiento del *script* de Visual Basic, se contemplo la posibilidad de traducir todas las funciones de dicho *script* al lenguaje de PERL, ya que este lenguaje de programación permite manipular fácilmente archivos de texto, lo cual, es un punto alto en la implementación de la solución.

Se realizaron pruebas para determinar si era posible establecer una comunicación tipo apretón de manos entre el *script* de Visual Basic y el *script* de PERL, es decir, se contemplo la posibilidad de modificar la entrada por defecto del *script* de Visual Basic, actualmente, el teclado del usuario, para que las instrucciones de *Helper Class* se recibieran del *script* de PERL y de esta manera poder ejecutar desde PERL las instrucciones de *Helper Class* sin necesitar de realizar en PERL todo el procedimiento de inicialización.

Posteriormente, se realizaron las entrevistas a los ingenieros encargados del desarrollo y la validación de las pruebas de Caché, SBFT y Scan, esto con le fin de determinar cuales funciones se deben desarrollar para implementar el procedimiento de validación de las fallas que se presenta en dichas pruebas.

Se planteó una solución inicial y se evaluó el grado de control del CMT que esta permitía, además, se contemplo que tan robusta esta era y que tal fácil sería realizar modificación en un futuro a la herramienta que se esta desarrollando, por que recordemos que el proyecto es de final abierto, y el desarrollo del mismo constituye la etapa inicial de un proyecto de mayor envergadura.

Por último, se investigo sobre los *UserSDK Tools API*, el nivel de control sobre el CMT que estos permiten, la forma que en se utilizan y la posibilidad de ejecutar dichas instrucciones desde un *script* en PERL

4.4. Implementación de la solución

Para implementar la primera solución, fue necesario investigar sobre comunicación entre diferentes programas, específicamente, redireccionamiento de la entrada por defecto y la salida por defecto de las aplicaciones, es decir, modificar el *script* de Visual Basic que Intel utilizaba para ejecutar *Helper Class* para que este no recibiera las instrucciones desde un teclado, sino para que las reciba de otro *script* en PERL.

Una vez que se implementó dicha solución, se realizaron pruebas para verificar su correcto funcionamiento y robustez, además, se realizan entrevistas a ingenieros con experiencia en el desarrolla de herramientas de automatización en el CMT, como Gilbert Figueroa y José Chico, los cuales apartaron sugerencias muy importante para modificar la solución actual y aumentar en nivel de control de la herramienta sobre el CMT.

Estos dos ingenieros criticaron el grado de dificulta que la implementación inicial presentada para futuras modificación, ya que la dependencia que el script de PERL tenía del *script* de Visual Basic, limitaba significativamente el potencial de PERL para implementar funciones más complejas, además, sugirieron utilizar los *UserSDK Tools API* para aumentar el nivel de control de la herramienta sobre el *Tester CMT*.

La solución final, implicó, traducir todo el proceso de inicialización de *Helper Class* que el *script* de Visual Basic realizaba, al lenguaje PERL, lo cual significó, aprender como se realiza una programación a nivel de objetos en PERL, para lo cual, se contó con el apoyo y guía de Gilbert Figueroa.

Además, fue necesario aprender a utilizar las instrucciones de *UserSDK Tools API* en PERL, para lo cual, las enseñanzas de Gilbert fueron de gran importancia porque al final se comprendió que *Helper Class* era un conjunto de librerías dinámicas que requieren de las instrucciones de *UserSDK Tools API* para poder trabajar.

Es importante destacar, que dicha implementación fue mucho más lenta que la inicial ya que el nivel de la programación aumento considerablemente, esto debido a que los *UserSDK Tools API* representan el más bajo nivel de programación en el CMT y la documentación sobre su utilización es limitada y escueta.

4.5. Reevaluación y rediseño

Como ya se ha mencionado, con las instrucciones de *Helper Class* se obtiene un control muy limitado de la ejecución de las pruebas en el *Tester CMT*, además, se tiene la dificultad de que no es posible acceder a todos los resultados de la ejecución de las pruebas que se imprimen en la consola del *Site Controller* de la interfaz gráfica.

Durante el desarrollo del proyecto, se contemplo la posibilidad de capturar estos resultados mediante las instrucciones de *UserSDK Tools API*, y esta fue la razón de mayor peso para modificar la solución inicial, sin embargo, después de realizar varias consultas a los ingenieros que desarrollar el *Tester CM*T y la interfaz gráfica del mismo, se llego a la conclusión de que actualmente con las instrucciones de *UserSDK Tools API* disponibles para el usuario eso no era posible.

Otra razón muy importante que motivó la modificación de la solución original fue el aporte de Gilbert Figueroa sobre el alto nivel de complejidad que la herramienta presentaba para implementar funciones más complejas y por ende se dificultaba la transferencia de conocimiento y la reutilización de la misma.

Capítulo 5: Descripción detallada de la solución

En este capítulo, se detalla el proceso que se realizó para implementar la herramienta que permitió dar solución al problema planteado, es decir, se comentan los principales detalles del trabajo realizado, entre los que destacan: la definición de una metodología de validación para las pruebas de Caché, Scan y SBFT y la implementación de la herramienta que permite la automatización de las órdenes de trabajo.

Antes de iniciar con la descripción de la solución implementada es necesario ubicar al lector en el entorno del proyecto, específicamente, el punto de partida del desarrollo de la solución.

Existes dos maneras en que un ingeniero puede realizar sus experimentos o validaciones en el CMT, la primera, es utilizando la interfaz gráfica del *System Controller*, esta implica una participación activa del ingeniero en el desarrollo de las pruebas, la otra posibilidad, es utilizar un *script* de Visual Basic para correr instrucciones de *Helper Class*, esta opción permite un nivel muy básico de automatización en la ejecución de las pruebas, por lo tanto, este fue el punto de partida del desarrollo del proyecto.

En la figura 5-1, se muestra la relación entre el usuario y las instrucciones de *Helper Class*, básicamente, se utiliza un *script* de Visual Basic para leer, interpretar y ejecutar en el CMT las instrucciones de *Helper Class* que el usuario define en una consola similar a la de DOS que es parte del *script* de Visual Basic.

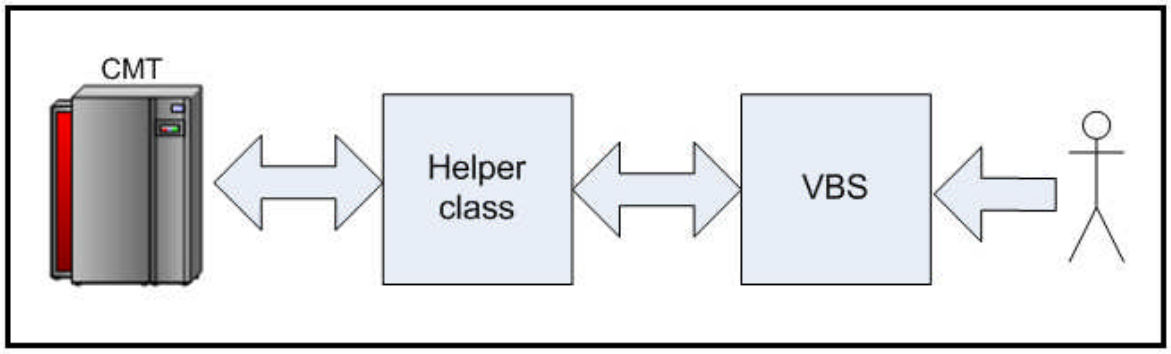

Figura 5-1 Diagrama de bloques del funcionamiento de *Helper Class*

5.1. Análisis de soluciones y selección final

5.1.1. Solución Inicial

En la figura 5-2, se muestra el diagrama de bloques de la solución inicial, la cual, consiste en modificar el *script* de Visual Basic para que la entrada por defecto de este concuerde con la salida por defecto del *script* en PERL y de esta manera logar la comunicación entre ambas aplicación.

La razón principal, por la que se definió esta solución inicial fue su rapidez de implementación, esto debido a que no era necesario preocuparse por la interpretación y ejecución de las instrucciones de *Helper Class* por que eso era responsabilidad del *script* en Visual Basic, y el problema se resumiría a la interpretación de los archivos de entrada usando PERL, el almacenamiento de los resultados y el desarrollo de una lógica que permitiera la comunicación entre PERL y Visual Basic.

A continuación, se detalla la lógica utilizada para lograr la comunicación entre Visual Basic y PERL.

Como el ambiente Windows no permite que dos aplicaciones corran de manera simultanea utilizando un mismo medio para comunicarse, fue necesario hacer uso de una implementación amo-esclavo, es decir, cada vez que el *script* de Visual Basic necesite una instrucción de *Helper Class*, este debe ejecutar el *script* de PERL, el cual, se encarga de análisis la información de los archivos de entrada (orden de trabajo y instrucciones) para definir la siguiente instrucción de la secuencia de ejecución.

Además, la salida por defecto del *script* en PERL concuerda con la entrada por defecto del *script* en Visual Basic, por lo tanto, cada vez que se imprima información desde PERL esta será captura por el *script* de Visual Basic y de esta manera se logra la comunicación de apretón de manos deseada.

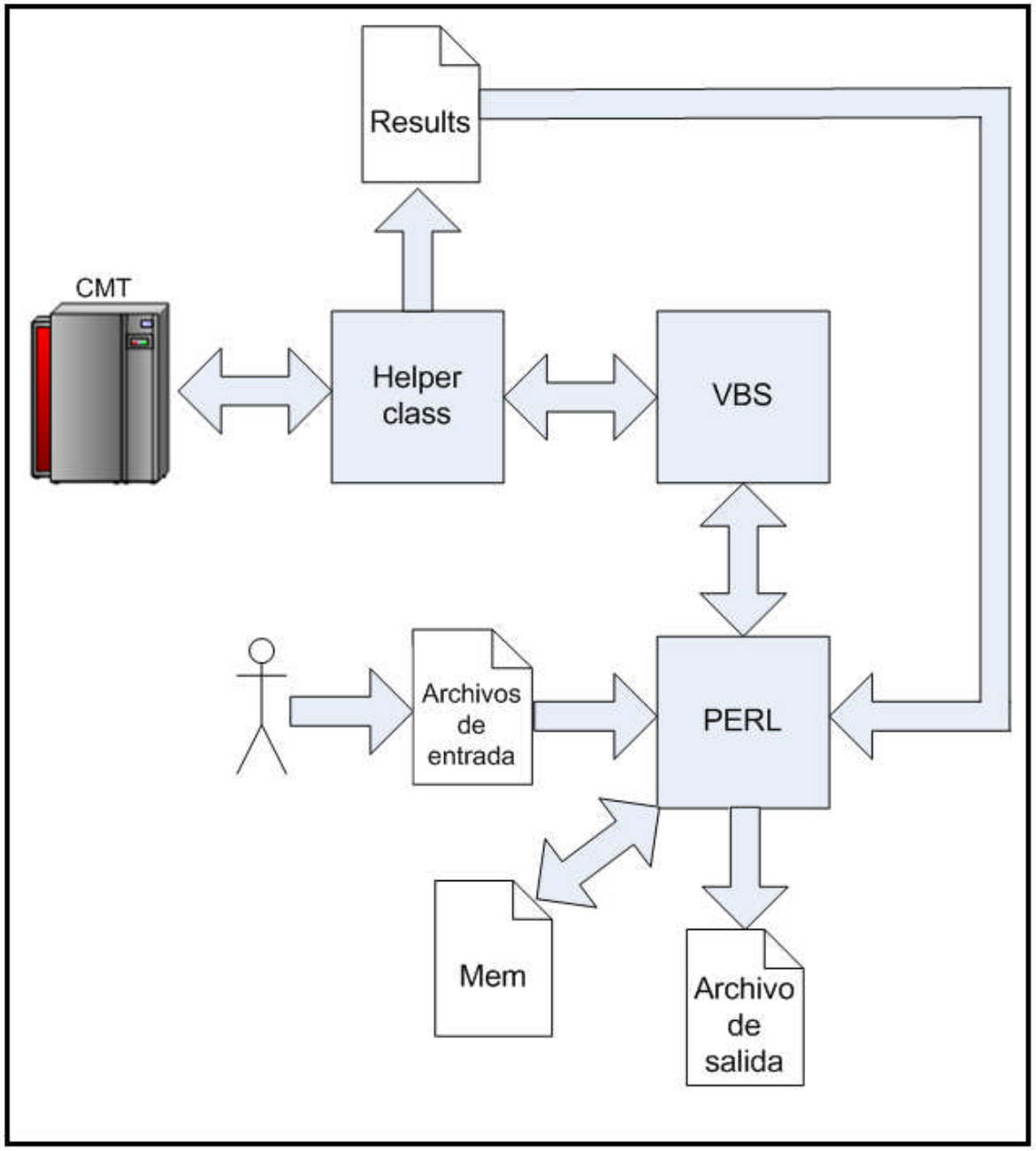

Figura 5-2 Diagrama de bloques de la solución inicial

Sin embargo, se presenta el problema de que las dos aplicaciones no pueden estar trabajando al mismo tiempo, por que el *script* de PERL debe terminar su ejecución justo después de haber transferido la siguiente instrucción de la secuencia de ejecución al *script* de Visual Basic, por lo tanto, la próxima que el *script* de Visual Basic ejecute el *script* de PERL, este último no conocería su estado anterior, es decir, no sabría cual fue la última instrucción que se ejecuto en el CMT.

Por este motivo, fue necesario utilizar un archivo de memoria para el *script* de PERL y de esta manera resolver el problema de la perdida de la información del estado anterior, entonces, antes de terminar su ejecución, el script de PERL almacena en dicho archivo la información de la instrucción que se va ha correr y antes de iniciar una nueva ejecución, lee del archivo de memoria la información de la ejecución anterior.

Esta solución fue implementada y se demostró su correcto funcionamiento, sin embargo, presentaba algunos problemas: el nivel de automatización no era el deseado, la lógica de la implementación dificultaba el desarrollo de funciones más complejas, el desempeño de la aplicación sería mayor si solo se utilizara un único lenguaje de programación ya que no se perdería tiempo en la comunicación entre ambos, además, el ingeniero Gilbert Figueroa sugirió, utilizar los *UserSDK Tools API* para aumentar el nivel de automatización de la herramienta, el desempeño y la robustez de la misma.

5.1.2. Solución Final

La solución final corresponde a una versión mejorada de la solución inicial, básicamente, las modificaciones realizadas a la solución inicial son: la incorporación de las instrucciones *UserSDK Tools API* para aumentar el nivel de control sobre CMT, el número de funciones o instrucciones que la herramienta puede interpretar y ejecutar y por ende aumentar el nivel de automatización de la herramienta.

Además, se omitió el uso del *script* de Visual Basic, lo cual, implicó traducir todo la secuencia de inicialización de *Helper Class* que se ese realizaba, con lo cual, se obtuvo un control centralizada y por ende una mayor flexibilidad para futuras mejoras, ya que la lógica de control y ejecución de las pruebas en el CMT que utiliza el *script* de PERL se simplificó considerablemente. En la figura 5-3, se muestra el diagrama de la solución final.

Sin embargo, hubo una función que no fue posible implementar, la captura de todos los resultados que se imprimen en la consola del *Site Controller*, cuando el CMT ejecuta una prueba en la consola del *Site Controller* se imprime los resultados obtenidos con dicha ejecución, los cuales, contiene información muy importante para la caracterización y la validación, por este motivo, se desea que la herramienta fuera capaz de capturarla automáticamente.

Se realizaron varas consultas a los ingenieros de *Advantest*, la empresa que en un trabajo conjunto con Intel desarrollo la plataforma de prueba CMT, para estudiar la posibilidad de lograr automatizar la captura de dichos resultados, sin embargo, después de varias semanas de pruebas se llegó a la conclusión que con las instrucciones de *UserSDK Tools API* disponibles para el usuario no fue posible lograr dicho cometido, para lograrlo, se debía utilizar funciones ocultas que son para uso exclusivo de la empresa *Advantest*.

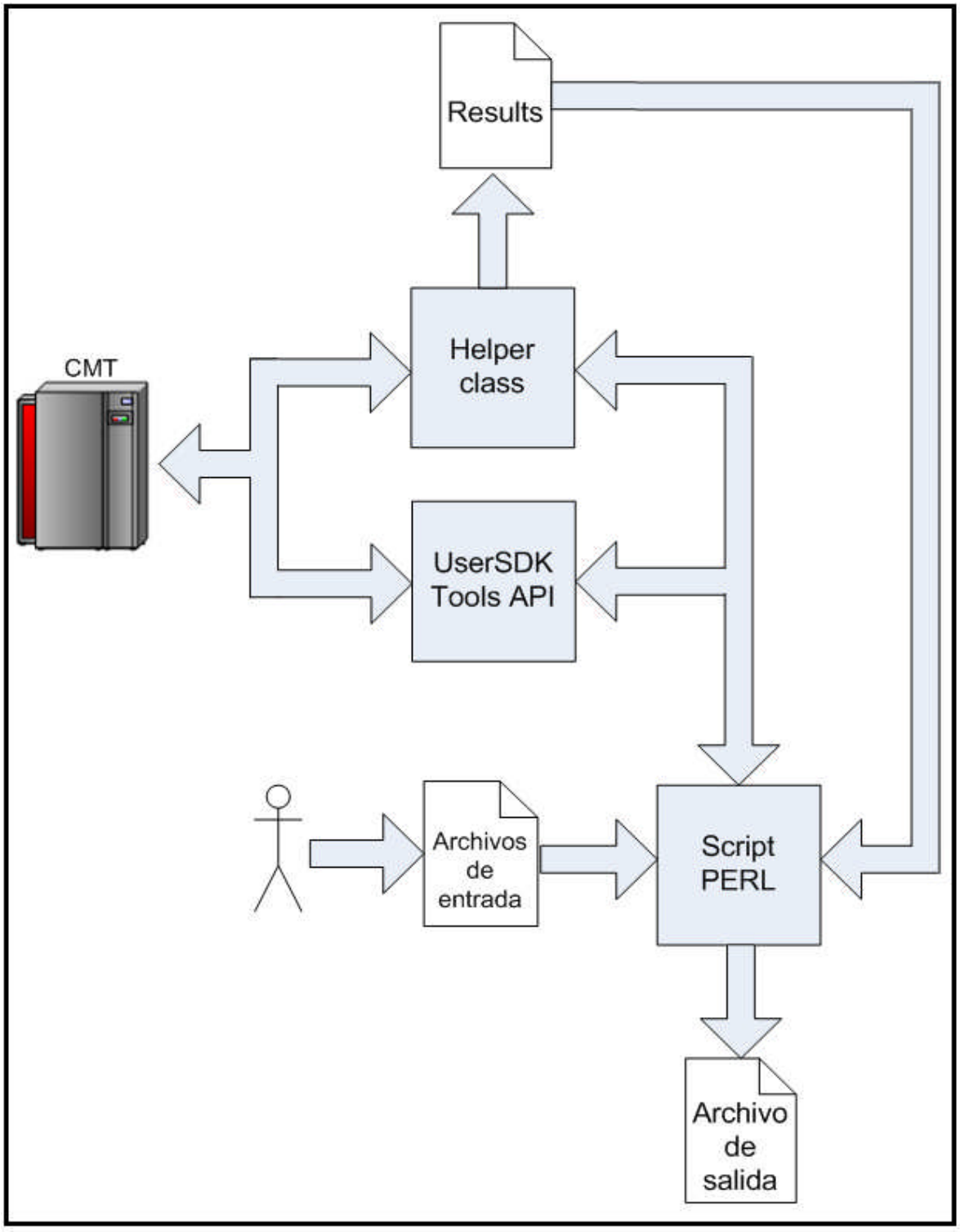

Figura 5-3 Diagrama de bloques de la solución final

A continuación, se realiza una descripción del funcionamiento de la herramienta, para que el lector tenga un panorama más claro de los alcances del trabajo realizado.

5.1.2.1. Los Archivos de entrada

Los archivos de entrada le permiten al usuario definir el procedimiento de validación o la secuencia de instrucciones que constituyen el experimento de ingeniería para cada unidad, seguidamente, se detalla sobre el formato y utilización de los cinco archivos de entrada de la herramienta desarrollada.

- *El archivo de configuración*

En el archivo de configuración, el usuario debe especificar la dirección donde se encuentran los archivos de: orden de trabajo (*WO_File*), instrucciones (*Command_File*), el archivo de etiquetas (*Labels_File*) y la ubicación donde desea que la herramienta almacene los resultados de la orden de trabajo (*Results_File*).

Además, en este archivo el usuario debe especificar la dirección IP del *Site Controller* (*SiteC1_IP*), esto para que la herramienta pueda hacer una lectura a través de la red para obtener los resultados de las pruebas.

Igualmente, el usuario tiene la posibilidad de especificar si desea que la herramienta se encargue del manejo de las unidades (*Unit_Handling* = 1), en cuyo caso, la aplicación trabaja con todas las unidades en el *handler*, o puede definir que trabaje con solo una unidad (*Unit_Handling* = 0). En la Tabla 1, se observa el formato del archivo de configuración.

Tabla 1 Formato del archivo de configuración

SiteC1_IP =	192.168.0.11
WO_File =	Z:\aguzmanr\Archivos_Conf\WO_File.txt
Command_File =	Z:\aguzmanr\Archivos_Conf\Commands.txt
Labels_File =	Z:\aguzmanr\Archivos_Conf\Labels.txt
Results_File =	Z:\aguzmanr\Result_Alex.txt
Unit_Handling =	0

- *El archivo de orden de trabajo*

En el archivo de orden de trabajo el usuario puede definir el procedimiento que desea que la herramienta realice a cada unidad de manera automática, para la cual, puede utilizar el Visual ID o el ULT para identificar la unidad.

En este archivo, el usuario puede definir para cada unidad, una validación o un experimento de ingeniería, en caso, de que desee realizar una validación, debe especificar un *SoftBin*[18], de lo contrario, debe especificar una etiqueta para vincular una metodología en el archivo de instrucciones y una etiqueta para vincular una prueba o un conjunto de pruebas en el archivos de etiquetas, tal cual, se puede observar en la Tabla 2.

Tabla 2 Formato del archivo de orden de trabajo

Visual ID	ULT	Methodology	TEST	SOFTBIN
2L633914Z0126	-	_MetoAlex3_	_AlexTest_	-
-	W6356328_351_-01_+02	_MetoAlex2_	_Test1_	-
2L640Y00Z0661	-	-	-	b41_06
All	-	_MetoAlex3_	_Test1_	-

- *El archivo de instrucciones*

En el archivo de instrucciones, el usuario puede especificar las metodologías a utilizar, es decir, puede utilizar las instrucciones de usuario para realizar la programación del procedimiento que desea que se realice como un experimento de ingeniería para cada unidad en prueba.

Cada procedimiento debe iniciar con el nombre de la etiqueta que el usuario quiere utilizar para identificarla, toda etiqueta debe iniciar y concluir con guión bajo (_), además, la definición de la metodología debe concluir con la etiqueta de *_end_,* tal cual, se observa en la figura 5-4.

[18] SoftBin: un número único que se utiliza para identificar un tipo de falla y las pruebas vinculadas con dicha falla.

```
_MetoAlex1_
        shmoo (_Alex1_)
_end_

_MetoAlex2_
        runFlow (SBFT_COMP)
        runForceFlow(SBFT_COMP,1)
        shmoo (_Alex1_)
_end_

_MetoAlex3_
        shmoo (_Alex1_)
        shmoo (FSBper_spec,5nS, 10nS,10,vcore_spec,0.6V,1.2V, 10,normal)
_end_

_Pruebas1_

        BypassTest (Xiu_check,OFF)
        ChangeTestParam (Xiu_check,debug_mode, OBNOXIOUS)
        ChangeUserVar (ieu_core1, "Alexander")
        runTest ()
        runFlow (FUSE)
        runForceFlow (FUSE,1)
_end_
```

Figura 5-4 Formato del archivo de instrucciones

- *El archivo de validación*

El archivo de validación, tiene el mismo formato del archivo de instrucciones, tal cual se observa en la figura 5-5, sin embargo, la diferencia entre ambos radica en que el archivo de validación se utiliza para definir las metodologías de validación y el archivo de instrucciones es utiliza para definir los procedimientos de los experimentos de ingeniería.

```
_b41_
        runFlow (Scan_comp)
        FullScan ()
        shmoo (FSBper_spec,5nS, 10nS,5,vcore_spec,0.6V,1.2V, 5,normal)

_end_
```

Figura 5-5 El archivo de validación

- *El archivo de etiquetas*

En el archivo de etiquetas, el usuario puede definir un vincular valores que con una etiqueta la cual puede ser utilizada para identificar una prueba o un conjunto de pruebas en el archivo de orden de trabajo, también, fue utilizar una etiqueta para definir la configuración de los parámetros de la instrucción, el cual se utiliza en los archivos de instrucciones y validación.

Lo antes expuesto, se puede ver claramente en la figura 5-6, la idea de estas etiquetas es facilitar la programación del usuario, ya que realiza la definición una única vez y puede utilizar dicha definición cuantas veces este desee.

```
# All the label names must start and end with (_),also they can't have spaces between caracters
# Use (=) to specify the values related with with label name
# The values specification must end with (;) and separated by (,)


# Format Shmoo Setup Label
# _LabelName_ = X,Xmin,Xmax,Xstep,Y,Ymin,Ymax,Ystep,Mode;
# The Mode can be [normal | transition | all_fails | alpg_fails | max_info]
        _Alex1_ = FSBper_spec, 5nS,10nS,3,vcore_spec,0.6V, 1.2V,2,all_fails;


# Format Test Names Label
# _LabelName_ = All Test Names separate by (,) and end with (;)
        _AlexTest_ = l2dw6c1_1000_nom1200_nom875,l2dw3c1_1000_nom1200_nom875;
        _Test2_ = l2dw6c1_1000_nom1200_nom875,l2dw3c1_1000_nom1200_nom875;
        _Test1_ = iscandualcore_100_min1200_min1025_10TCK;
```

Figura 5-6 Formato del archivo de etiquetas

5.2. Metodología de validación

En esta sección se describe los principales conceptos de la metodología de prueba que se utiliza para las pruebas de SBFT, Scan y Caché, esto con el fin de informar y ubicar al lector sobre el propósito de dichas pruebas y establecer un punto de partida para la definición de la metodología de validación para las fallas que se presentan es esas pruebas.

5.2.1. Pruebas de SBFT

SBFT son pruebas funcionales basadas en una metodología Estructural, es decir, son pruebas que permiten valorar el estado de la lógica interna del procesador a partir del análisis de los resultados de las pruebas que se realizan en una *plataforma de prueba* [14].

Básicamente, lo que se realiza es desarrollar un programa en ensamblador que requiera la intervención de los módulos internos del procesador que se desean valorar, por ejemplo, si se desea verificar el correcto funcionamiento de la unidad punto flotante entonces se realiza una rutina en ensamblador que mueva los operando de memoria hacia los registro de entrada de la unidad punto flotante, que realice una operación matemática y que coloque el resultado obtenido en una posición especifica de memoria, esto con el fin de poder comparar el resultado obtenido con el resultado esperado y de esa manera poder determinar el correcto funcionamiento de la unidad punto flotante.

Sin embargo, para poder realizar el procedimiento antes descrito, se debe emular en una plataforma de pruebas el funcionamiento normal del procesador, es decir, en una plataforma de pruebas no se cuenta con una memoria externa ni se puede hacer uso del bus de datos externo del procesador, lo cual, impide el funcionamiento normal del procesador ya que en teoría la información de buteo, el código de la rutina a ejecutar y los operandos de esta deben estar en la memoria externa, tal cual, se observa en el diagrama de la figura 5-7.

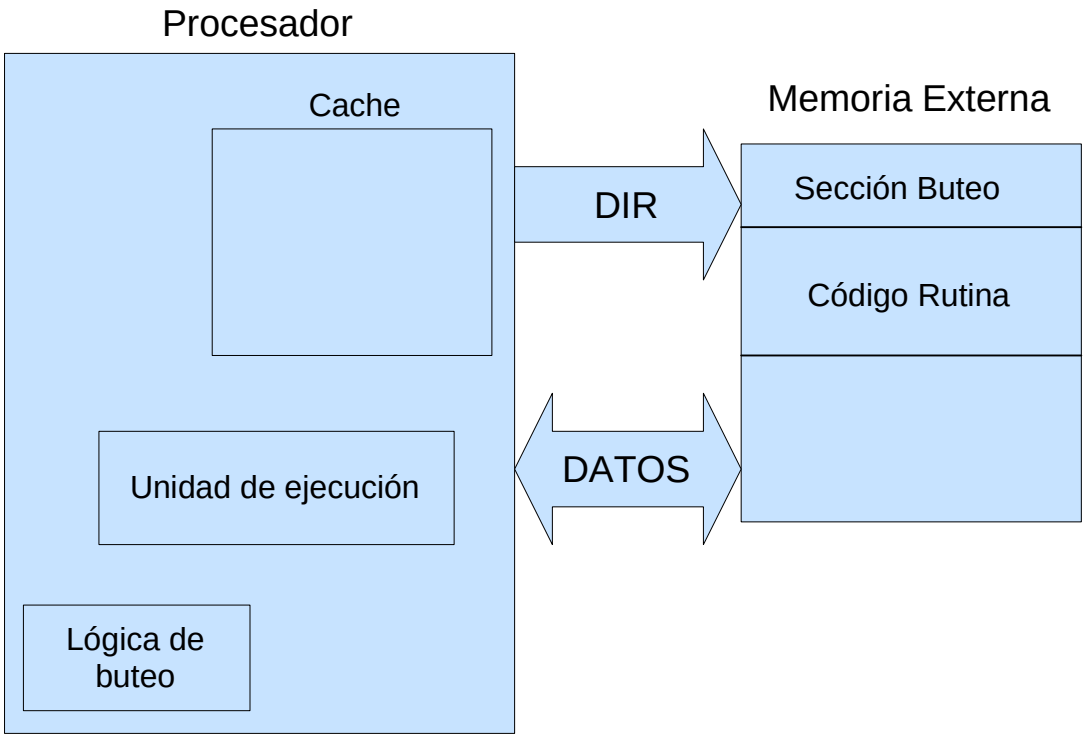

Figura 5-7 Operación normal del procesador

Para poder realizar las pruebas de SBFT, se debe modificar la lógica de buteo del procesador para que este no busque las instrucciones de inicialización en la memoria externa sino que realice la búsqueda en la memoria caché, además, se debe cargar en la memoria caché la rutina de la prueba de SBFT que se desea ejecutar, a dicha rutina se le llama *patrón de prueba* [14], además, se debe garantizar que todos los accesos a memoria que el procesador realice para leer una instrucción y los operandos de esta se debe hacer a la memoria caché, tal cual se observa en la figura 5-8.

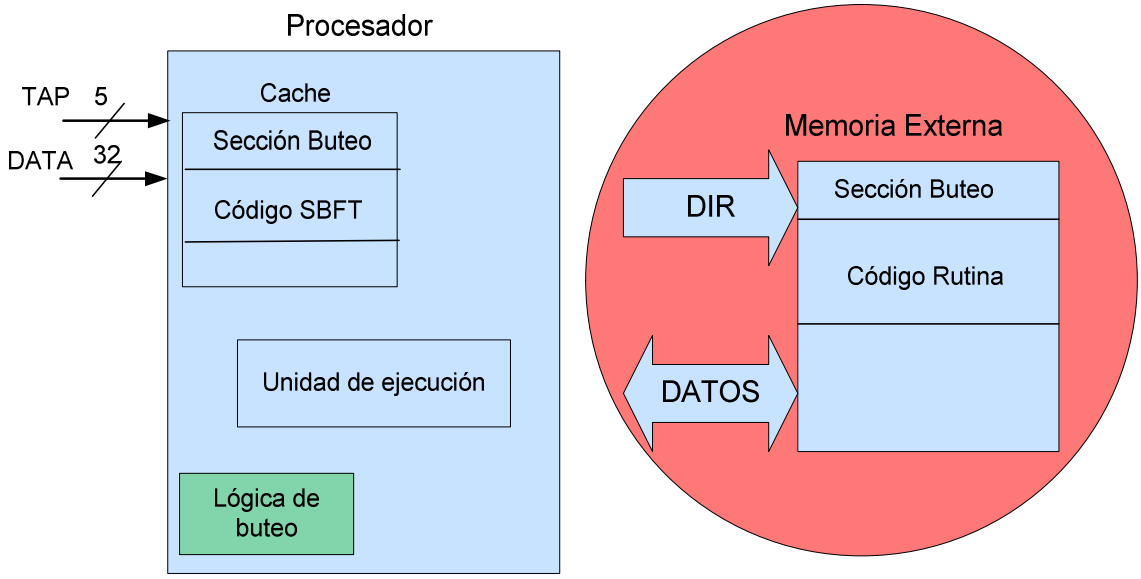

Figura 5-8 Diagrama de bloques del funcionamiento del procesador en una prueba de SBFT

Falta únicamente definir la manera como se introducen los datos a la memoria caché desde la plataforma de prueba, para esto se utilizan el protocolo de comunicación serie TAP (*Test Access Port*), el cual, se encarga de habilitar y conectar las 32 líneas de DATA que se utilizan para transferir el patrón de prueba a la memoria caché, tal cual, se puede observar en la figura 2.

Además, es importante mencionar que para el caso específico de Montvale, las pruebas de SBFT requieren de 6 arreglos de la memoria caché por núcleo para poder cargar todo el contenido requerido para ejecutar el o los patrones de prueba, cada arreglo tiene una capacidad de 1MB y por núcleo hay disponibles 12 arreglos, por ende la caché de Montvale tiene un tamaño de 24MB.

Una vez realizada una breve descripción de la metodología de prueba de SBFT, procederemos a definir la metodología de validación. El *HardBin* 44 es el tipo de falla relacionada con las pruebas de SBFT, cuando se presente este tipo de falla la herramienta desarrollada debe realizar el siguiente procedimiento:

1. Correr el flujo de *Fuses* para determinar cuales arreglos de la memoria caché trabajan correctamente.
2. Correr el flujo de Caché L2D para verificar el correcto funcionamiento de los arreglos de caché que fueron considerados buenos con la lectura de los fusibles que se realizó en el paso anterior, esto con el fin, de descartar algún error en la quema de los fusibles que se realizara en pruebas pasadas o en etapas anteriores del proceso de producción.
3. Utilizar el *SoftBin* para identificar la prueba dentro del flujo de SBFT que fallo.
4. Correr un *Shmoo* de la prueba que falló, el eje X del *shmoo* debe ser el parámetro *FSBper_spec* con un valor mínimo de 3.5ns, un máximo de 12ns e incrementos de 250ps, además, el eje Y debe ser el parámetro *vcore_spec* con u valor mínimo de 800mV, un máximo de 1.4V e incrementos de 25mV.

5.2.2. Pruebas de Scan

La metodología de prueba de Scan permite verificar el correcto funcionamiento de módulos internos del procesador que de otra manera fueran imposibles de probar o que se requeriría de una metodología de prueba muy compleja y de mucho tiempo de prueba.

Las pruebas de Scan requieren de la sustitución de los *latches* y *flip-flops* normales que se encuentran dentro del procesador por un versión de ISCAN *latches o flip-flops*, tal cual, se puede observar en la figura 5-9, es decir, se requiere que el dispositivo ha probar tenga una constitución particular y especifica que permita la implementación de las pruebas [15].

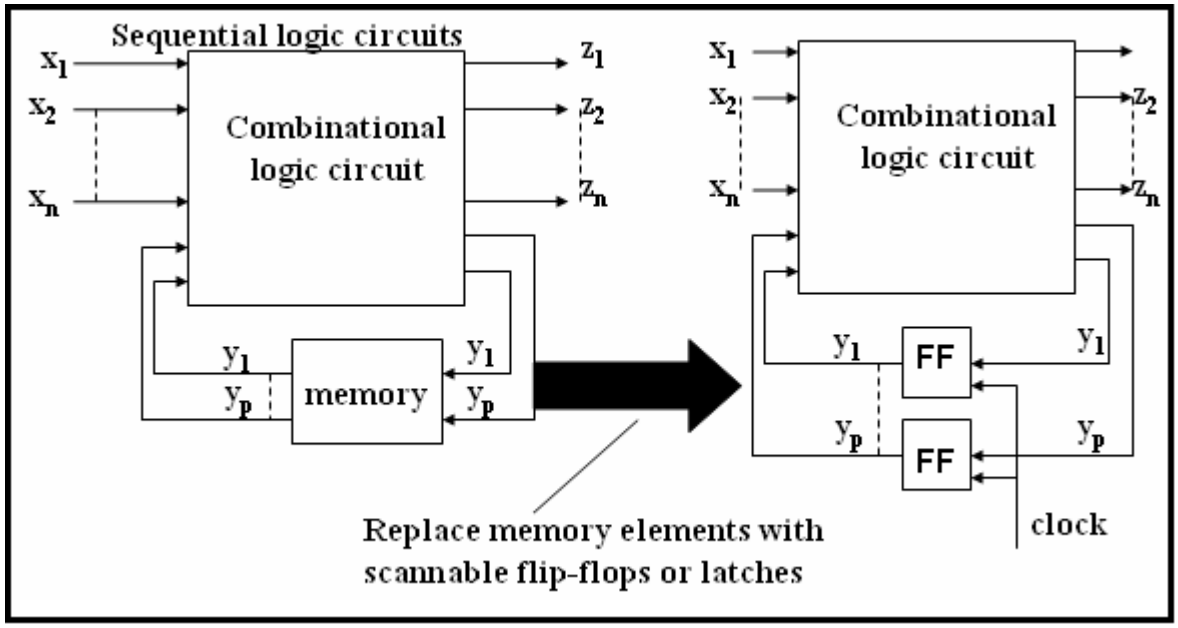

Figura 5-9 Diagrama de la lógica involucrada en las pruebas de Scan [15]

Con la sustitución de los elementos de memoria (*laches* y *flip flop*) por los elementos de ISCAN, se obtiene un circuito combinacional, lo cual, implica la transformación de una metodología de prueba secuencial en una metodología combinacional, lo cual, aumenta el nivel de controlabilidad y observalidad de los módulos de hardware internos del procesador involucrados en las pruebas de Scan.

Las pruebas de Scan se fundamentan en una metodología de prueba en cadena, en la figura 5-10, se muestra el algoritmo que se implementa para estas pruebas, el cual, consiste en paralizar el reloj para realizar la carga de todos los bits de la cadena de prueba en la lógica a probar, esta carga se realiza por medio del protocolo de comunicación serie TAP (*Test Access Port*).

Luego, los bits de la cadena de prueba se propagan a través de la lógica del módulo que se esta probando, se obtiene el resultado y se compara con un patrón de resultados establecidos para terminar el funcionamiento del módulo, por último, se activa el reloj y se descarga la cadena de Scan de la lógica en prueba, tal cual, se observa en la figura 5-10.

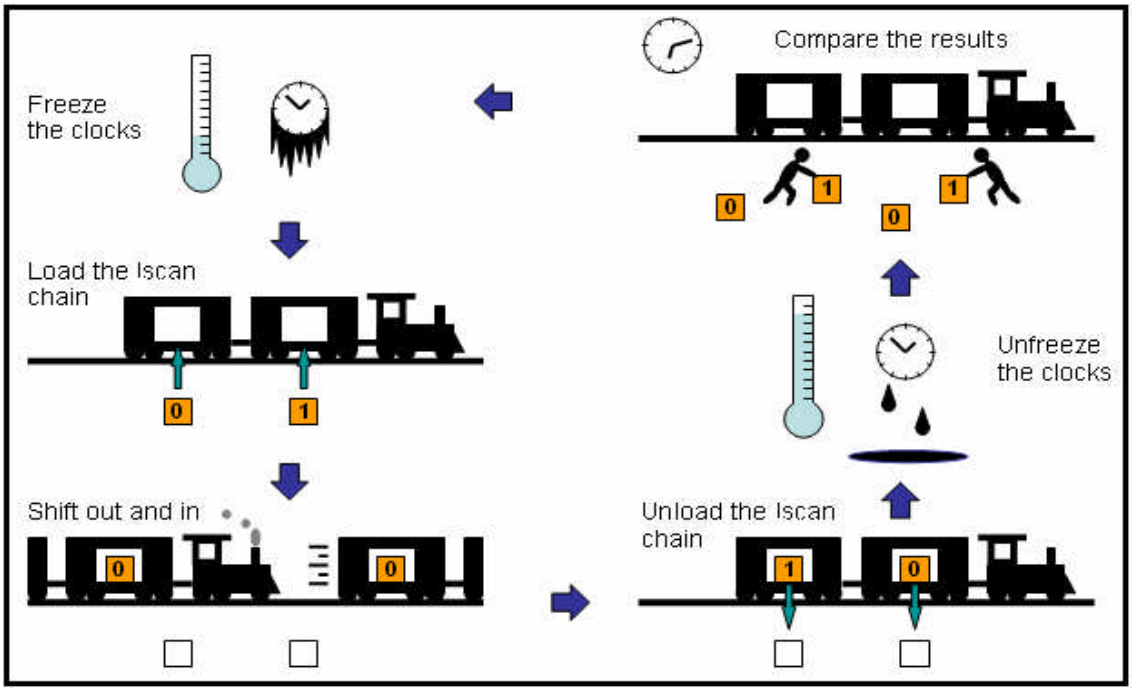

Figura 5-10 Diagrama del algoritmo que se emplea en las pruebas de Scan [15]

Una vez realizada una breve descripción de la metodología de prueba de Scan, procederemos a definir la metodología de validación. Los *HardBines* 41 y 42 son los tipos de fallas relacionadas con las pruebas de Scan, cuando se presente estos tipos de fallas la herramienta debe realizar el siguiente procedimiento:

1. Se debe correr el flujo *Vcc* y *Shops* para descartar problemas de alimentación y continuidad en la plataforma de prueba.
2. Se debe ejecutar el flujo de Scan para verificar que la unidad realmente presenta la falla y que esta no se presentó por un problema en el CMT, el TIU o el programa de prueba.
3. Utilizar el *SoftBin* para identificar la prueba dentro del flujo de Scan que fallo.
4. Por último, se debe correr un *Shmoo* de la prueba que falló, el eje X del *shmoo* debe ser el parámetro *FSBper_spec* con un valor mínimo de 5ns, un máximo de 15ns e incrementos de 250ps, además, el eje Y debe ser el parámetro *vcore_spec* con u valor mínimo de 800mV, un máximo de 1.4V e incrementos de 25mV.

5.2.3. Pruebas de Caché

El término caché hace referencia a todos los arreglos internos de memoria del procesador, los cuales, son más rápidos que los arreglos de memoria externos como la memoria RAM y ROM ya que la velocidad de las memorias depende de la distancias entre ellas y el procesador [16]. La memoria caché tiene la función de reducir los estados de espera del procesador, es decir, aumentar la velocidad de procesamiento de las instrucciones.

Existen tres niveles de memoria caché, cada uno tiene características y realizan funciones específicas, estos niveles dependen de la velocidad de los arreglos de memoria y por ende de la proximidad de estos al procesador, en la figura 5-11, se muestra la distribución de los tres niveles de caché para el procesador Pentium 4, desde el punto de vista de ingeniería y del punto de vista de mercado.

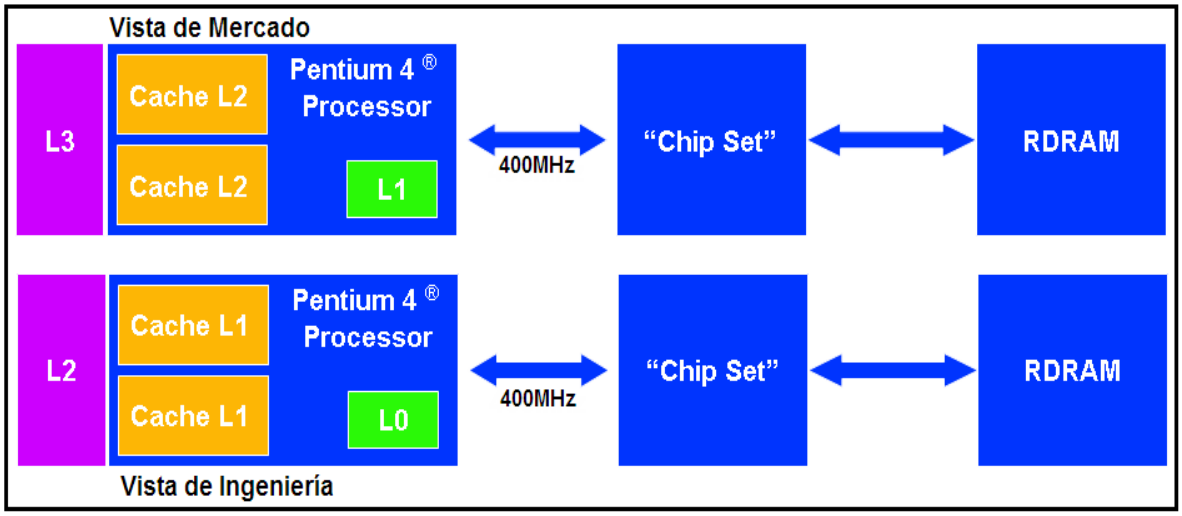

Figura 5-11 Los tres niveles de caché desde un punto de vista de ingeniería y de mercado [17]

Para sistemas de alto desempeño que realicen operación de punto flotante y operaciones de matemáticas de alta complejidad tales como CAD, CAE, simulación, animación entre otras requieren de un alto desempeño del procesador y por ende un alto consumo e utilización de los recursos de la caché [17].

Los sistemas como servidores y estaciones de trabajo requieren de procesadores de alto desempeño y de una memoria caché con gran capacidad de almacenamiento, ya que de esta manera pueden ejecutar una mayor cantidad de procesos de manera simultanea.

La descripción de los tres niveles de caché se realizará desde el punto de vista de ingeniería:

- *El nivel L0*

Este nivel de memoria esta constituido por SRAM y se encuentra implantado junto al núcleo del procesador, por lo tanto, es el nivel de memoria más rápido y de menor tamaño de los tres [16], aproximadamente cada uno tiene una capacidad entre los 8KB y los 32KB.

El nivel L0 se conoce también como UL0 y se divide en dos áreas: un área para instrucciones (trace) y otra para datos (ALG0) [17], además, los arreglos del nivel L0 son irregulares, es decir, no todos tiene la misma capacidad de almacenamiento, por lo tanto, se utiliza un direccionamiento específico.

- *El nivel L1*

Este nivel de memoria esta constituido por SRAM y se encuentra integrado con el procesador, tiene una mayor capacidad de almacenamiento que el nivel L0 (128KB-1MB), sin embargo, se encuentra a mayor distancia del núcleo del procesador, lo cual, la hace más lenta que la memoria L0.

Este nivel de memoria se conoce también con UL1 y se comunica con el procesador por medio del bus de memoria de este, al igual que la caché L0, los arreglos de la caché L1 son irregulares, por lo tanto, se debe utilizar direccionamiento específico.

- *El nivel L2*

Este nivel de memoria esa integrado al unidad de control del procesador, tiene la mayor capacidad de almacenamiento de los tres niveles (1MB – 24MB) y su capacidad de almacenamiento se esta incrementado considerablemente, por otra parte, es la memoria caché que se encuentra más lejos del núcleo del procesador por lo tanto, es la más lenta de las tres [17].

Normalmente, se utiliza en procesadores para servidores o procesadores para computadoras personales de alto desempeño, en el caso de Montvale, la caché L2 tiene 12 arreglos por núcleo de 1MB cada uno, tal cual, se puede observar en la figura 5-12, por lo tanto, la capacidad de almacenamiento máximo de la caché L2 de Montvale es 24MB

Sin embargo, se presentar ocasiones en las que 1 o mas arreglos de la memoria caché L2 no operan correctamente, en cuyo caso, se refiere a este como un arreglo defectuoso, tal cual, se muestra en la figura 5-12; el objetivo principal de las pruebas que se realizan a los arreglos de la memoria caché es determinar cuales de ellos operan correctamente y cuales no, ya que se debe garantizar el correcto funcionamiento y desempeño del procesador.

La diferencia entre los arreglos de la caché L2, es básicamente el direccionamiento que se debe realizar para acceder a cada uno de ellos, sin embargo, todos tiene un tamaño y distribución regular [16].

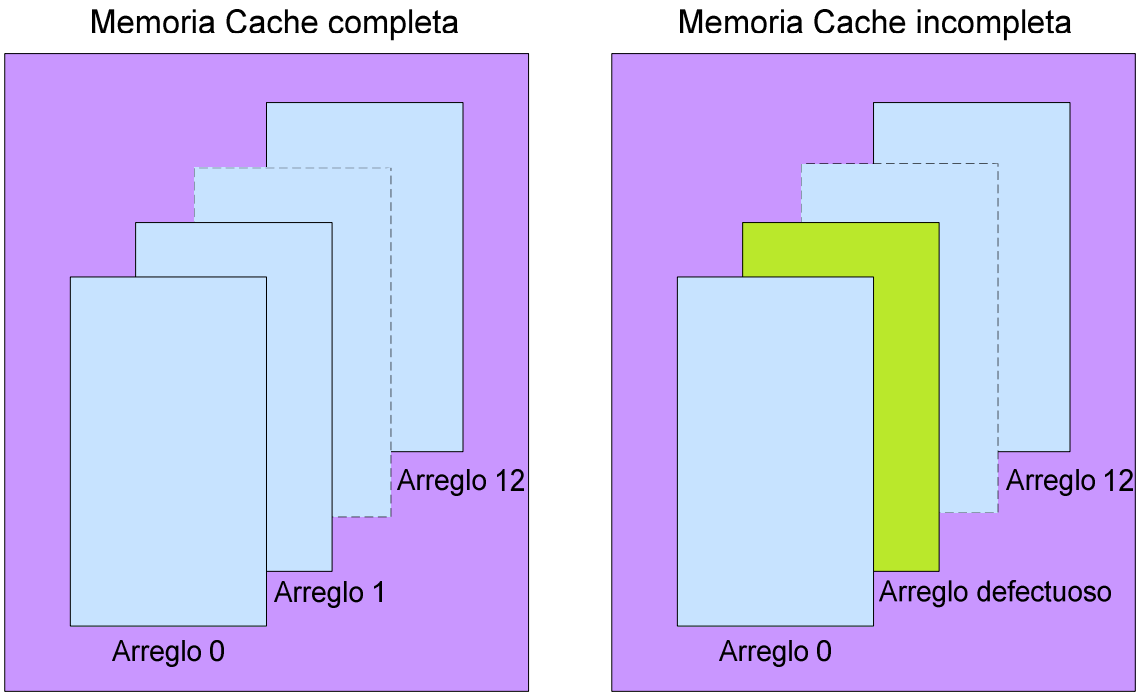

Figura 5-12 Diagrama de la memoria caché L2 de Montvale

En las pruebas de caché, se usan patrones de prueba estándar para cada nivel de memoria caché de todos los productos de Intel, grupos de patrones de prueba específicos para cada arreglo y todas las pruebas utilizan un algoritmo en común, la diferencia radia en el direccionamiento, la temporización, la cantidad y el tamaño de los arreglos de cada producto.

Una vez realizada una breve descripción de la metodología de prueba de Caché, procederemos a definir la metodología de validación. El *HardBin* 20 se refiere a fallas que ocurren en el nivel L0 y el *HardBin* 60 se vincula con las fallas que corren en el nivel L1 y el *HardBin* 62 corresponde a las fallas en L2.

La validación de las fallas de L2 requieren de una metodología de prueba específica, la cual, no es posible implementar con la herramienta desarrollada, por lo tanto, solo de especificará una metodología de validación común para los bines 20 y 60, la cual, consiste en:

1. Se debe correr el flujo *Vcc* y *Shops* para descartar problemas de alimentación y continuidad en la plataforma de prueba.
2. Se debe ejecutar el flujo de Caché para verificar que la unidad realmente presenta la falla y que esta no se presentó por un problema en el CMT, el TIU o el programa de prueba.
3. Utilizar el *SoftBin* para identificar la prueba dentro del flujo de Caché que fallo.
4. Se debe correr un *Shmoo* de la prueba que falló, el eje X del *shmoo* debe ser el parámetro *FSBper_spec* con un valor mínimo de 5ns, un máximo de 15ns e incrementos de 250ps, además, el eje Y debe ser el parámetro *vcore_spec* con u valor mínimo de 800mV, un máximo de 1.4V e incrementos de 25mV.
5. Por último, se debe correr un Full Scan a la prueba que falló para identificar cuales patrones de prueba son los que están fallando.

5.3. Descripción del software

En las secciones anteriores, se realizó una descripción de los archivos de entrada de la aplicación y se comentó sobre la metodología que se definió con la investigación realizada, únicamente, falta por comentar el funcionamiento de la herramienta, es decir, el procedimiento de interpretación y ejecución de las instrucciones de usuario en el CMT.

Iniciaremos con la descripción de la rutina principal del programa, en la figura 5-13, se muestra el diagrama de flujo de dicha rutina, la cual, se que encarga de verificar los parámetros de entrada, especifica la dirección del archivo de configuración que el usuario debe indicar a la hora de corre la aplicación, si el usuario, no introduce una dirección valida la herramienta despliega en la consola un mensaje de error.

Una vez que la herramienta ha leído el archivo de configuración esta conoce las direcciones de los otros cuatro archivos de entrada, en la figura 5-14, se muestra el diagrama de flujo de la rutina que se encarga de realizar dicha labor.

Después, desde la rutina principal se crea un objeto de tipo *System Controller,* el cual, permite tener acceso a las instrucciones de *UserSDK Tools API* y por ende a las instrucciones de *Helper Class*, si la aplicación no puede obtener del sistema dicho objeto se presenta un mensaje de error y se concluye con la ejecución de la herramienta.

El siguiente paso, es realizar un análisis sintáctico del archivo TPL[19], en el cual, se especifica por medio de un seudo código la definición de los flujos del programa de prueba, dicho análisis busca leer del archivo toda la información vincula con los flujos, los elementos en estos y las condiciones de paso de dichos elementos según el resultado de las pruebas que se realizan en ellos, en la figura 5-19, se muestra el diagrama de flujo de la rutina principal que se encarga de realizar el análisis sintáctico del archivo TPL

[19] Archivo TPL: es el archivo donde se define por medio de un seudo código la información de todos los flujos y las pruebas que constituyen el programa de prueba

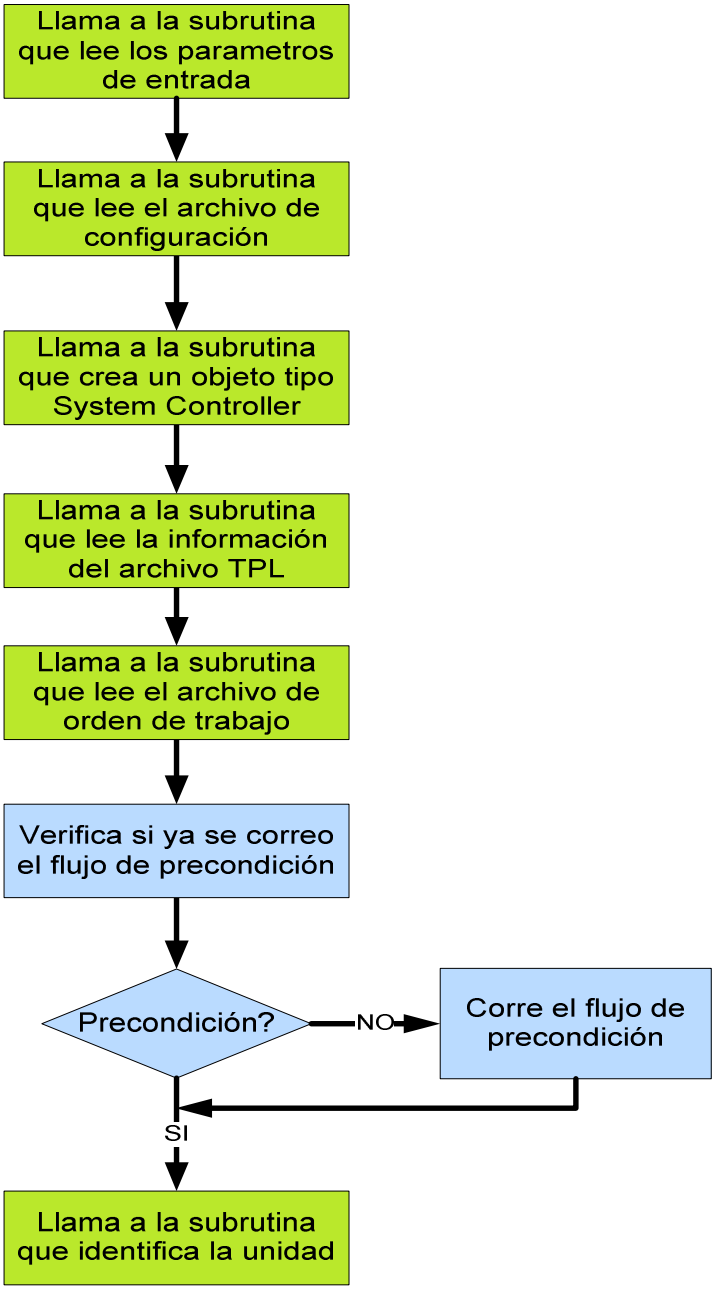

Figura 5-13 Diagrama de flujo de la rutina principal

Eso con el fin de poder construir las estructuras de memoria que se muestran en las figuras: 5-15, 5-16, 5-17 y 5-18 y de esta manera poder disponer de una seudo base de datos para poder reconstruir de manera dinámica los flujos del programa de prueba y poder utilizar las instrucciones de *UserSDK Tools API* para ejecutar un flujo y un flujo forzado en el *tester CMT*.

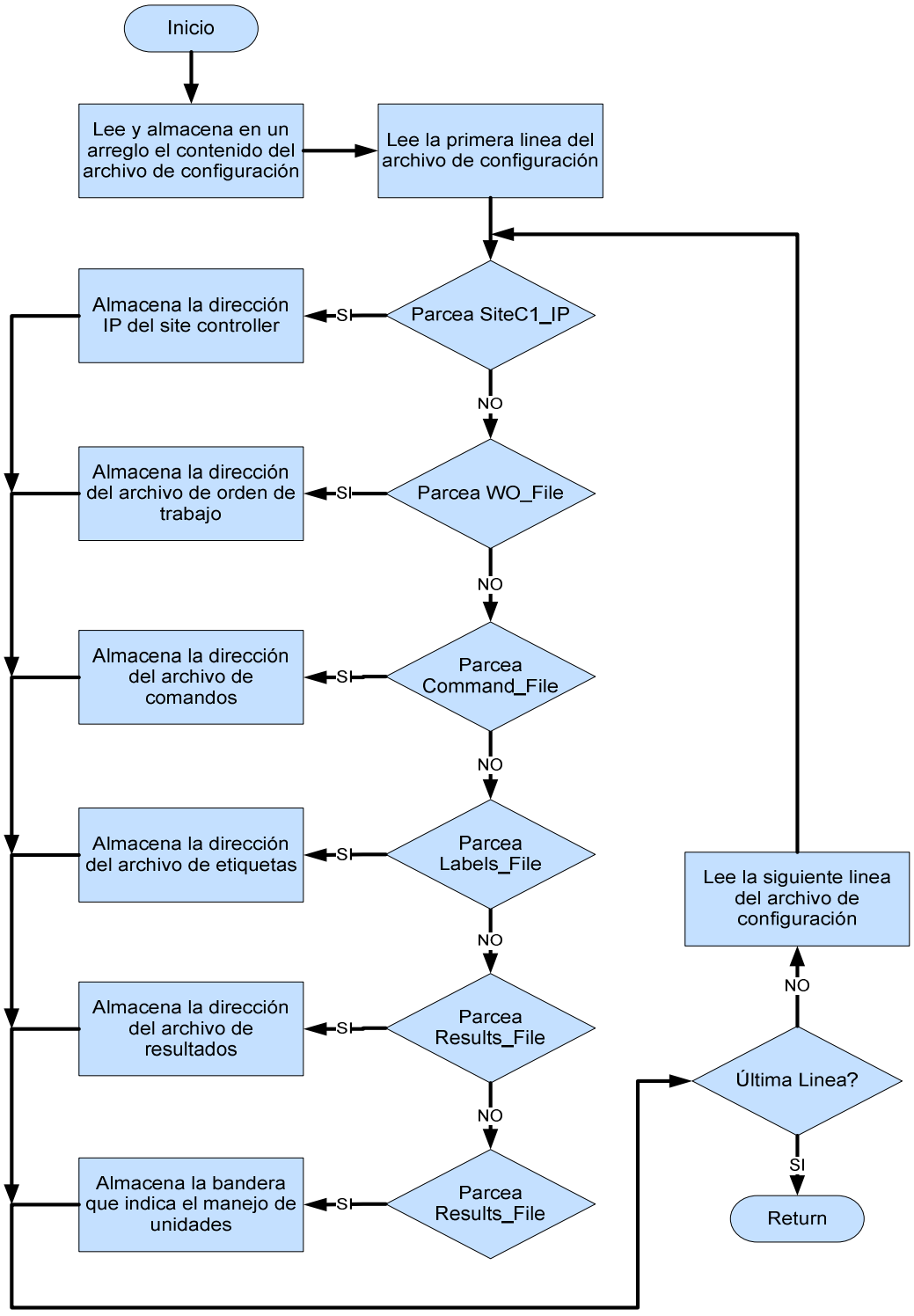

Figura 5-14 Diagrama de flujo de la rutina que lee el archivo de configuración

Para facilitar la comprensión del lector, se realiza una descripción de cada una de las estructuras de memoria que se crean con el análisis sintáctico, del archivo TPL.

La estructura de memoria de la figura 5-15, se crea para poder vincular el nombre de un flujo con todos los elementos que forman parte de este, los elementos pueden ser pruebas aisladas o un conjunto de pruebas (compuesto), es decir, un flujo puede contener otros flujos, lo cual, se puede visualizar como un flujo padre y uno o más flujos hijos.

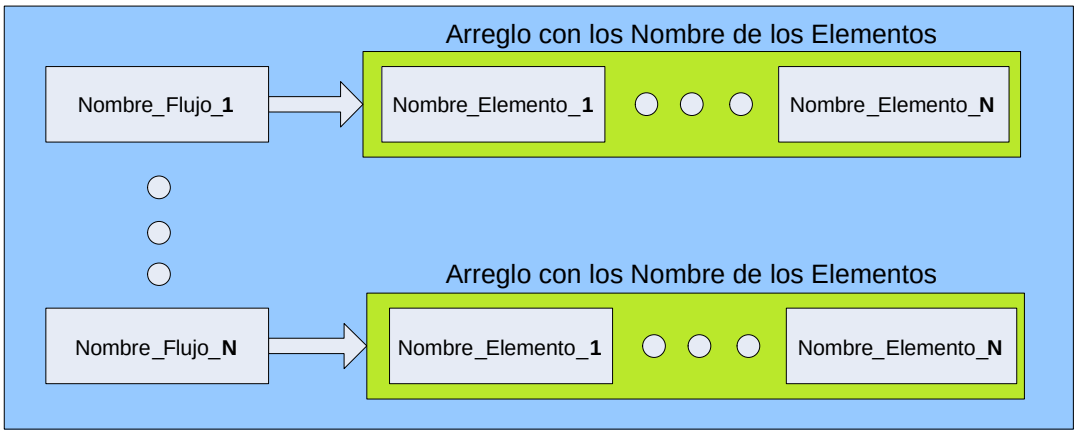

Figura 5-15 Diagrama de la estructura de memoria que contiene la información de los flujos

En las figura 5-20 y 5-21, se muestran los diagramas de flujo de las rutinas que se encargan de crear la estructura de memoria de la figura 5-15, la rutina *Busca_Flujos*, se encarga de buscar y almacenar el nombre de los flujos, luego llama a la rutina *Busca_FlowItem*, la cual, se encarga de leer y almacenar el nombre de todos los elementos que pertenecen a cada flujo.

Para poder realizar la reconstrucción y ejecución dinámica de los flujos desde la PERL, es necesario conocer cual es la siguiente prueba que se debe ejecutar según el resultado obtenido con la ejecución de la prueba actual, por este motivo, la herramienta construye estructura de memoria que se observa en la figura 5-16, en la cual, se almacena toda la información correspondiente a los elementos de los flujos.

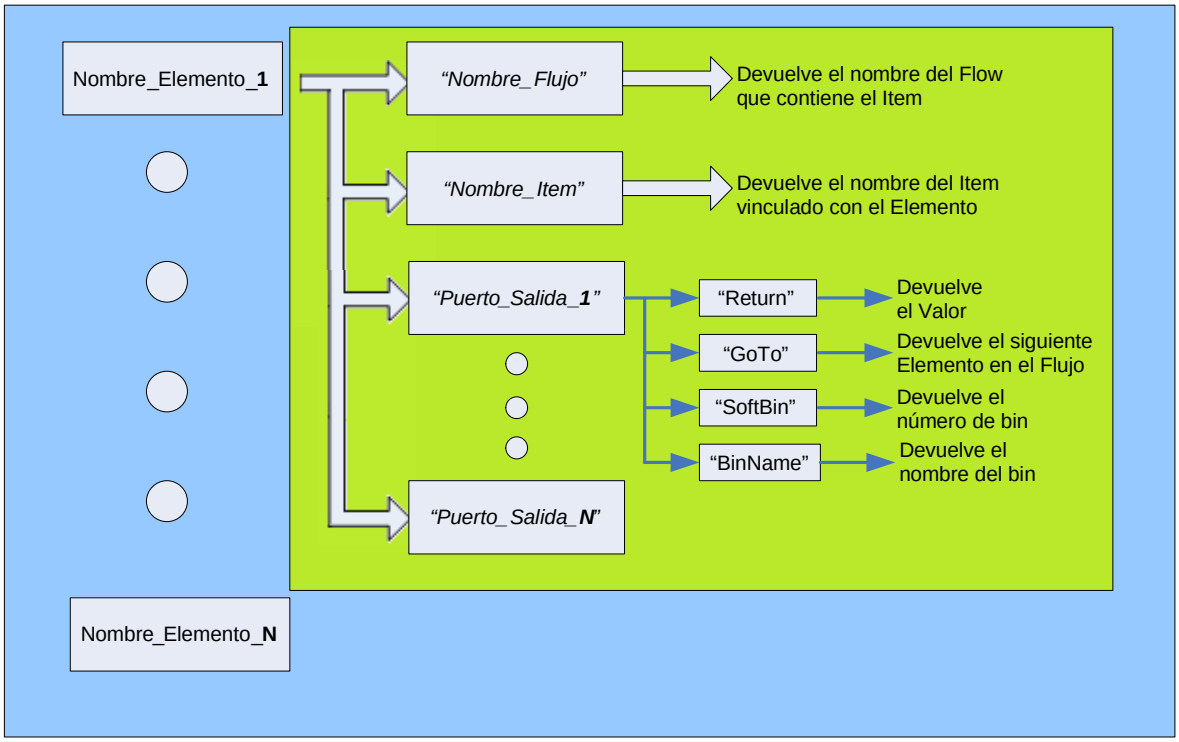

Figura 5-16 Diagrama de la estructura de memoria con la información de los elementos

La estructura de memoria de la figura 5-16, permite utilizar el nombre del elementos y el resultado de la ejecución de esta para consultar el siguiente elementos en la ejecución del flujo, en las figuras 5-21 y 5-22, se muestran los diagramas de flujo de las rutinas de *Busca_FlowItem* y *Busca_FlowItemInf* respectivamente, las cuales se encargan de leer y almacenar toda la información de los elementos que están definidos en el archivo TPL.

Ahora bien, una prueba puede formar parte de más de un flujo e inclusive, una misma prueba se puede repetir en un mismo flujo, por lo tanto, fue necesario, crear una estructura de memoria que vincule el nombre de la prueba con todos los elementos relacionados con ella, recordemos que elementos es un concepto genérico que agrupa las pruebas o flujos dentro de un flujo en particular.

En la figura 5-17, se puede observar la estructura de memoria que contiene la información de todas las pruebas que se definen en el archivo TPL, nuevamente, las rutinas *Busca_FlowItem* y *Buscan_FlowItemInf* se encargan de leer y almacenar la información de las pruebas.

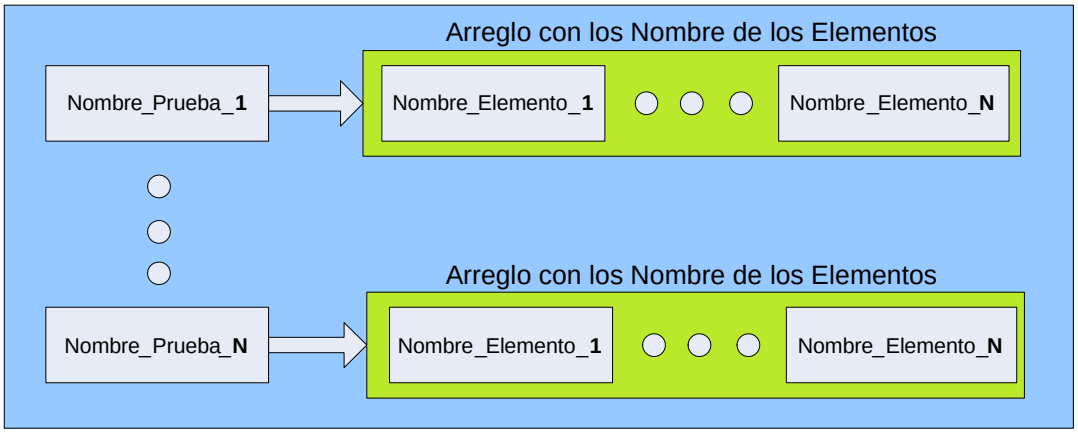

Figura 5-17 Diagrama de la estructura de memoria con la información de las pruebas

Únicamente, faltaría crear una estructura de memoria que permita vincular un SoftBin con una prueba o un conjunto de pruebas, ya que de esta manera se puede utilizar el SoftBin que el usuario define en el archivo de orden de trabajo para realizar una validación con la prueba o el grupo de pruebas en que la unidad presentó problemas.

En la figura 5-18, se muestra el diagrama de la estructura de memoria que contiene la información de los bines de falla, en la rutina Busca_FlowItemInf de la figura 5-22, se realiza la lectura y almacenamiento de dicha información.

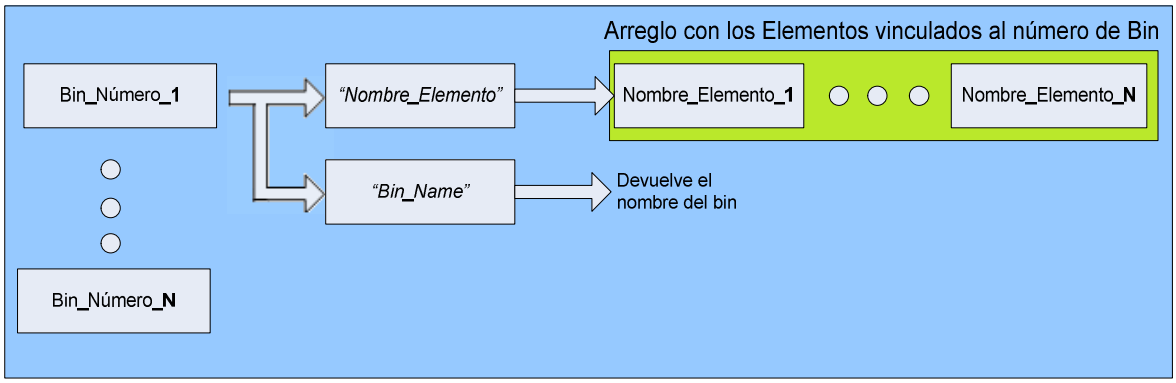

Figura 5-18 Diagrama de la estructura de memoria con la información de los bines

En las figuras 5-19, 5-20, 5-21 y 5-22 se muestran los diagramas de flujo de todas las rutinas involucradas en el análisis sintáctico del archivo TPL, los

diagramas están ordenadas según la secuencia lógica de ejecución de la herramienta desarrollada.

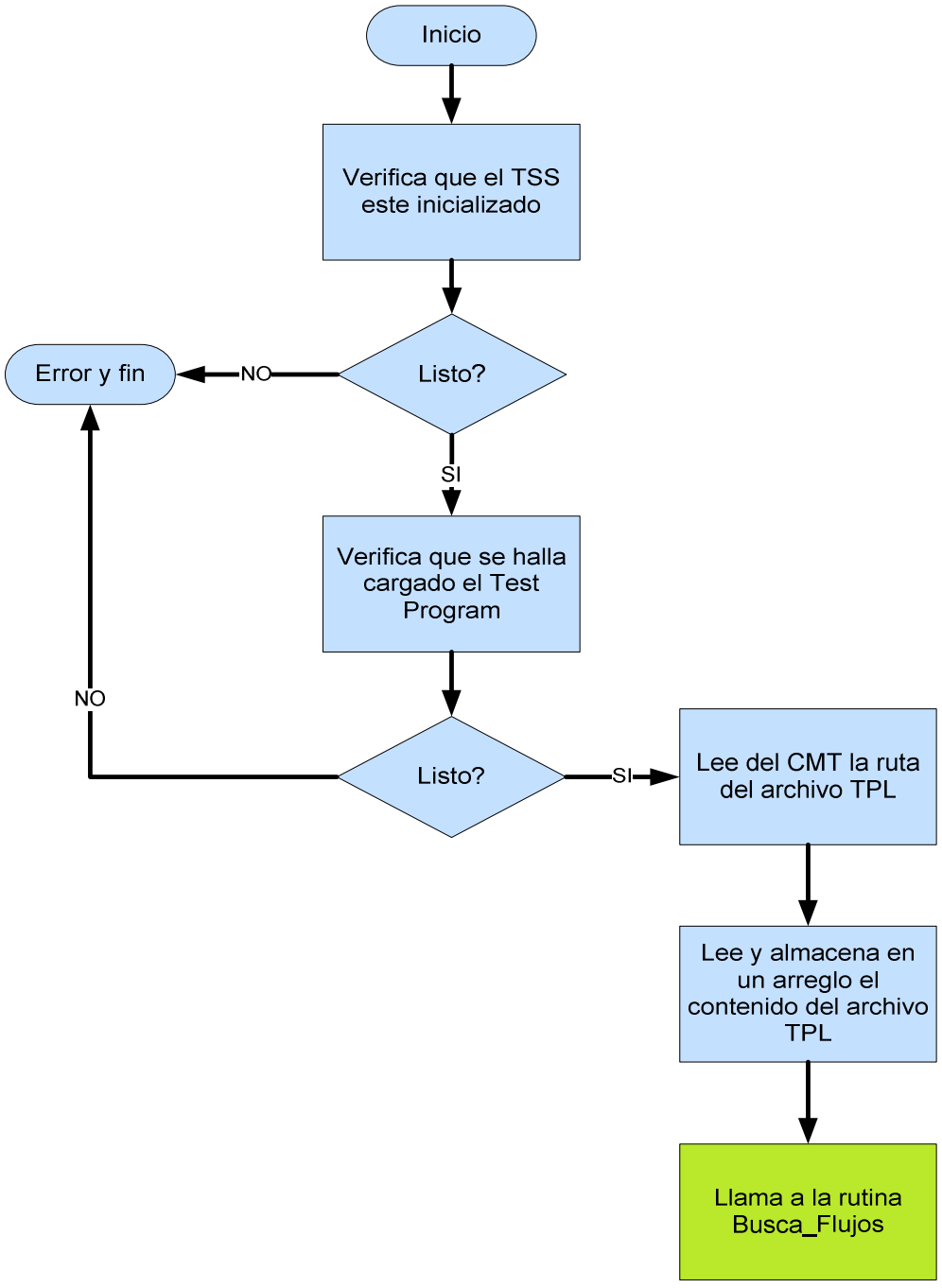

Figura 5-19 Diagrama de flujo de la rutina que analiza la información del archivo TPL

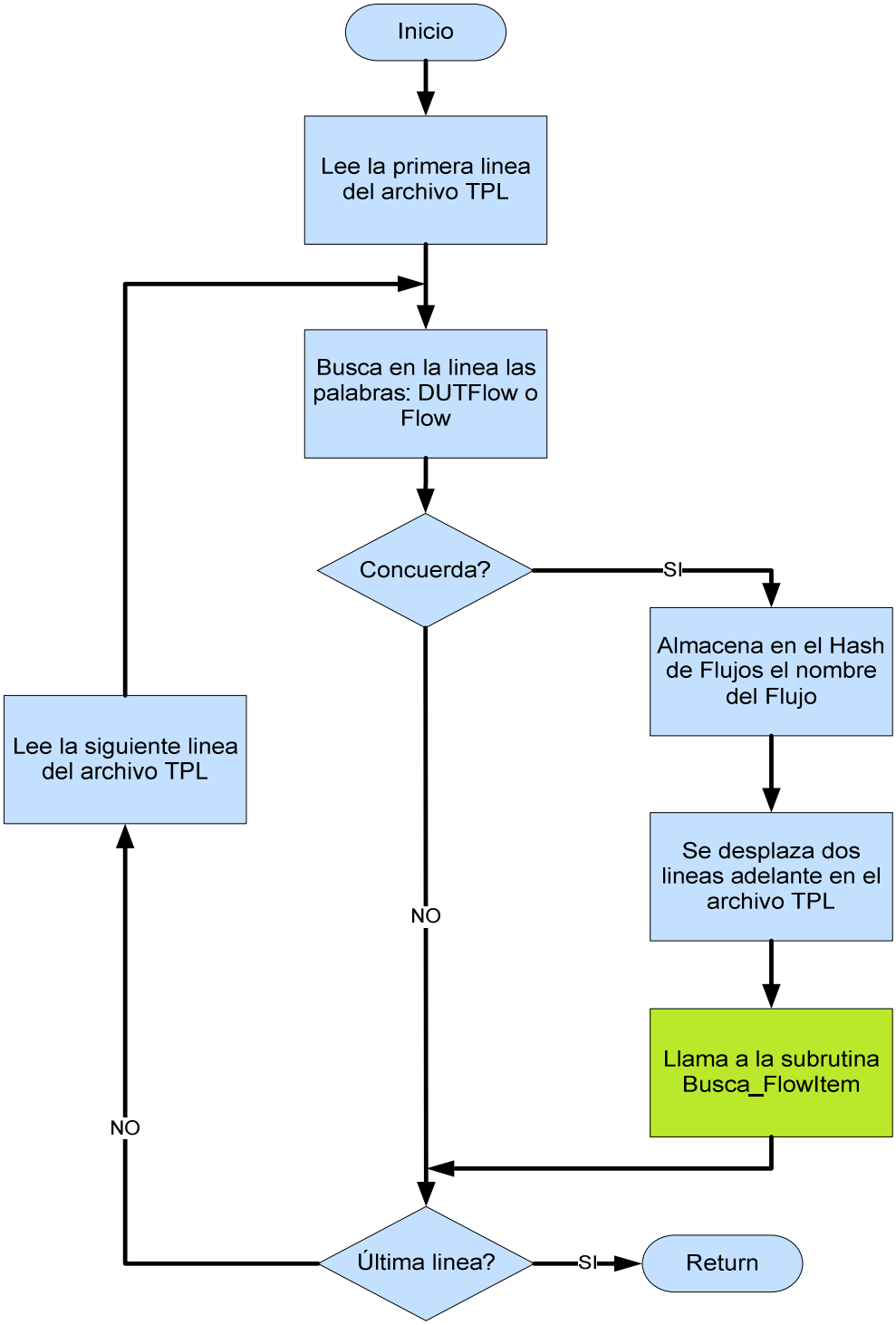

Figura 5-20 Diagrama de flujo de la rutina Busca_Flujos

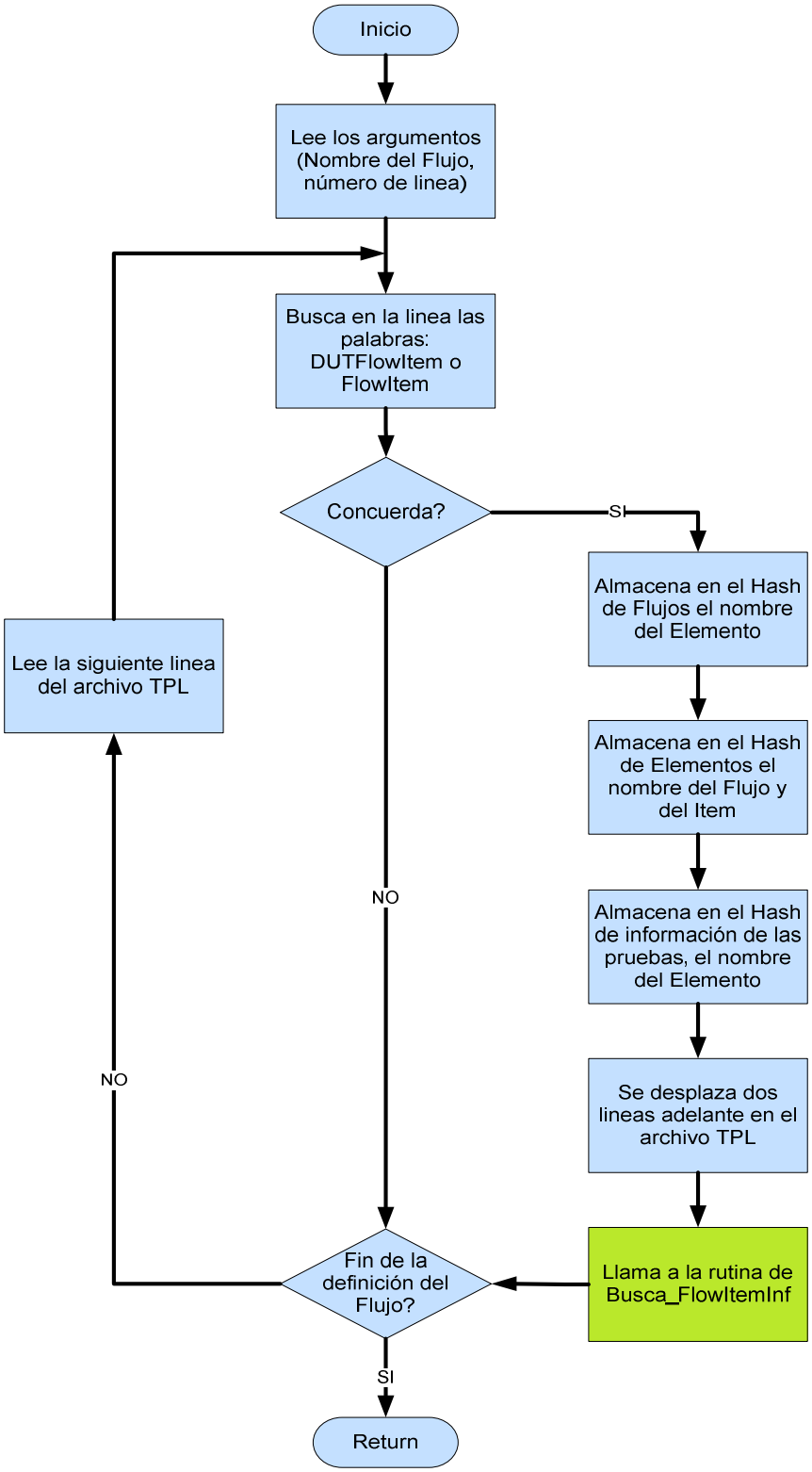

Figura 5-21 Diagrama de flujo de la rutina Busca_FlowItem

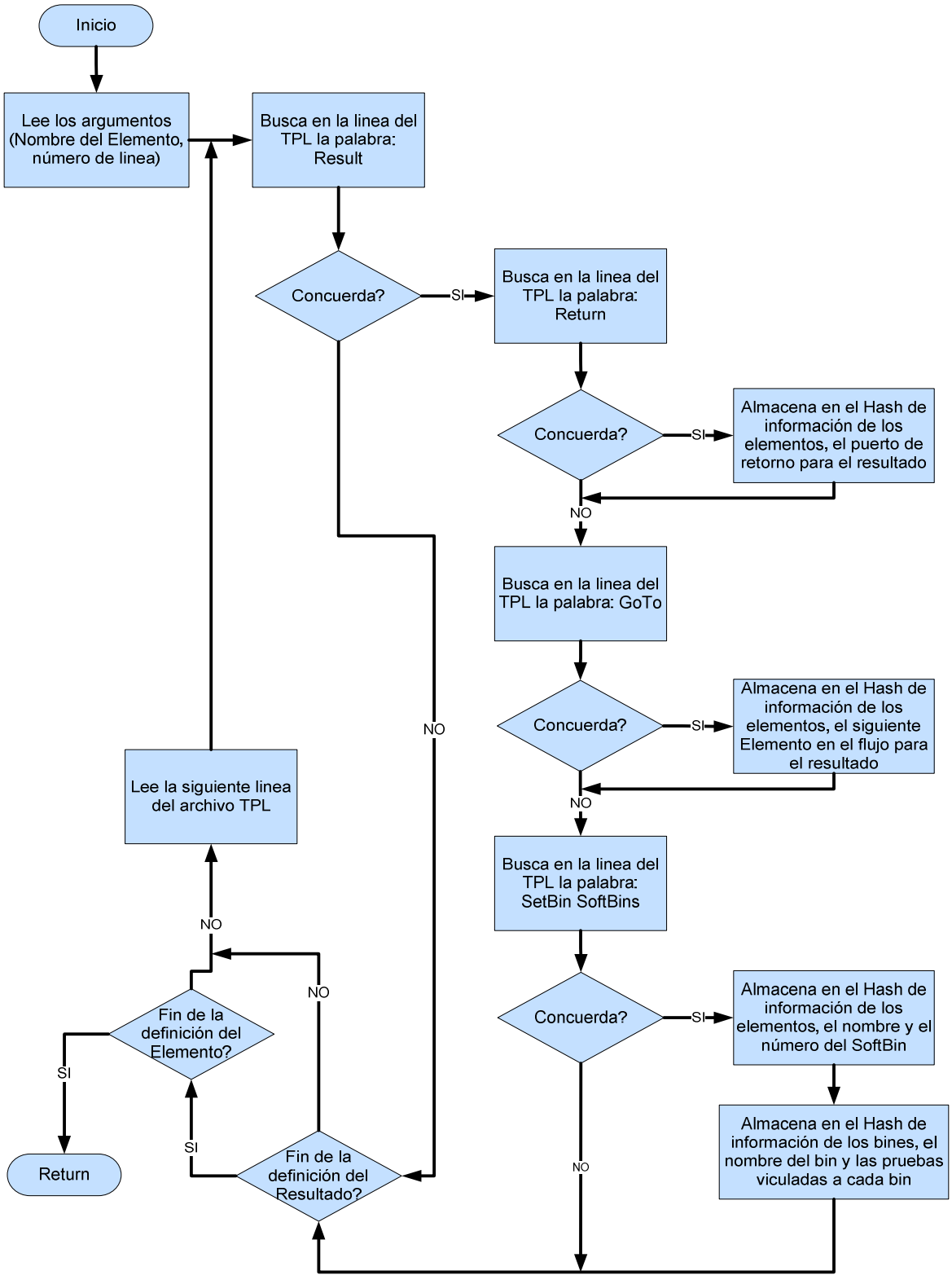

Figura 5-22 Diagrama de flujo de la rutina Busca_FlowItemInf

Si observamos el diagrama de flujo de la rutina principal en la figura 5-13, podemos observar que el siguiente paso de la ejecución después de hacer terminado con el análisis del archivo TPL, es realizar la lectura del archivo de orden de trabajo, en la figura 5-23, se muestra el diagrama de flujo de la rutina que se encarga de realizar dicha labor.

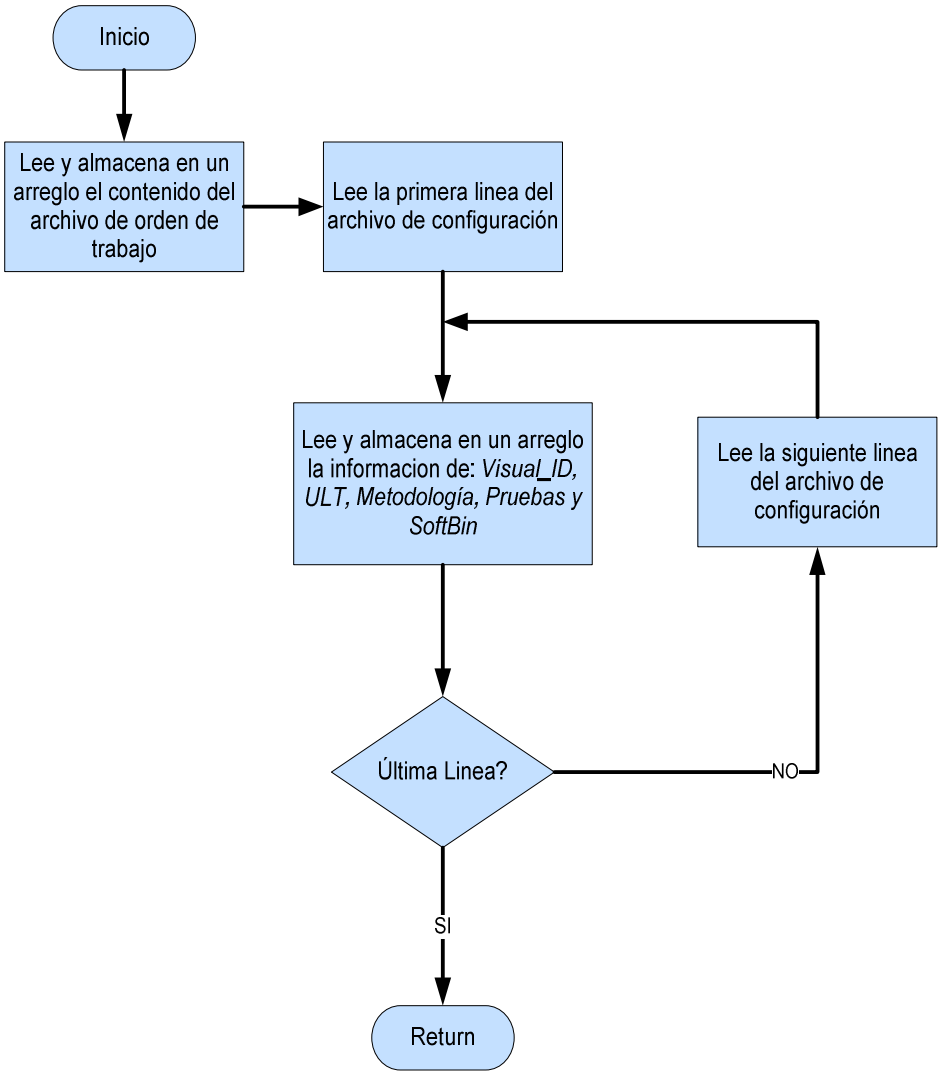

Figura 5-23 Diagrama de flujo de la rutina que lee el archivo de orden de trabajo

Con la lectura del archivo de orden de trabajo, se almacena en arreglos separados la información relacionada con el Visual ID, el ULT, la metodología, las pruebas y el SoftBin que el usuario define, esto para facilitar su consulta y ejecución en las otras rutinas.

Una vez, que se ha concluido la lectura del archivo de orden de trabajo, la herramienta se encarga de verificar si ya se ejecutó la rutina que preacondiciona el *tester CMT*, esta rutina se encarga de ejecutar el flujo de inicial y el flujo que se encarga de la lectura de los fusibles del procesador, el diagrama de flujo de la misma se muestra en la figura 5-24. La ejecución de dicha rutina es indispensable para garantizar el correcto funcionamiento del *tester CMT* y por ende de la aplicación desarrollada.

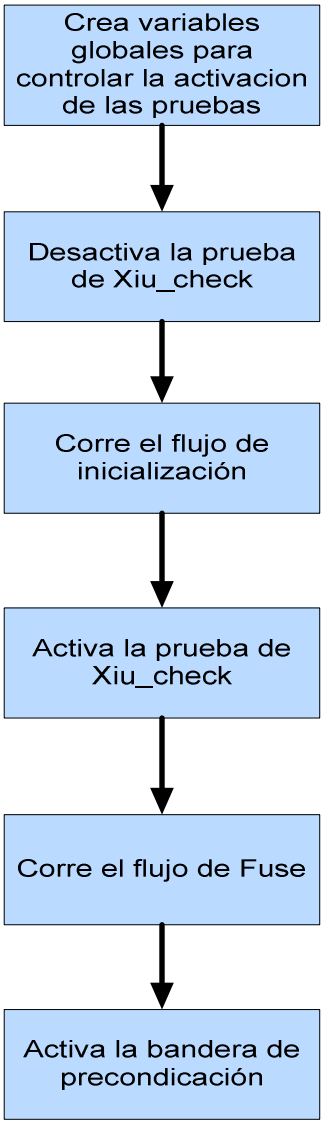

Figura 5-24 Diagrama de flujo que se encarga de preacondicionar el CMT

El último proceso que realiza la rutina principal es identificar la unidad, en la figura 5-25, se muestra el diagrama de flujo de la rutina que se encarga de realizar dicha labor, dicha rutina hace uso de varas funciones de usuario de fueron desarrollas por José David Gómez, las cuales, se encargan de establecer una comunicación con el *handler* y permiten desarrollar una lógica para el manejo de las unidades.

Esta rutina aparte del control de las unidades se encarga de leer el Visual ID y el ULT de la unidad y posteriormente llama a la rutina que se encarga de correr la orden de trabajo, el diagrama de flujo de dicha rutina se muestra en la figura 5-26.

La rutina que corre la orden de trabajo se encarga de comparar el Visual ID y el ULT de la unidad que esta siendo probada con los valores que anteriormente, la herramienta leyó del archivo de orden de trabajo.

En caso de que el Visual ID o el ULT de la unidad en prueba concuerde con alguno de los definidos por el usuario, la herramienta verifica si el usuario definió un SoftBin para dicha unidad, de ser así, se realiza una validación de lo contrario, la herramienta corre un experimento de ingeniería, en caso de que el Visual ID o el ULT de la unidad no concuerde con alguno de los especificados por el usuario, la herramienta verifica si el usuario utilizó la etiqueta *_All_* en la definición del Visual ID para especificar una metodología por defecto.

Tal cual se observa en el diagrama de la figura 5-26, una vez que se identificado la unidad y el procedimiento vinculado con ella, se procede a leer las instrucciones de la validación o del experimento de ingeniería según sea el caso.

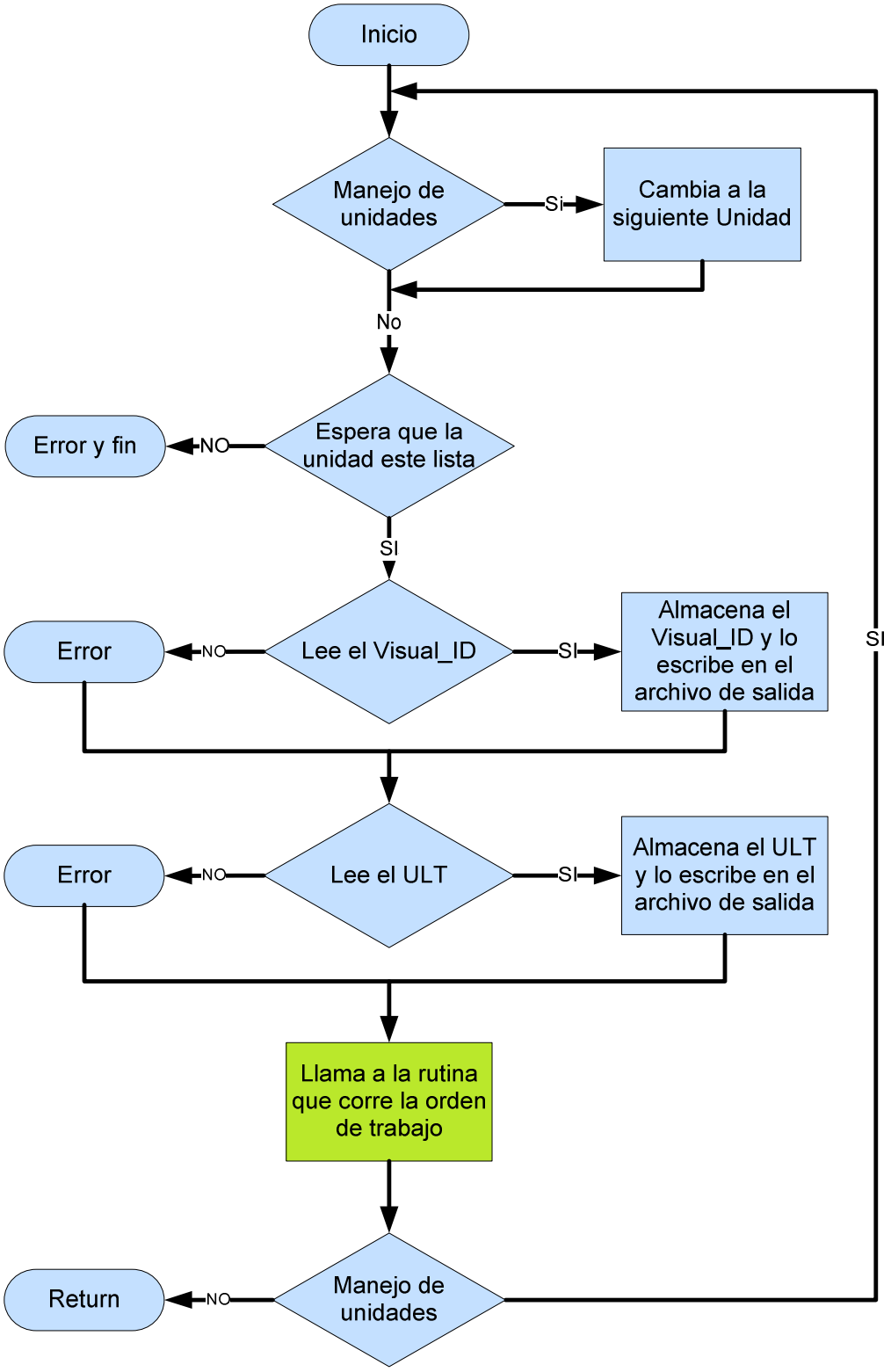

Figura 5-25 Diagrama de flujo de la rutina que se encarga de identificar la unidad

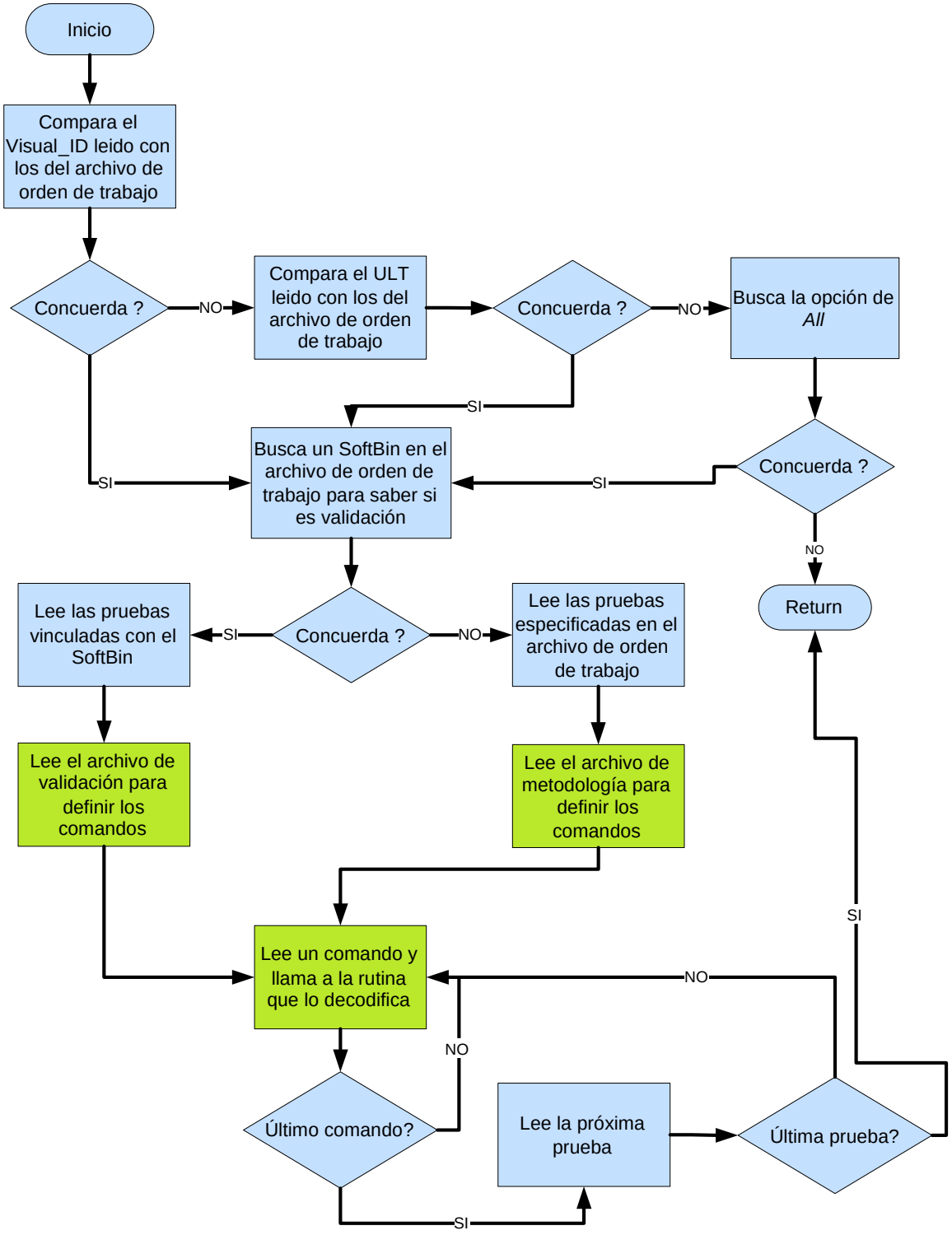

Figura 5-26 Diagrama de flujo de la rutina que corre la orden de trabajo

En caso de ser una validación, se utiliza el SoftBin para identificar las pruebas vinculadas con la falla y obtener el HardBin[20] de la unidad, con el cual se realiza una búsqueda en el archivo de validación para identificar y almacenar un arreglo todas las instrucciones especificadas por el usuario, en la figura 5-27, se muestra el diagrama de flujo de la rutina que se encarga de leer las instrucciones de usuario ya sea para una validación o para un experimento de ingeniería.

Para un experimento de ingeniería se realiza un procedimiento similar al anterior, la diferencia esta en que la definición de las pruebas a utilizar se realiza a partir de la lectura de la etiqueta que el usuario definió para la unidad en el archivo de orden de trabajo, además, la lectura de las instrucciones se realiza en el archivo de instrucciones y se utiliza para dicha búsqueda la etiqueta que el usuario definió para la unidad en el apartado de metodología en el archivo de orden de trabajo.

Una vez que se han definido las pruebas o utilizar y la secuencia de instrucciones a ejecutar para dichas pruebas, se inicia con la ejecución de las instrucciones, para lo cual, se hace uso de un decodificador de instrucciones, en la figura 5-28, se muestra el diagrama de flujo de la rutina que se encarga de dicha labor.

Las instrucciones se ejecutan una a una, y se repite la secuencia de instrucciones definidos por el usuario, llámese validación o experimentos de ingeniería con todas las pruebas que el usuario defina.

[20] HardBin: un código único que identifica a un conjunto de fallas.

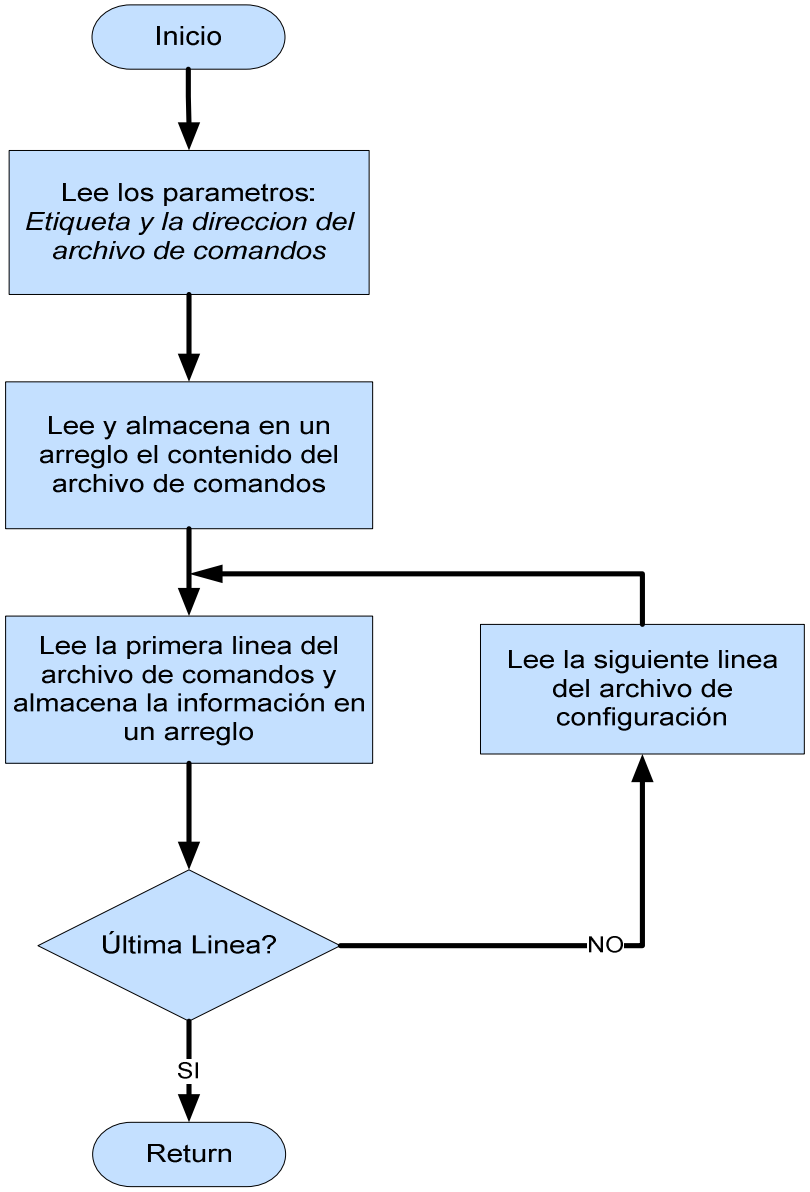

Figura 5-27 Diagrama de flujo de la rutina que lee las instrucciones de usuario

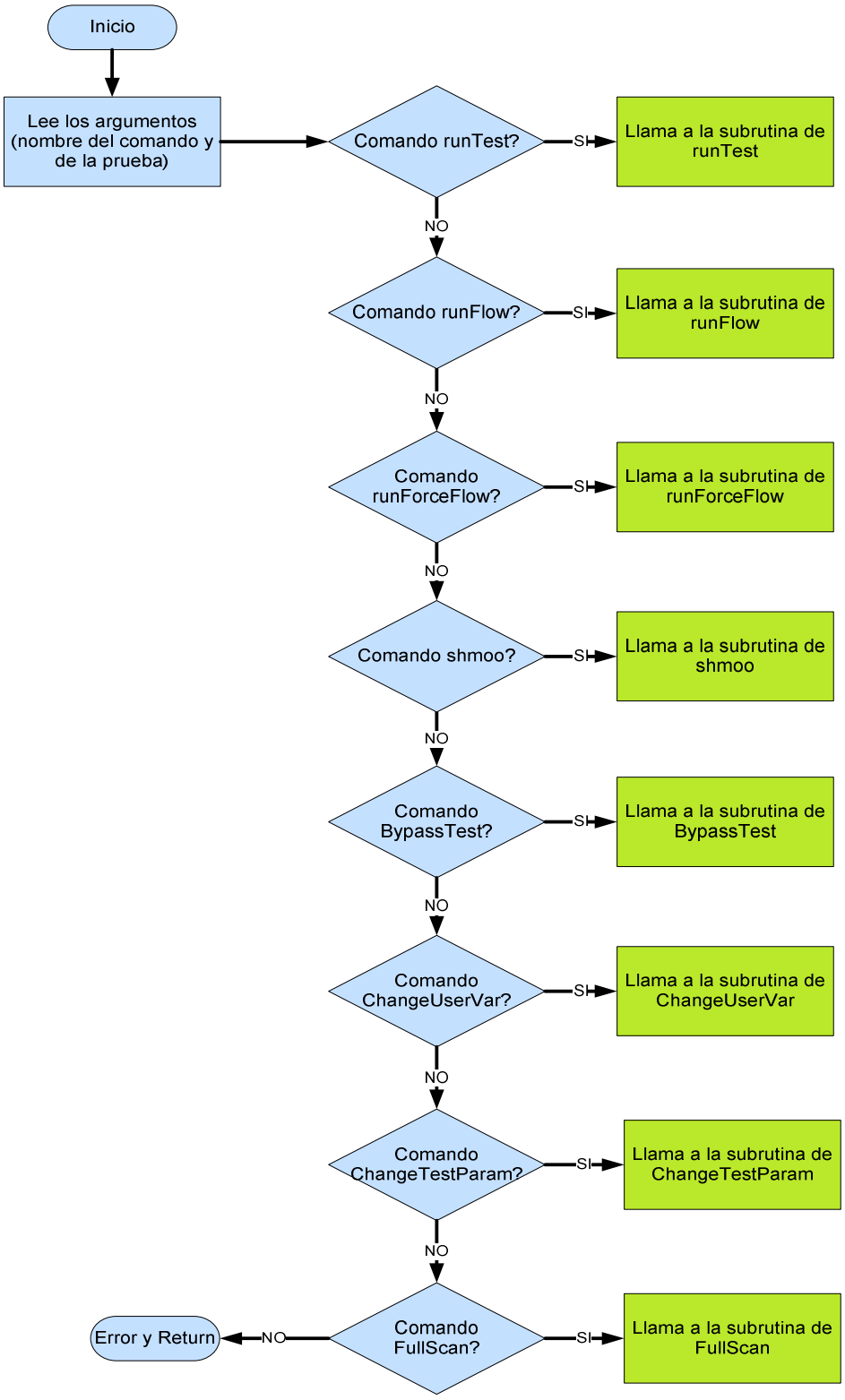

Figura 5-28 Diagrama que la rutina que decodifica las instrucciones del usuario

5.3.1. Las instrucciones de Usuario

Las instrucciones de usuario pueden ser utilizadas en el archivo de instrucciones para definir un experimento de ingeniería o en el archivo de validación para definir una metodología de validación para un grupo de fallas.

5.3.1.1. La instrucción *runTest*

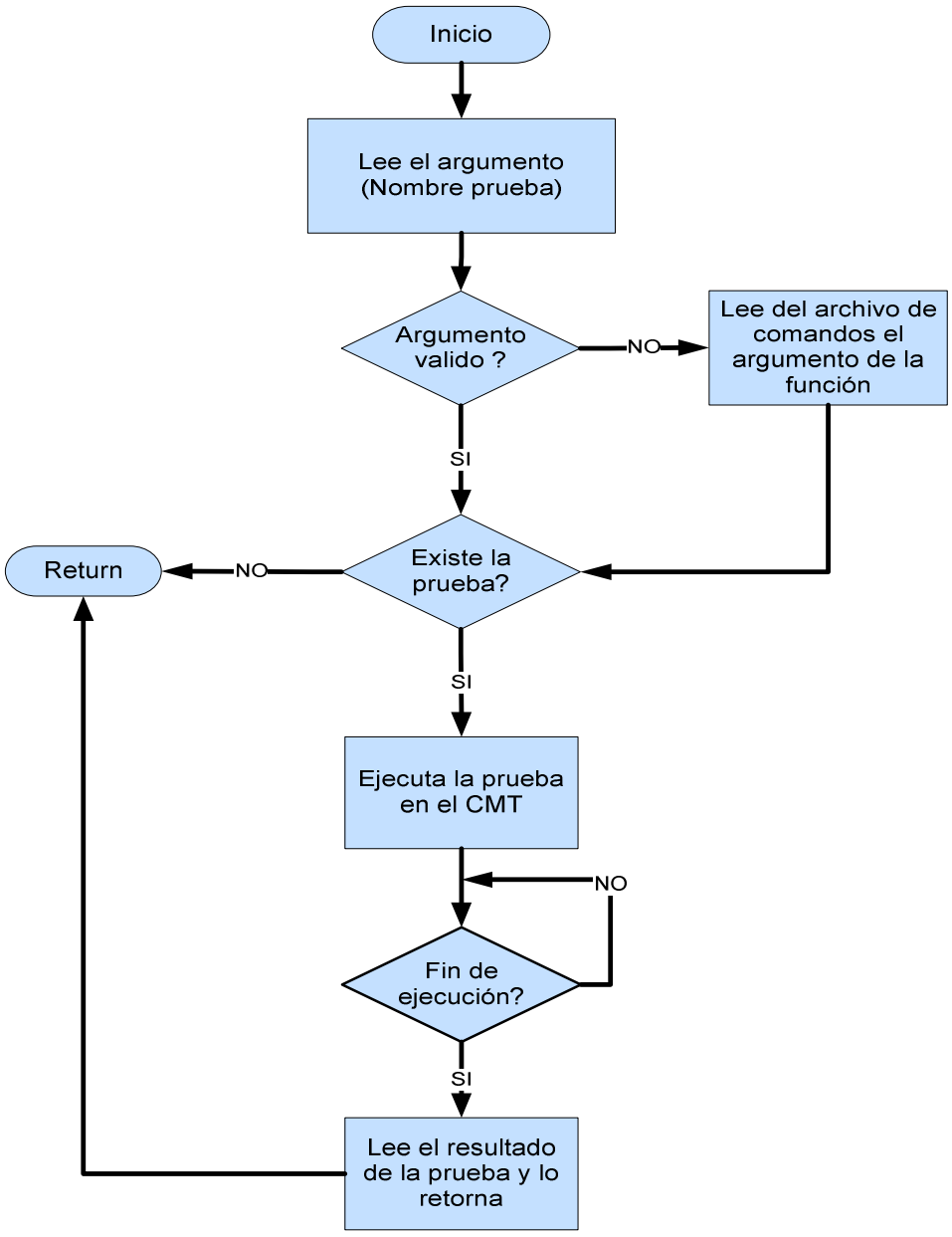

Figura 5-29 Diagrama de flujo de la rutina que ejecuta la instrucción *runTest*

La instrucción *runTest* se utiliza para correr una prueba en el *tester CMT*, en la Tabla 3, se muestra la sintaxis de la instrucción, además, en la figura 5-29, se puede observar el diagrama de flujo de la rutina que se encarga de ejecutar la instrucción.

Tabla 3 Sintaxis de la instrucción *runTest*

Formato	Descripción
runTest ()	Sin argumento, se utilizan las pruebas que se definen en el archivo de orden de trabajo
runTest (*Nombre de la prueba*)	Se puede especificar el nombre de la prueba que se desea correr como argumento

5.3.1.2. La instrucción *runFlow*

La instrucción *runFlow* ejecuta un flujo de manera dinámica, es decir, ejecuta el primer elemento del flujo, lee el resultado de su ejecución, realiza una consulta a las estructuras de memoria creadas con el análisis sintáctico del archivo TPL para definir el siguiente elemento y repite de manera recursiva el procedimiento anterior hasta ejecutar el último elementos del flujo.

Tabla 4 Sintaxis de la instrucción *runFlow*

Formato	Descripción
runFlow (*Nombre del flujo*)	Se debe especificar el nombre del flujo como argumento de la instrucción

En la Tabla 4, se muestra la sintaxis de la instrucción *funFlow* y en la figura 5-30, se puede observar el diagrama de la rutina que da servicio a dicho instrucción, la cual, se encarga de leer del archivo de instrucciones o del archivo de validación el nombre del flujo y llamar a la rutina *runFlowItem*, la cual, se encarga de implementar el algoritmo que permite la ejecución dinámica de los flujos, en la figura 5-31, se muestra el diagrama de flujo del la rutina *runFlowItem*.

La rutina *runFlowItem*, se encarga de identificar si el elemento del flujo que se va ha ejecutar es una prueba o un subflujo, en caso de ser una prueba, llama a la rutina *runTestItem*, la cual se encarga de correr la prueba y leer el resultado de esta, en la figura 5-32, se muestra el diagrama de flujo de dicha rutina.

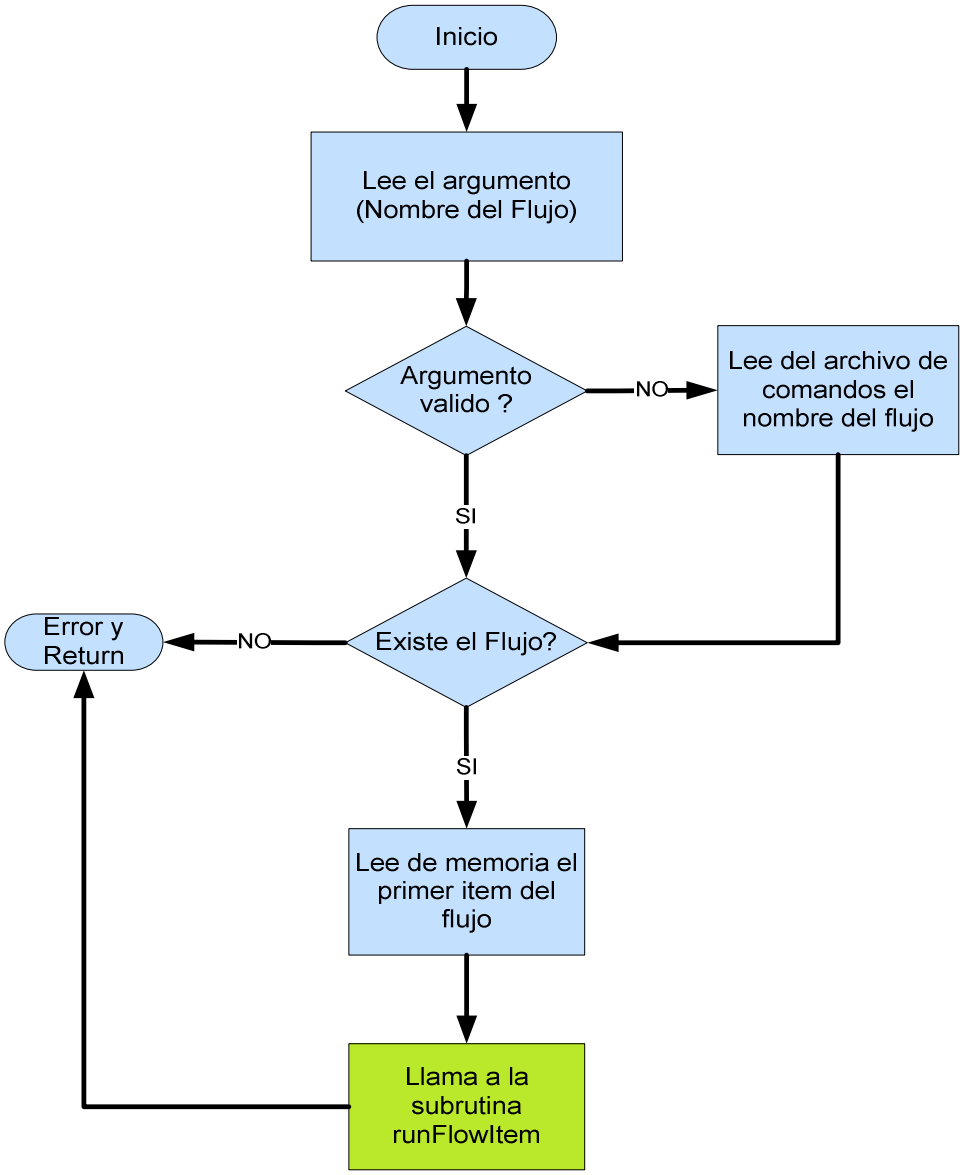

Figura 5-30 Diagrama de flujo de la rutina que corre la instrucción *runFlow*

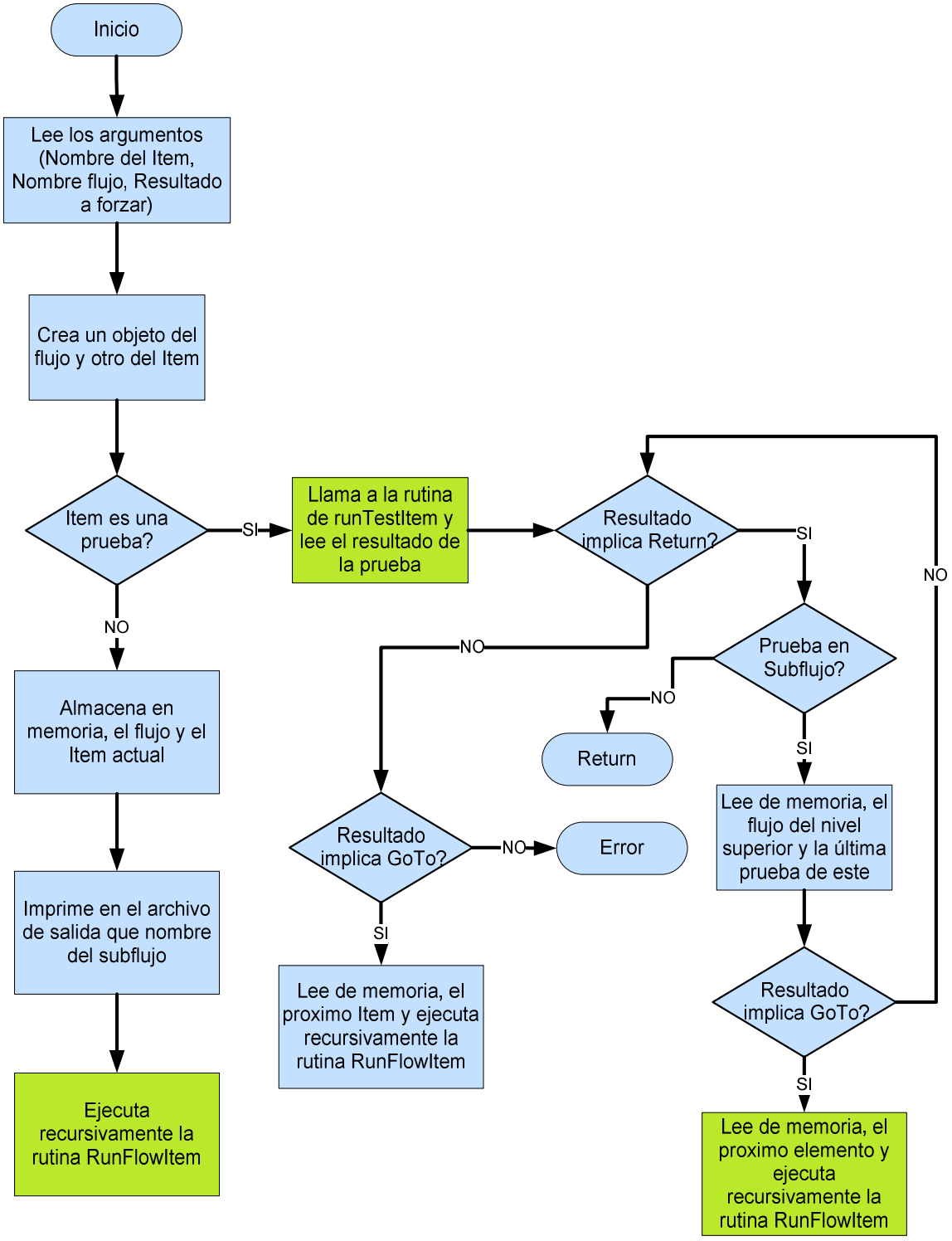

Figura 5-31 Diagrama de flujo de la rutina *runFlowItem*

Si el elemento no es una prueba, la herramienta asume que es un subflujo, por lo tanto, esta almacena en una pila la información del flujo actual y del último elemento ejecuta y ejecuta de manera recursiva la rutina *runFlowItem*, tal cual, se observa en la figura 5-31.

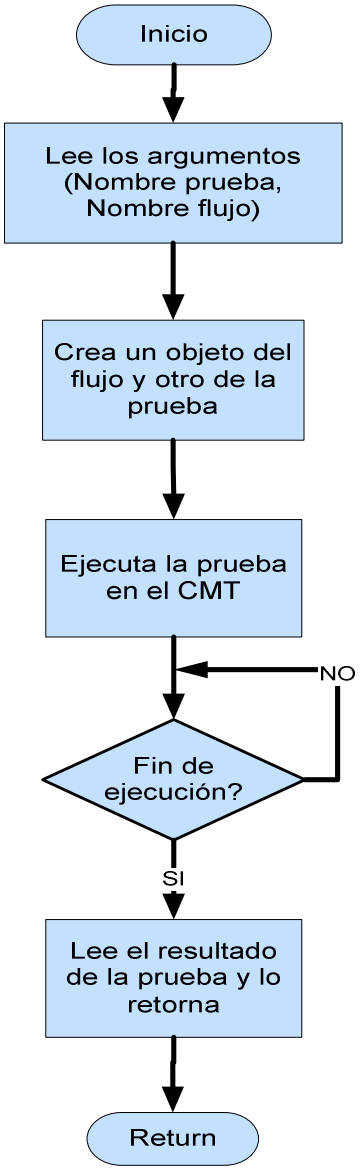

Figura 5-32 Diagrama de flujo que la rutina *runTestItem*

5.3.1.3. La instrucción *runForceFlow*

La instrucción *runForceFlow* se utiliza para ejecutar un flujo forzado, es decir, corre un flujo de forma dinámica tal cual lo hace la instrucción *runFlow*, la diferencia radica en que en un flujo forzado el siguiente elemento a ser ejecuta no se define por el resultado de la prueba o subflujo actual, sino que se define a partir de un resultado que el usuario defina.

Por ejemplo, con esta instrucción un usuario puede ejecutar un flujo e indicar que el resultado a forzar sea 1, entonces, sin importar el resultado de la prueba actual, el siguiente elemento del flujo que se ejecutaría sería el elemento vincula con el resultado 1, en la Tabla 5, se puede observar la sintaxis de la instrucción *runForceFlow*.

Tabla 5 Sintaxis de la instrucción *runForceFlow*

Formato	Descripción
runForceFlow (*Nombre del flujo, Resultado*)	Se debe especificar el nombre del flujo y el resultado a forzar como argumentos

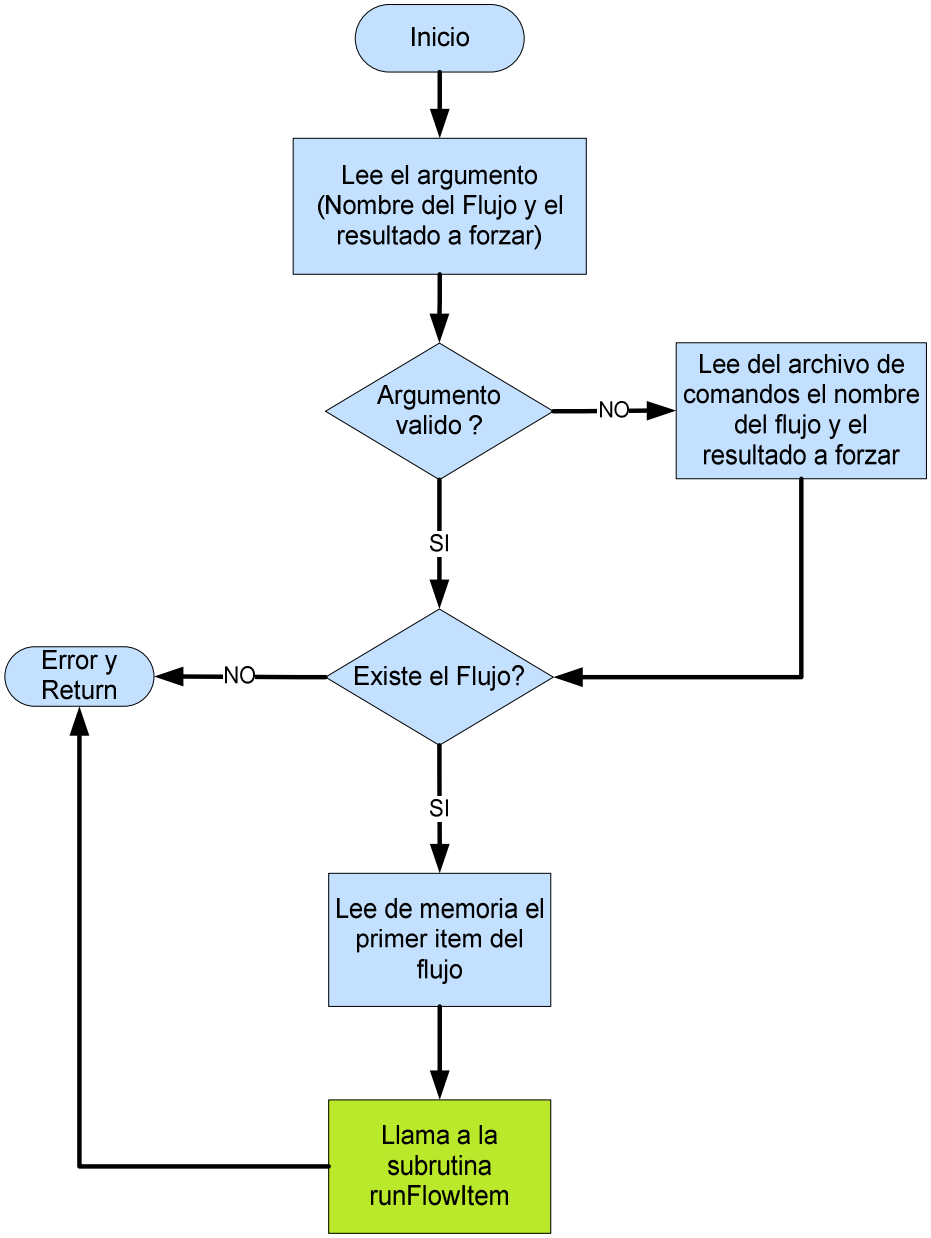

Figura 5-33 Diagrama de flujo de la rutina que corre la instrucción *runForceFlow*

5.3.1.4. La instrucción *shmoo*

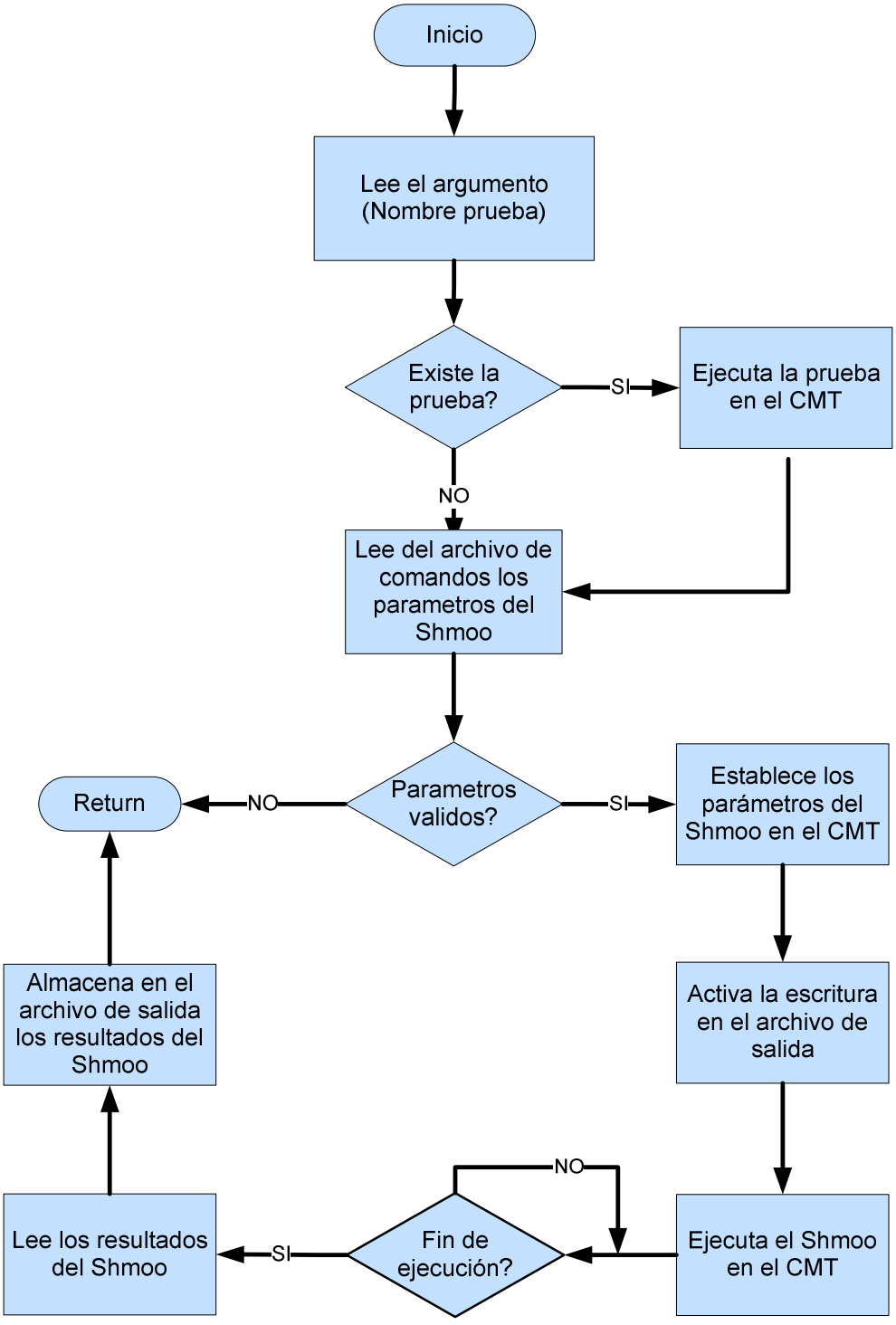

Figura 5-34 Diagrama de flujo de la rutina que corre la instrucción *shmoo*

La instrucción *shmoo*, se utiliza para realizar un gráfico bidimensional de dos parámetros de una prueba, por ejemplo la tensión en el núcleo en función de la velocidad de bus.

Tal cual se observa en la Tabla 6, existen dos posibilidad para definir un *shmoo*, una es especificar la configuración del *shmoo* con argumentos de la función y la otra es utilizar una etiqueta para definir la configuración del *shmoo*, de igual manera, existe dos posibilidades de vincular el *shmoo* con una prueba, la primera es utilizar la instrucción *runTest* para ejecutar la prueba desea justo antes de definir la instrucción *shmoo* en la programación del archivo de instrucciones o el archivo de validación, la otra posibilidad es definir una prueba o un conjunto de pruebas en el archivo orden de trabajo para la unidad en prueba, en cuyo caso, la herramienta realiza un *shmoo* a cada uno de las pruebas que se especifiquen en dicho archivo.

Tabla 6 Sintaxis de la instrucción *shmoo*

Formato	Descripción
shmoo (X,Xmin,Xmax,Xstep,Y,Ymin,Ymax,Ystep,Mode)	Se debe especificar los valores mínimos, máximos y el incremento para cada eje, así como el modo del *shmoo* que se desea
shmoo (*_Nombre de etiqueta_*)	Se puede utilizar una etiqueta para vincular el *shmoo* con una configuración particular

5.3.1.5. La instrucción *BypassTest*

La instrucción *BypassTest*, se utiliza para permitir o omitir la ejecución de una prueba, por ejemplo, si es *bypass* de una prueba esta activa esta no sea ejecuta de lo contrario si es ejecutada, en la Tabla 7, se muestra la sintaxis de la instrucción *BypassTest*.

Tabla 7 Sintaxis de la instrucción *BypassTest*

Formato	Descripción
BypassTest (*Nombre de la prueba, ON* | *OFF*)	Se debe especificar la prueba que se desea, además se debe utilizar *ON* para desactivar la prueba y *OFF* para activarla

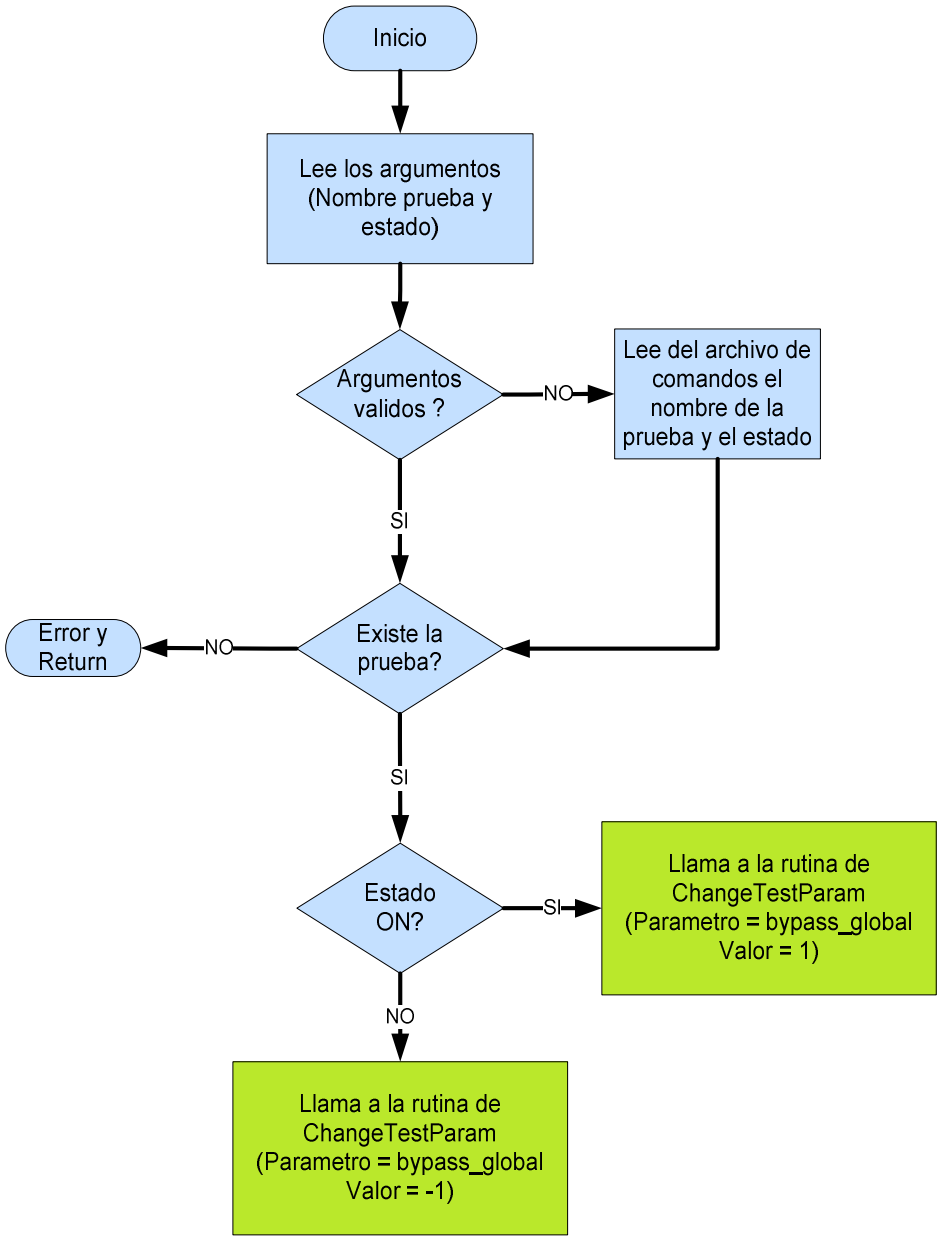

Figura 5-35 Diagrama de flujo de la rutina que corre la instrucción *BypassTest*

5.3.1.6. La instrucción *ChangeTestParam*

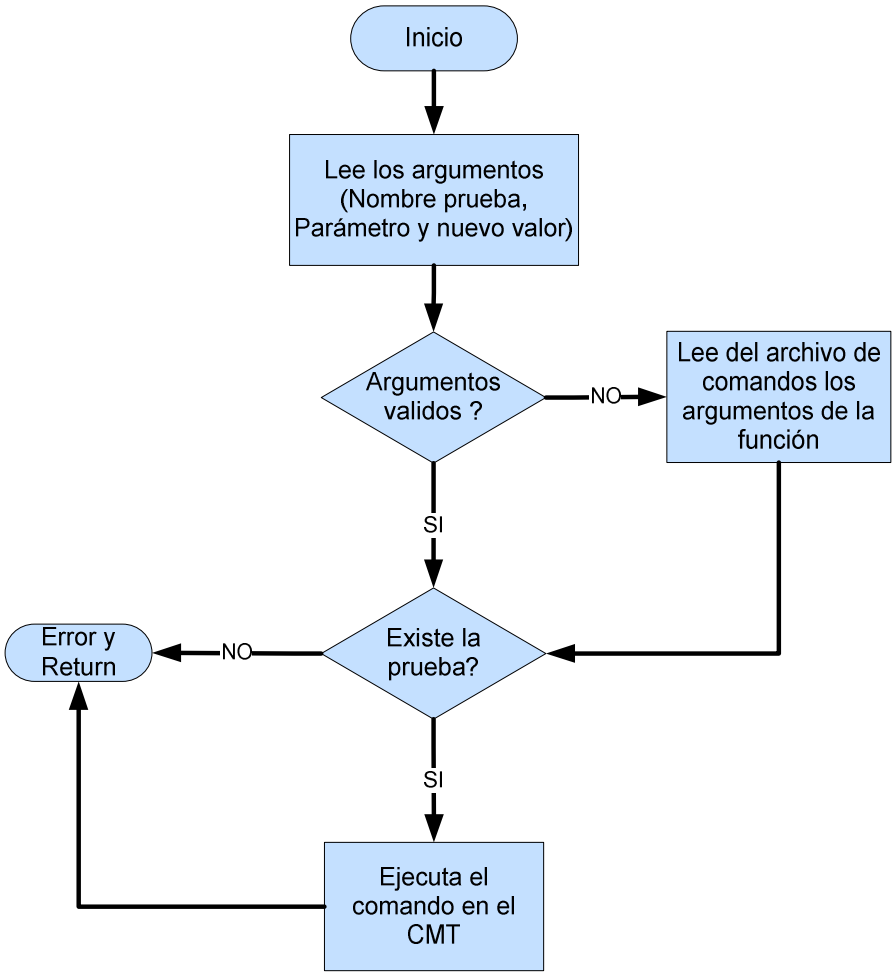

Figura 5-36 Diagrama de flujo de la rutina que corre la instrucción *ChangeTestParam*

La instrucción *ChangeTestParam*, se utiliza para cambiar los parámetros de una prueba, es decir, se modifica la configuración de que prueba que se pretende realizar, se puede cambiar: el patrón de prueba, la temporización, los niveles de las señales y demás opciones especificas de cada prueba.

En la figura 5-36, se muestra el diagrama de flujo de la rutina que da servicio a la instrucción *ChangeTestParam* y en la Tabla 8 se puede observar la sintaxis de dicha instrucción.

Tabla 8 Sintaxis de la instrucción *ChangeTestParam*

Formato	Descripción
ChangeTestParam (Prueba, *parámetro, valor*)	Se puede especificar la prueba, el parámetro y el nuevo valor deseados.
ChangeTestParam (*Parámetro, valor*)	Se puede especificar el parámetro y el nuevo valor deseados y se modifican todas las pruebas especificadas en la orden de trabajo

5.3.1.7. La instrucción *ChangeUserVar*

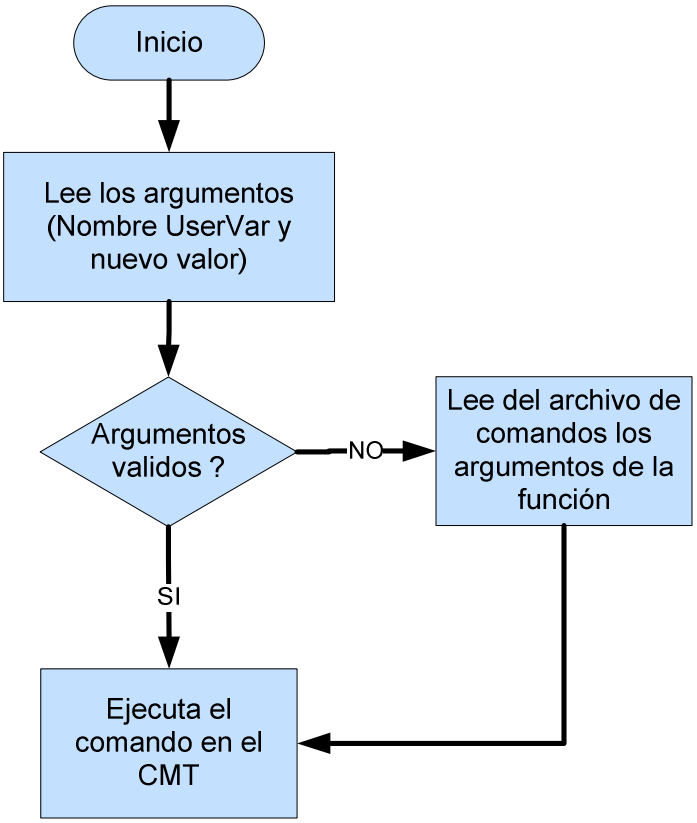

Figura 5-37 Diagrama de flujo de la rutina que corre la instrucción *ChangeUserVar*

Tabla 9 Sintaxis de la instrucción *ChangeUserVar*

Formato	Descripción
ChangeUserVar (*Variable, valor*)	Se debe especificar el nombre de la variable y el nuevo valor para esta

La instrucción *ChangeUserVar* se utiliza para modificar el valor de alguna variable global del programa de prueba, las variables globales son muy utilizadas para controlar la ejecución de las pruebas, para configurar las pruebas y para almacenar información importante de las pruebas.

En la figura 5-37, se muestra el diagrama de flujo de la rutina que se encarga de dar servicio a la instrucción *ChangeUserVar* y en la Tabla 9, el lector puede consulta la sintaxis de dicha instrucción.

5.3.1.8. La instrucción *FullScan*

La instrucción *FullScan*, se utiliza para ejecutar todos los patrones de prueba vinculados con cada prueba y poder determinar cuales patrón son los que presentan problemas y de esta manera poder caracterizar una falla.

En la figura 5-38, se muestra el diagrama de flujo de la rutina que se encarga de ejecutar la instrucción *FullScan* y en la Tabla 10 se muestra la sintaxis de dicha instrucción.

Tabla 10 Sintaxis de la instrucción *FullScan*

Formato	Descripción
FullScan ()	Sin argumento, se realiza un *FullScan* las pruebas que se definen en el archivo de orden de trabajo
FullScan (TestName)	Se puede especificar el nombre de la prueba a la cual se desea realizar el *FullScan*

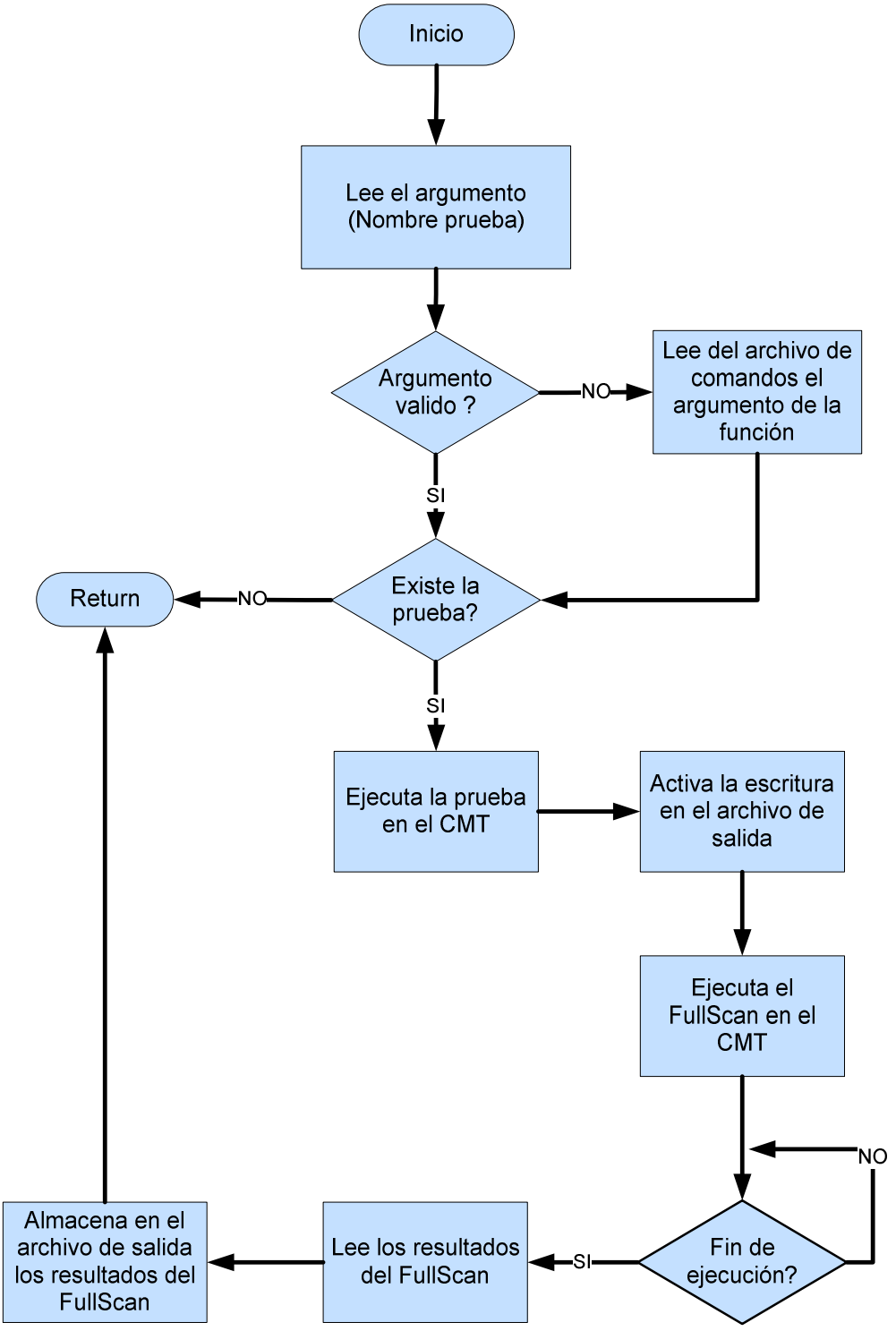

Figura 5-38 Diagrama de flujo de la rutina que corre la instrucción *FullScan*

Capítulo 6: Análisis de Resultados

En este capítulo se muestran los principales resultados obtenidos con la herramienta desarrollado, también, se muestran resultados obtenidos gráficamente con la interfaz del TSS, esto con le fin de poder establecer una comparación entre ambos y demostrar el correcto funcionamiento de la aplicación y con esto el cumplimiento de los objetivos del proyecto.

6.1. Resultados

Como ya se mencionó, la herramienta tiene la capacidad de ejecutar dos tipos de operaciones: una validación de una falla y experimento de ingeniería, ambas de enmarcan en el concepto de orden de trabajo, por este motivo, es que se presentan dos tipos de resultados experimentales: los resultados de una validación y los resultados de un experimento de ingeniería.

6.1.1. Resultados de una validación

6.1.1.1. Resultados obtenidos con la herramienta desarrollada

```
_____________________________________________

     START THE DEDUG-CHARACTERIZATION
_____________________________________________

Start with the execution of the Test WAIT_UNIT
The result of the Test WAIT_UNIT is: 1

Start with the execution of the Test GET_VISUALID
The result of the Test GET_VISUALID is: 1

****************************************
      THE UNIT VISUAL ID IS: 2L640Y00Z0661
****************************************

Start with the execution of the Test MON_FUSE_READ_M1
The Test MON_FUSE_READ_M1 have more than one Flow Item
The result of the Test MON_FUSE_READ_M1 is: 1

****************************************
    THE UNIT ULT IS: W6356328_085_+04_+03
****************************************
_____________________________________________________
 THE RESULT OF THE TEST chainc0_tdo_100_min1200_min1025
-----------------------------------------------------
```

Figura 6-1 Identificación de la unidad

Tabla 11 Comparación del archivo de orden de trabajo

Visual ID	ULT	Methodology	TEST	SOFTBIN
2L633914Z0126	-	_MetoAlex3_	_AlexTest_	-
-	W6356328_351_-01_+02	_MetoAlex2_	_Test1_	-
2L640Y00Z0661	-	-	-	b41_06
All	-	_MetoAlex3_	_Test1_	-

```
Start with the execution of the Flow Scan_comp

The Flow Item chainc0_180_min1200_min1025_10TCK_eos0 is a Test
The result of the FlowTest chainc0_180_min1200_min1025_10TCK_eos0 is: 0

The Flow Item chainc0_100_min1200_min1025_10TCK_eos0 is a Test
The result of the FlowTest chainc0_100_min1200_min1025_10TCK_eos0 is: 1

The Flow Item chainc1_100_min1200_min1025_10TCK_eos0 is a Test
The result of the FlowTest chainc1_100_min1200_min1025_10TCK_eos0 is: 1

The Flow Item chainc0_100_max1200_max1025_10TCK_eos0 is a Test
The result of the FlowTest chainc0_100_max1200_max1025_10TCK_eos0 is: 1

The Flow Item chainc1_100_max1200_max1025_10TCK_eos0 is a Test
The result of the FlowTest chainc1_100_max1200_max1025_10TCK_eos0 is: 1

The Flow Item iscandualcore_100_min1200_min1025_10TCK_eos0 is a Test
The result of the FlowTest iscandualcore_100_min1200_min1025_10TCK_eos0 is: 1

The Flow Item iscandualcore_100_max1200_max1025_eos0 is a Test
The result of the FlowTest iscandualcore_100_max1200_max1025_eos0 is: 1

The Flow Item iscauncore_mpe_100_min1200_min1025_10TCK_eos0 is a Test
The result of the FlowTest iscauncore_mpe_100_min1200_min1025_10TCK_eos0 is: 1

The Flow Item iscanuncore_mpe_100_max1200_max1025_10TCK_eos0 is a Test
The result of the FlowTest iscanuncore_mpe_100_max1200_max1025_10TCK_eos0 is: 1

The Flow Item iscanuncore_qr_100_min1200_min1025_10TCK_eos0 is a Test
The result of the FlowTest iscanuncore_qr_100_min1200_min1025_10TCK_eos0 is: 1

The Flow Item iscanuncore_qr_100_max1200_max1025_10TCK_eos0 is a Test
The result of the FlowTest iscanuncore_qr_100_max1200_max1025_10TCK_eos0 is: 1

The Flow Item iscanmanual_100_min1200_min1025_10TCK_eos0 is a Test
The result of the FlowTest iscanmanual_100_min1200_min1025_10TCK_eos0 is: 1

The Flow Item iscanmanual_100_max1200_max1025_10TCK_eos0 is a Test
The result of the FlowTest iscanmanual_100_max1200_max1025_10TCK_eos0 is: 1

Finish with the execution of the Flow Scan_comp
```

Figura 6-2 Ejecución del flujo de Scan con la herramienta desarrollada

```
Start with the execution of the Test chainc0_tdo_100_min1200_min1025
The result of the Test chainc0_tdo_100_min1200_min1025 is: 1

The Full Scan is :

DUT ID : 1              Pattern list : st2_chain_c0_narrow_list         Burst # 255
+-------------------------------------------------------------------------------------------------------------+
1 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------+
2 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------+
3 :  FAILED : SP0_C0_rand_r1_for_conv (default) Capture Count = 1  (MAX = 448)
+-------------------------------------------------------------------------------------------------------------+
4 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------+
5 :  PASSED : SP1_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------+
6 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------+
7 :  PASSED : SP2_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------+
8 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------+
9 :  PASSED : SP3_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------+
10 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------+
11 :  PASSED : SP4_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------+
12 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------+
13 :  PASSED : SP5_E0_rand_r2_for_conv (default)
+-------------------------------------------------------------------------------------------------------------+
14 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------+
15 :  PASSED : SP6_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------+
16 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------+
17 :  PASSED : SP7_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------+
18 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------+
19 :  PASSED : SP8_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------+
20 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------+
21 :  PASSED : SP9_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------+
22 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------+
23 :  PASSED : SP10_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------+
24 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------+
25 :  PASSED : SP11_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------+
26 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------+
27 :  PASSED : SP12_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------+
28 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------+
29 :  PASSED : SP13_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------+
30 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------+
31 :  PASSED : SP14_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------+
32 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------+
33 :  PASSED : SP15_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------+
34 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------+
35 :  PASSED : SP16_E0_rand_r2_for_conv (default)
+-------------------------------------------------------------------------------------------------------------+
36 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------+
37 :  PASSED : SP17_E0_rand_r2_for_conv (default)
+-------------------------------------------------------------------------------------------------------------+
38 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------+
39 :  PASSED : SP18_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------+
```

Figura 6-3-I Primera parte del *FullScan* de la validación del bin 41

```
+-------------------------------------------------------------------------------------------------------------------+
40 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------------+
41 :  PASSED : SP19_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------------+
42 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------------+
43 :  PASSED : SP20_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------------+
44 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------------+
45 :  PASSED : SP21_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------------+
46 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------------+
47 :  PASSED : SP22_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------------+
48 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------------+
49 :  PASSED : SP23_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------------+
50 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------------+
51 :  PASSED : SP24_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------------+
52 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------------+
53 :  PASSED : SP25_C3_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------------+
54 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------------+
55 :  PASSED : SP26_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------------+
56 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------------+
57 :  PASSED : SP27_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------------+
58 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------------+
59 :  PASSED : SP28_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------------+
60 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------------+
61 :  PASSED : SP29_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------------+
62 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------------+
63 :  PASSED : SP30_E0_rand_r2_for_conv (default)
+-------------------------------------------------------------------------------------------------------------------+
64 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------------+
65 :  PASSED : SP31_E0_rand_r2_for_conv (default)
+-------------------------------------------------------------------------------------------------------------------+
66 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------------+
67 :  PASSED : SP32_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------------+
68 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------------+
69 :  PASSED : SP33_C0_rand_r1_for_conv (default)
+-------------------------------------------------------------------------------------------------------------------+
70 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------------+
71 :  PASSED : SP34_E0_rand_r2_for_conv (default)
+-------------------------------------------------------------------------------------------------------------------+
72 :  PASSED : scan_narrow_preamble_bf106_core0_10TCK (default)
+-------------------------------------------------------------------------------------------------------------------+
73 :  PASSED : SP35_E0_rand_r2_for_conv (default)
+-------------------------------------------------------------------------------------------------------------------+
PASSED Pattern count: 72
FAILED Pattern count: 1

================================
Total PASSED Pattern count: 72
Total FAILED Pattern count: 1
```

Figura 6-3-II Segunda parte del *FullScan* de la validación del bin 41

```
Start with the execution of the Test chainc0_tdo_100_min1200_min1025
The result of the Test chainc0_tdo_100_min1200_min1025 is: 1

The Shmoo is :

*** 2D normal (ASCII Display) Mon Nov 20 15:31:52 2006 ***
Pattern List Name  : st2_chain_c0_narrow_list;
Plan File : D:\T2000\SITEC1ROOT\tmp\SiteC_1\cache\SiteC\SiteC_1\dlls\mtvst2cmt.dll
Plan Name : mtvst2cmt
TestName : chainc0_tdo_100_min1200_min1025

*** Shmoo ***
Header  :
DUT     : 1
          Y: vcore_spec=1.200V(1.2v)|
          delta:150.0mv

               1.200V >|*****
               1.050V  |*****
               900.0mV |*****
               750.0mV |.
               600.0mV +.....
                       +---^
                     5.000nS
                     X: FSBper_spec=10.0nS(10.00nS)
                     delta:1.250nS

Category Definitions
  - : Not Tested   <*> : Pass   < .> : Fail   $ : Over Range
```

Axis	Var/Unit/ Edge/Pattern	Test condition/Pattern_List	Pins	WFS/ PER	WFT	WFC	Start	Stop	Reso	St-eps	equation
X	FSBper_spec	st2_scan_narrow_timing_fsb10	***	***	***	***	5.000nS	10.00nS	1.250nS	5	***
Y	vcore_spec	st2_scan_levels_vcoremin1200	***	***	***	***	600.0mV	1.200V	150.0mV	5	***

Figura 6-4 El *shmoo* de la validación del bin 41

6.1.1.2. Resultados obtenidos gráficamente con el TSS

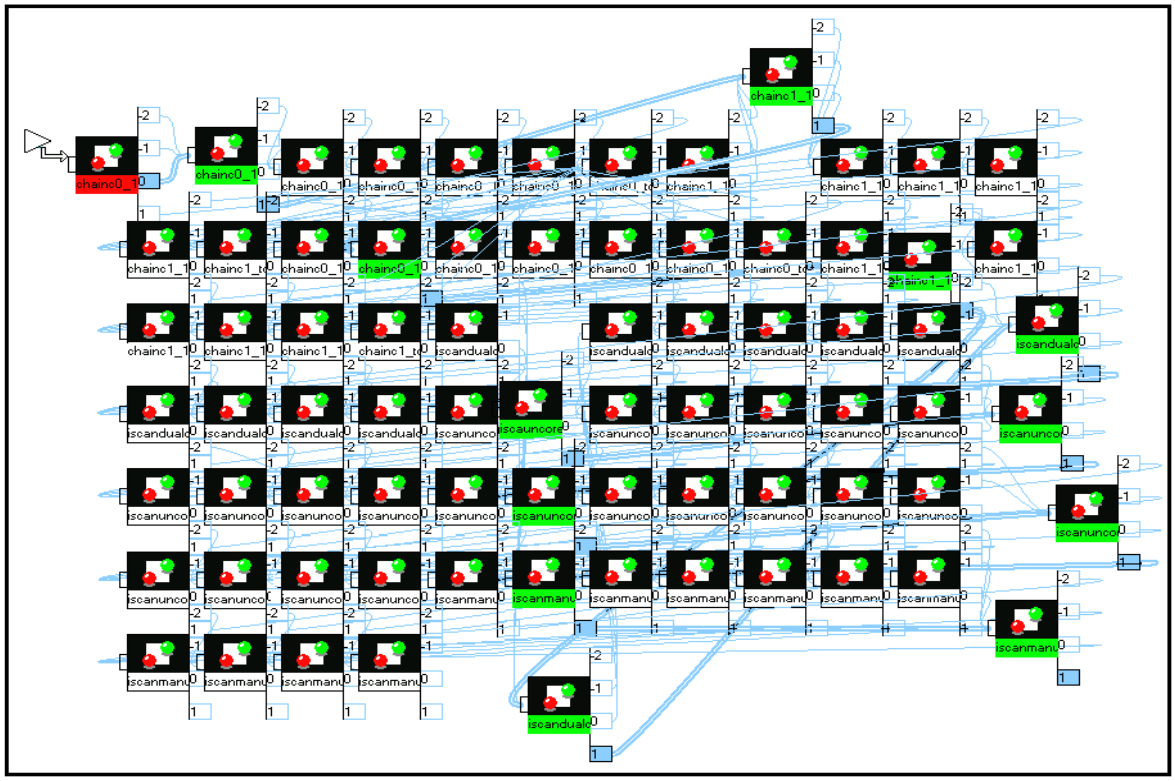

Figura 6-5 Ejecución del flujo Scan con la interfaz del TSS

6.1.2. Resultados de un experimento de ingeniería

6.1.2.1. Resultados obtenidos con la herramienta desarrollada

```
_____________________________________________

     START THE DEDUG-CHARACTERIZATION
_____________________________________________

Start with the execution of the Test WAIT_UNIT
The result of the Test WAIT_UNIT is: 1

Start with the execution of the Test GET_VISUALID
The result of the Test GET_VISUALID is: 1

****************************************
      THE UNIT VISUAL ID IS: 2L640Y00Z0038
****************************************

Start with the execution of the Test MON_FUSE_READ_M1
The Test MON_FUSE_READ_M1 have more than one Flow Item
The result of the Test MON_FUSE_READ_M1 is: 1

****************************************
    THE UNIT ULT IS: W6356328_351_-01_+02
****************************************
_____________________________________________________
 THE RESULT OF THE TEST iscandualcore_100_min1200_min1025_10TCK
_____________________________________________________
```

Figura 6-6 Identificación de la unidad

Tabla 12 Comparación del archivo de orden de trabajo

Visual ID	ULT	Methodology	TEST	SOFTBIN
2L633914Z0126	-	_MetoAlex3_	_AlexTest_	-
-	W6356328_351_-01_+02	_MetoAlex2_	_Test1_	-
2L640Y00Z0661	-	-	-	b41_06
All	-	_MetoAlex3_	_Test1_	-

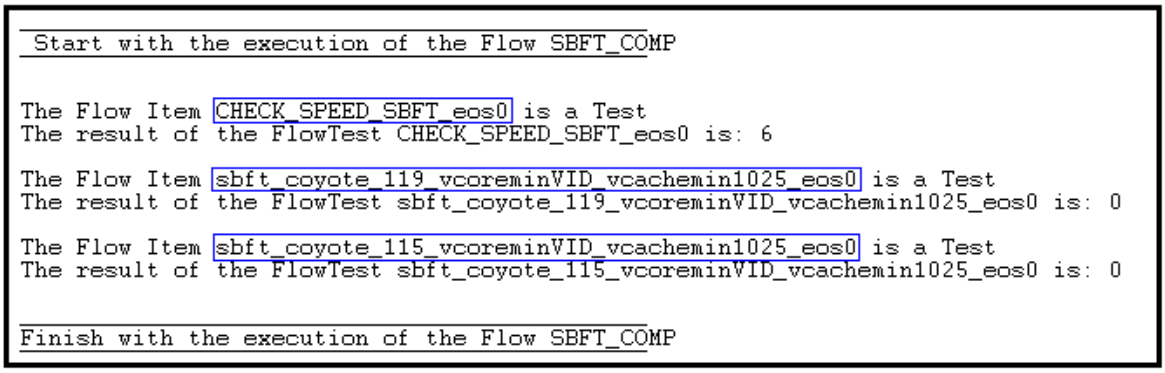

Figura 6-7 Ejecución del flujo de SBFT con la herramienta desarrollada

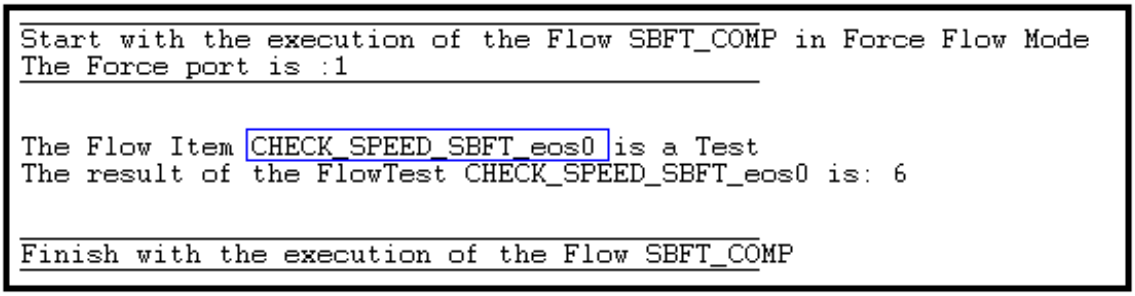

Figura 6-8 Ejecución del flujo forzado de SBFT con la herramienta desarrollada

```
Start with the execution of the Test iscandualcore_100_min1200_min1025_10TCK
The result of the Test iscandualcore_100_min1200_min1025_10TCK is: 1

The Shmoo is :

*** 2D all_fails (ASCII Display) Tue Nov 28 18:58:42 2006 ***
Pattern List Name  : st2_scan_list_10TCK;
Plan File : D:\T2000\SITEC1ROOT\tmp\SiteC_1\cache\SiteC\SiteC_1\dlls\mtvst2cmt.dll
Plan Name : mtvst2cmt
TestName : iscandualcore_100_min1200_min1025_10TCK

*** Shmoo ***
Header  :
DUT     : 1
          Y: vcore_spec=1.200V(1.2V)
          delta:600.0mV

               1.200V >|***
              600.0mV  +***
                       +-^
                     5.000nS
                     X: FSBper_spec=10.0nS(10.00nS)
                     delta:2.500nS

Category Definitions
  - : Not Tested   <*> : Pass   < .> : Fail   $ : Over Range


----+------------+----------------------------+---------+-----+-----+------+---------+---------+---------+---+----------
Axis|Var/Unit/   |Test condition/Pattern_List |Pins     | WFS/| WFT | WFC  | Start   | Stop    | Reso    |St-| equation
    |Edge/Pattern|                            |         | PER |     |      |         |         |         |eps|
----+------------+----------------------------+---------+-----+-----+------+---------+---------+---------+---+----------
X   |FSBper_spec |st2_scan_timing_fsb100_scan |***      |***  |***  |***   |5.000nS  |10.00nS  |2.500nS  |3  |***
Y   |vcore_spec  |st2_iscan_levels_vcoremin120|***      |***  |***  |***   |600.0mV  |1.200V   |600.0mV  |2  |***


All fails(normal) report

---+---+---+
x  |y  |z  |Fail Patterns(Fail addresses)
---+---+---+
```

Figura 6-9 El shmoo del experimento de ingeniería _MetoAlex2_

6.1.2.2. Resultados obtenidos gráficamente con el TSS

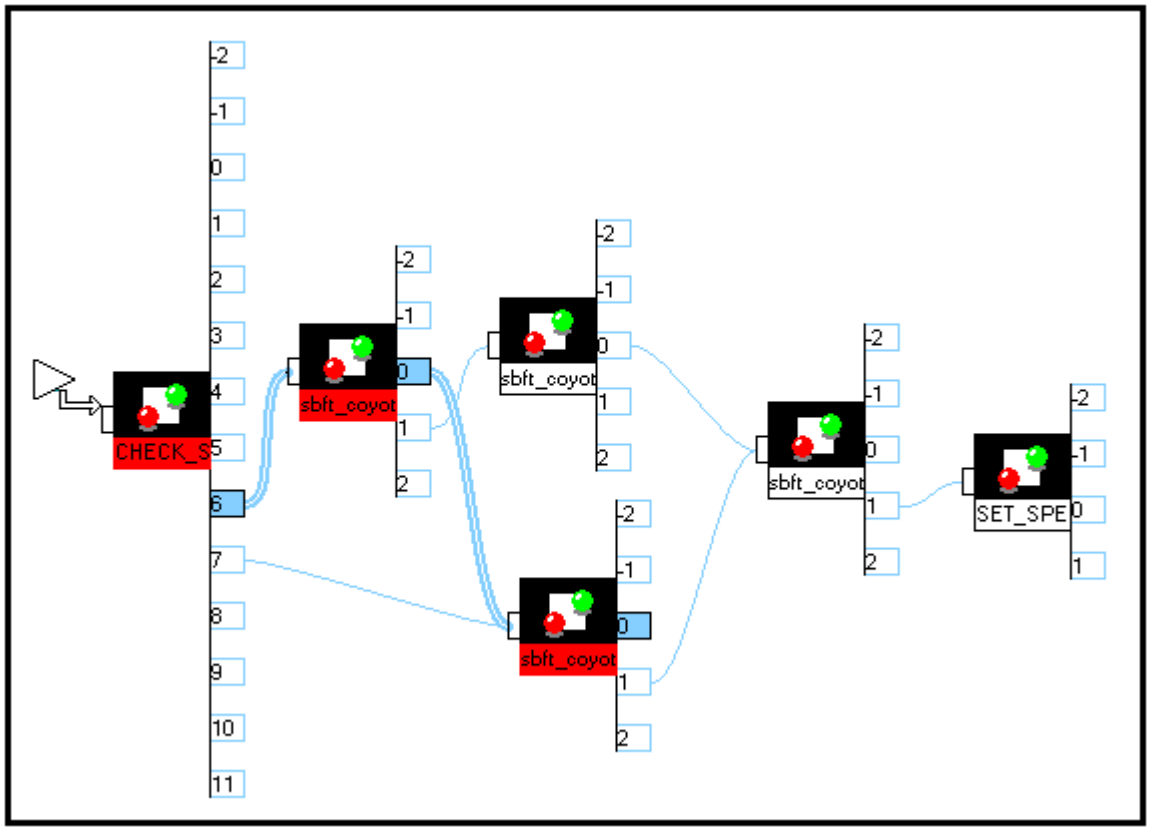

Figura 6-10 Ejecución del flujo SBFT con la interfaz del TSS

6.2. Análisis

Al igual que los resultados experimentales dividiremos esta sección en dos: el análisis de los resultados de la validación y el análisis de los resultados del experimento de ingeniería.

Partimos del hecho que una unidad esta acoplada en el *handler* y se ha alcanzado la temperatura de prueba, después de eso, se ejecuta la herramienta que se desarrollo para este proyecto.

Después de realizar la configuración inicial y la lectura de los archivos de entrada, la herramienta procede a identificar la unidad que se esta probando, tal como se comentó en el capítulo 5 del presente informe, en la figura 6-1, se puede observar el resultado del proceso de identificación de la unidad, de la cual, se puede inferir que fue posible la lectura del Visual ID y del ULT y que estos tiene un valor 2L640Y00Z0661 y W6356328_085_+04_+03 respectivamente, es importante mencionar que dicha figura es un fragmento del archivo de salida de la herramienta desarrollada.

Seguidamente, la herramienta se encarga de comparar los valores de Visual ID y ULT leídos como los definidos por el usuario en el archivo de orden de trabajo en la tabla 11 se puede observar el resultado de la comparación, en la cual, se puede notar que para dicha unidad el usuario definió un SoftBin b41_06, lo cual, implica que la herramienta debe realizar una validación y que el HardBin a utilizar es b41.

Luego, la herramienta utiliza el SoftBin para determinar que la prueba que presentó problemas es *chainc0_tdo_100_min1200_min1025*, tal cual se observa en la figura 6-1, además utiliza el HardBin para leer las instrucciones especificadas en el archivo de validación para dicho grupo de fallas y de esta manera poder iniciar con la ejecución del procedimiento de validación.

En la figura 5-4, se puede observar el procedimiento definido para realizar una validación de un *HardBin* b41, de la cual, se puede notar que la primera instrucción ha ejecutar es *runFlow(Scan_comp)* y en la figura 6-2, se puede observar el resultado obtenido de la ejecución del flujo de Scan, al igual que la figura 6-1, esta representa un fragmento del archivo de resultados que la herramienta produce de manera automática.

Ahora bien, si comparamos la figura 6-2 con la figura 6-5, en la cual, se muestra el resultado gráfico de la ejecución del flujo de Scan por medio de la interfaz gráfica del TSS, podemos observar que en ambas se presenta la misma secuencia de ejecución y que las pruebas ejecutas presentan los mismos resultados por lo tanto, podemos concluir que el algoritmo desarrollado para implementar la ejecución dinámica de un flujo trabaja correctamente.

Una vez que se concluye con la ejecución de la instrucción *runFlow(Scan_comp)*, la herramienta lee la siguiente instrucción, el cual es *FullScan()*, como esta instrucción no tiene como argumento el nombre de la prueba, la herramienta concluye que el usuario desea realizar un *FullScan* de la prueba vincula con el *SoftBin* b41-06, en las figuras 6-3-I y 6-3-II, se muestran los resultados obtenidos con la ejecución del *FullScan*, en la cual, se puede observar que todos los patrones de prueba que pertenecen a la prueba *chainc0_tdo_100_min1200_min1025,* fueron ejecutas y solo el patrón *SP0_C0_rand_r1_for_conv* de los 72 restantes presentó problemas.

Posteriormente, la herramienta lee la próxima instrucción de la metodología de validación el cual es *shmoo (FSBper_spec,5nS, 10nS,5,vcore_spec,0.6V,1.2V, 5,normal)*, lo cual, implica que la herramienta debe ejecutar un *shmoo*, en el cual, el eje X debe ser el parámetro *FSBper_spec* que corresponde a la velocidad en el bus y el eje Y debe ser el parámetro *vcore_spec*, el cual, corresponde a la tensión en el núcleo del procesador.

En la figura 6-4, se puede observar la gráfica del *shmoo*, en la cual, se puede notar que el valor máximo del eje Y es 1.2V y que el mínimo es 600mV, además, se pueden observar 5 incrementos del valor mínimo al máximo en dicho eje tal cual el usuario definió con los argumentos de la instrucción *shmoo*.

De igual manera, se puede observar que el valor máximo del eje X es 10ns, el valor mínimo es 5ns y que existen 5 incremento desde el valor mínimo hasta el valor máximo, tal cual, el usuario lo definió con los argumentos de la instrucción *shmoo*, por lo tanto, podemos concluir que la herramienta desarrollada tiene la capacidad de ejecutar de manera automática un *shmoo*. La figura 6-4 también corresponde a un fragmento del archivo de resultados que la herramienta desarrollada crea de manera automática.

Con la ejecución del *shmoo*, se termina la ejecución de la metodología de validación para el bin b41_06, tal cual se puede observar en la figura 5-5, es decir, la rutina que ejecuta la orden de trabajo termina su ejecución y retorna a la rutina que identifica la unidad, la cual, también concluye su ejecución ya que en usuario definición en el archivo de configuración que no desea realizar el manejo automático de las unidades, tal como se puede observar en la Tabla 1.

Por último, como la herramienta utiliza el *SoftBin* para identificar la falla que presenta problemas y la validación de la misma se inicia justo de después de la ejecución de la prueba fallida, se puede concluir que fue posible modificar el flujo por defecto del programa de prueba para implementar un proceso de validación automática de la falla.

Hemos concluido con el análisis de los resultados de la validación por lo tanto, procederemos con el análisis de los resultados del experimento de ingeniería, como se explicó en el capítulo 5 del presente informe, el principio de ejecución de un experimento de ingeniería es prácticamente igual al principio de ejecución de una validación, la diferencia radica en que se utilizan archivos diferentes para definir la metodología de prueba y que para un experimento de ingeniería es necesario especificar el nombre del las pruebas con la que se desea trabajar.

Por lo tanto, el primer paso del experimento de ingeniería después de que la herramienta realiza el proceso de inicialización y configuración inicial, es la identificación de la unidad, en la figura 6-6, se puede observar el resultado del proceso de identificación de la unidad, de la cual, se puede concluir que la herramienta fue capaz de realizar la lectura del Visual ID y del ULT y que estos tiene un valor 2L640Y00Z0038 y W6356328_351_-01_+02 respectivamente.

Al igual que en la validación, el siguiente paso es determinar cual procedimiento el usuario definió para esta unidad y a cuales pruebas se debe utilizar para la ejecución de las instrucciones que constituyen el experimento de ingeniería., en la Tabla 12, se muestra el resultado de la comparación del Visual ID y del ULT de la unidad con los definidos por el usuario en el archivo de orden de trabajo.

En dicha tabla, se puede observar que el usuario desea realizar la Metodología etiquetada como *_MetoAlex2_*, la cual debe estar definida en el archivo de instrucciones y que desea utilizar las pruebas vinculadas con la etiqueta *_Test1_*, la cual, debe estar definida en el archivo de etiquetas.

Seguidamente, la herramienta utiliza la etiqueta *_MetoAlex2_* para leer las instrucciones que el usuario definió como el experimento de ingeniería y utiliza la etiqueta *_Test1_* para leer el nombre de las prueba que el usuario definió para utilizar en el experimento.

En la figura 5-6, se muestra el archivo de etiquetas, en el cual la herramienta utiliza la etiqueta de *_Test1_* para obtener el nombre de las pruebas que debe utilizar en este caso *iscandualcore_100_min1200_min1025_10TCK*, de la misma manera, la herramienta utiliza la etiqueta _MetoAlex2_ para buscar en el archivo de instrucciones la secuencia de instrucciones que el usuario definió, tal cual se puede observar en la figura 5-4.

Según la figura 5-4, la primera instrucción que la herramienta debe ejecutar es el *runFlow (SBFT_COMP),* en la figura 6-7, se muestran los resultados obtenidos con la ejecución de dicha instrucción, si comparamos dicha figura con la figura 6-10, podemos notar que ambas tiene la misma secuencia de ejecución del flujo de SBFT y que los resultados de las pruebas ejecutas son los mismos en ambas figuras, por lo tanto, podemos concluir que la herramienta y el algoritmo de ejecución dinámica implementado trabaja correctamente.

La siguiente instrucción en la ejecución del experimento de ingeniería es *runForceFlow(SBFT_COMP,1)*, lo cual, implica que la herramienta deje ejecutar el flujo SBFT forzando el resultado o puerto de salida de la prueba al valor de 1, en la figura 6-8, se muestran los resultados obtenidos con la ejecución de dicha instrucción.

Si comparamos la figura 6-8 con la figura 6-10, podemos notar que en la ejecución del flujo forzado únicamente se ejecutó la primera prueba del flujo, es decir, *CHECK_SPEED_SBFT_eos0*, esto se debe a que en la definición del flujo SBFT en el programa de prueba no se especificó una siguiente prueba para el puerto 1 de la primera prueba del flujo, tal cual se puede observar en la figura 6-10 y por ende los resultados obtenidos son correctos, es importante aclarar que con la interfaz gráfica del TSS no es posible ejecutar un flujo forzado, por lo tanto, esta es una buena función que se introdujo con la herramienta desarrollada.

La última instrucción que el usuario definió para el experimento de ingeniería _MetoAlex2_, es el shmoo (_Alex1_), para poder ejecutar esta instrucción, la herramienta debe leer del archivo de etiquetas la configuración del shmoo que el usuario definió como _Alex1_.

En la figura 5-6, se puede observar que la configuración del shmoo que el usuario desea es: *FSBper_spec, 5nS, 10nS, 3, vcore_spec, 0.6V, 1.2V, 2, all_fails*, lo cual implica, que el eje X del *shmoo* se debe se el parámetro FSBper_spec y el eje Y del *shmoo* debe ser el parámetro *vcore_spec*, tal cual, se puede observar en la figura 6-9.

También en la figura 6-9, se puede observar que el eje X tiene tres valores de *FSBper_spec* muestreados, esto se debe a que en la definición de la configuración del *shmoo* se especificó tres incrementos desde el valor mínimo (5ns) hasta el valor máximo (10ns) incluyendo estos dos valores limites, además, se observa que el eje Y tiene muestreadas únicamente los valores limites de 1.2V y 600mV, esto se debe a que se definieron únicamente dos incrementos para dicho eje los cuales corresponden a los dos valores límites.

Por último, al ser el *shmoo* la última instrucción definido por el usuario, con la ejecución de este termina la ejecución de la herramienta, además, es importante que el lector tenga noción con la concatenación de las figuras 6-6, 6-7, 6-8 y 6-9, se obtiene el archivo de resultado que la herramienta crea de manera automática, el cual, posteriormente es utilizado por un ingeniero para realizar un análisis detallado de los resultados y poder establecer las conclusiones pertinente, recordemos, que el fin del proyecto es obtener una herramienta que permita automatizar la colección de los resultados de las validaciones y los experimentos de ingeniería.

Otra cosa que el lector debe tener conciencia, es que los *shmoo* que se presentan como resultados experimentales son con fines demostrativos, es decir, nunca se realizan en las pruebas de ingeniería gráficos en baja resolución, la razón por la cual se realizaron, fue que se tuvo un tiempo limitado de utilización del *tester CMT* y para obtener un *shmoo* de alta definición se requiere de al menos 30 minutos.

Es importante mencionar que durante el desarrollo del proyecto, se contempló la posibilidad de captura de manera automática la información que el TSS imprime en la consola del *Site Controller* ya que esta es fundamental para realizar una correcta caracterización de la falla.

Se estudió la posibilidad de hacer uso de las instrucciones de *UserSDK Tools API* para implementar una rutina que sea capaz de captura estos resultados he almacenarlos en el archivo de salida de la aplicación, sin embargo, después de realizar varias consultas a los ingenieros de soporte de *Advantest*, se llegó a la conclusión que no era posible con las instrucciones disponibles.

También, se contempló la posibilidad de modificar la definición de las funciones que utiliza *Cortex* para imprimir en la consola, esto con el fin de poder redireccionar esta información a un archivo de texto en el *Site Controller* y por medio de la red leer desde el *script* que se encuentra en el *System Controller* dicha información.

Sin embargo, después de realizar una consulta a los ingenieros desarrolladores de *Cortex* se concluyó que esta implementación se alejaba de los alcances del proyecto, ya que según la normativa interna de Intel, la modificación de dichas funciones es responsabilidad de los desarrolladores de *Cortex* y por ende, se debe esperar una buena versión que contemple las modificación propuestas.

Capítulo 7: Conclusiones y recomendaciones

7.1. Conclusiones

1. Para poder utilizar las instrucciones de *Helper Class* desde un *script* en PERL se requiere utilizar las instrucciones de *UserSDK Tools API* con el fin de establecer un servicio desde el *script* con el TSS.
2. Todas las pruebas del programa de prueba desarrolladas con *Cortex*, utilizan el método "*sendConsoleMessage*" para imprimir la información en la consola del *Site Controller*.
3. No existe un método vinculado con el objeto "*MsgHandlerServerBase*" de las instrucciones de *UserSDK Tools API* que permita leer la información que se imprima en el *Site Controller*.
4. Es posible crear una estructura de memoria por medio del análisis sintáctico del archivo TPL, para ejecutar un flujo del programa de prueba desde un *script* en PERL.
5. Es posible crear un archivo de texto en el *Site Controller* de tipo "*datalog stream"* por medio del objeto "*IDatalogManagerProxy"* de los *UserSDK Tools API*, sin embargo, no fue posible vincular este archivo con la información de la última prueba que la herramienta ejecutó en el *Tester CMT.*
6. La herramienta desarrollada permite definir una secuencia de instrucciones para ejecutar de manera automática una metodología de prueba, sin embargo, la información que la herramienta recolecta y almacena en el archivo de salida es insuficiente para caracterizar una falla.
7. Es posible desde un *script* en PERL tener control absoluto de la ejecución de las pruebas del programa de prueba y de esa manera modificar dinámicamente el flujo del este.

7.2. Recomendaciones

Para lograr la automatización de la captura de la información que el TSS imprime en la consola del *Site Controller* se recomienda:

- Modificar la definición de las funciones que utiliza *Cortex* para imprimir en la pantalla, es decir, modificar en la definición de las funciones *iC_zPrint, iC_tPrintZDatalog*, entre otras que se realiza en el programa *OASIS_core.cpp*, para que estas no solo utilice los *UserSDK Tools AP*I para imprimir en la consola sino que también envié esa información a un archivo de texto para poder ser acezada desde el *script* en PERL.
- Investigar como se puede vincular un archivo tipo "*datalog stream"* como el tipo de *datalog* de cada prueba, ya que de esta manera talvez se pueda lograr que toda la información que se produce con la ejecución de la prueba se almacene en el archivo tipo "*datalog stream".*
- La otra posibilidad es no utilizar la consola del TSS, sino crear una consola propia que sustituya la consola por defecto del *Site Controller* y de esta manera poder acceder la información que el TSS envía a la consola del *Site Controller* por medio de los *UserSDK Tools API* y las propiedades de la mueva consola.

Además, se recomienda implementar nuevas funciones que permita reducir la intervención humana y que aumenten el grado de automatización e inteligencia de la herramienta desarrollada, tales como:

- Implementar una función que permita interpretar la información del *shmoo*, es decir, que tenga la capacidad de detectar fallas sólidas, fallas marginales, pisos, techos y paredes en la gráfica del *shmoo*.

- Automatizar la carga de los patrones de pruebas que se desean utilizar en la validación o el experimento de ingeniería, para lo cual, se debe configurar el *Tester* en modo de producción para que este utiliza una lista de patrones de prueba que se define en un archivo de texto. Para esto, se podría utilizar el método "*setProductionMode"* del objeto "*ISystemControllerProxy"* de los *Tools API*, sin embargo, se debe investigar la secuencia previa y posterior a la utilización de dicho método.

Bibliografía

1. Advantest Corporation. *T2000 Basic Applications Course Student Workbook. 2003. [Intranet]:*

   <http://tmgtraining.intel.com/trainingmaterials/Advantest/1%20T2000-CMT%20Tester/Prev%20Equip%20Docs/CMV021292_%20T2000_Basics_Apps_Manual%20rev%201.2.%20April%2004.pdf> [Consulta: 12 Julio 2006]

2. Campos, H. *CMT Hardware Overview*. SDM-CR PDE Team, 2005. [Intranet]:

   <\\Crspublic002\Shareddata\EMG\PDE\Training\IDP\CMT_Trainig\2005\CMT_training_day_1.ppt> [Consulta: 12 Julio 2006]

3. Jiménez, P. *Levels on CMT Tester*. SDM-CR PDE Team, 2005. [Intranet]:

   <\\Crspublic002\Shareddata\EMG\PDE\Training\IDP\CMT_Trainig\2005\Levels on CMT Tester.ppt> [Consulta: 14 Julio 2006]

4. Jiménez, P. *Pindef on CMT Tester*. SDM-CR PDE Team, 2005. [Intranet]:

   <\\Crspublic002\Shareddata\EMG\PDE\Training\IDP\CMT_Trainig\2005\Pindef on CMT Tester.ppt> [Consulta: 14 Julio 2006]

5. Jiménez, P. *Timing on CMT Tester*. SDM-CR PDE Team, 2005. [Intranet]:

   <\\Crspublic002\Shareddata\EMG\PDE\Training\IDP\CMT_Trainig\2005\timing on CMT Tester.ppt> [Consulta: 14 Julio 2006]

6. Ramírez, M. *CMT Introduction.* SDM-CR PDE Team, 2005. [Intranet]:

   <\\Crspublic002\Shareddata\EMG\PDE\Training\IDP\CMT_Trainig\2005\CMT_Intro.ppt> [Consulta: 12 Julio 2006

7. Soto, L. *Tester Hardware training*. SDM-CR PDE Team, 2005. [Intranet]: <\\Crspublic002\Shareddata\EMG\PDE\Training\IDP\General Test Training\New Hire Tester Hardware training_1.ppt > [Consulta: 17 Julio 2006]

8. Advantest. *T2000 ToolsAPI Programming Seminar*, Advantest Corporation, Octubre 2006

9. Advantest. *T2000 System Software Helper Class Tool User's Guide*, Advantest Corporation, Octubre 2004

10. Smith, J.E. *An overview of Virtual Machine Architectures*, 2004 [Online]:< http://www.cis.upenn.edu/~cis700-6/04f/papers/smith-vm-overview.pdf > [Consulta: 9 Agosto 2006]

11. Wikipedia the free encyclopedia. *Interpreter (computing),* [Online]: <http://en.wikipedia.org/wiki/Interpreter_%28computer_software%29> [Consulta: 6 Agosto 2006]

12. Garlan, D. *An Introduction to Software Architecture*, 1994 [Online]: <http://www.cs.cmu.edu/afs/cs/project/vit/ftp/pdf/intro_softarch.pdf> [Consulta: 10 Agosto 2006]

13. Risto, S. *Programming Languages mini-HOWTO*, 200 [Online]:< http://www.ibiblio.org/pub/Linux/docs/HOWTO/other-formats/pdf/Programming-Languages.pdf > [Consulta: 13 Agosto 2006]

14. Serrano, K. *SBFT basics for Montecito*. SDM-CR PDE Team, 2005. [Intranet]: <\\Crspublic002\emg\PDE\Training\IDP\SBFTtraining\Intro_SBFT.ppt> [Consulta: 11 Septiembre 2006]

15. Solano, J. *iSCAN Methodology.* SDM-CR PDE Team, 2005 [Intranet] : <\\Crspublic002\emg\PDE\Training\IDP\SCAN_training\Scan_Material\Scan Basic.ppt> [Consulta: 18 Septiembre 2006]

16. Intel. *Chapter 9, SRAM and Caché Testing*. Intel Confidential [Intranet]: <http://itmh.intel.com/pdf/chapter9.pdf > [Consulta: 26 Septiembre 2006]

17. Morales, R. *Caché Testing Methodology.* Intel Confidential, Product Development Engineering, 2006

Apéndices

A.1 Glosario, abreviaturas y simbología

- *Bin de falla*: es un código único que identifica el tipo de falla y la prueba en la cual se produjo la falla
- *DUT*: las siglas en ingles de *Device Under Testing*, es un acrónico que se utiliza para referirse a la unidad que esta siendo probada
- *EMG*: las siglas en ingles de *Enterprise Microprocessor Group*, es el departamento del área de microprocesadores de Intel donde se desarrolla en proyecto
- *Handler:* una máquina que se encarga de controlarla temperatura y la presión de las pruebas que se realizan a las unidades, además, se encarga de manipular y clasificar las unidades que son probadas
- *Helper Class:* un conjunto de instrucciones que permite controlar desde una consola la operación del Tester CMT, es decir, ejecutar pruebas y modificar las condiciones de estas
- *Ituff:* un archivo de formato ASCII que contiene los resultados obtenidos con la ejecución de un programa de prueba
- *SDM:* las siglas en ingles de Server Development and Manufacture, la subdivisión del departamento de EMG de Intel donde se realiza el proyecto
- *Tester CMT:* las siglas en ingles de Configurable Modular Tester, es la plataforma de pruebas que utiliza el departamento para probar las unidades
- *Programa de prueba:* es el programa que controla las condiciones de cada una de las pruebas que se realizan a las unidades, además define el flujo que deben seguir las pruebas a realizar

- *Montecito:* Es el nombre que utiliza Intel para referirse al microprocesador de la familia Itanium II, el cual, tiene 24 MB de caché y 1.7 billones de transistores
- *Montvale:* Es el nombre que utiliza Intel para referirse a la versión mejora de Montecito, este microprocesador también tiene 24MB de caché y 1.7 billones de transistores, solo que Montvale será probado únicamente con el Tester CMT
- *Patterns:* cada uno de los vectores de la tabla de verdad que se utiliza para probar el correcto funcionamiento de la unidad
- *Pruebas de Caché:* son pruebas que se realizan a todas las diferentes memorias que se encuentran dentro del microprocesador (L0-L2)
- *Pruebas de SBFT:* las siglas en ingles de *Structural Basic Funtion Test*, son pruebas estructurales que se realizan para determinar el correcto básico funcionamiento de un microprocesador
- *Pruebas de Scan*: son pruebas combinacionales para determinar el estado de los *flip-flops* que se encuentran alrededor del procesador
- *Pruebas Estructurales*: Verifican la integridad física de los módulos internos del procesador, es decir, comprueban el estado del hardware asociado a los módulos en prueba.
- *Pruebas Funcionales*: Verifican el correcto funcionamiento de uno o más módulos internos del procesador, es decir, comprueban el correcto funcionamiento de la lógica relacionada con los módulos en prueba
- .*Shmoo*: son barridos de una o más condiciones de las prueba que se realizan a una unidad, por ejemplo, se realizan *shmoo* frecuencia contra tensión para caracterizar el comportamiento de la unidad al variar estos dos parámetros

- *Full Scan*: es un herramienta que realiza pruebas funcionales para determinar cuales de los *patterns* que se están corriendo en una prueba presentan problemas
- *Site Controller*: es el control centralizado de todos los módulos de hardware programable que utiliza el Tester CMT para realizar un conjunto de pruebas a una unidad.
- *Software TSS*: es el software que permite la comunicación entre el *System Controller* y el *Site Controller*
- *System Controller*: una computadora que permite la interacción entre el usuario y el *Site Controller*
- *TIU*: las siglas en ingles de Tester Interfase Unit, es una tarjeta que realiza el acople entre la cabeza de pruebas del Tester CMT y la unidad que se desea probar
- *Visual_ID*: es un código único que identifica a cada unidad
- *Wafer*: es un círculo de silicio que contiene muchos circuitos independientes, cada uno de los cuales constituye la unidad de control de un microprocesador

A.2 Información sobre la empresa

En marzo de 1998, la empresa Componentes Intel de Costa Rica inicio sus operaciones en La Ribera de Belén, ubicada en la provincia de Heredia. Las instalaciones se dividen en dos plantas de manufactura y un centro de distribución, estas constituyen las únicas instalaciones de Intel en Latinoamérica, lo cual representa una inversión acumulada de aproximadamente $400 millones y unas 2500 fuentes de empleo directas para Costa Rica.

La planta de Componentes Intel de Costa Rica se dedica exclusivamente al ensamblaje y prueba de microprocesadores y *chipsets*, los cuales, son los dos principales productos manufacturados por Intel a nivel mundial.