Generador Automático de Código que Describe Máquinas de Soporte Vectorial para Implementarlas en Hardware Re-configurable

Omar Piña Ramírez

22 de marzo de 2011

UNIVERSIDAD AUTÓNOMA METROPOLITANA

Casa abierta al tiempo

Unidad Iztapalapa

Generador automático de código que describe máquinas de soporte vectorial en hardware reconfigurable

---

TESIS QUE PRESENTA

OMAR PIÑA RAMÍREZ

PARA OBTENER EL GRADO DE

MAESTRO EN CIENCIAS (INGENIERÍA BIOMÉDICA)

Autor:

Ing. Omar PIÑA RAMÍREZ

Asesores:

Dr. Humberto CERVANTES MACEDA

Dra. Raquel VALDÉS CRISTERNA

Jurado calificador:

Presidente: Dr. Miguel Octavio ARIAS ESTRADA INAOE.

Secretario: M. en I. Oscar YAÑÉZ SUÁREZ UAM-I.

Vocal: Dr. Humberto CERVANTES MACEDA UAM-I.

22 de marzo de 2011

Índice general

Agradecimientos

Esta tesis está dedicada a Amelia Carrillo Hernandez a quien de debo lo que soy.

Te agradezco Ranulfo por haberme inculcado la lectura y todas las platicas amenas que me has regalado.

Dra. Yola te agradezco por ser la mejor mamá del mundo y por alentarme en todas mis locuras y experimentos.

Quiero dedicar esta tesis a mi persona favorita del mundo a mi hermano Moy.

También quiero agradecer a mi papá por no haber claudicado en la reposición de todos los componentes electrónicos que queme.

Te agradezco Mat. Ana Irene Tovar Ehlers por compartir tantos momentos juntos y haberte convertido en la mujer de mi vida. Te amo Nauz

Les agradezco a quienes he llamado mis padres académicos: Raquel gracias por mostrarme el maravilloso mundo digital. Juan Carlos Echeverria gracias por tu entusiasmo en clase. Salvador Carrasco gracias por las magnificas clases de fisiología. Y gracias Herr Oscar, que a pesar de que nunca he sido tu alumno formal de ti he aprendido mucho más que nadie.

También quiero agradecer a Irene, a Mauricio y al dios de Mauricio por tantas platicas agradables y constructivas que se realizaron en el laboratorio, de ellas surgieron muchas ideas, muchos proyectos y muchos libros que leer.

No me podía olvidar de agradecer a mi compañeros del LINI, al Teo, el Erit y el Gabo.

Finalmente quiero agradecer a CONACyT por su valioso apoyo para realización de mi maestría

Prefacio

La presente es la idónea comunicación de resultados del proyecto denominado *Generador Automático de Código que Describe Máquinas de Soporte Vectorial para Implementarlas en Hardware Re-configurable*, el cual fue desarrollado por el I.B. Omar Piña Ramírez para obtener el grado de Maestro en Ciencias (Ingeniería Biomédica). El proyecto se desarrolló en el Laboratorio de Investigación en Neuroimagenología en la Universidad Autónoma Metropolitana Unidad Iztapalapa, en el periodo comprendido entre Enero de 2007 y Marzo de 2008. Los asesores para la realización del mismo fueron el Dr. Humberto Cervantes Maceda y la Dra. Raquel Valdés Cristerna.

Capítulo 1

Introducción

1.1. Problemática: Discapacidad Motora e Interfaces Cerebro Computadora

Se conoce como Discapacidad Motora a la pérdida parcial o total del control neuro-muscular. Quienes la padecen no pueden ejecutar ni total ni parcialmente movimientos voluntarios. En los casos más graves se manifiesta como una parálisis total. La discapacidad motora es ocasionada por diversas patologías, entre las que se encuentran: la esclerosis múltiple, paraplegia, hemiplegia, parálisis cerebral, daños en la columna vertebral, etc [24].

A la discapacidad motora se le considera un problema de salud pública, ya que según datos de la Organización Mundial de la Salud (OMS), en todo el mundo existen aproximadamente 600 millones de personas que la padecen. La incidencia en hombres en edad productiva (25 - 45 años) es del 65 %, el resto se reparte entre mujeres, niños y personas de la tercera edad. La mayoría de los casos son ocasionados por daños en la columna vertebral que, generalmente, son el resultado de accidentes automovilísticos o laborales; también puede ser resultado de violencia doméstica o social, entre otras [4, 24].

El 80 % del total de los afectados con discapacidad motora son personas de escasos recursos, que no cuentan con los servicios de salud adecuados que les brinden el cuidado y atención que su estado requiere[1]. En México, hay aproximadamente 2 millones de personas que padecen discapacidad motora, la mayoría de ellos de escasos recursos y sin un programa de seguridad social que soporte el tratamiento de su padecimiento [24].

[1] Información extraída de un informe publicado en internet por la OMS [24]

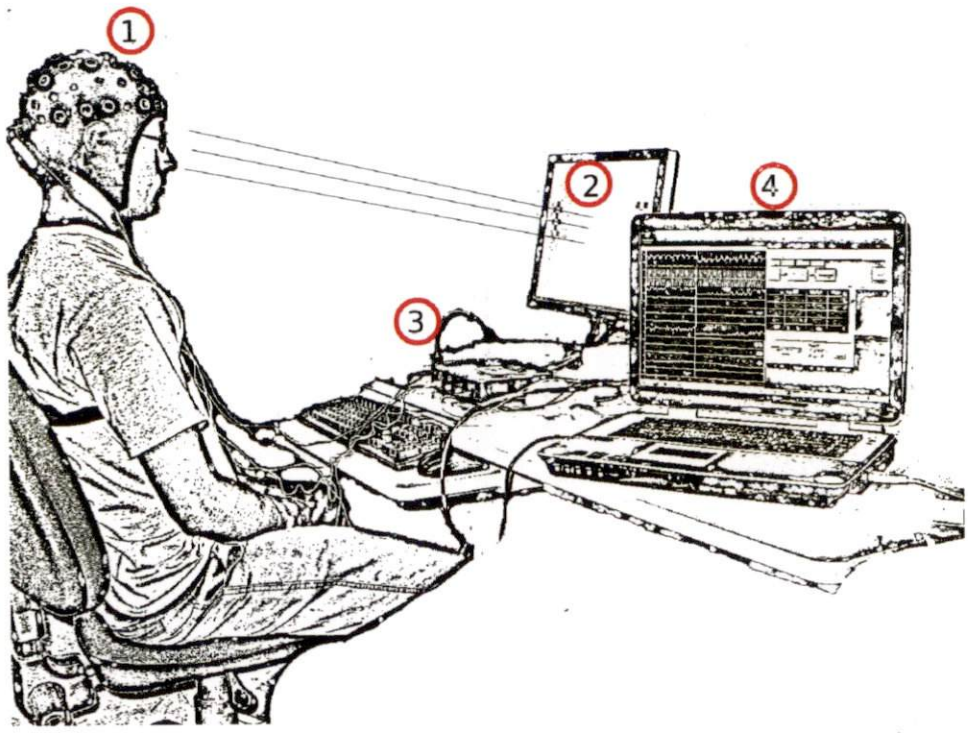


Figura 1.1 – *Elementos que componen a una BCI. Adquisición (1. Electrodos, 2. Estimulación, 3. Amplificadores), Procesamiento (4. Visualización de las señales y procesamiento) y Ejecución de Comando en la figura no se muestran pero pueden ser controladores de sillas de ruedas, deletreadores, inclusive, controladores de prótesis robóticas*

1.1.1. Interfaces Cerebro Computadora

Para ayudar a los pacientes que padecen discapacidad motora grave, surgieron las Interfaces Cerebro-Computadora (BCI Brain-Computer Interfaces), que son sistemas que extraen información de la actividad eléctrica cerebral de los pacientes[2] e identifican si en ella están presentes patrones bien definidos que se interpretan como comandos, p. ej. mover una silla de ruedas o un brazo mecánico, incluso deletrear una palabra en la computadora [27,47].

Las BCI se componen de tres etapas (figura 1.1) que son:

1. **Adquisición de la Señal de Comunicación**. La Actividad Eléctrica Cerebral es producida por el funcionamiento de las neuronas, que puede ser medida usando técnicas como el Electroencefalograma (EEG) que utiliza un arreglo de electrodos fijado a la superficie del cuero cabelludo por medio de una gorra especial (figura 1.1-1). El número de electrodos varia entre 2 y 128, inclusive hasta 256. La forma en que son colocados y su nomenclatura está regida por el estándar 10-20. Se prefiere el uso del EEG a otras técnicas como p. ej. el Electrocorticograma (ECoG) en el cual el arreglo de electrodos se coloca directamente en la corteza cerebral lo que la hace una técnica sumamente invasiva [3].

   Un registro típico de EEG tiene un comportamiento de ruido aleatorio no estacionario cuya magnitud máxima

[2] en la mayoría de los casos de discapacidad motora el cerebro no se encuentra afectado [24]

es del orden de los 10 μV. Ambos factores dificultan la interpretación del registro. Por otra parte, se ha demostrado que cuando se aplican cierto tipo de estímulos visuales, auditivos o somato-sensoriales, en el EEG aparecen potenciales (más o menos regularmente) del orden de los 10 mV, es decir, se provoca una respuesta cerebral a través de un estimulo. A los protocolos para generar este tipo de respuestas se les llama Potenciales Evocados [3, 55].

El P300 es un tipo de Potencial Evocado que se presenta (300 ± 100) $ms.$ después de un estímulo y se caracteriza por ser un potencial positivo de gran amplitud (en promedio 25 μV). El P300 está asociado con el reconocimiento de un estímulo infrecuente y por ello es asociado a la parte cognitiva [14,55]. En este proyecto se trabajó con potenciales evocados P300 de estimulación visual (figura 1.1-2).

2. **Procesamiento**. Se le llama Reconocimiento de Patrones a los métodos que se ocupan para detectar regularidades en un conjunto de datos, para llevarlo a cabo se requieren al menos de dos etapas: la Extracción de Características y la Clasificación. La primera se encarga de obtener la información más relevante del conjunto de datos de interés y lo expresa como un vector llamado vector de características; en la segunda se realiza un proceso automático de asignación de etiquetas, es decir, con base a ciertas reglas se decide a que clase pertenece un vector de características [16–18].

   Se le llama Clasificador al sistema que realiza la asignación automática de etiquetas y su ciclo de vida se compone de tres fases:

   I) **Entrenamiento**. Es la búsqueda de los criterios para discriminar las clases. Entre los clasificadores más usados se encuentran las Redes Neuronales Artificiales y los Clasificadores de Discriminante Lineal cada uno con su propio algoritmo de entrenamiento, que a su vez sigue uno de dos paradigmas; con supervisión o sin supervisión [12].

   Asumiendo que $\mathbb{X}$ es el conjunto de todos los vectores $\mathbb{R}^n$ que resultan de algún método de extracción de características para un experimento dado y que $\mathbb{Y} \subset \mathbb{Z}$ son las etiquetas de las clases a las que pueden corresponder cada uno de los vectores de entrenamiento; entonces se dice que el entrenamiento supervisado parte de un conjunto de entrenamiento formado por las parejas $\{(\mathbf{x}_k, y_k) | x_k \in \mathbb{X}; y_k \in \mathbb{Y}; k \in \mathbb{N}\}$, donde $\mathbf{x}_k$ denota al k-ésimo vector de entrenamiento y y_k es la etiqueta de clase que le corresponde. El objetivo del entrenamiento supervisado es encontrar la función $y(\mathbf{x}) | y : \mathbb{X} \to \mathbb{Y}$, usando el conjunto de entrenamiento, que mejor relaciona los vectores con su respectiva etiqueta de clase, en otras palabras la evaluación de dicha función para un vector $\mathbf{x} \in \mathbb{X}$ da como resultado la clase a la que pertenece. La llamada clasificación binaria es aquella que solo discrimina entre dos clases; donde usualmente $\mathbb{Y} = \{-1, 1\}$.

   De forma general la función de clasificación se describe como $y(\mathbf{x}, \rho)$ donde ρ es el vector de parámetros adicionales que el clasificador requiere para funcionar. El vector ρ depende de cada tipo de clasificador.

   II) **Pruebas y reentrenamiento**. Durante esta fase se evalúa el desempeño del clasificador $y(\mathbf{x})$ para vectores $\mathbf{x}$ distintos a los vectores de entrenamiento, pero cuya clase también es conocida, a estos vectores se les llama vectores de prueba o vectores no vistos. Se le llama *generalización* a la capacidad de los clasificadores de discriminar adecuadamente los vectores no vistos.

   La evaluación de un clasificador consiste en usarlo para clasificar un conjunto de vectores de prueba y contabilizar las discrepancias entre la clase real y la obtenida como resultado de la clasificación. El

resultado de la evaluación se expresa con la *tasa de clasificación* que mide la capacidad de generalización del clasificador y usualmente se calcula como el cociente de la diferencia entre la cuenta total de discrepancias y el número de vectores de prueba entre este último.

La tasa mínima de clasificación permitida depende de la aplicación del clasificador. Si un clasificador presenta tasas por debajo del umbral fijado se pueden seguir varios caminos para aumentarla: El primero es modificar los parámetros de entrenamiento del clasificador; otro es cambiar el tipo de clasificador; y por último se puede utilizar otro método de extracción de características.

III) **Uso del clasificador**. Una vez que el clasificador satisface los requerimientos mínimos de la aplicación, ya puede ser empleado para clasificar vectores de características reales en el entorno de la aplicación para la que fue diseñado. Tomando en cuenta que el método de extracción de características debe ser el mismo con el que se obtuvieron los vectores de entrenamiento y prueba.

Como nota adicional; el procesamiento de las señal de EEG puede realizarse *online*, es decir, durante la adquisición u *offline*; posterior a la adquisición, esta última casi siempre se lleva a cabo con propósitos de investigación [17, 18].

Las Máquinas de Soporte Vectorial (SVM Support Vector Machines) son clasificadores de discriminante lineal, cuya capacidad de generalización es superior a la de clasificadores tradicionales como las Redes Neuronales Artificiales Multi-capa [12]. Las SVM fueron desarrolladas a principios de la década de los 90's y desde entonces su uso se ha ido extendiendo en un amplio campo de aplicaciones [12, 15, 18, 35].

3. **Ejecución de Comandos**. Consiste en interpretar el resultado de la clasificación y actuar en consecuencia, es decir ejecutar alguna acción. Los dispositivos que realizan la ejecución del comando se les conoce, en el argot de las BCI, como actuadores y los que más se utilizan son los deletreadores (deletrear palabras para escribir texto o síntesis de voz), controladores de sillas de ruedas, prótesis mecano-electrónicas, controladores del ambiente de una casa, etc [14, 27, 30, 47].

Se han desarrollado BCI que utilizan Potenciales Evocados P300 de estímulo visual como el lenguaje de comunicación entre el cerebro y la computadora. De manera general el protocolo de adquisición consiste en realizar varios estímulos formados de ciclos aleatorios de iluminación-oscurecimiento de distintas porciones de una pantalla. El P300 se presenta cada vez que se ilumina la parte de la pantalla en la que el sujeto está enfocando su atención.

En síntesis: La Adquisición se encarga del registro y acondicionamiento del EEG, el acondicionamiento consiste en minimizar la información espuria inherente al proceso de medición. La Extracción de Características se aplica a segmentos del EEG y se tiene un vector de características por cada segmento, una colección de esos vectores conforman los conjuntos de entrenamiento y prueba del clasificador; la clase real de los vectores del conjunto de prueba está dada manualmente por un experto. Durante la Clasificación se utiliza un clasificador (en este caso SVM) previamente entrenado y evaluado satisfactoriamente; para sistemas BCI las tasas de clasificación típicas son aproximadamente del 75 % [14, 26, 30]. Finalmente dependiendo del resultado de la Clasificación los actuadores ejecutan los comandos correspondientes.

1.1.2. Diseño e Implementación de Sistemas Digitales en un FPGA

Los Sistemas Integrados de Propósito Específico (SIPE), en inglés conocidos como *Embedded Systems*[3] han tenido un gran auge en los últimos años p. ej. celulares, reproductores portátiles de música o video, aparatos portátiles de uso médicos o doméstico. Como su nombre lo indica son sistemas que ejecutan un conjunto definido y limitado de tareas y que están contenidos en una tarjeta de circuitos integrados. A diferencia de las computadoras de propósito general, los SIPE son portátiles, tienen bajo consumo de energía, no requieren de un sistema operativo y en el caso de requerirlo es sencillo. Sin embargo los SIPE son más complicados de diseñar e implementar que los sistemas que se implementan en sistemas de propósito general y no son fácilmente portables.

Los Arreglos Lógicos Configurables en el Entorno (FPGA Field Programmable Gate Array) son circuitos VLSI compuestos de una gran cantidad de bloques lógicos que contienen tablas de búsqueda (LUT Look-Up Tables) y flip-flops que se conectan para implementar una máquina de estados finitos; en otras palabras, sistemas digitales. Los FPGA se distinguen de otros Dispositivos de Lógica Configurable (*Programable*)[4] por su alta flexibilidad para implementar casi cualquier sistema digital, inclusive, los más complejos, siempre y cuando la implementación no exceda la cantidad de recursos con los que cuenta el FPGA. Otra ventaja es la facilidad con la que se pueden volver a configurar, ya que no requieren de un dispositivo especial para hacerlo. También son capaces de implementar varios sistemas digitales concurrentemente. Por ello los FPGA son idóneos para implementar SIPE. Sin embargo, no cuentan con una Unidad Aritmética Lógica y tampoco y debido a su carencia de contador de programa no pueden implementar estructuras de control cíclicas nativamente. [6, 8, 39].

Para describir algún sistema digital e implementarlo en un FPGA se utilizan los llamados Lenguajes de Descripción de Hardware (HDL Hardware Description Languages). Los HDL más usados son VHDL, Verilog, JHDL y Handel-C. Este último pertenece a Celoxica Co. y sigue un esquema de diseño algorítmico de programación imperativa con una sintaxis similar a la de ANSI-C, además de sus propias palabras reservadas, que permiten hacer un mejor uso del paralelismo y de los recursos con los que cuenta el FPGA [6].

Sin importar el HDL usado, la descripción debe ser traducida en las conexiones entre los bloques lógicos del FPGA necesarias para implementar el sistema digital. El resultado de la traducción se almacena en un archivo denominado de configuración, cuya extensión es *BIT* para los FPGA fabricados por Xilinx Inc., el cual se le envía por medio de algún protocolo de comunicación (p.ej. paralelo, USB, JTAG) y de esta forma queda configurado [6,8]. Un mismo FPGA puede utilizarse para muy diversas aplicaciones ya que su funcionamiento depende del sistema digital que esté implementando.

El número de bloques lógicos que contienen los FPGA depende del modelo al que pertenecen. Por otra parte, en la actualidad algunos FPGA incorporan otro tipo de dispositivos además de las LUT p.ej. multiplicadores de punto fijo, unidades MAC (Multiplier-Accumulator), y memorias RAM internas, por mencionar algunos. La cantidad de dispositivos adicionales también depende del modelo del FPGA [8].

[3] Suele traducirse Embedded como Embebido, pero esta palabra no existe en español.

[4] Estrictamente un Dispositivo de Lógica Programable no ejecuta ningún algoritmo ya que no se comporta como una máquina de Turing, por ello no puede programarse.

En términos de la implementación FPGA de algún sistema digital, existe un compromiso entre el número de LUT usadas y el tiempo de respuesta del sistema digital, debido a los retrasos inherentes a la conexión entre bloques lógicos. El manejo de dicho compromiso, requiere de conocimientos específicos del HDL y sobre el diseño, implementación y optimización de sistemas digitales.

Otro factor que debe considerarse para la descripción HDL es que las implementaciones deben operar con una frecuencia de reloj relativamente pequeña, ya que esto contribuye a un menor consumo de potencia.

El diseño en Handel-C no es complicado debido a las bondades del lenguaje, pero generalmente el número de dispositivos que se requieren para implementar un sistema digital descrito en Handel-C, excede en mucho a los que se utilizarían, por ejemplo, con una descripción VHDL del mismo sistema. Por ello es necesario utilizar técnicas sintácticas y semánticas del lenguaje Handel-C, propuestas por el fabricante, que permiten reducir la cantidad de recursos y el número de retrasos. Esto incrementa el tiempo de implementación y hace más compleja la descripción.

1.2. ¿Cómo implementar clasificadores SVM en un FPGA?

Cuando una BCI ha sido desarrollada y evaluada satisfactoriamente puede que su aplicación final exija que sea implementada en un SIPE; dadas las ventajas que ofrecen. La migración del clasificador SVM al SIPE no es usual debido a que los grupos de investigación relacionadas a BCI o SVM no la llevan a cabo. Esto se refleja en la escasez de publicaciones relacionadas a la implementación de BCI y SVM en SIPE.

En el Laboratorio de Investigación en Neuroimagenologia (LINI) de la Universidad Autónoma Metropolitana Unidad Iztapalapa en la ciudad de México se han desarrollado clasificadores SVM aplicados a BCI que se requieren implementar en FPGA. El objetivo a mediano plazo es implementar una BCI completa ,(adquisición, preprocesamiento y ejecución de comandos) en un FPGA. Por cuestiones de notación, en adelante se denominará al código Handel-C que describe una SVM como descripción FPGA; y como clasificador FPGA a la implementación FPGA de dicha descripción.

Para los propósitos de esta tesis se utilizaron clasificadores entrenados para identificar si un potencial P300 ha ocurrido o no. Lo anterior fija las restricciones del tiempo de clasificación que debe ser menor a 300 ms una vez ocurrido una P300, es decir, antes de que ocurra otro P300. En otras palabras, el clasificador FPGA debe realizar su tarea en menos de 300 ms.

En cuanto a las restricciones de la cantidad de recursos se debe tomar en cuenta el objetivo a mediano plazo, es decir, que en un único FPGA deben estar todas las etapas de la BCI. Por ello el clasificador FPGA debe utilizar la menor cantidad posible de recursos.

1.2.1. Solución Propuesta: Generador Automático de Descripciones FPGA

Tomando en cuenta que de una descripción SVM se puede obtener un clasificador FPGA entonces:

Cada SVM que se requiere implementar en FPGA tiene asociada una única descripción SVM. Dicha descripción requiere cumplir tanto con las restricciones de tiempo como las de recursos.

El punto de partida de este proyecto fue una descripción SVM de máquinas lineales con lenguaje Handel-C descrita en [39, 40]. Se realizó la optimización de recursos de dicha descripción y a este código mejorado se le realizó un análisis lineal. Tomando como base la descripción SVM anterior se diseñó una descripción SVM para una máquina gaussiana; de los análisis se concluyó que las descripciones correspondientes a otras SVM lineales o gaussianas tienen unas estructuras casi idénticas que sólo variarían en ciertas partes muy específicas de la descripción. De lo anterior se plantean las siguientes preguntas ¿Cómo se podría generar automáticamente la descripción SVM? ¿Qué datos se requieren para generar la descripción? ¿Qué procesamiento requieren los datos?

La solución propuesta fue diseñar e implementar una aplicación de software *stand-alone* (independiente) capaz de generar automáticamente la descripción Handel-C de la SVM requerida por el usuario, asumiendo que los datos para generarla se encuentran almacenados en un archivo tipo MAT de Octave-Matlab ya que es muy común usarlos con SVM. Se denominó *gCodeGen* a la aplicación de software *stand-alone* que se desarrolló para facilitar el proceso de generación de la descripción Handel-C, ya que incluye una GUI, manejo de archivos tipo MAT y almacenamiento de la descripción SVM generada.

La parte fundamental del gCodeGen es el algoritmo para generar el código de la descripción SVM, el cuál se basó en el análisis realizado a las descripciones SVM lineal y gaussiana. La idea fundamental es tener una estructura genérica del código, es decir, una plantilla, la cual se adapta a través de una serie de transformaciones para que describa la SVM requerida por el usuario. Se supuso que los datos necesarios para generar el código serían únicamente los parámetros del modelo SVM obtenido con Octave o Matlab

El gCodeGen también fue diseñado para realizar la evaluación del clasificador FPGA, dicho de otra forma, realiza una comparación entre la clasificación realizada con software (Octave o Matlab) y la llevada a cabo en el FPGA. Esto para medir el deterioro de la tasa de clasificación debido a la migración a un SIPE. La evaluación requiere de dos archivos: uno que contenga los vectores de prueba y otro que almacene los resultados de su clasificación llevada a cabo en software; el gCodeGen lee ambos archivos y envía los vectores de prueba al FPGA, éste los clasifica y regresa el resultado al gCodeGen que procede a comparar el resultado leído del archivo con el obtenido con el FPGA; contabilizando las discrepancias, para finalmente mostrar dicho conteo y el error de migración que es el conteo entre el total de vectores. Ya que no se encontró en la literatura consultada cual es el error tolerable para una migración de SVM a un SIPE se fijó arbitrariamente en 2 % . El error de migración se atribuyó principalmente a la perdida de precisión de la representación Qn.m $\mid m+n = 15 \; \forall \; m, \; n \in \mathbb{N}$ que se usó en el FPGA, tomando en cuenta que en Octave y Matlab la clasificación se lleva a cabo con precisión doble de punto flotante.

Para que el gCodeGen fuera una aplicación fácilmente extensible se utilizaron algunas prácticas de ingeniería de software; aunque no fue el propósito de este proyecto llevar a cabo completa y estrictamente las prácticas de ingeniería de software. El gCodeGen fue diseñado y desarrollado para ser extendido por medio de *plugins* que son pequeños módulos agregados por el usuario que extienden o aumentan la funcionalidad inicial de la aplicación; con el objetivo de agregar fácilmente lo siguiente: decodificadores de archivo distintos al tipo MAT; gestores de puertos de comunicación p. ej. USB; algoritmos para generar código, inclusive para otros HDL; diferentes tipos de protocolos de evaluación, p. ej. envío de un flujo de vectores y recibir sus clases o enviar un vector y esperar a recibir su clase para enviar el siguiente, etc.

El gCodeGen se implementó con Java usando el IDE RCP Eclipse v3.2 que es una versión del IDE Eclipse que permite desarrollar aplicaciones extensibles por plugins. Por otra parte, se utilizaron algunas partes del proceso OpenUP para el desarrollo del gCodeGen [43].

Como nota adicional se remarca que el usuario del gCodeGen tendrá que hacer manualmente la implementación FPGA del código de la descripción SVM generada, ya que si se hubiera incluido la opción de implementación en el gCodeGen se hubieran limitado las capacidades de síntesis de las aplicaciones especializadas como *Quantus II* de Altera o *ISE Project Navigator* de Xilinx. El proceso de implementación del código de la descripción SVM generada consiste en:

1. Crear un proyecto en el Development Kit de Handel-C.
2. Añadir como fuente el archivo del código generado (Descripción SVM).
3. Compilar el código para generar el archivo EDIF de conexiones.
4. Generar el archivo de configuración del FPGA.

Para esta tesis se utilizó el IDE DK 4.0 de Celoxica Co. y el ISE Xilinx *Project Navigator* v8.2. Las cuatro etapas anteriores se realizaron completamente en el DK, aunque este usa al *Project Navigator* en modo *background* para las dos últimas etapas. Se utilizó una tarjeta de desarrollo RC10 que contiene un FPGA Spartan 3 S31500L-4 de Xilinx. Mismo que es configurado a través de USB con la aplicación FTU3 también de Celoxica.

1.3. Descripción de la estructura de la Tesis

Esta tesis se compone de cuatro capítulos que son: Introducción, Marco Teórico, Desarrollo, Resultados y Discusión; además de tres secciones complementarias: Conclusiones, Apéndices, y Bibliografía.

2. **Marco Teórico**. Se describe formalmente a las SVM como clasificadores; se describen las particularidades de Handel-C; posteriormente se detallan algunos conceptos sobre el desarrollo de software, arquitecturas de software y patrones de diseño; se realiza la descripción del estado de la técnica sobre implementaciones BCI

en SIPE, clasificadores FPGA y software para generar código; por último se plantean los objetivos generales y específicos del proyecto.

3. **Desarrollo**. Se describe el análisis y construcción de las descripciones SVM lineal y gaussiana que conforman la plantilla, así como las consideraciones técnicas de su descripción e implementación; se detalla el proceso de desarrollo del gCodeGen y la creación de los plugins que lo componen y de su gestor.

4. **Resultados y Discusión**. Se muestran y analizan los resultados tanto del desarrollo e implementación de la descripción SVM plantilla y su implementación; como del proceso de desarrollo del gCodeGen y del algoritmo de generación de código; también se describen las consideraciones y cambios adicionales que se realizaron para que tanto el hardware como el software cumplieran los requerimientos del proyecto.

   Conclusiones. En las conclusiones se discuten menos rigurosamente los resultados; se describen los alcances del proyecto, el trabajo pendiente y la visión a futuro del proyecto.

Capítulo 2

Marco Teórico

2.1. Clasificadores de Discriminante Lineal

La clasificación automatizada es un problema abierto en investigación y existen diversos paradigmas que intentan solucionarlo, un ejemplo es la inteligencia artificial, representada por las redes neuronales; otro ejemplo son los clasificadores que se basan en la estadística de los datos, usan el teorema de Bayes y el principio de máxima verosimilitud con el objetivo de estimar la probabilidad de que un vector pertenezca a una u otra clase [15, 18].

En la tabla 2.1 se muestra una lista de los métodos de clasificación más utilizados. Se hace énfasis en los llamados clasificadores de discriminante lineal ya que son ampliamente usados por factores como la sencillez de su modelo y a pesar de eso, sus altas tasas de clasificación y de generalización. [15].

Asumiendo que todos los vectores de entrenamiento, prueba y reales de una aplicación especifica forman el conjunto $\mathbb{X} \subset \mathbb{R}^n$; se dice, que los clasificadores de discriminante lineal realizan su tarea usando un hiperplano $h(\mathbf{x})|\mathbf{x} \in \mathbb{X}$ que divide en dos partes al espacio $\mathbb{R}^n$ que lo contiene (figura 2.1).

Dicho hiperplano $h(\mathbf{x})$ puede definirse a partir de su vector normal $\mathbf{w}$ y su sesgo b, ecuación 2.1

$$h(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = 0 \tag{2.1}$$

donde $\mathbf{x}$ es el vector a clasificar también denominando vector de características. Finalmente, por convención se le llama característica a cada componente de $\mathbf{x}$.

La asignación de la etiqueta de clase del vector $\mathbf{x}$ se lleva a cabo de acuerdo su posición con respecto al hiperplano $h(\mathbf{x})$. Si el resultado de evaluar la ecuación 2.1 es positivo se dice que el vector $\mathbf{x}$ pertenece a una clase, en otro caso se encuentra en otra clase. Frecuentemente las etiquetas de clase son -1 y 1, por tanto, los

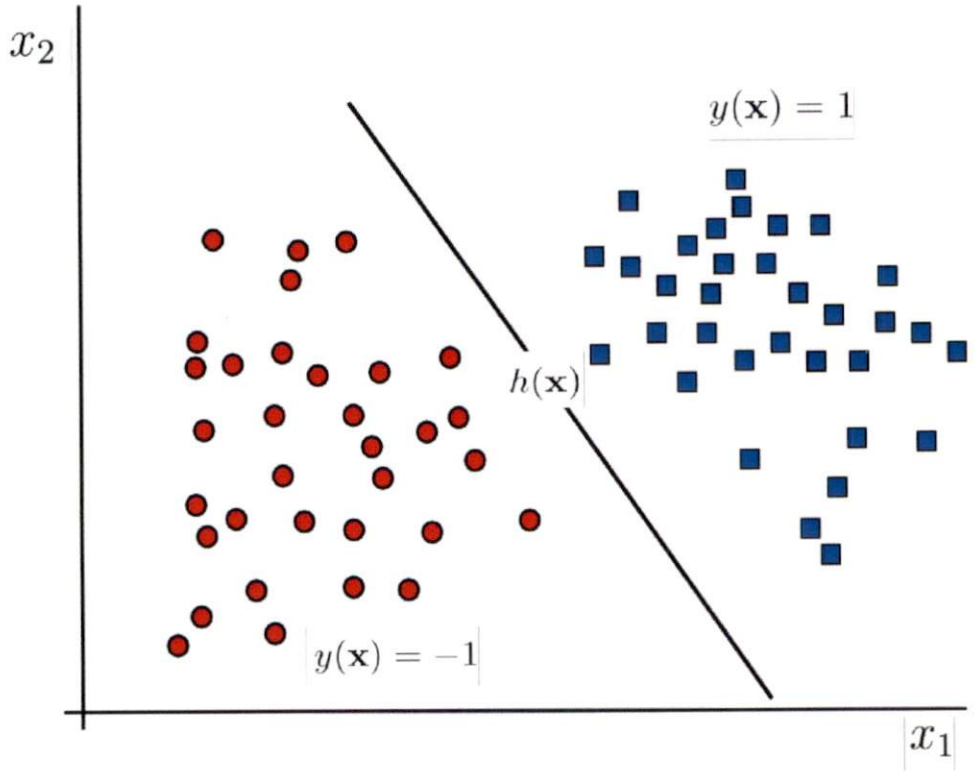


Figura 2.1 – *Ejemplo de un clasificador de discriminante lineal, descrito con el hiperplano $h(\boldsymbol{x}) \in \mathbb{R}$ que separa a las clases $y = 1$ e $y = -1$*

clasificadores de discriminantes lineal son clasificadores binarios. Por otra parte, para discriminar entre más de dos clases se pueden usar arreglos de clasificadores de discriminante lineal conectados a una función de decisión que interprete el resultado de las clasificaciones [12, 15, 18, 35].

Por otra parte, redefiniendo $\mathbf{w}$ y $\mathbf{x}$ de la ecuación 2.1 como $\mathbf{w} = [\mathbf{w};\ b]$ y $\mathbf{x} = [\mathbf{x};\ 1]$ y así esta ecuación puede escribirse como la ecuación 2.2

$$h(\mathbf{x}) = \mathbf{w}^T\mathbf{x} = 0 \tag{2.2}$$

De esta forma, los clasificadores de discriminante lineal se entrenan con el paradigma de entrenamiento supervisado y consiste en estimar el vector $\mathbf{w}$ de la ecuación 2.2 más adecuado para clasificar correctamente los vectores de entrenamiento.

Para el entrenamiento de los clasificadores de discriminante lineal se consideran dos casos:

1) Clases linealmente separables. En general, que dos clases sean linealmente separables implica, que existe al menos un hiperplano capaz de separarlas completamente, es decir, no hay vectores del lado opuesto al que les corresponde. Gráficamente esto se ve como dos clases que no se mezclan. Ejemplos de clases linealmente

Métodos de Clasificación	
1.	Clasificadores de Distancia Mínima (Euclidiana y Mahalanobis)
2.	Clasificador de Correlación
3.	Estimadores de Maxima Verosimilitud
4.	Discriminante Lineal
5.	Clasificador de los k Vecinos más Cercanos
6.	Criterio de la Entropia
7.	Comparación de Cadenas de Caracteres
8.	Parsing
9.	Aprendizaje Sintáctico
10.	Autómatas de estado finito
11.	Redes Neuronales
12.	Clustering
13.	Cuantización de Vectores de Aprendizaje

Tabla 2.1 – *Muestra una lista de los métodos de clasificación más usados para el Reconocimiento de Patrones. Modificada de [16]*

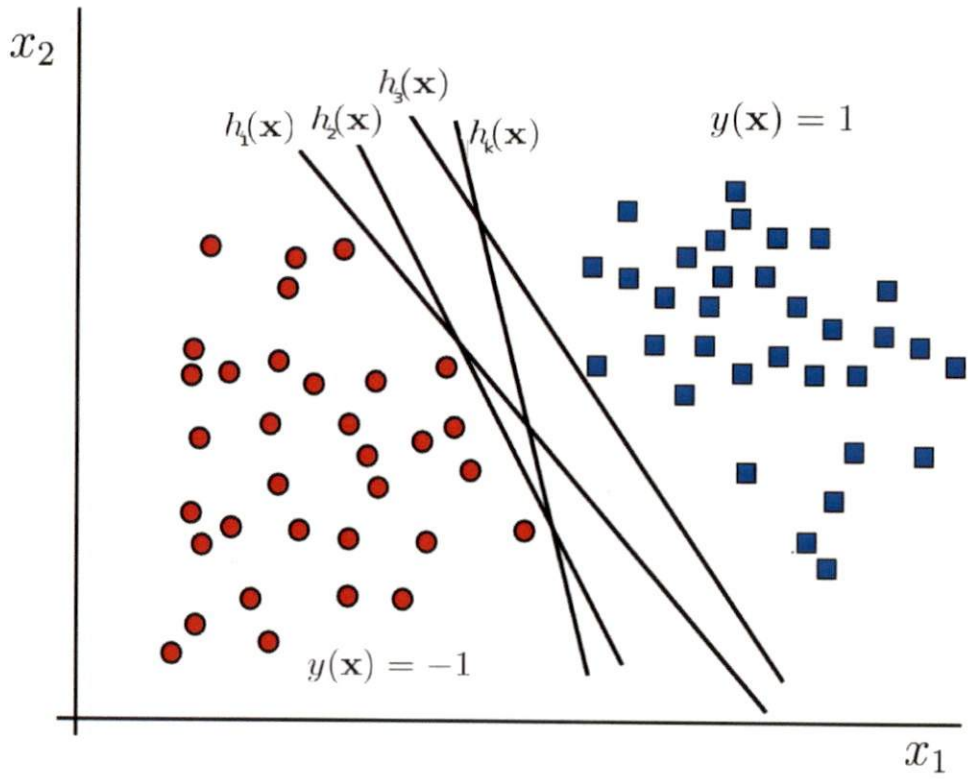


Figura 2.2 – *Algunos hiperplanos $h(\boldsymbol{x})$ de separación en $\mathbb{R}^2$*

separables son las figuras 2.1 y 2.2.

2) Clases no linealmente separables. Es el caso opuesto al anterior, que es cuando las clases se mezclan y no existe un hiperplano que pueda dividir el espacio de tal forma que cada clase quede confinada a un solo lado del mismo.

Dependiendo de la naturaleza de los datos y del proceso de extracción de características, las clases pueden ser linealmente o cuasi linealmente separables, este último es un caso muy frecuente. Para cualquiera de ambos existe un único hiperplano que maximiza la separación entre clases, es decir, no favorece la prevalecia de ninguna. En la figura 2.2 se muestran ejemplos de hiperplanos que no maximizan la separación, mientras que en la figura 2.3 se muestra un hiperplano que sí la maximiza; su principal ventaja es su excelente generalización [12, 15, 35].

Para el caso de las clases cuasi linealmente separables, los métodos de entrenamiento encuentran el hiperplano que minimiza la cantidad de errores de clasificación. Aunque esto no necesariamente garantiza una buena generalización. Cuando las clases se mezclan demasiado son necesarios otros métodos para mejorar los resultados de la clasificación. Aunque existen casos extremos donde es necesario cambiar de método de extracción de características [15, 18, 19].

La asignación de etiquetas puede realizarse de varias formas, una de las más utilizadas es la función de activación signo $sign : \mathbb{R} \rightarrow \{-1, 1\}$ definida como:

$$sign(a) = \begin{cases} 1 & \text{si } a \geq 0 \\ -1 & \text{en otro caso} \end{cases}$$

En otras palabras; si $sign(h(\mathbf{x}) \geq 0)$ el vector $\mathbf{x}$ pertenece a la clase 1, en otro caso pertenece a la clase -1 [2, 12, 35].

2.1.1. Máquinas de Soporte Vectorial

Las SVM son clasificadores de discriminante lineal, cuyo hiperplano de separación es aquel que maximiza la separación entre clases [12, 35]. En adelante al hiperplano de separación que usan las SVM se le llamara hiperplano óptimo.

Se ha demostrado que en aplicaciones BCI las SVM tienen mejor desempeño con respecto a otros métodos de clasificación, inclusive que las Redes Neuronales Multicapa [12, 17, 30].

2.1.1.1. Construcción del hiperplano óptimo

Dado un conjunto de entrenamiento $S = \{(\mathbf{x}_k, y_k) | x_k \in \mathbb{X}; y_k \in \mathbb{Y}; k \in \mathbb{N}\}$ donde $\mathbb{X}$ es el conjunto de todos los vectores de n componentes resultado de la extracción de características de una aplicación específica; $\mathbb{Y} = \{-1, 1\}$

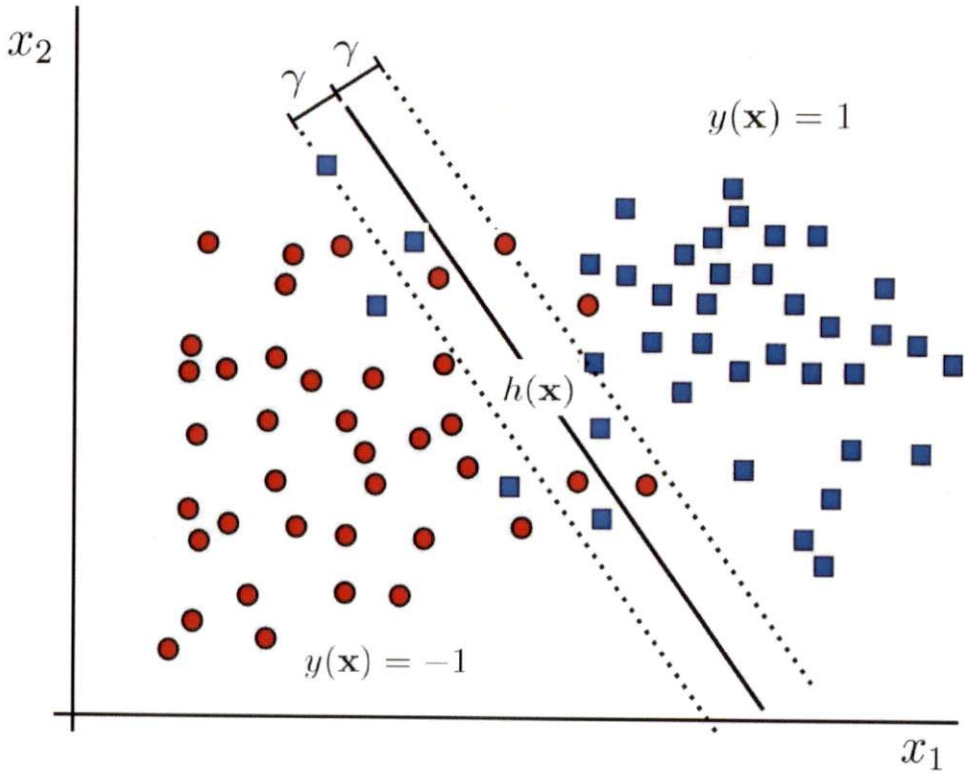


Figura 2.3 – *Muestra gráficamente la construcción de las fronteras f_1 y f_2 y del hiperplano óptimo.*

son todas las etiquetas de clase de los vectores en $\mathbb{X}$; la pareja $(\mathbf{x}_k, y_k)$ denota al k-ésimo vector de entrenamiento y su etiqueta de clase. En la figura 2.3 se muestran ejemplos de vectores de entrenamiento en $\mathbb{R}^2$ de dos clases linealmente separables, los cuales están representados con los círculos para la clase -1 (abajo del hiperplano) y con los cuadrados la clase 1 (arriba del hiperplano). Los vectores de la clase -1 que están más próximos a la clase 1 forman una frontera f_1; de forma recíproca para la frontera f_2. Se llama γ_1 a la distancia de f_1 al hiperplano óptimo; de manera similar para γ_2. Asumiendo que no se prevalece ninguna clase: $\gamma = \gamma_1 = \gamma_2$. Es γ el margen de separación entre clases que maximiza el hiperplano óptimo. En la figura 2.3 se muestra gráficamente el planteamiento antes mencionado.

Encontrar el vector $\mathbf{w}$ del hiperplano óptimo a partir de S se plantea como un problema de optimización con restricciones de Lagrange y dada la dualidad del mismo, el problema se resuelve con técnicas de optimización cuadrática; de tal forma que $\mathbf{w}$ queda en función de combinaciones lineales de los elementos del conjunto de entrenamiento S (ecuación 2.3); recordando que el sesgo b es la componente $N+1$, lo que implica que $\alpha_{N+1} = 1$ y $y_{N+1} = 1$.

$$\mathbf{w} = \sum_{k=1}^{N+1} \alpha_k y_k \mathbf{x}_k \tag{2.3}$$

las α_k son constantes positivas de ponderación que reflejan la contribución de cada vector de entrenamiento al calculo de $\mathbf{w}$. En la práctica, la mayoría de las α_k son cero o despreciables, excepto para aquellos vectores que están más cerca de las fronteras f_1 y f_2, dicho de otra forma, el vector $\mathbf{w}$ queda en función de un conjunto $D \subset S$ pequeño comparado con S. Los vectores de entrenamiento que se encuentran en D se les llama vectores de soporte; y $|D|$ es el número de vectores de soporte. De tal forma que $\mathbf{w}$ se escribe en función de los vectores de soporte como la ecuación 2.4

$$\mathbf{w} = \sum_{i=1}^{|D|+1} \alpha_i y_i \mathbf{x}_i \tag{2.4}$$

donde $(\mathbf{x}_i, y_i) \in D$ y α_i es la i-ésima constante de ponderación; la componente $|D| + 1$ es el sesgo b.

Finalizando el planteamiento, el hiperplano óptimo $h_{opt}(\mathbf{x})$ se obtiene al sustituir (2.4) en (2.1) resultando la ecuación 2.5

$$h_{opt}(\mathbf{x}) = \sum_{i=1}^{|D|+1} \alpha_i y_i \mathbf{x}_i^T \mathbf{x} = 0 \tag{2.5}$$

La cardinalidad de D es el número de vectores de soporte (nvs)

2.1.1.2. Generalización de las SVM para clases no linealmente separables

Intuitivamente la idea de generalizar las SVM para clases no linealmente separables, consiste en aplicarles una transformación no lineal que los transporte a un espacio de características de mayor dimensión que el espacio original. La probabilidad de que en ese nuevo espacio los vectores sean lineal o cuasi linealmente separables aumenta.

Si $\Phi(\cdot)$ representa la transformación no lineal y $\langle \cdot, \cdot \rangle$ el producto interno; entonces el hiperplano óptimo en el nuevo espacio está descrito como la ecuación 2.6

$$h_{opt}(\mathbf{x}) = \sum_{i=1}^{nvs+1} \alpha_i y_i \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}) \rangle \tag{2.6}$$

Otra forma de evaluar $\langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}) \rangle$ es usar las llamadas funciones *kernel* ecuación 2.7, su uso es extendido, debido a que las transformaciones no se evalúan directamente para cada vector, lo que disminuye el trabajo computacional.

$$K(\mathbf{x}_i, \mathbf{x}) = \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}) \rangle \tag{2.7}$$

Las siguientes son las funciones *kernel* más comunes [12, 15, 35, 54]:

$$K_l(\mathbf{x}_i, \mathbf{x}) = \mathbf{x}_i^T \mathbf{x} = \langle \mathbf{x}_i, \mathbf{x} \rangle \quad \text{lineal} \tag{2.8}$$

$$K_p(\mathbf{x}_i, \mathbf{x}) = (1 + \mathbf{x}_i^T \mathbf{x})^P : P \in \mathbb{Z} \quad \text{polinomial} \tag{2.9}$$

$$K_g(\mathbf{x}_i, \mathbf{x}) = \mathbf{e}^{-\rho \|\mathbf{x} - \mathbf{x}_i\|} : \rho \in \mathbb{R} > 0 \quad \text{gaussiano} \tag{2.10}$$

$$K_s(\mathbf{x}_i, \mathbf{x}) = arctan(\langle kx_i, x\rangle + c) : k, c \in \mathbb{R} \quad \text{sigmoidal} \tag{2.11}$$

El *kernel* lineal (ecuación 2.8) y el de base radial gaussiana (ecuación 2.10) son ampliamente usados en aplicaciones BCI [30, 47]. El *kernel* lineal se reduce a un producto interno, es decir, al hiperplano óptimo de la ecuación 2.5. En el caso del *kernel* de base radial gaussiana se requiere de la evaluación de una exponencial; el parámetro $\gamma > 0$ está relacionado a la distribución de las clases.

El hiperplano óptimo generalizado en función del *kernel* se obtiene al sustituir (2.7) en (2.6) resultando la ecuación 2.12.

$$h_{opt}(\mathbf{x}) = \sum_{i=1}^{nvs+1} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}) \tag{2.12}$$

El entrenamiento de las SVM generalizadas se modifica de acuerdo al tipo de *kernel*. Sin embargo, se mantiene la idea de maximizar el margen de separación en el nuevo espacio de características.

Finalmente, la asignación de la etiqueta de clase $y(\mathbf{x})$ del vector $\mathbf{x}$ se realiza con la función de activación $sign(\cdot)$ como lo muestra la ecuación 2.13.

$$y(\mathbf{x}) = sign\left(\sum_{i=1}^{nvs+1} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x})\right) \tag{2.13}$$

Concluyendo, la evaluación de la función $sign(\cdot)$ en el hiperplano óptimo (ecuación (2.13) es el clasificador SVM generalizado. Para la nomenclatura; se le adosa a *"Clasificador SVM"* el tipo de *kernel* usado, p. ej. si se usa el *kernel* gaussiano, se le llamará clasificador SVM gaussiano o simplemente SVM gaussiana.

De tal forma que el clasificador SVM lineal es la ecuación 2.14, mientras que el gaussiano es 2.15

$$y(\mathbf{x}) = sign\left(\sum_{i=1}^{nvs+1} \alpha_i y_i \mathbf{x}_i^T \mathbf{x}\right) \tag{2.14}$$

$$y(\mathbf{x}) = sign\left(\sum_{i=1}^{nvs+1} \alpha_i y_i e^{\rho||\mathbf{x}-\mathbf{x}_i||}\right) \tag{2.15}$$

Entrenamiento de SVM. Para realizar el entrenamiento de las SVM se ocupan herramientas de software de cálculo numérico p.ej. Octave, Matlab, Scipy, LIBSVM (*A Library for Support Vector Machines*) [34], Shogun (A Large Scale Machine Learning Toolbox) [51], etc.

Aplicación de clasificadores SVM. Con una SVM entrenada y evaluada se procede al uso en la aplicación real; a diferencia del entrenamiento que puede requerir mucho tiempo; en las aplicaciones BCI la clasificación debe realizarse en unos cuantos milisegundos. Recordar que en esta tesis se requiere detectar la ocurrencia de una P300, es decir, el tiempo de clasificación debe ser menor de 300 ms.

2.2. Implementación de Sistemas Digitales en FPGA

Debido a las características de los FPGA mencionadas anteriormente, son una buena opción para implementar sistemas BCI, ya que, estos últimos evolucionan rápidamente, tienen que ser fácilmente adaptables, portables y tienen que funcionar en tiempo real.

En esta sección se detallará sobre el proceso de implementación de sistemas digitales en FPGA y de los lenguajes de descripción de hardware más utilizados, aunque se enfatizará Handel-C.

2.2.1. Descripción e Implementación de Sistemas Digitales en FPGA

De manera más o menos general, el diseño, desarrollo e implementación de sistemas digitales en FPGA requiere de las siguientes seis etapas:

1. Diseño y descripción del sistema. En esta parte se realiza el diseño y descripción del sistema digital usando algún HDL (lenguaje de descripción de hardware).

2. Simulación del sistema. La descripción es implementada en un modelo de simulación dependiente del dispositivo donde se va a implementar. En dicho modelo puede verificarse el buen funcionamiento de la descripción.

3. Construcción del modelo de implementación. La compilación de la descripción al modelo de implementación da como resultado la lista de conexiones *netlist* que sigue el estándar EDIF (Electronic Design Interchange Format)[1]. En este modelo que está ligado a las características y limitaciones del dispositivo en el que se va a implementar se pueden realizar análisis de tiempos de propagación más reales que los simulados; se pueden gestionar los dispositivos y sus conexiones para reducir ya sea el número de retrasos o la cantidad de dispositivos que se requieren.

4. Generación de la implementación FPGA del sistema digital. A través de un proceso denominado *Place and Route* que está compuesto de dos subprocesos: el "*emplazamiento*", el cual consiste en decidir que dispositivos del FPGA van a utilizarse para la implementación; el otro subproceso es la "*creación de rutas*" que da como resultado la lista de conexiones entre todos los dispositivos seleccionados en la etapa previa.

5. Generación del archivo de configuración del FPGA. Consiste en traducir el resultado del *Place and Route* en los códigos de operación del FPGA que le permiten habilitar los dispositivos y la lista de conexiones. Los códigos se almacenan en un archivo llamado *archivo de configuración.*

6. Configuración al FPGA. Se realiza cuando un archivo de configuración es enviado al FPGA usando algún protocolo de comunicación p. ej.: paralelo-JTag, USB-JTag ó USB-USB, aunque también suele usarse una memoria EEPROM serial conectada a pines específicos del FPGA; en dicha memoria se almacenan los códigos de operación del FPGA. De esta manera el FPGA implementa la descripción HDL [40].

[1] Estándar desarrollado en la década de los 80's para promover la portabilidad de los diseños electrónicos ASIC independientemente del fabricante [44]

Los fabricantes de FPGA han desarrollado herramientas IDE que facilitan el proceso de desarrollo e implementación de sistemas digitales en FPGA. Un ejemplo de esto es el IDE de Xilinx llamado *ISE Project Navigator*. El cual fue usado para el desarrollo de este proyecto.

2.2.1.1. Handel-C: Lenguaje de Descripción de Hardware

El lenguaje Handel-C es un lenguaje imperativo de descripción de hardware; es un subconjunto de ANSI-C con palabras reservadas adicionales que explotan las características de la arquitectura de los FPGA. Handel-C fue desarrollado en el Reino Unido por Celoxica Co. La principal diferencia de Handel-C con respecto a otros HDL tradicionales es que en él la descripción se realiza algoritmicamente, en vez de la descripción de la máquina de estados finitos y sus componentes [5, 6].

El lenguaje Handel-C es parte de un sistema de desarrollo llamado *DK* (Developer Kit) de Celoxica Co., que, entre otras herramientas, incluye: bibliotecas de operaciones de punto fijo y punto flotante; manejo de dispositivos periféricos como: pantallas VGA, puertos PS2, Paralelo, RS232 y USB; memorias RAM y RAM-flash, además de herramientas para simulación y síntesis del diseño . En la figura 2.4 se muestra un diagrama de bloques de la arquitectura del *DK*, la cual está compuesta del núcleo Handel-C que es el que se encarga de la compilación del código Handel-C a VHDL o Verilog o a EDIF; también incluye un simulador y algunas herramientas de síntesis [6].

Handel-C fue diseñado para que cada instrucción en el FPGA se realice en 1 ciclo de reloj. Por ello una instrucción compleja requerirá de muchas LUT y como consecuencia también aumentará el número de retrasos. Para reducir los retrasos se requiere que el diseño y descripción de los sistemas sea realizado por medio de instrucciones sencillas; de tal forma que las instrucciones complejas deben ser divididas en dos o más instrucciones sencillas [5,6]. En la tabla 2.2 se hace una comparación entre las principales características de los lenguajes ANSI-C y Handel-C. Handel-C maneja funciones, procedimientos y macros de forma muy similar a ANSI-C, además de otras opciones como la replicación de funciones en código y declaración de bloques de instrucciones. El lenguaje también provee de mecanismos de comunicación entre bloques de instrucciones, funciones, procedimientos y macros, que permiten la transferencia de datos y establecen el control del flujo [6].

El problema es que al realizar una traducción de un programa escrito en ANSI-C a su equivalente inmediato en Handel-C, la implementación FPGA utiliza recursos de más; para optimizar el número de recursos del FPGA se debe refinar la descripción Handel-C utilizando un conjunto de estrategias tanto sintácticas como semánticas, que permiten el uso óptimo de recursos [5]. Estas técnicas se basan en el uso de comparaciones simples que se evalúan en las estructuras de control cíclicas y condicionales.

De forma nativa, Handel-C realiza las siguientes operaciones aritméticas: suma, resta, multiplicación y modulo $2^n : n \in \mathbb{N}$; las operaciones lógicas: *and, or, not*; y las operaciones para manipulación de bits: corrimiento a la derecha y a la izquierda y extracción de una subpalabra de bits [6]. Pero, al mismo tiempo, la representación punto flotante no se maneja nativamente, lo cual se debe a la arquitectura del FPGA. Para el manejo de datos con representación punto flotante se puede utilizar la biblioteca *Celoxica floating-point library* [7].

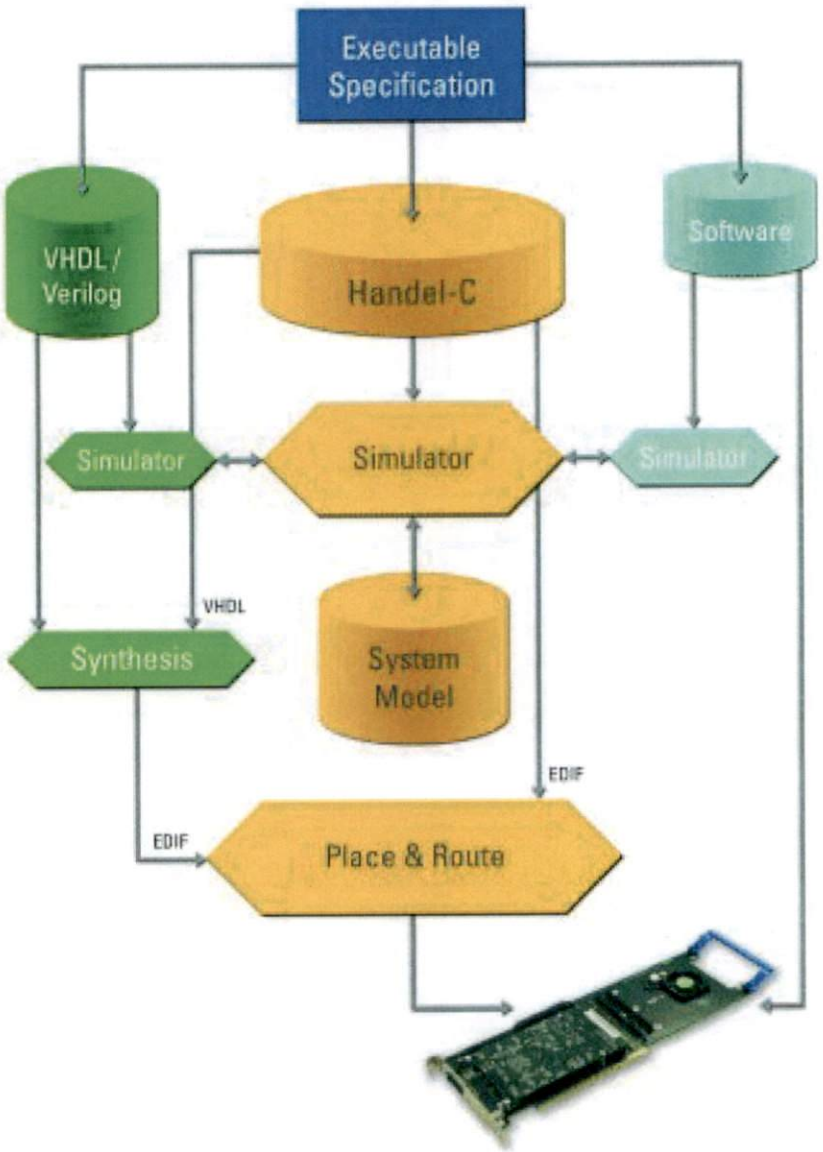


Figura 2.4 – *Muestra un diagrama a bloques de la arquitectura del Celoxica Co. Developer Kit. Extraído de [6].*

Característica	ANSI-C	Handel-C
Sentencias concurrentes	No incluida	Sentencia *par*
Resultado de compilación	Código objeto	*netlist*
Multiples funciones principales (*main*)	No permitidas	Permitidas
Estructuras de control	*if-else, switch-case, for, do-while, while, for*	*if-else, switch-case, for, do-while, while, for*
Replicación de código	No existe	Sentencias *par, macro, functions, procedures*
Tamaño de los datos tipo *int*	4 bytes	Arbitrario; a criterio del programador
Uso de la memoria RAM	Con las funciones *malloc y sizeof*	Palabras reservadas *ram, with block=1* †

Tabla 2.2 – *Muestra una tabla comparativa entre los lenguajes ANSI-C y Handel-C [6,31].* † *Indica mapeo a la memoria RAM interna del FPGA.*

La multiplicación $*$ se trata de manera distinta dependiendo de los operandos. En el caso de que al menos uno de los operandos sea constante, el compilador se encarga de generar la LUT que implementa la multiplicación; en el caso en que ambos operandos sean variables el compilador infiere la operación a los multiplicadores punto fijo *MULT 18×18* con los que cuentan algunos FPGA.

Handel-C permite la declaración de dos tipos de arreglos unos de acceso paralelo y otros de acceso secuencial. Los primeros se convierten en un arreglo de registros, mientas que los segundos se mapean a la memoria RAM interna del FPGA.

Para concluir, algunas consideraciones generales que se deben tomar en cuenta para realizar descripciones con Handel-C son las siguientes:

- A pesar de la descripción algorítmica que permite Handel-C, una traducción directa de ANSI-C a Handel-C no es factible si se requiere optimizar recursos.
- El desarrollo de descripciones Handel-C optimizadas requiere no solo de conocimientos de programación sino también de sistemas digitales.
- El entorno DK 4.0 de Celoxica hace diferencia entre la descripción Handel-C de simulación y de implementación, es decir, que la descripción de simulación y la de implementación son diferentes.

2.3. Desarrollo de Software

De acuerdo con el *Software Engineering Project 2004* [50] la ingeniería de software se define como: *"La aplicación de una técnica sistemática, disciplinada y cuantificable para el desarrollo, operación y mantenimiento de software"* [49].

2.3.1. Proceso de desarrollo de software

Las prácticas de la ingeniería de software surgen para evitar o disminuir las consecuencias que implica un desarrollo desorganizado del software; ejemplos de estas consecuencias son: inestabilidad del software; mal manejo de los errores y excepciones; poca o nula extensibilidad, entre otros [49, 50]

La práctica común de desarrollar software de forma desorganizada conducía a que tuviera errores, fuera difícilmente adaptable, entre otras desventajas. Para solucionarlo se propuso el llamado *proceso de desarrollo de software en cascada* que surgió en la decada de los 70 [28, 38].

De forma general el desarrollo de software se compone de 5 etapas [28, 38, 49, 50]:

1. **Planteamiento de los requerimientos del software.** Consiste en plantear y especificar las necesidades que el software debe satisfacer. A estas necesidades se les conoce como *requerimientos* que pueden ser de dos tipos: el primero son los requerimientos funcionales que denotan acciones concretas a las cuales se las conoce como *casos de uso*; el segundo tipo son los requerimientos no funcionales que denotan las cualidades que el software debe cumplir, entre los requerimientos funcionales más usuales se encuentran los siguientes: eficiencia, extensibilidad, rendimiento, portabilidad, extensibilidad, etc.

2. **Diseño del software.** Consiste en diseñar y especificar las partes que constituyen al software y como éstas se relacionan para que se satisfagan adecuadamente los requerimientos.

3. **Construcción del software.** Es la etapa donde se realiza la codificación del software en algún lenguaje de programación.

4. **Evaluación del software.** Consiste en la planeación y ejecución de las pruebas a las que el software será sometido para evaluar si cumple con los requerimientos. Cuando la evaluación es satisfactoria el software se libera para su uso.

5. **Mantenimiento del software.** Consiste en la actualización, modificación y adaptación de nuevos requerimientos al software ya liberado. También se corrigen los errores que no fueron descubiertos en la etapa de evaluación.

Un elemento destacado del desarrollo de software es la *arquitectura de software*, este consiste en especificar los componentes y sus relaciones que satisfacen los requerimientos. En la etapa de diseño se construye la *arquitectura*

de software de diseño que se caracteriza por ser una descripción en alto nivel; durante la etapa de construcción se realiza la *arquitectura de software de construcción* que a diferencia de la arquitectura anterior, en esta se hace una descripción detallada.

Existen diversas formas de organizar los elementos de la arquitectura de software, entre éstas se encuentran: los modelos multi-capa, el modelo vista controlador, arquitectura extensible con elementos agregados tardíamente. Ésta última se caracteriza por tener una estructura central fija o núcleo al cual se le pueden agregar o quitar fácilmente elementos que modifican la funcionalidad del software. Los elementos que se agregan se llaman *componentes* si el grupo de desarrollo es el único que puede agregarlos, es decir, se requiere del código; y se llaman *plugins* si cualquier persona puede agregarle o quitarle elementos al núcleo sin necesidad de conocer el código fuente. Finalmente se les llama *puntos de extensibilidad tardía* a las partes del núcleo que pueden extenderse con plugins o componentes [21, 28, 38].

Otros elementos relacionados con la arquitectura de software son los *patrones de diseño*, estos son soluciones reusables a problemas recurrentes en el diseño y construcción de software. Son soluciones en alto nivel, es decir, que su adaptación y aplicación es labor del grupo de desarrollo de software [20].

2.3.2. Modelado de Software

El software puede analizarse estructural o funcionalmente, es común que este análisis se realice con abstracciones gráficas llamadas modelos que deben ser fáciles de entender y describir adecuadamente y sin ambigüedades lo que representan. El Lenguaje Unificado de Modelado (UML Unified Model Language) es una propuesta para estandarizar los diagramas que se usan para el modelado de software, aunque, actualmente también se ocupan para modelar procesos de producción y de hardware. La versión 2.0 de UML define 15 diagramas que modelan tanto la parte estructural como la funcional [21, 28, 38, 43, 46].

En las variantes del proceso de desarrollo de software que consideran a la arquitectura de software como el elemento fundamental para desarrollarlo, es imprescindible el modelado de dicha arquitectura. Para hacerlo se utilizan, generalmente, los diagramas de componentes que define UML. Otros diagramas muy usados son: de casos de uso, de actividades, de colaboración, de secuencias [43, 46].

2.3.3. Gestion del proceso de desarrollo de software

El Proceso Racional Unificado (RUP Rational Process Unified) y una modificación de este, el Proceso Abierto Unificado (OpenUP) son ejemplos de métodos de gestión del desarrollo de software, cada uno plantea metas bien definidas para las etapas del desarrollo de software, mismas que redefinide [28, 38, 43]. Cada etapa del OpenUP concluye con la entrega de productos, por ejemplo: modelos, documentos, reportes, etc. Estos gestores siguen el paradigma iterativo e incremental de desarrollo, lo cual implica que sus productos son depurados y especificados a través de varias iteraciones; en el caso particular de la implementación de la arquitectura de software; consiste en

implementar en cada iteración ciertas partes de la arquitectura hasta completarla.

Otra tarea importante de los gestores antes mencionados es la *administración de riesgos*. Los riesgos son aquellos impedimentos que impiden concluir en tiempo y forma el desarrollo del software. La administración de riesgos se encarga de categorizar los riesgos de acuerdo a la gravedad y probabilidad de su ocurrencia, esto se hace con el fin de realizar un plan de prevención y contingencia de los riesgos de desarrollo [38, 42].

El OpenUP a diferencia del RUP es menos complejo y requiere de menos productos. Esta compuesto de 4 etapas que son Incepción, Elaboración, Construcción y Transición [38, 42, 43].

2.3.4. Eclipse RCP para desarrollar aplicaciones Extensibles con plugins

Las *Rich Client Platforms* (RCP) son plataformas para desarrollar aplicaciones extensibles a base de plugins. Estas están constituidas de un núcleo (Core) de funciones básicas como son: manejo de Interfaces Gráficas de Usuario (GUI's Graphic User Interfaces); interacción con el sistema operativo y con los periféricos; gestión de los plugins, es decir, controlar su creación, registro y manejo la información que se les envía o es recibida de ellos; entre otras. El núcleo provee las funciones que requieren la mayoría de las aplicaciones la especificidad de cada una se hace a través de los plugins agregados. El Eclipse RCP es un IDE que permite desarrollar RCP con Java [43].

2.4. Trabajos relacionados

2.4.1. Implementación de SVM en hardware

El artículo más antiguo que se encontró referente a la implementación de una clasificador de discriminante lineal en hardware es [1] de Anguita *et al.*, y consiste en la implementación de un *perceptron*. Ídem realizaron la implementación del algoritmo de entrenamiento de una SVM [2] lineal en un FPGA, en sus resultados encontraron que una representación $Qn.m$ de 16 bits es suficiente para no tener errores de cuantización.

En [32] Faisal M. Khan, et al., presentan una implementación FPGA de varias SVM, mismas que utilizan el Sistema Numérico Logarítmico (LNS Logarithmic Number System). Se realizaron varias implementaciones SVM una de kernel lineal, dos gaussianos y un sigmoideo. Se realizaron comparaciones entre las clasificaciones realizadas en el FPGA y las hechas en la computadora. Encontraron que con una representación $Q2.13$ se pueden alcanzar un 99 % de parecido entre la clasificación FPGA y la realizada en la PC. Aunque en algunas aplicaciones la precisión $Qn.m$ tuvo que ser de al menos 20 bits.

Roman Genov, et al., propusieron en [22] un hardware analógico-digital, que evalúa la función de discriminación de una SVM lineal. Es un dispositivo masivamente paralelo, ya que su núcleo contiene 128 *cores* que evalúan el producto punto, las multiplicaciones de tal operación se realizan analógicamente y de forma concurrente, mientras

que las sumas de acumulación se realizan de forma digital. El desempeño del hardware es alto, aunque, no es fácilmente escalable ya que está construida con tecnología VLSI-CMOS.

En [40] Omar Piña et al., describen la implementación FPGA de una SVM lineal de 6 características y 78 vectores de soporte, los resultados demostraron un 98 % similitud entre la clasificación realizada en Matlab con punto flotante de doble precisión y la realizada en un FPGA con representación de punto fijo. El tiempo de clasificación de un vector es de 2 ms, pero utilizó el 75 % de los recursos totales del FPGA.

2.4.2. Aplicaciones que generan automáticamente código

Aunque existen varias aplicaciones que permiten generar código Handel-C ó de VHDL a partir de la especificación de parámetros o del código en otros lenguajes; en las referencias consultadas no se encontró alguna que generara código VHDL ó Handel-C que describan SVM. A continuación se describen algunas aplicaciones que generan código que se implementa en FPGA.

Inferno C++ to Verilog es una aplicación de código abierto que genera el código Verilog o VHDL, a partir de código C++.

V2DIVGEN. En [25] se describe el desarrollo de una aplicación llamada V2DIVGEN que genera el código Handel-C que implementa una división rápida, usando los multiplicadores internos del FPGA. Este divisor utiliza pocos recursos del FPGA, aunque, tiene poca precisión. El V2DIVGEN es una aplicación parametrizable, es decir, tiene una ventana donde el usuario establece los parámetros para adecuar la generación del código; en otras palabras la aplicación está dedicada a la implementación de divisores y únicamente los desarrolladores pueden modificar el algoritmo de generación de código o agregar otro tipo de implementaciones.

Icarus Verilog es una aplicación para diseñar e implementar hardware utilizando Verilog. Entre sus funciones, puede traducir entre Verilog y VHDL a través del siguiente proceso; Código en Verilog → *netlist* Verilog → Traducir netlist a código VHDL [56].

vMagic. Es una aplicación que genera automáticamente código VHDL. Para hacerlo se requiere de una plantilla VHDL, la cual, vMagic convierte a un meta-objeto; los cuales pueden ser manipulado con operaciones de adhesión y supresión de funciones o de puertos. A partir de algún meta-objeto manipulados se puede generar el código en VHDL que lo implementa. El uso de meta-objetos permite visualizar la implementación desde un nivel alto de abstracción y evita que el usuario requiera de un amplio conocimiento de VHDL. Sin embargo para generar un meta-objeto se requiere, como ya se dijo, de un código VHDL plantilla. La aplicación vMagic también permite la simulación de las implementaciones y de los meta-objetos [45].

Capítulo 3

Desarrollo del generador de código Handel-C que describe SVM para aplicaciones BCI.

A continuación se realiza la formalización del problema que resolvió en este proyecto de investigación así como el desarrollo de su solución.

3.1. Descripción general del generador de código Handel-C que describe SVM para aplicaciones BCI

El objetivo general del proyecto es la implementación en FPGA de sistemas BCI completos dadas las ventajas que tienen. Este proyecto está enfocado únicamente a la implementación del clasificador SVM parte de las BCI. Recordando que es el P300 el paradigma de potenciales evocados para los que serán utilizados los clasificadores FPGA de este proyecto; se tienen dos restricciones: primera, el número de dispositivos del FPGA requeridos para la implementación del clasificador tiene que ser reducido para que haya suficientes recursos para la implementación del resto de la BCI; segunda, el tiempo de clasificación debe ser menor a 300 ms para detectar dos P300 consecutivos.

En un trabajo previo presentado en [39,40] se realizó la implementación de una SVM lineal descrita en Handel-C que, únicamente, cumple con la segunda restricción. Esta descripción fue el punto de partida de este proyecto.

Se realizó un análisis de la descripción Handel-C de [39, 40]; se le aplicaron las técnicas de optimización de recursos descritas en [5]. De esta manera se obtuvo una descripción Handel-C optimizada de una SVM lineal.

En aplicaciones BCI, cada sujeto requiere de una SVM particular, inclusive, para una misma persona, puede requerirse un reajuste de la SVM cada determinado tiempo. Nuevos resultados de investigación sobre algoritmos de extracción de características y nuevos clasificadores involucran una actualización de las SVM. Esto genera un problema para la implementación FPGA del clasificador ya que tiene que modificarse.

Para solucionar el problema enunciado se propuso realizar un generador automático del código Handel-C que describe la SVM requerida por el usuario. El algoritmo generador de código se obtuvo a partir del análisis de la descripción Handel-C optimizada. La hipótesis fue que a partir de los datos que definen a alguna SVM era posible generar el código Handel-C que la implementa en FPGA.

Se desarrolló una aplicación llamada *general code generator (gCodeGen)* que es independiente (*stand-alone*), portable, de código abierto y extensible con plugins; para auxiliar el proceso de generación de código, es decir, seleccionar archivos necesarios para generar el código, visualizar y almacenar el código generado. El gCodeGen también auxilia en la evaluación de la FPGA configurada con una SVM, es decir, permite seleccionar los archivos necesarios para la evaluación, ejecutar algún protocolo de evaluación y visualizar los resultados de la misma. El gCodeGen tiene varios puntos de extensibilidad que permiten agregar nuevos algoritmos de generación de código, protocolos de evaluación, puertos de comunicación y decodificadores de archivos.

La principal ventaja del gCodeGen desde el punto de vista del usuario es que una vez que se cuente con los parámetros que describen a alguna SVM su implementación en FPGA es inmediata y el que la realiza no requiere de conocimientos de sistemas digitales ni de Handel-C ni de la arquitectura del FPGA. Otra ventaja es que la evaluación automática requiere únicamente que el usuario configure el puerto de transferencia de datos, y que seleccione los archivos de vectores de prueba y de la clasificación patrón; para que luego se lleve a cabo automáticamente la evaluación; el resultado de la evaluación debe desplegarse en pantalla.

La principal ventaja del gCodeGen desde el punto de vista del desarrollador de plugins para generar código es que no tiene que preocuparse de agregar datos a menús, ni controlar el flujo de datos entre su plugin y otros, ya que el núcleo del gCodeGen fue diseñado para realizarlo, de tal forma que el desarrollador solo necesita implementar el algoritmo generador de código. De forma muy similar ocurre para el resto de los puntos de extensibilidad que se describirán más adelante en este capítulo.

Como ya se mencionó en el capítulo de Introducción, se deja al usuario la implementación del código Handel-C generado, para lo cual requiere del compilador Handel-C a EDIF y de alguna herramienta para compilar de EDIF a archivo de configuración. En esta tesis se ocupó el DK 4.0 para lo primero; y para lo segundo ISE Project Navigator.

En resumen; en este proyecto se realizó la optimización del código Handel-C que implementa SVM lineales, se analizó dicho código para construir un algoritmo generador de código. También se desarrollo un software auxiliar para facilitar la generación de código y la evaluación de implementaciones.

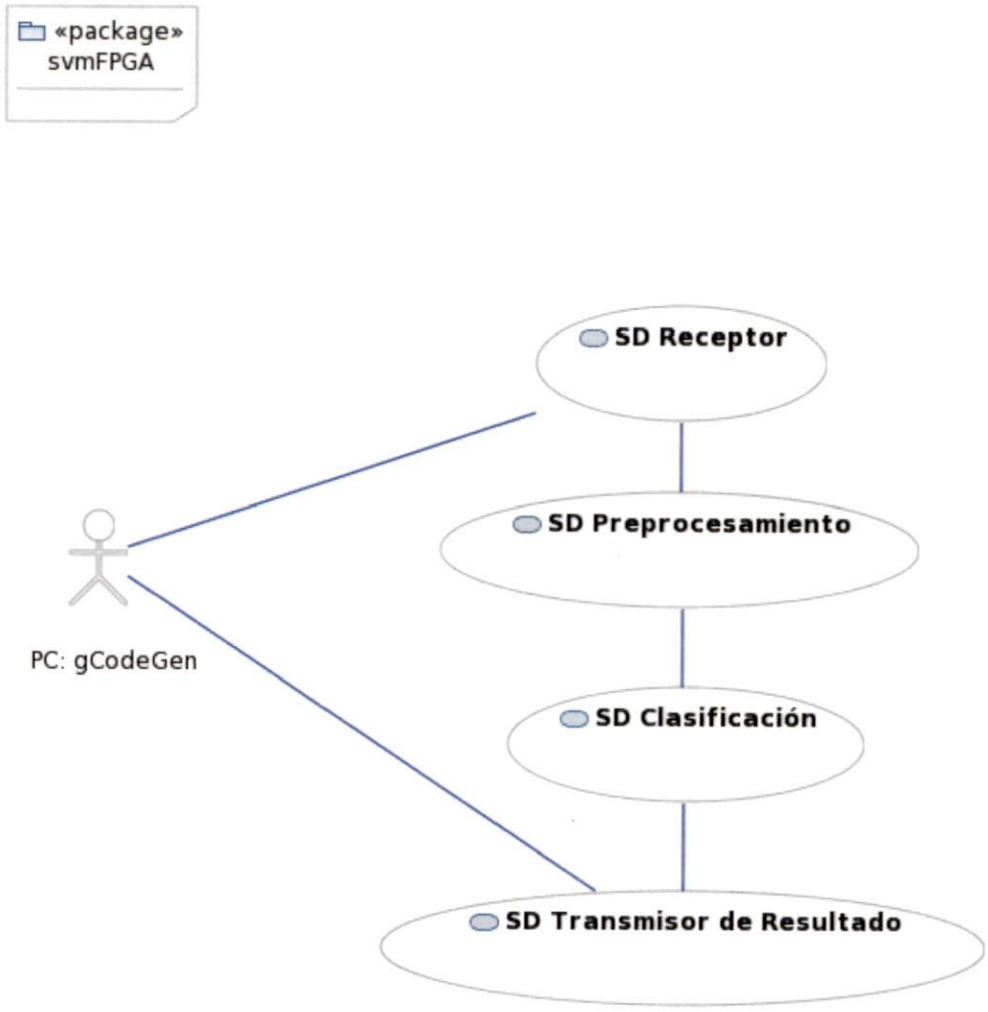


Figura 3.1 – *Representa el modelo de casos de uso del FPGA para realizar la clasificación con una SVM. Cada caso de uso representa a un sistema digital.*

3.2. Desarrollo de las descripciones Handel-C de las SVM

En adelante se conocerá como *descripción plantilla* a la descripción Handel-C resultado de la optimización de la descripción de [39,40], a esta última se le conocerá como descripción original. La plantilla fue desarrollada con el Celoxica Developer Kit 4.0, usando las bibliotecas *Platform Abstraction Layer* (PAL) y *Platform Support Library* (PSL). La implementación se realizó en una tarjeta de desarrollo RC10 Spartan 3 S31500L-4.

Las características iniciales de la descripción plantilla fueron ligeramente modificadas del original ya que la SVM lineal que implementa es de 4 vectores de soporte, 8 características.

3.2.1. Consideraciones de las descripciones Handel-C que implementan SVM.

Se realizó el modelado de la descripción original a través de diagramas de casos de uso y de actividades. En la figura 3.1 se muestran los casos de uso que requiere la descripción original y por ende el FPGA; para realizar la clasificación de un vector. Cada caso de uso es un sistema digital independiente y se comunican a través de canales sincrónicos como se muestran en dicha figura. La comunicación entre el gCodeGen y el FPGA se realiza con algún protocolo de comunicación; en este caso RS232. También se modelaron las actividades necesarias para realizar cada caso de uso que fueron utilizados para obtener la descripción plantilla.

Se realizó el análisis, diseño e implementación de algunos elementos de una plantilla SVM gaussiana, la cual no llegó a término ya que el método para para evaluar e^x contemplado para este proyecto sólo sirve para $x \in [0, 1]$ [41]. De la ecuación 2.15 se observa que el argumento de la exponencial siempre es negativo en consecuencia siempre se está fuera del intervalo válido para x. Aunque se completó el análisis de los elementos necesarios para su descripción, no se pudo realizar la implementación.

3.2.2. Construcción de la descripción Handel-C optimizada de la SVM lineal

Para optimizar recursos del FPGA se realizó una reducción de los parámetros que este requiere almacenar para llevar a cabo la clasificación. Reescribiendo la ecuación del clasificador SVM lineal con el sesgo explícito.

$$y(\mathbf{x}) = sign\left(\sum_{i=1}^{nvs} \alpha_i y_i \mathbf{x}_i^T \mathbf{x} + b\right)$$

Para una SVM entrenada los parámetros α_i, y_i y $\mathbf{x}_i$ son constantes por ello se define al vector $\overline{\mu}$ como la ecuación 3.1

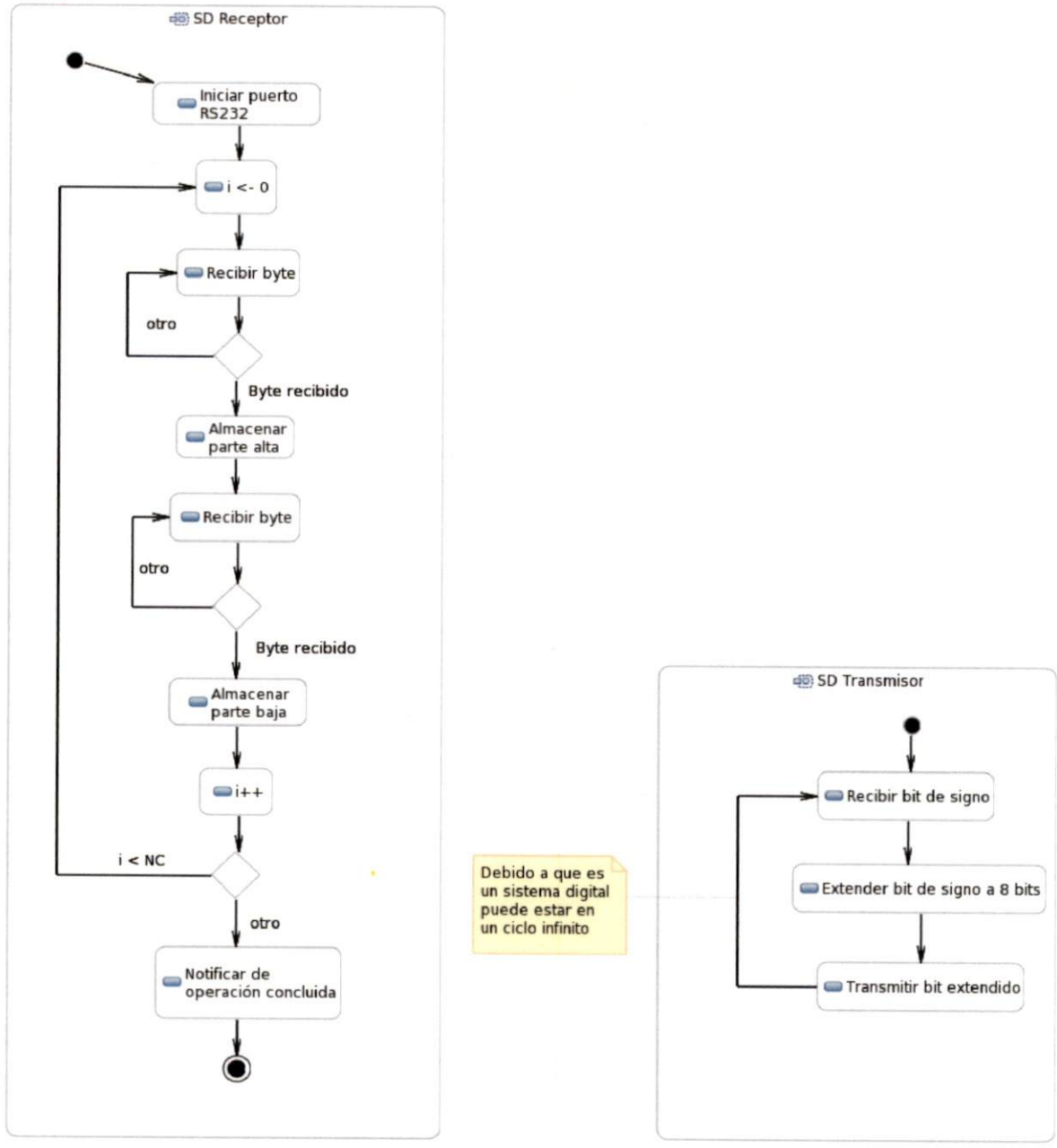
$$\overline{\mu} = \sum_{i=1}^{nvs} \alpha_i y_i \mathbf{x}_i \tag{3.1}$$

La ecuación 3.1 puede calcularse fuera del FPGA con Octave o Matlab; de esta forma la ecuación 3.2 es la que hay que implementar en el FPGA

$$y(\mathbf{x}) = sign\left(\overline{\mu}^T \mathbf{x} + b\right) \tag{3.2}$$

3.2.2.1. Modelado de los sistemas digitales que componen la descripción optimizada de la SVM lineal

Se realizó el modelo secuencial de la ecuación 3.2 y posteriormente se paralelizó con ayuda del modelo de la descripción original. Al modelo que resultó de este proceso se le aplicaron las técnicas de optimización de recursos descritas en [5]. Los modelos finales de los 4 sistemas digitales se muestran en las figuras 3.2(a), 3.2(b) y 3.3. Todos los modelos se construyeron de acuerdo al paradigma iterativo e incremental.



(a) Sistema digital de recepción

(b) Sistema digital transmisión

Figura 3.2 – *Modelos de los sistemas digitales de recepción y transmisión requeridos para recibir un vector de prueba y enviar el resultado de la clasificación*

Descripción del sistema digital receptor. Este realiza la recepción de los $2nc$ bytes que corresponden a las nc componentes del vector de prueba. Cada una requiere de dos bytes; una parte alta y otra baja debido a la representación Qn.m n+m=15 y al protocolo RS232 que envía bloques de 8 bits (figura 3.2(a)). Aquel tipo de representación fue utilizada debido a los resultados de [33, 40]. Este sistema implementa el protocolo RS232 para lo que crea una instancia del puerto por medio de la biblioteca PSL de Celoxica DK 4.0. La recepción de datos se realiza con una macro de la misma biblioteca, finalmente el control y almacenamiento de los datos se realizó con replicadores de hardware. La PC *host* es la encargada del envío de los vectores de prueba.

Descripción del sistema digital transmisor. Este recibe un bit que indica la clase del vector de prueba a través de un canal sincrónico de comunicación; al que le concatena 7 bits con valor cero cada uno; en la parte más significativa para formar la palabra que se envía por el puerto de comunicación hacia la PC-*host*. Este sistema digital también ocupó la biblioteca PSL para implementar el transmisor RS232 y uso una macro de la misma para el envío de la clase a la que pertenece el vector de prueba. La clase es enviada a una PC *host*. La interpretación del byte enviado es la siguiente: si el byte es cero la clase correspondiente es 1, mientras que si es uno la clase correspondiente es -1 (figura 3.2(b)).

Diseño de los sistemas digitales preprocesamiento y clasificación Por simplicidad en el modelo aparecen como un solo sistema digital (figura 3.3). La primera tarea del preprocesamiento es el acondicionamiento que consiste en recomponer los datos recibidos y almacenados en el sistema receptor. Se definieron dos arreglos de acceso paralelo XH[·] y XL[·] de $8{\times}nc$ bits; en donde se almacenan los bytes recibidos. La parte alta de la i-ésima componente del vector de prueba $\mathbf{x}$ se almacena en XH[i] y de forma similar para sus bits menos significativos que son almacenados en XL[i]. La recomposición consta de concatenar las componentes de XH[·] y XL[·]; el resultado se almacena en el banco de registros paralelos X[·] de $16{\times}nc$. Fue necesaria una extensión de signo de las componentes de X[·] a 32 bits; el resultado de esta operación se almacenó en el arreglo XE[·] de $32{\times}nc$.

Por otra parte, las componentes del vector $\overline{\mu}$ se almacenaron en el arreglo Mu[·] de $32{\times}nc$ y el sesgo b en el registro B de 32 bits. En el caso de las componentes de Mu[·], toda la información esta contenida en los 16 bits menos significativos de la palabra, el resto es una extensión de signo. El caso de B no es tan directo y se explicará más adelante.

Todas las concatenaciones se realizaron paralelamente, al igual que todas las extensiones de signo, excepto las de Mu[·] ya que sus componentes son inicialmente de 32 bits. Para dichas operaciones se ocupó el operador de concatenación de Handel-C "@" y la macro para extender el signo.

Terminado el preprocesamiento se inicia la clasificación. La clasificación en función de los registros es la ecuación 3.3 y la evaluación de la función *sign* es la ecuación 3.4

$$y = \sum_{i=0}^{NC-1} Mu[i]XE[i] + B \tag{3.3}$$

$$clase = MSB(y) \tag{3.4}$$

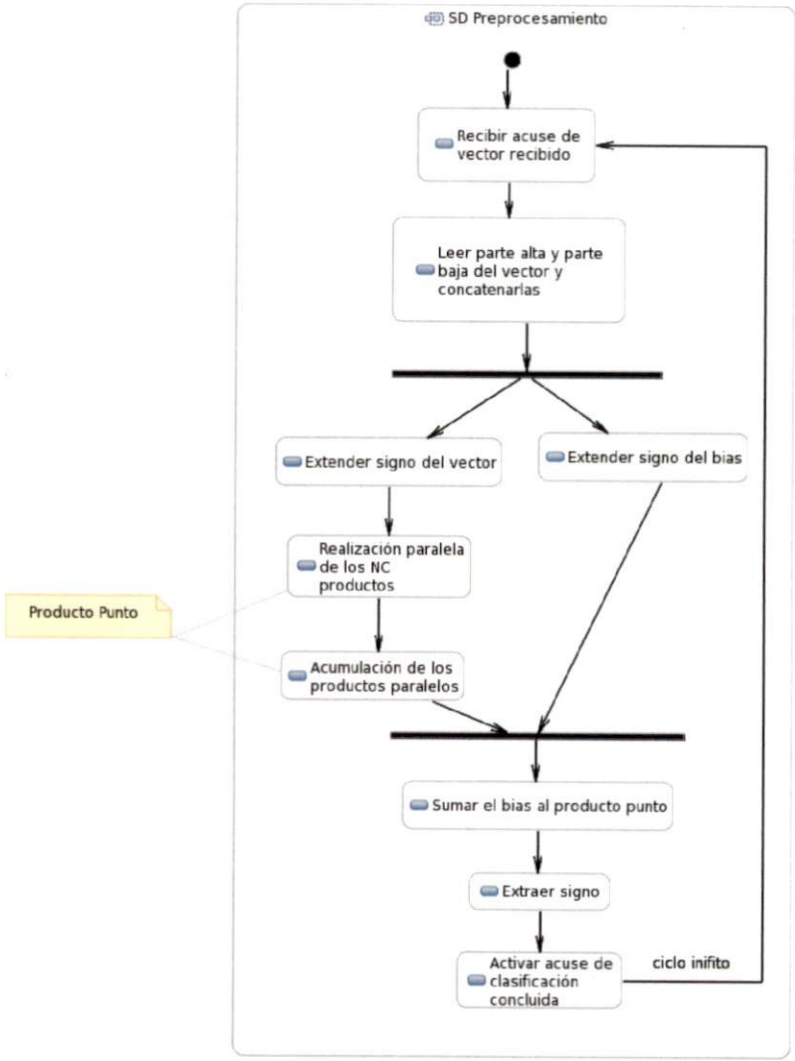


Figura 3.3 – *Modelo del preprocesamiento y la clasificación*

La extensión de signo a 32 bits de las componentes de X[·] y Mu[·] fue necesaria debido a que se debe realizar el producto componente a componente entre dichos arreglos y de acuerdo a la sintaxis de Handel-C; el tamaño de los operandos debe ser el mismo que el del resultado y este debe ser de 32 bits para contener todos los acarreos. El resultado de los productos se almacena en el arreglo P[·] de $32 \times nc$; esto se observa en ciclo 1 del *datapath* de la figura 3.4.

A diferencia de la descripción original; en la descripción optimizada se consideró que los vectores fuesen de 8 características; esto es una ventaja en el cómputo de la acumulación ya que puede realizarse en capas de sumas paralelas como se muestra en los ciclos 2, 3 y 4 de la figura 3.4. En el 2° ciclo la acumulación se almacena en un arreglo SP0[·] de 33×4; obsérvese que la precisión creció en un bit para contener el acarreo de la suma. De forma muy similar ocurre para el 3° ciclo donde el arreglo SP1[·] de 34×2 almacena la suma. La acumulación termina el

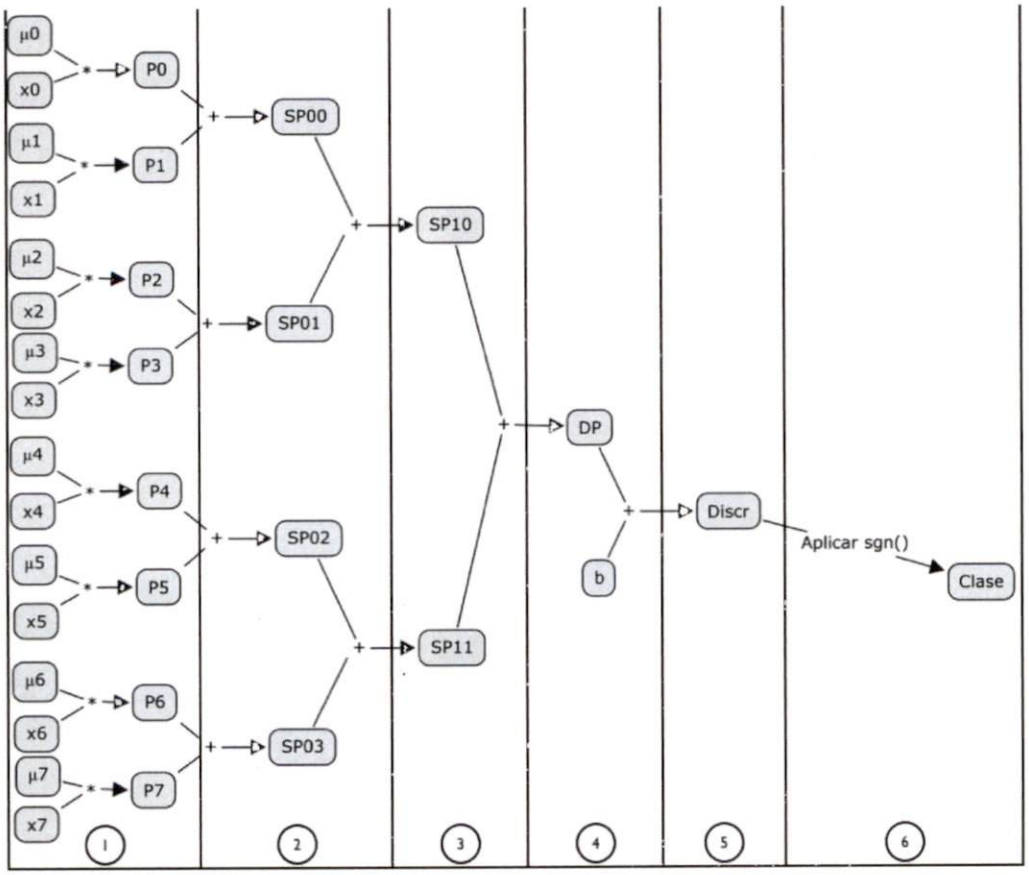


Figura 3.4 – *Muestra el datapath para realizar la clasificación en el FPGA. 1) Productos paralelos. 2) Primera capa de sumas paralelas. 3) Segunda capa de sumas paralelas. 4) Acumulación final. 5) Suma del sesgo. 6) Clasificación (evaluación de la función sign).*

4° ciclo donde el resultado se almacena en el registro DP de 35 bits. Esta forma de acumular solo es válida si nc es una potencia de dos.

La descripción Handel-C de los productos y las sumas paralelas fueron especificados con la sentencia *par* que indica que las operaciones deber realizarse paralelamente. De igual forma se utilizó dicha palabra reservada para cada capa de sumas paralelas, de ésta forma se asegura que todas las sumas de cada etapan se realizen concurrentemente.

Por otra parte, concurrentemente al cálculo del producto punto se realiza el acondicionamiento de B. Debido a que la precisión se ha modificado en cada ciclo es necesario un reajuste de B que se almacena en el registro BE para posteriormente realizar la suma de DP con BE que se almacena en el registro discr de 36 bits.

En general el reajuste de B Qm.n a BE Qent.frac se hace de la siguiente forma (ver diagrama de la figura 3.5):

1. Se inicializa BE con cero.

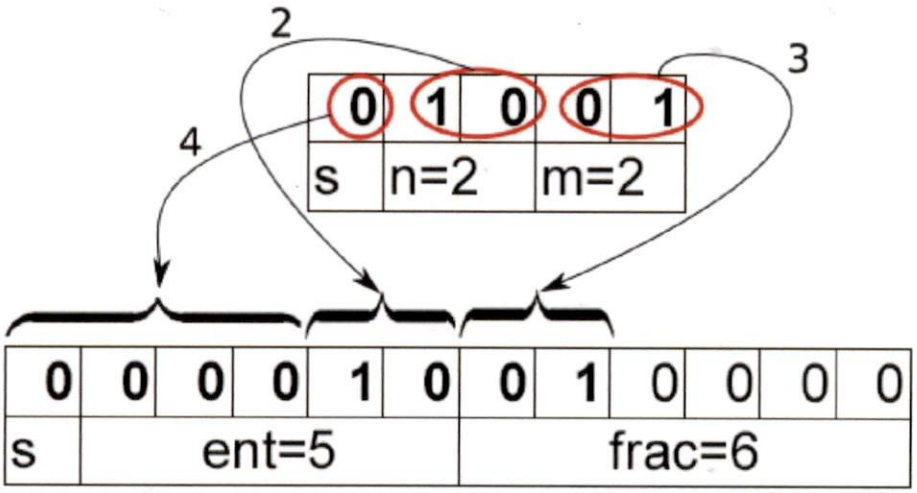


Figura 3.5 – *Muestra un ejemplo del reajuste de una B Q2.2 a una BE Q5.6. 1) Se asigna el vector BE con cero. 2) Se copian los bits enteros de B a la parte entera de BE. 3) Se copian los bits fraccionarios de B a la parte fraccionaria de BE. 4) Se extiende el signo en BE de acuerdo al MSB de B.*

BE$\leftarrow 0$.

2. Se copian los bits enteros de B a la parte entera de BE.
   BE$[frac + n - 1 : frac] \leftarrow$B$[n + m - 1 : m]$.

3. Se copian los bits fraccionarios de B a la parte entera de BE.
   BE$[frac - 1 : frac - m] \leftarrow$B$[m - 1 : 0]$.

4. Se realiza la extensión de signo de BE.
   BE$[ent + frac : frac + n] \leftarrow$B$[n + m]$.

En el ejemplo de la figura 3.5; m=2, n=2, ent=5, frac=6. Para la evaluación real los datos cumplen que m+n=15; ent+frac=36 y en general $ent = 2n + \ \#capas + 2$ y $frac = 2m$.

Se realizó también una descripción para la misma SVM agregándole dos componentes en cero, es decir, los vectores son de 10 características, aunque las dos últimas siempre son cero. Esta descripción sirvió para desarrollar el acumulador cuando las características no son potencia de dos. La acumulación se realizó de forma iterativa, en este caso particular se requieren de 9 ciclos para completarse. No son necesarios los arreglos de las capas de sumas paralelas intermedias; en cambio se usó un registro DP de 36 bits que contiene todos los acarreos de las iteraciones. Finalmente, la representación del *discr* fue Qent.frac $ent = 2n + \ \#iteraciones + 2$ y $frac = 2m$

Para ambos casos la clasificación consiste en extraer el MSB de *discr* y se almacena en el bit *clase*, que en general se expresa como, $clase = discr[ent + frac - 1]$.

3.2.2.2. Criterios de diseño de la descripción Handel-C optimizada del clasificador SVM lineal

El desarrollo de la plantilla fue guiado por resultados, esto es, se implementaban parcialmente el clasificador y se evaluaba la implementación parcial. Se corregían los errores; se optimizaba el proceso de clasificación y se continuaba con la siguiente iteración hasta completar el clasificador. También se consideraron para la evaluación de las implementaciones parciales los siguientes aspectos: Cantidad reducida de recursos y que el tiempo de realización no excediera 300 ms.

Debido al compromiso entre tiempo y recursos se buscó empíricamente el equilibrio. Por otra parte, la falta de referencias sobre implementaciones FPGA de SVM aplicadas a BCI impide que se tenga una idea de cual es la cantidad de recursos requeridos para su implementación. Por ello se decidió, arbitrariamente, un limite de los recursos del FPGA que deben utilizarse en la implementación FPGA del clasificador. El límite es del 20 %.

En cuanto a la descripción, el simulador de DK 4.0 requiere la inclusión de bibliotecas que no son sintetizables en FPGA, por ello la descripción de simulación difiere en algunos aspectos a la de implementación. Esto se consideró como un riesgo para el proyecto y es por eso que únicamente se desarrollo la descripción de implementación.

3.2.2.3. Proceso de clasificación

Una vez configurado el FPGA con el clasificador SVM lineal el resto de la BCI se emula con una PC que le envía con el protocolo RS232 un vector de prueba; la implementación SVM lo clasifica y transmite el resultado a la PC usando el mismo protocolo.

De igual forma se realiza la evaluación de la implementación; la diferencia radica en que la PC envía un conjunto de vectores de prueba de clase conocida, obtenida usando la misma SVM lineal implementada, sólo que utilizando un programa de cálculos matemáticos como Octave o Matlab, esta clasificación es llamada *clasificación patrón*. La clasificación del FPGA se transmite a la PC donde es cotejada con la clasificación patrón. En otras palabras, se está midiendo la similitud entre la implementación Octave y la implementación FPGA de una misma SVM lineal.

3.2.2.4. Análisis para desarrollar el algoritmo para generar código Handel-C que describe la SVM lineal requerida por el usuario

El vector $\overline{\mu}$ es el acumulado de los parámetros α_i, y_i y $\mathbf{x}_i$, es decir, como una especie de promedio. Por lo tanto, para cada SVM la precisión Qn.m debe ajustarse. La identidad n+m=15 tiene que variar de acuerdo al entero supremo de las componentes de $\overline{\mu}$ y de b.

La primera tarea del algoritmo generador de código es encontrar n y m de acuerdo a la regla antes mencionada. A continuación se transforman $\overline{\mu}$ y b a la representación Qn.m adecuada.

A priori se sabe que hay partes de la descripción SVM lineal que no se modifican. Siempre están los mismos sistemas digitales involucrados en la clasificación y además lo único que cambia de entre SVM lineales es el número de características de $\mathbf{x}$; los vectores de soporte; e implícitamente cuantos de ellos hay. Entonces se asume que tomando la descripción SVM lineal optimizada descrita en 3.2.2.1 como plantilla; puede generarse el código Handel-C de la SVM lineal requerida por el usuario aplicándole transformaciones que le adapten las tres características variables mencionadas.

3.2.3. Diseño y descripción de clasificadores SVM gaussianos y su implementación FPGA

Reescribiendo la ecuación de la SVM gaussiana

$$y(\mathbf{x}) = sign\left(\sum_{i=1}^{nvs} \alpha_i y_i e^{\rho||\mathbf{x}-\mathbf{x}_i||} + b\right)$$

se observa que el vector $\overline{\mu}$ se modifica (ecuación 3.5) ya que el $\mathbf{x}_i$ es parte del argumento de la exponencial. Se obtiene la función a implementar, ecuación 3.6

$$m = \sum_{i=1}^{nvs} \alpha_i y_i \tag{3.5}$$

$$y(\mathbf{x}) = sign\left(m \sum_{i=1}^{nvs} e^{\rho||\mathbf{x}-\mathbf{x}_i||} + b\right) \tag{3.6}$$

En la figura 3.6 se muestran los pasos requeridos para evaluar la ecuación 3.6. Mismos que se detallan a continuación:

1. Restas paralelas. Asumiendo que $\mathbf{x}$ y $\mathbf{x}_i$ son de 4 características; lo primero que se evalúa es la resta concurrente componente a componente entre dichos vectores. La i-ésima resta se almacena en el i-ésimo elemento del arreglo P.

2. Productos paralelos. Para obtener la norma de P se requiere de un producto punto. El primer paso es multiplicar componente a componente el arreglo P consigo mismo. Los resultados se almacenan en el arreglo PP.

3. Sumas paralelas, capa 0. Se realiza la acumulación de las componentes de PP a través de capas de sumas paralelas. En esta etapa los resultados se almacenan en el arreglo SP0.

4. Sumas paralelas, capa 1. Se continua con las capas de sumas paralelas. En esta etapa los resultados se en el registro DP.

5. Producto de ρ. Se lleva a cabo el producto entre DP y ρ, el resultado se almacena en el registro Arg.

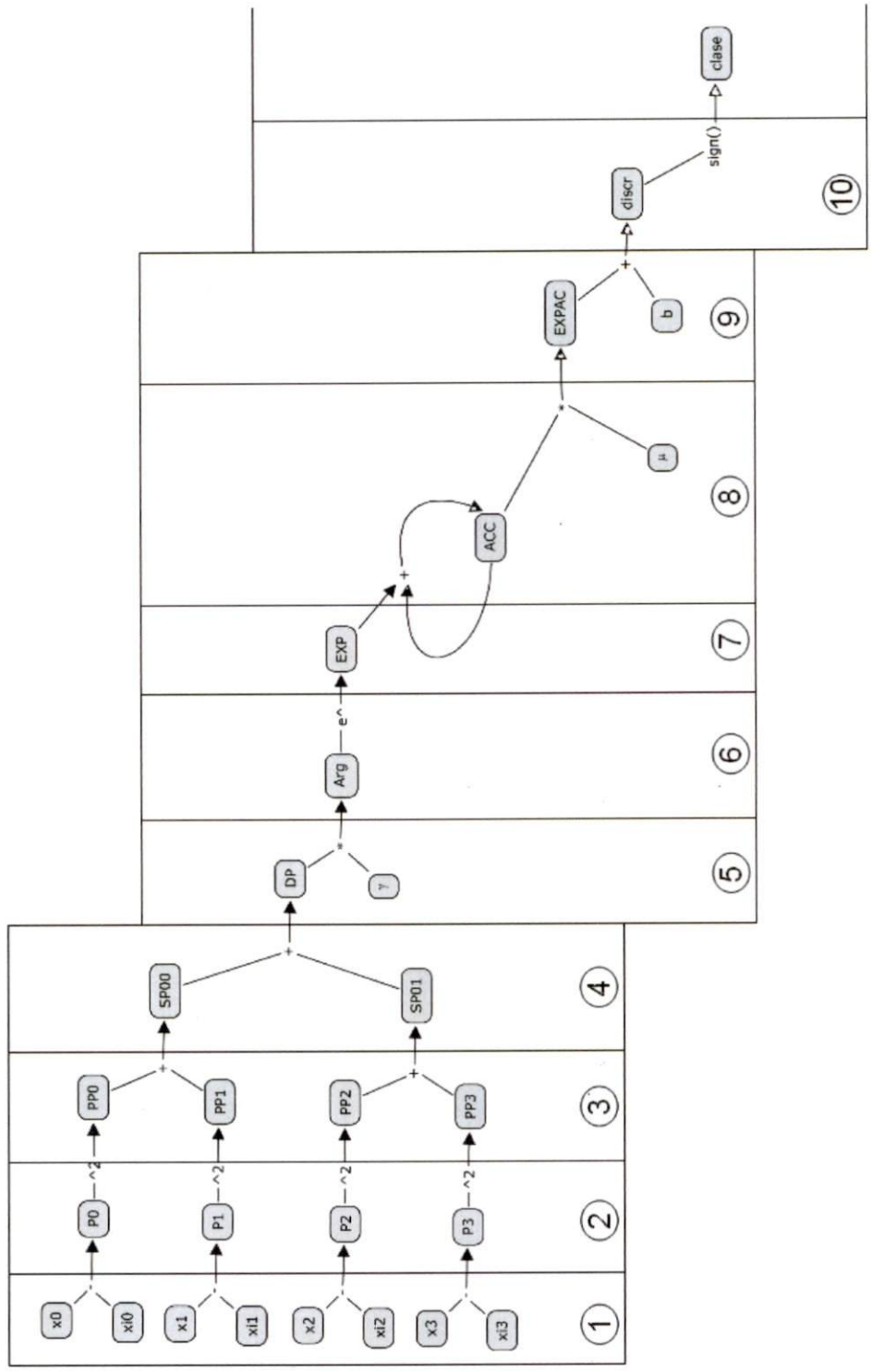


Figura 3.6 – *Muestra el datapath de la SVM gaussiana para vectores de 4 características. 1) Restas paralelas. 2) Productos paralelos (elevar al cuadrado). 3) y 4) Sumas paralelas. 5) Producto de ρ. 6) Evaluar la exponencial. 7) Acumular el resultado. 8) Multiplicar por el vector $\overline{\mu}$. 9) Sumar el sesgo b. 10) Clasificar (evaluar la función sign).*

6. Evaluar la exponencial. Como su nombre lo indica, consiste en evaluar la función exponencial. Se decidió el uso de CORDIC para tal propósito [13,41,48]. El resultado se almacena en el registro EXP.

7. Acumular el resultado. Consiste en sumar los resultados de EXP para cada vector de soporte, es decir, se tienen que repetir las etapas de 1 a 7 *nvs* veces. El almacenamiento se realiza en el registro ACC

8. Multiplicar por m. Al final de la acumulación se realiza el producto de ACC con el parámetro m. El resultado se almacena en el registro EXPAC.

9. Sumar el sesgo b. Se realiza la suma discr = EXPAC + b

10. Clasificar. Se extrae el MSB, es decir, se evalúa la función sign y se asigna al bit de clase.

De la etapa 1 a la 5 es esencialmente la misma descripción que para la SVM lineal. Por ello se procedió a la evaluación de la exponencial con el algoritmo de CORDIC.

Dicha descripción se realizó con *pipeline* con 16 iteraciones. cada iteración esta representada como un bloque de hardware.

Como se mencionó anteriormente no se consideró inicialmente el intervalo válido del CORDIC lo cual impidió completar la evaluación de la SVM gaussiana.

3.3. Desarrollo de la aplicación gCodeGen

El gCodeGen fue desarrollado en un principio para facilitar las tareas generar código y evaluar implementaciones FPGA, es decir, fue planteada como una aplicación auxiliar y experimental, por lo cual, no se contempló desarrollarla usando las prácticas de ingeniería de software. Pero, se tenía experiencia con el uso de Proceso Unificado de Desarrollo de Software [28] y con el OpenUP; se conocían las ventajas de su aplicación en el desarrollo de software, fue por ello que se retomaron algunas prácticas del último para el desarrollo del gCodoGen, pero se deja en claro que en este proyecto no fue un objetivo desarrollarlo usando las prácticas de la ingeniería de software.

Fue construido el *glosario de términos* del dominio del problema, se identificaron los requerimientos del sistema y se construyo el modelo de casos de uso que se muestra en la figura 3.7

Cada caso de uso se especificó con diagramas de actividad; en la tabla 3.1 se muestra un resumen de los casos de uso. También se realizaron pruebas de escritorio para evaluar si la especificación de los casos de uso satisfacía los requerimientos.

Se usaron las herramientas del RCP Eclipse para generar un núcleo de funcionalidad mínima de una aplicación a la cual se le agregó una capa intermedia para el manejo de los casos de uso y la ejecución de su flujo genérico, también permite el manejo de los plugins. Adicionalmente, se le agregaron un conjunto de plugins que implementan

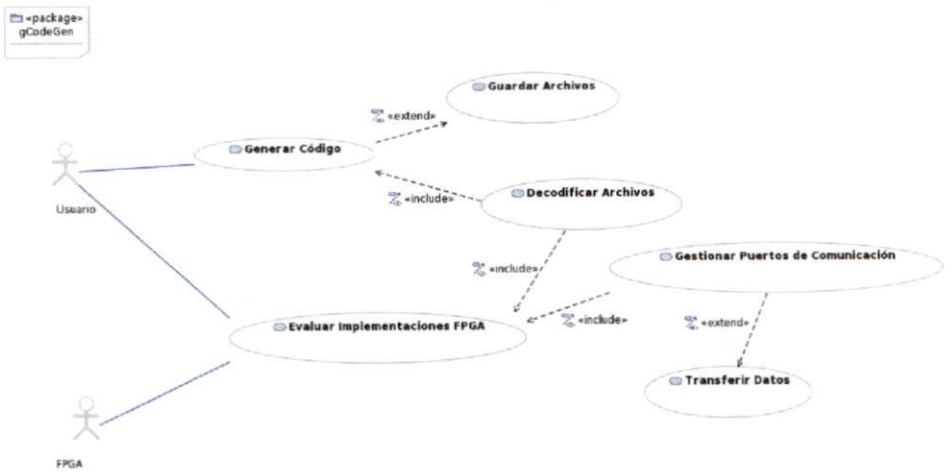


Figura 3.7 – *Muestra el diagrama de casos de uso de la aplicación gCodeGen.*

el flujo extendido de los casos de uso. Esta arquitectura se muestra en la figura 3.8 y consta de 4 partes o *packages* y las interfaces de los puntos de extensibilidad tardía:

1. Gestores. Son controladores de alto nivel, se ocupan de la gestión de la información recibida o enviada a los plugins. La gestión puede tiene dos propósitos principales que son: la actualización automática de la GUI y la realización de los casos de uso usando los plugins adecuados.
2. Herramientas. Son utilidades usadas frecuentemente en el entorno.
3. Controladores. Se encargan de la creación de instancias, del registro, transferencia de información y llamadas a los métodos de los plugins.
4. RCP Eclipse. Esta compuesta del *runtime kernel* para el manejo de bajo nivel de los plugins; algunos plugins de Eclipse IDE que le dan la funcionalidad mínima al gCodeGen.

   Por otra parte las interfaces de los puntos de extensibilidad tardía es donde los plugins son agregados para extender la funcionalidad del gCodeGen.

   De forma esquemática el gCodeGen esta compuesto de tres capas, la primera es el núcleo de bajo nivel (parte 4. RCP Eclipse); la segunda es el núcleo de alto nivel (partes 1, 2 y 3); la tercera capa está compuesta por los plugins que extienden la funcionalidad del gCodeGen (parte 5).

3.3.1. Desarrollo e implementación del gCodeGen

Se desarrolló el gCodeGen usando el RCP de Eclipse. A continuación se detalla el proceso.

3.3.1.1. Descripción del núcleo de bajo nivel del gCodeGen

El núcleo de bajo nivel del gCodeGen fue construido con los *wizards* del RCP de Eclipse versión 3.2. Este núcleo consta del *runtime kernel*, el manejador de vistas, el manejador de menús y del núcleo OSGi para el manejo de

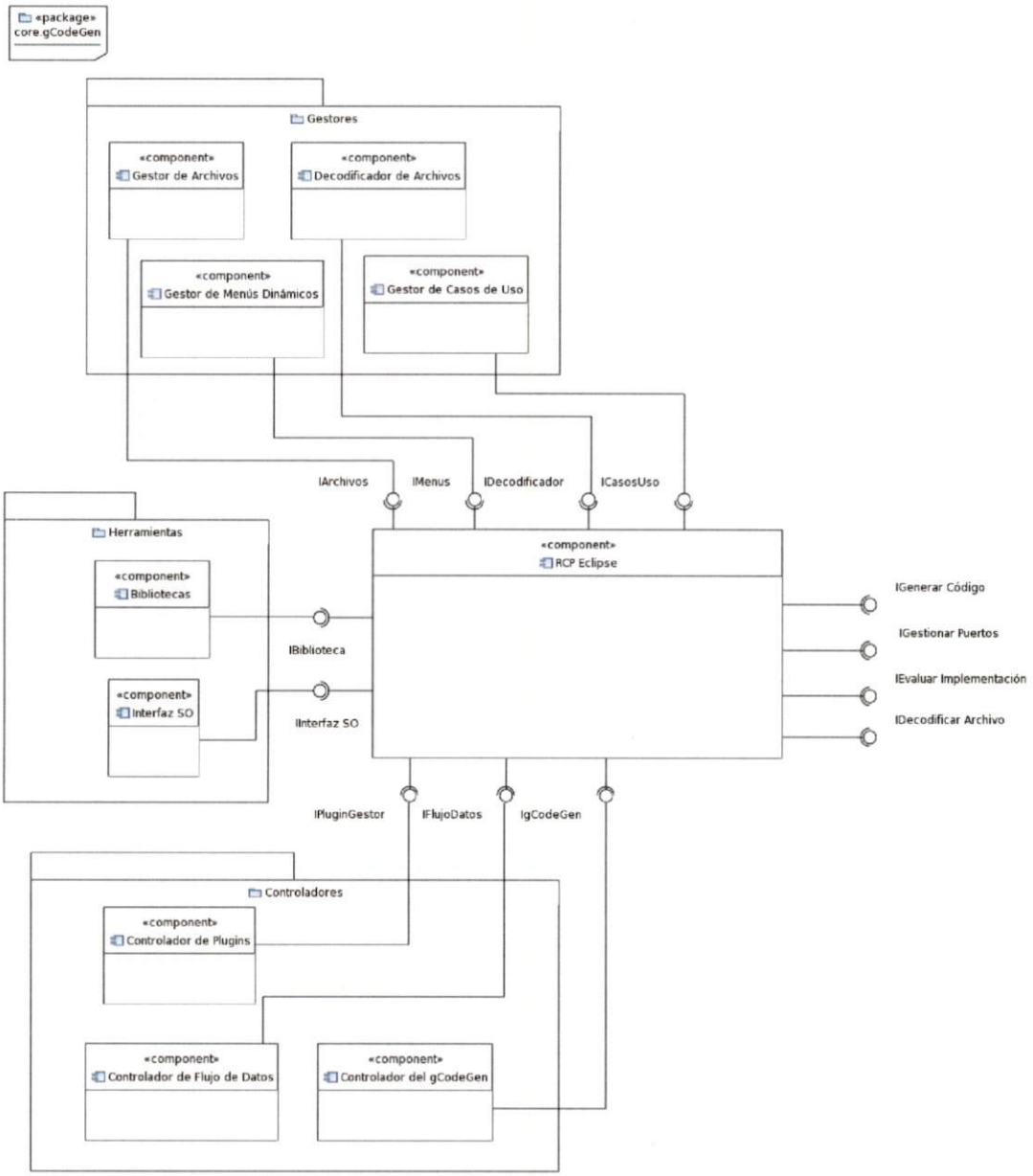


Figura 3.8 – *Muestra la arquitectura de diseño del gCodeGen. Se compone de 4 packages: 1) Gestores; clases que se dedican a la administración del gCodeGen. 2) Herramientas; contiene herramientas de uso frecuente como interfaz con el sistema operativo o bibliotecas de funciones. 3) Controladores; Se encargan de la inicialización y manejo del funcionamiento de los plugins y su interacción con el resto del gCodeGen. 4) RCP Eclipse; está compuesto del runtime kernel de eclipse y de algunos plugins básicos como son las GUI, los menús y el controlador RCP de plugins; y de las interfaces de los puntos de extensibilidad tardía.*

Nombre	ID	Descripción
Generar Código	UC001	Se encarga de la generación automática de código a partir de los parámetros especificados por el usuario en un archivo llamado *archivo fuente.*
Guardar Archivos	UC002	Almacena en un archivo de texto el código generado.
Evaluar Implementaciones FPGA	UC003	Realiza la evaluación automática de implementaciones FPGA. Compara los resultados obtenidos con dicha implementación con los obtenidos con una implementación patrón. Requiere de dos archivos un es el *archivo de prueba* que contiene los datos que enviarán al FPGA para que sean procesados; el otro es el *archivo patrón* es donde se almacenan los resultados esperados o patrón con los que se compararán los resultados del FPGA.
Decodificar Archivos	UC004	Interpreta la información contenida en los archivos fuente, archivos prueba y archivos patrón.
Gestionar Puertos de Comunicación	UC005	Realiza la conexión del gCodeGen con el sistema operativo para entablar la comunicación con el puerto seleccionado por el usuario. Los parámetros de configuración también son dados por él.
Transferir Datos	UC006	Se encarga de enviar y recibir información por el puerto seleccionado por el usuario.

Tabla 3.1 – *Muestra un resumen de las características principales de los casos de uso.*

los plugins.

3.3.1.2. Descripción del núcleo de alto nivel del gCodeGen

Controladores. Las instancias que están directamente ligadas al núcleo de bajo nivel son las que pertenecen a la parte de control del sistema; y se distribuyen de la siguiente forma: sobre el núcleo de alto nivel se monta la parte de control de los plugins relacionados al gCodeGen. A esta parte se conecta el control de flujo de información. Ambas partes son comandadas por el control del gCodegen. Nótese que hay dos controladores de plugins; uno es del núcleo de bajo nivel, es parte del sistema RCP, y maneja todos los plugins proveyendo los métodos para crear instancias y para acceder a los métodos de los plugins. El otro control del plugins que se encuentra en el núcleo de alto nivel, se encarga de usar adecuadamente los métodos de su homónimo para los fines del gCogeGen.

Aquí se mencionan los puntos de extensibilidad del gCodeGen, aunque más adelante se detallarán. 1) IGenerar

Código. 2) IGestionar Puertos. 3) IEvaluar Implementación. 4) IDecodificar Archivo. Dicho esto, los métodos más importantes del control de alto nivel son:

- Registro de plugins: se encarga de leer todos los plugins conectados a algún punto de extensibilidad, crea las instancias de cada uno y almacena una referencia a ellas en un arreglo denominado *registro de punto de extensibilidad*
- Obtener referencia: regresa la referencia al plugin localizado en la i-ésima posición del registro de algún punto de extensibilidad.
- Transferencia de información: se encarga de recibir datos de un plugin y enviarlos a otro plugin. Estos pueden o no pertenecer al mismo punto de extensibilidad.

Herramientas. Son dos las herramientas que se incluyeron en el núcleo gCodeGen; la primera es el decodificador de archivos MAT que pertenece al *package* Bibliotecas. Octave y Matlab son muy utilizados para el entrenamiento de SVM por ello se decidió incluir en el núcleo un decodificador de archivos tipo MAT.

El decodificador de archivos MAT Matlab-Octave se montó sobre la biblioteca para abrir archivos MAT llamada *JMatIO* [23]. La aportación fue la construcción de métodos que facilitan el uso de dicha biblioteca. Se implementaron métodos para transformar los datos decodificados con la biblioteca *JMatIO*, en arreglos de doble precisión que son más fáciles de manejar que los tipos de datos propuestos en dicha biblioteca.

El proceso de decodificación de archivos MAT se puede resumir en los siguientes pasos:

a) Usuario selecciona archivo.

b) Verificar que el archivo sea tipo MAT.

c) Abrir un flujo de datos del archivo.

d) Extraer el contenido de las matrices almacenadas en el archivo[1]. Las matrices que se requieren leer deben especificarse con su identificador. Si algún identificador no concuerda con las matrices almacenadas, se lanza una ventana de error *"dato no almacenado en el archivo"*

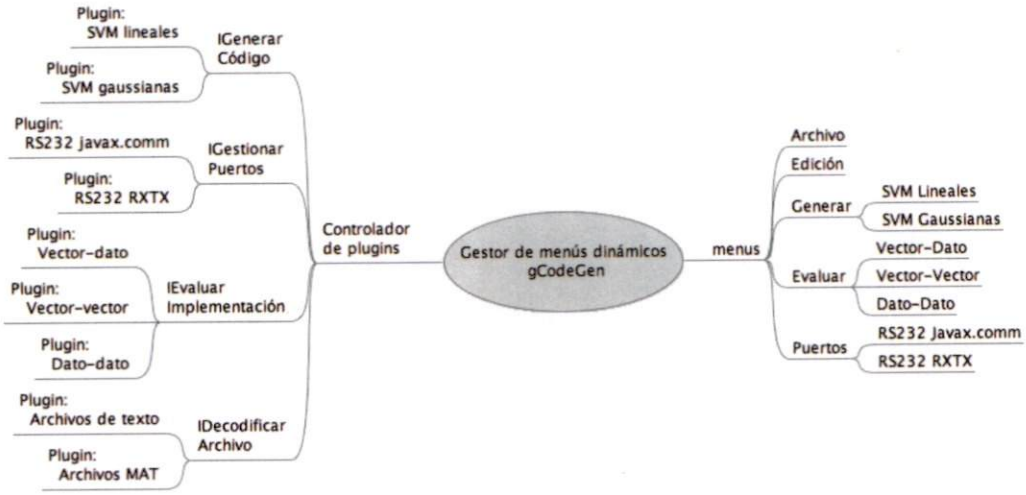
e) Transformar en arreglos de doble precisión de Java.

La otra herramienta que se incluyó en el gCodeGen es una *biblioteca para trabajar con datos en código binario*, también incluida en el *package* Bibliotecas. Esta implementa las siguientes operaciones:

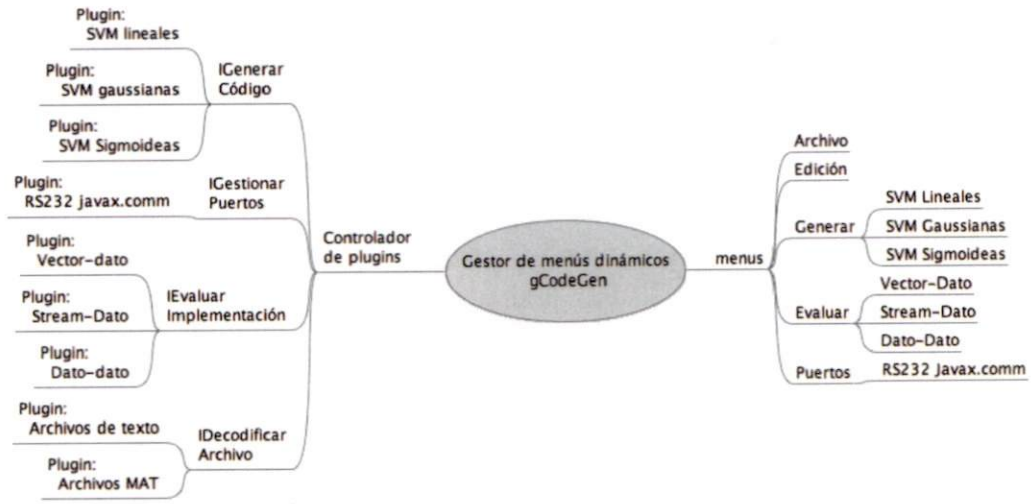
Transformar datos de doble precisión punto flotante a una representación Qn.m de 16 bits. La transformación solo se lleva a cabo si la precisión permite representar el dato; en caso contrario el resultado de la transformación se satura en positivo o negativo según sea el caso. El cálculo de la parte entera n se realiza automáticamente.

La transformación funciona de la siguiente forma: asumiendo que el dato a trasformar es r. Este es recorrido m bits a la izquierda; se trunca hasta el punto decimal y se verifica que el valor obtenido este entre -2^n y $2^n - 1$. Si se cumple la condición se regresa el número encontrado y en caso contrario se satura el resultado.

[1]Los datos almacenados en archivos MAT están contenidos en matrices de números de doble precisión o cadenas de caracteres. Cada matriz tiene un identificador [53]



(a) Antes



(b) Después

Figura 3.9 – *Muestra un esquema comparativo de los menús cuando algunos plugins son agregados y otros eliminados. La figura a muestra el estado inicial de los menús y la figura b se muestra como los menús se modifican de acuerdo a la agregación y eliminación de plugins.*

Dividir palabras en bytes. Sirve para separar en los bytes necesarios un dato de que se almacene en más de 8 bits. Esto es muy útil para enviarlo por partes a través de un puerto serial; ya que en la mayoría de estos, el tamaño de las palabras transmitidas es de 8 bits.

En el caso de que la palabra a dividir x sea de 16 bits; la parte baja es $x \; mod \; 256$ y la parte alta es $x - x \; mod \; 256$. Este método puede generalizarse para palabras de más de 16 bits.

Gestores. Son los controladores integradores de los núcleos tanto de bajo como de alto nivel. Se compone de las siguientes partes:

Gestor de Archivos. Se encarga de la apertura de los archivos. Verifica que los archivos seleccionados por el

usuario sean los requeridos por el caso de uso.

Decodificador de Archivos. Tiene un registro de los plugins que decodifican archivos y una referencia que le ayuda a saber que tipo de archivos decodifican. Implementa una variación del patrón de diseño *cadena de responsabilidad* [20] para la selección automática del decodificador del archivo seleccionado por el usuario. La variación del patrón de diseño consiste en que el decodificador de archivos es el encargado de la decisión de seguir buscando un decodificador adecuado; cada plugin llamado por el decodificador de archivos regresa un booleano que indica si puede o no decodificar el archivo. Si se recorre todo el registro de los plugins decodificadores y no se haya un decodificador adecuado, se lanza una ventana de error *"no hay un decodificador adecuado para el archivo seleccionado"*. Esta herramienta es utilizada por el UC004 Decodificar Archivos.

Gestor de Menús Dinámicos. El gestor de menús dinámicos se encarga de actualizar automáticamente los menús de acuerdo a los plugins que se estén agregados. En la figura 3.9 se muestra un ejemplo del funcionamiento de este gestor, a continuación se explica: en la figura 3.9(a) se observa al gestor de menús dinámicos conectado al controlador de plugins (lado izquierdo); el que controla los cuatro puntos del gCodeGen; cada punto de extensibilidad tiene sus propios plugins conectados. Del lado derecho se observan los 5 menús que componen al gCodeGen. Notese la correspondencia que existe entre los plugins y elementos de los menús para los siguientes casos: Plugins Generar Código→Menú Generar; Plugins Gestionar Puertos→Menú Puertos; Plugins Evaluar Implementación→Menú Evaluar. En la figura 3.9(b) se muestra como al modificarse la estructura de plugins los menús también se modifican.

La implementación de este gestor se realizó usando el registro de plugins de cada punto de extensibilidad provisto por el control de plugins de alto nivel. Sabiendo que únicamente los plugins de los puntos de extensibilidad Generar Código, Gestionar Puertos y Evaluar Implementación son los que modifican la estructura de los menús. Para cada uno de ellos; el gestor de menús implementa un método que recorre sus respectivos registros; a cada plugin del mismo se le requiere el nombre del elemento que ha de agregarse al menú.

La creación de menús se realizó con el núcleo de bajo nivel que requiere de cuatro parámetros: el nombre del menú donde se agrega el elementos, el nombre del submenu, nombre del elemento, y la acción que se realiza al dar clic en el elemento.

Gestor de Casos de Uso. En la figura 3.10 se muestra un ejemplo de como funciona el gestor de caso de uso. *1)* El usuario selecciona un elemento de cualquiera de los 5 menús. El núcleo de bajo nivel detecta la acción. *2)* El gestor de casos de uso es notificado de la acción y a través de tablas de *hash* se selecciona el controlador del caso de uso adecuado para la acción; de dichas tablas también se obtienen los índices de de los registros de puntos de extensibilidad en donde se almacenan las referencias a los plugins requeridos para realizar el caso de uso (plugins involucrados). *3)* El gestor de casos de uso pide al controlador de plugins de alto nivel las referencias de los plugins involucrados; para esta operación el controlador de plugins requiere de los indices obtenidos en la etapa anterior. *4)* El gestor de casos de uso tiene las referencias a los plugins involucrados y al controlador del caso de uso correspondiente. *5)* El gestor de casos de uso ejecuta el controlador del casos de uso y le a este envía como parámetros las referencias de los plugins involucrados.

3.3.1.3. Descripción de los puntos de extensibilidad del gCodeGen

Durante el diseño de la arquitectura de implementación se definieron y especificaron los puntos de extensibilidad tardía. En la tabla 3.2 se hace un compendio de los puntos de extensibilidad; el caso de uso que extienden; y su

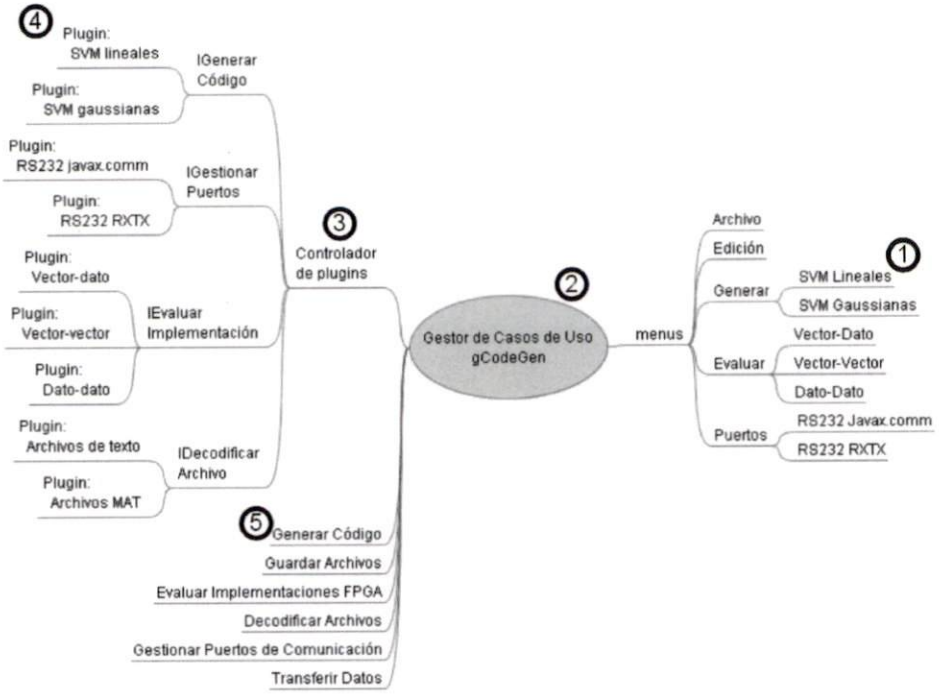


Figura 3.10 – *Muestra un ejemplo de como se lanzan los casos de uso en el gCodeGen. El ejemplo es la secuencia de ejecución del caso de uso Generar Código; aunque es muy similar para todos los casos de uso. 1) El usuario selecciona la acción Generar→SVM lineales. 2) El gestor de casos de uso con la ayuda del núcleo de bajo nivel, detecta la acción. 3 y 4) Con tablas de* hash *y requisiciones al controlador de plugins de alto nivel se identifica el controlador de casos de uso (en este caso es controlador de generar código) y los plugins involucrados en la ejecución del caso de uso (en este caso SVM lineales del punto se extensibilidad Generar Código). 5) Finalmente el gestor de casos de uso indica al controlador de casos de uso seleccionado que ejecute el caso de uso.*

objetivo.

3.3.2. Descripción de la implementación de los casos de uso.

Cada caso de uso tiene dos flujos de ejecución: *flujo genérico* que se lleva a cabo en le núcleo de alto nivel y consiste en todos las tareas comunes que el caso de uso realiza. el otro flujo es el denominado *flujo extendido* que es llevado a cabo por cada plugin de algún punto de extensibilidad y es el que da la especificidad al caso de uso. La comunicación entre ambos flujos es a través de mensajes de comandos o de información.

3.3.2.1. Diseño e implementación del UC001 Generar Código

La generación del código es la parte medular del gCodeGen. Para llevarla a cabo requiere del caso de uso decodificar archivo UC004. En la figura 3.11 se muestra el modelo de secuencia de actividades de este caso de uso; esta se describe a continuación:

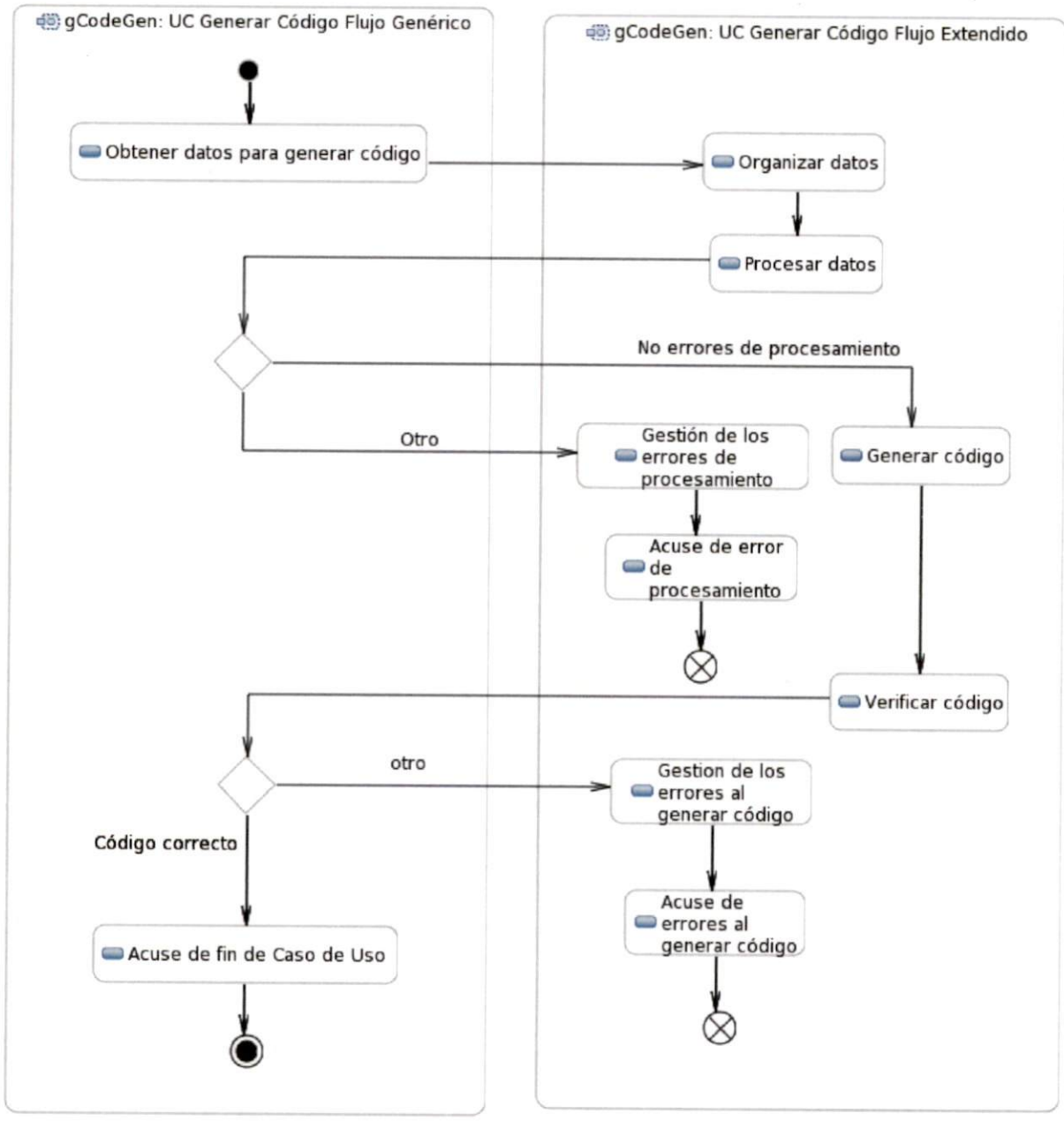

gCodeGen: UC Generar Código Flujo Genérico
gCodeGen: UC Generar Código Flujo Extendido
Obtener datos para generar código
Organizar datos
Procesar datos
No errores de procesamiento
Otro
Gestión de los errores de procesamiento
Generar código
Acuse de error de procesamiento
Verificar código
otro
Gestion de los errores al generar código
Código correcto
Acuse de errores al generar código
Acuse de fin de Caso de Uso

Figura 3.11

UC	Puntos de Extensibilidad	Objetivo
UC001	IGenerar Código	Para poder implementar más algoritmos generadores de código. Diversificar el proceso de optimización de recursos o el lenguaje en el que se genera el código.
UC003	IEvaluar Implementaciones	Poder incrementar los tipos de protocolos de evaluación de dispositivos. Incluso que estos no sean FPGA. Así como la diversificación de la medición del error.
UC004	IDecodificar Archivos	Ampliar el tipo de archivos que pueden ser leídos, incluso leer la información de una base de datos.
UC005	IGestionar Puertos	Diversificar los protocolos de comunicación para incrementar las aplicaciones del gCodeGen.

Tabla 3.2 – *Muestra los puntos de extensibilidad; a qué caso de uso pertenecen; su objetivo.*

- *Flujo genérico:* Usa al UC004 para abrir y decodificar un archivo fuente (contiene los parámetros necesarios para la generación de código).
- *Flujo extendido:* Organiza y procesa los datos leídos del archivo para iniciar la generación de código. Todos los métodos que realizan lo anterior registran su éxito en un arreglo.
- *Flujo extendido:* Lee dicho arreglo y si encuentra alguna falla indica al flujo extendido que gestione el error ocurrido; en otro caso sigue la ejecución normal del flujo extendido.
- *Flujo extendido:* Genera el código y lo verifica.
- *Flujo genérico:* Si la verificación no tuvo errores termina el caso de uso exitosamente en caso contrario delega al flujo extendido el manejo del error.
- *Flujo extendido:* Maneja el error y lo reporta al núcleo de alto nivel.

El código generado se despliega en la pantalla principal de gCodeGen.

Se desarrollaron e implementaron dos plugin para el punto de extensibilidad Generar Código. Uno para generar código Handel-C que describe SVM lineales y otro para SVM gaussianas.

Plugin Generar Código SVM lineales: Este es lo que se ha denominado a lo largo de la tesis el algoritmo generador de código.

Se requiere que el archivo fuente tenga los parámetros α_i, y_i, x_i y b. De acuerdo al análisis que se realizó a la descripción Handel-C de la SVM lineal el proceso de generación de código es el siguiente:

Organización y procesamiento de la información. Leídos los parámetros del archivo son utilizados para el cálculo del vector $\overline{\mu}$. Seguidamente se obtiene su representación Qn.m y la de b. La parte entera n se calcula evaluando el logaritmo base dos al supremo de las componentes de $\overline{\mu}$ y b este resultado es truncando al entero superior

más próximo. La parte fraccionaria m es el resultado de n-15. Si m > 15 la SVM no puede implementarse con 16 bits; este error es registrado en el arreglo de éxito.

Conocidos n y m se utilizan los métodos provistos por la biblioteca de datos en código binario para obtener la representación Qn.m.

Generar Código. Del análisis del código de la descripción óptima se encontró que la estructura de la descripción se mantiene constante para todas las SVM lineales, excepto para los siguientes puntos:

a) Número de características de $\overline{\mu}$.

b) Número de bits de la parte entera n.

c) Número de bits de la parte fraccionaria m.

d) Número de bits para representar el resultado del producto punto.

e) Número de bits para representar la suma del producto punto y b.

El generador de código construye la arquitectura invariante de la descripción SVM lineal. Una vez concluido; continua con la adaptación de los cinco puntos anteriores de acuerdo a los parámetros leídos del archivo.

El algoritmo generador de código utiliza 6 funciones para lograr su objetivo, estas se detallan a continuación:

a) *función agregar información.* Agrega a manera de comentarios la información sobre el autor, lugar de desarrollo y forma de contacto.

b) *función agregar declaraciones.* Se realizan las declaraciones de variables globales, constantes globales, funciones, procedimientos, macros y canales de comunicación.

c) *función crear función principal.* Crea la función *main* encargada del control de la evaluación de la SVM lineal; ya que llama al resto de las funciones y macros en el orden adecuado.

d) *función crear transmisión serial en modo enviar.* Se encarga de crear la función que realiza una instancia del puerto serial RS232 de la tarjeta RC10 configurada para el envío de datos. El código usa la biblioteca PSL del Celoxica Co.

e) *función crear transmisión serial en modo recibir.* Se encarga de crear la función que realiza una instancia del puerto serial RS232 de la tarjeta RC10 configurada para la recepción de datos. El código usa la biblioteca PSL de Celoxica Co.

f) *función crear el secuenciador del clasificador.* Crea el código que recompone las componentes del vector $\overline{\mu}$ y de b que ya han sido recibidas por el puerto serial. Luego realiza los nvs productos paralelos. Realiza la acumulación ya sea por capas de sumas paralelas o iterativamente. Suma el producto punto con b, no sin antes haber acondicionado b. Finalmente se extrae el MSB del resultado. Para que sea enviado por el puerto serial.

Plugin Generar Código SVM gaussianas: También se desarrollo un algoritmo generador de código Handel-C que describe SVM gaussianas; esencialmente sigue los mismos pasos que el plugin anterior; solo que se le

aumentan restas paralelas y la evaluación de la exponencial para cada vector de soporte. Debido a que el evaluador de la exponencial descrito en la descripción SVM gaussiana no funcionó como se esperaba este algoritmo solo se concluyó con la finalidad de probar que se podían agregar plugins y se modifican los menús; ya que la descripción SVM gaussiana no se completó.

3.3.2.2. Diseño e implementación del UC002 Guardar Archivos

Debido a que la generación de código resulta siempre un archivo de texto o binario no se consideró necesario crear un punto de extensibilidad en este caso de uso. Por ello solo tiene flujo genérico y consiste en abrir un *OutputStream* que escribe el código generado; que también es desplegado en la ventana principal de la GUI; en un archivo. cuya extensión es dada por el usuario.

3.3.2.3. Diseño e implementación del UC003 Evaluar Implementaciones FPGA

Depende de los casos de uso: UC004 Decodificar Archivos; UC005 Gestionar Puertos de Comunicación; UC006 Transferir Datos. En la figura 3.12. A continuación se resume la ejecución de este caso de uso.

a) *flujo genérico:* Se abren dos archivos, el de prueba y el patrón con invocando los métodos del UC004 y sus plugins. Se verifica que puedan decodificarse y que contengan la información correcta.

b) *flujo extendido:* Abre una ventana para la selección del protocolo de transferencia de datos; ejemplos de estos son: se envía un vector y se recibe un dato; se envía un dato y se recibe un vector; o se envía un dato y se recibe un dato; incluso; se envía un dato y no se espera recibir algo).

c) *flujo genérico:* Verifica la selección correcta del protocolo y que se haya configurado un puerto para la transferencia de los datos. Si al menos una verificación falla se reporta el error y termina el caso de uso. Si no se continua con el flujo extendido.

d) *flujo extendido:* Ejecuta la evaluación de la implementación usando el protocolo de transferencia seleccionado. Se mide el error entre los resultados patrón y los obtenidos por el FPGA y se despliega en una pantalla auxiliar que requiere ser cerrada por el usuario.

e) *flujo genérico:* Termina la ejecución del caso de uso, es decir, cierra todos los archivos y regresa a la ventana principal.

Se desarrollaron e implementaron varios plugin del punto de extensibilidad Evaluar Implementaciones FPGA la mayoría destinados al proceso de desarrollo y evaluación de las descripciones SVM plantilla (las que se ocuparon para desarrollar los algoritmos generadores de código); muy pocos se dejaron en la versión liberada del gCodeGen.

Plugin Evaluar implementaciones FPGA de SVM. El gCodeGen se encarga de enviar, con algún protocolo de comunicación, los vectores de prueba a la implementación SVM misma que regresa al gCodeGen el resultado de la clasificación. Se denominó protocolo de transferencia a como el este plugin envía y recibe datos; en el caso de las SVM el protocolo de trasferencia es enviar todas las componentes del vector a clasificar y recibir un byte con el resultado de la clasificación.

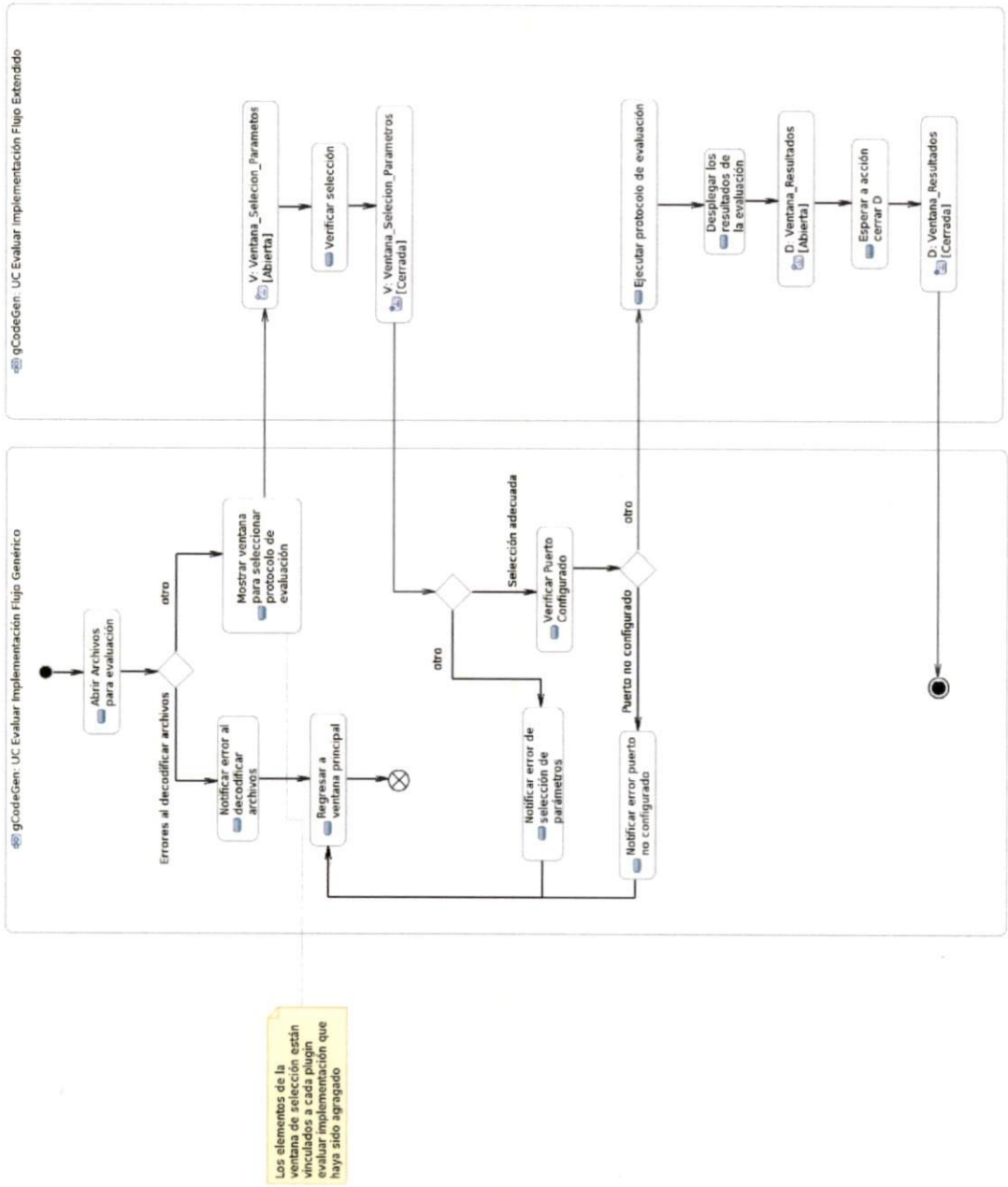


Figura 3.12 – *Muestra el modelo de secuencia de actividades requeridas para la ejecución del UC003 Evaluar Implementación FPGA*

Tipos de archivo. Los archivos con los vectores de prueba y el patrón pueden estar en diferentes codificaciones. Los vectores obtenidos del primero se tienen que tratar de la misma forma que para generar código (obtener la representación Qn.m y partirla en bytes para la transmisión por el puerto). Los datos obtenidos del segundo son únicamente bytes listos para compararse.

Evaluación de la implementación FPGA. Recuerdese que esta evaluación mide la equivalencia entre el clasificador que realiza los cálculos con punto flotante en doble precisión en la PC y la realizada en el FPGA precisión punto fijo de 16 bits.

Se realizó la comparación de la clasificación patrón con la realizada en el FPGA elemento por elemento. Cada discrepancia incrementa en uno un acumulador. Al final de prueba que es cuando se han clasificado todos los vectores de prueba, el error es el cociente de la diferencia entre el número total de vectores de prueba y el acumulador y el primer operando de la diferencia. El resultado de esta operación se lanza en una ventana.

3.3.2.4. Diseño e implementación del UC004 Decodificar Archivos.

Este caso de uso realiza una secuencia de configuración que únicamente se ejecuta la primera vez que se realiza este caso de uso. En la figura 3.13 se muestra el modelo de diagrama de actividades del UC004 Decodificar Archivo mismo que se describe a continuación:

- *Flujo normal:* Se realiza la secuencia de configuración que consta de pedir al controlador de plugins de alto nivel una referencia al registro de los plugins conectados al punto de extensibilidad decodificar archivo. Se abre una ventana de navegación de archivos donde el usuario elige alguno. Si se cancela la operación se termina el flujo del caso de uso.
- *Flujo extendido:* Se abre el archivo. Luego intenta decodificarlo, si logra hacerlo, regresa una cadena de objetos; en caso contrario regresa una cadena nula.
- *Flujo normal:* Notifica a su controlador de caso de uso que terminó la decodificación.

Controlador del caso de uso Decodificar Archivo. A diferencia los casos de uso anteriores en los cuales el controlador de casos de uso solamente llama al control genérico del caso de uso y le envía como parámetro una referencia del plugin que extiende el flujo. Este controlador es más complejo; como se describe a continuación: En la figura 3.14 se encuentra un esquema de los elementos involucrados en la decodificación de archivos:

a) Alguno de los casos de uso que requiere de la información de un archivo realiza una llamada al caso de uso decodificar archivo.

b) Realiza la secuencia de configuración.

c) Solo requerido en la secuencia de configuración. En el caso de la figura 3.14 regresa el registro de ese punto de extensibilidad; en este caso de 4 elementos.

d) Llama al Decodificador de Archivos del núcleo de alto nivel. Le envía el registro de plugins como parámetro.

e) Ejecuta la cadena de responsabilidad modificada. Es decir, recorre cada elemento del registro para encontrar el decodificador adecuado. Regresa un entero $i$que corresponde al índice de registro que contiene al decodificador adecuado. Si no hay un decodificador adecuado se regresa un -1.

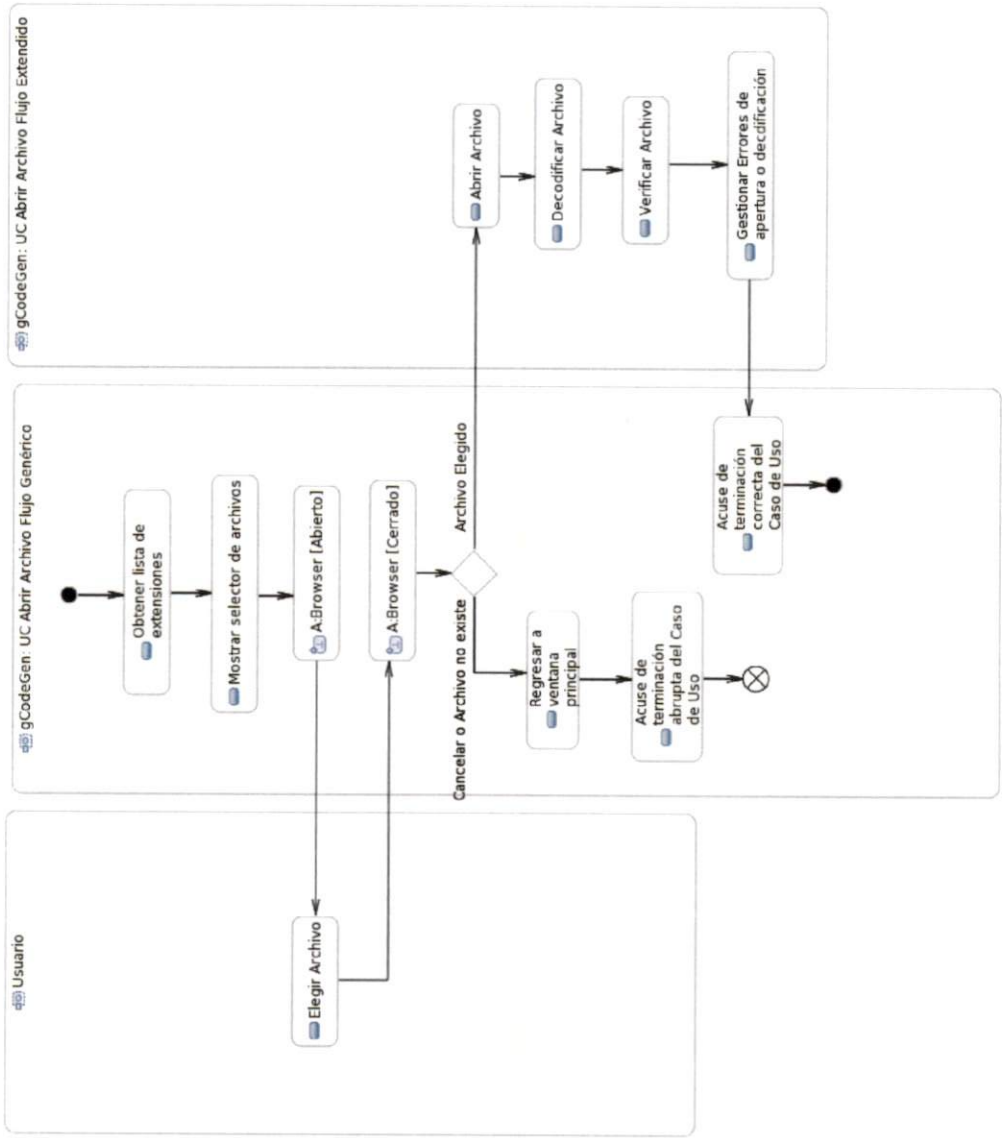


Figura 3.13 – *Muestra el modelo de secuencia de actividades requeridas para ejecutar el UC004 Decodificar Archivo*

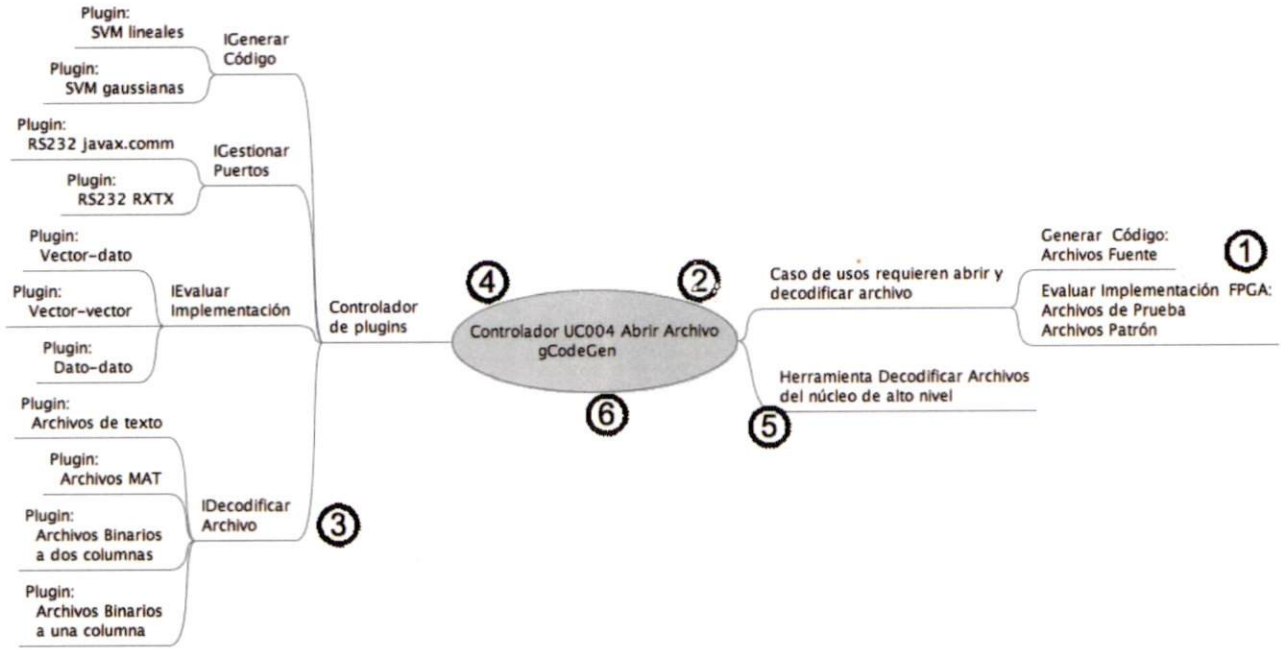


Figura 3.14 – *Muestra todos los elementos involucrados en la decodificación de archivos.*

f) Si hay un decodificador adecuado llama al controlador genérico del UC004 Decodificar Archivos y le envía como parámetro la i-ésima referencia del registro de plugins. En caso contrario se lanza un mensaje de error "No hay decodificador adecuado".

Plugin Decodificador de Archivos de texto en una sola columna. Estos archivos tienen una cabecera para indicar que tipo de SVM implementan; cuantos vectores de soporte; y de cuantas características tienen. El resto de la información se encuentra en una sola columna. Los datos se almacenan como tipo texto ASCII.

Este plugin lee la cabecera y los datos; y hace a conversión de cadena de caracteres a datos en doble precisión cuando es requerido.

Plugin Decodificador de Archivos de tipo MAT. El decodificador tipo MAT originalmente se implementó como un plugin, pero al ver su potencial uso se decidió migrarlo al núcleo de alto nivel del gCodeGen.

Finalmente, observese que el proceso de decodificación es automático y solo depende de que exista un plugin adecuado. Por otra parte, las cabeceras no requieren de un formato específico ya que el decodificar adecuado se encarga de interpretarlas.

3.3.2.5. Diseño e implementación del UC005 Gestionar Puertos de Comunicación

En la figura 3.15 se muestra el diagrama de actividades del UC005 Gestionar Puertos de Comunicación. A continuación se detalla:

a) *flujo genérico:* Llama inmediatamente al flujo extendido, cuya referencia es enviada por el controlador de este caso de uso. La referencia está en función del elemento seleccionado en el menú Puertos.

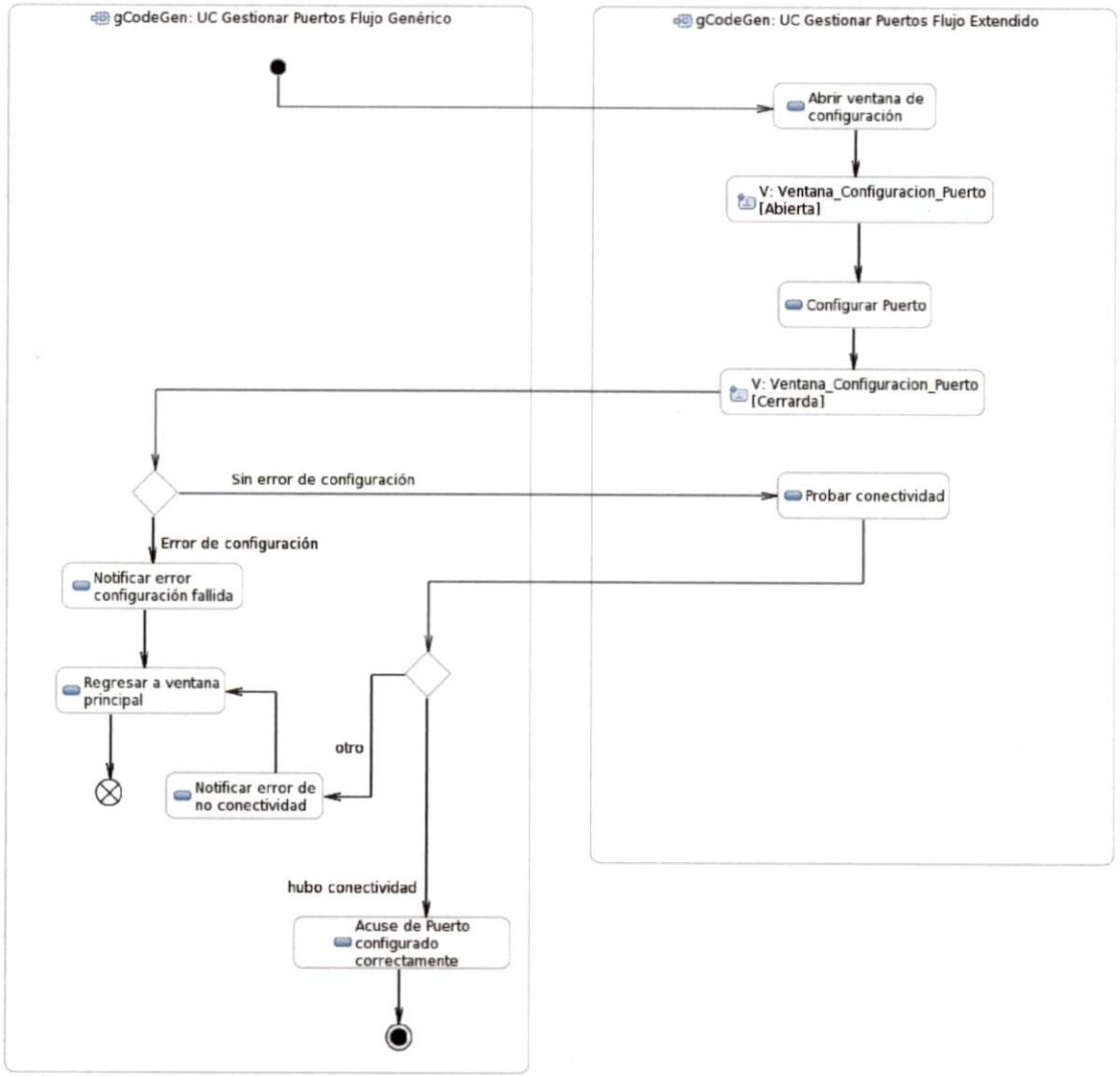


Figura 3.15 – *Muestra el modelo de diagrama de actividades correspondiente al UC005 Gestionar Puertos de Comunicación*

b) *flujo extendido:* Abre su ventana de configuración; elegidos los parámetros adecuados se inicia la configuración. Regresa flujos de datos uno de entrada y otro de salida (*InputStream* y *OutputStream*) que son en donde se escriben o leen los datos del puerto. Se regresa nulo si no se pudo configurar el puerto.

c) *flujo genérico:* Verifica que los flujos de datos no sean nulos. Si al menos uno lo es se lanza una ventana de error "El puerto no responde".

d) *flujo extendido:* Verifica la disponibilidad y conectividad del puerto configurado.

e) *flujo genérico:* Termina el caso de uso registrando los flujos de datos en el denominado registro de puerto, cuya referencia es enviada al UC003 Evaluar Implementaciones FPGA; esto sucede siempre y cuando no haya problemas de disponibilidad o conectividad. Si los hay se lanza una ventana de error "Puerto no disponible"

Todos los plugins de este punto de extensibilidad deben incluir un método para decodificar la información que va a transmitirse.

Plugin Gestor de puerto serial de protocolo RS232 que usa la biblioteca *java communication* [52]. Se usó la versión 2.0 de dicha biblioteca que es compatible con windows.

Dicha biblioteca provee de los controladores y de las clases Java que permiten establecer la comunicación serial. Se diseño una ventana para que el usuario seleccione los parámetros adecuados para la comunicación serial; seleccionados, el plugin usa los métodos de la biblioteca para crear la instancia del puerto serial y regresar una referencia los flujos de datos.

Plugin Gestor de puerto serial de protocolo RS232 que usa la biblioteca *RXTX* [29]. Se utilizó la misma ventana que la usada en el plugin anterior pero se usó la biblioteca libre RXTX para la configuración del puerto.

Ambos plugins el método para decodificar para la transmisión usan la biblioteca para el manejo de datos binarios del núcleo de alto nivel. El proceso consiste en transformar los datos decodificados el archivo de prueba cuyo tipo es doble, en su representación Qn.m y luego dividirla en bytes para enviarla por el puerto serial. En este caso se dividió en dos bytes.

3.3.2.6. Diseño e implementación del UC006 Transferir Datos

Su controlador únicamente ejecuta el protocolo de evaluación; recibe como parámetros un arreglo de datos tipo doble que contienen la información ya lista para transmitirse, una referencia al método que controla el protocolo de transmisión y las referencias a los flujos de información. En otras palabras es un método integrador que únicamente llama, usando la secuencia correcta, a los elementos necesarios para realizar la transmisión de datos; recordando que evaluar implementación es el único caso de uso que requiere de los puertos.

3.3.3. Descripción del plan de evaluación del gCodeGen

Además de la evaluación durante el proceso de desarrollo del gCodeGen y de los modelos SVM lineal y gaussiana, se realizó una evaluación global de la aplicación en dos partes; la primera consistió en lo siguiente:

1. Generar automáticamente el código Handel-C que describe la misma SVM lineal utilizada como plantilla. En otras palabras, el código generado debería de parecerse al de la plantilla. Para ello se realizó un comparación entre el código generado y el código de la plantilla usando la aplicación *Winmerge* que sirve para comparar texto [37].

2. Seguidamente se evaluó la implementación FPGA de ese código generado, usando el gCodeGen. La clasificación patrón de la plantilla SVM lineal se realizó con Octave.

3. Se realizó una comparación de los recursos ocupados en la implementación de la SVM lineal no optimizada con respecto a la implementación del código generado (que es el mismo que el de la plantilla).

La segunda parte de la evaluación consistió en lo siguiente:

1. Generar automáticamente el código para 3 distintas SVM lineales y luego implementarlas, una a la vez, en el FPGA.

2. Después se realizó la evaluación para cada una de las implementaciones cada una con su propios vectores de prueba y sus respectivas clasificaciones patrón obtenidas con Octave-Shogun.

3. Finalmente se realizó un análisis de los recursos requeridos para cada una de las implementaciones.

Capítulo 4

Resultados y Discusión

En este capitulo se muestran, analizan y discuten los resultados obtenidos del desarrollo de las implementaciones plantilla de las SVM lineales y gaussianas y del desarrollo de la aplicación gCodeGen.

4.1. Resultados generales

Se desarrolló una descripción Handel-C optimizada de una SVM lineal apta para aplicaciones BCI. Basados en esta descripción se desarrolló un algoritmo generador de código Handel-C que genera automáticamente la descripción de la SVM lineal especificada por el usuario.

Se desarrollaron los protocolos de evaluación de implementaciones FPGA de SVM lineales para conocer los errores de clasificación debidos a la perdida de precisión de la representación en punto fijo.

Para auxiliar a las tareas anteriormente mencionadas se desarrolló la aplicación gCodeGen; aunque se usaron algunos elementos del OpenUP para su desarrollo, no fue el propósito de esta tesis llevarlo a cabo ni formal, ni completamente. Esto se compensa con la arquitectura de plugins y el manejo de los mismos que permite el RCP Eclipse.

Entonces, las funciones del gCodeGen es generar automáticamente código Handel-C que describe SVM lineales y gaussianas a partir de un archivo que contiene los parámetros que las describen; la segunda función es evaluar automáticamente las implementaciones FPGA de SVM, es decir, compara la clasificación realizada en el FPGA con su homólogo en la PC, técnicamente, es una comparación de la clasificación usando representación Qn.m de 16 bits y una representación punto flotante de doble precisión.

Se desarrolló un algoritmo generador de código Handel-C que describe la SVM lineal requerida por el usuario. Los datos necesarios para generar el código se obtienen de un archivo fuente. El algoritmo se obtuvo a partir del análisis de la descripción SVM lineal original.

Parámetro	SVM Or	SVM Op	R ini
Cantidad de recursos (NAND)	860,000 (86 %)	69000 (6.9 %)	<20 %
Cantidad de recursos (FF)	1800 (18 %)	1200 (12 %)	<20 %
Tiempo de clasificación [ns]	3000	156.16	$< 3 \times 10^5$
Similitud con la implementación patrón	98 %	100 %	>80 %

Tabla 4.1 – *Muestra un resumen de los resultados obtenidos de la descripción Handel-C optimizada e implementación FPGA de la SVM lineal plantilla. Los porcentajes fueron calculados para la tarjeta de desarrollo RC200 con un Virtex II de un millón de compuertas. La nomenclatura es la siguiente: SVM Or es SVM original; SVM Op es SVM optimizada; y R ini son las restricciones iniciales.*

4.2. Plantillas SVM descritas en Handel-C e implementadas en FPGA

En la tabla 4.1 se muestra una lista de los resultados obtenidos para la implementación FPGA de la SVM lineal. Se muestra que cumple con todas las restricciones iniciales y mejora todos los aspectos de la SVM lineal original. A continuación se detalla la forma en que se obtuvieron los resultados presentados en dicha tabla.

Fueron necesarios 4 sistemas digitales para la descripción Handel-C de SVM: sistema digital receptor, sistema digital preprocesador, sistema digital clasificador y sistema digital transmisor. Dichos sistemas son generales para cualquier tipo de SVM y cualquier protocolo de transferencia de datos, inclusive es tan general que se puede considerar protocolos de envío y recepción distintos.

Son dos los canales de comunicación sincrónicos entre los sistemas digitales:

- receptor→preprocesador
- clasificador→transmisor

y pueden, fácilmente, utilizarse para soportar procesamiento con *pipeline*, debido al bloqueo característico del los canales sincrónicos se asegura el flujo del mismo; siempre y cuando la velocidad de llegada de los vectores a clasificar sea la adecuada. Los sistemas digitales preprocesamiento y clasificación pueden considerarse como uno solo, aunque en el modelo se contemplan como separadas figura 3.1.

4.2.1. Análisis de los recursos del FPGA ocupados en las implementaciones de SVM lineales

En la tabla 4.2 se muestran todos los bits de almacenamiento requeridos para la implementación FPGA de la SVM lineal optimizada (plantilla).

Dato	Denominación en Handel-C	Tamaño [bits]
1. vector $\overline{\mu}$	Mu[·]	$32\times nc$
2. sesgo b	B	32
3. vector de prueba **x**	X[·]	$32\times nc$
4. productos componente a componente	P[·]	$32\times nc$
5. primera capa de acumulación	SP0[·]	$33\times\frac{nc}{2}$
6. segunda capa de acumulación	SP1[·]	$34\times\frac{nc}{4}$
7. Producto punto	DP	35
8. Bias extendido	BE	35
9. Discriminante	Discr	36
10. Clase	c	1

Tabla 4.2 – *Muestra todos los registros ocupados para el almacenamiento en la implementación de la SVM lineal.*

4.2.1.1. Almacenamiento de los parámetros requeridos para la clasificación.

Los datos que se requieren para implementar la SVM lineal optimizada son: α_i, y_i, $\mathbf{x_i}$ y b.

Recordando que la implementación patrón (no optimizada) de donde partió este proyecto, almacena todos los anteriormente mencionados, a excepción de b que no se tomó en cuenta, con representación Q1.14: se ocupan $16nvs(nc+1)$ bits para almacenarlos. En la implementación optimizada para el mismo propósito se ocuparon únicamente $32(nc+1)$ bits, debido al cálculo fuera del FPGA del vector $\overline{\mu}$. Para $nc > 8$ la implementación optimizada ahorra, aproximadamente, $\frac{nvs}{2}$ bits. De lo anterior se deduce que el ahorro de bits de almacenamiento es proporcional al número de vectores de soporte nvs. Por otra parte los parámetros $\overline{\mu}$ y b se habrían almacenado en la memoria RAM interna al FPGA, de no ser que ésta es de acceso secuencial, y habrían sido necesarios $nc+1$ ciclos de lectura para leer las componentes de $\overline{\mu}$ y el bias b.

Los parámetros almacenados que son requeridos para la clasificación son los elementos 1, 2 de la tabla 4.2. Adicionalmente debe haber una arreglo de $32\times nc$ bits para almacenar al vector de prueba (elemento número 3 de la misma).

4.2.1.2. Almacenamiento de los cálculos intermedios que son necesarios para obtener la clasificación.

Los cálculos intermedios que fueron necesarios para obtener la clase del vector **x** son los elementos del 4 al 10 en la tabla 4.2

Analizando dichos datos, se puede generalizar para cualquier nc que sea potencia de 2 cual es la cantidad de bytes CB requeridos para los cálculos intermedios según la ecuación 4.1

$$CB = 32nc + \sum_{i=0}^{k-1}(33+i)\frac{nc}{2^{i+1}} + 3k + 101 \tag{4.1}$$

donde $k = log_2(nc)$ es decir el número de capas, aunque se comienzan a enumerar desde cero.

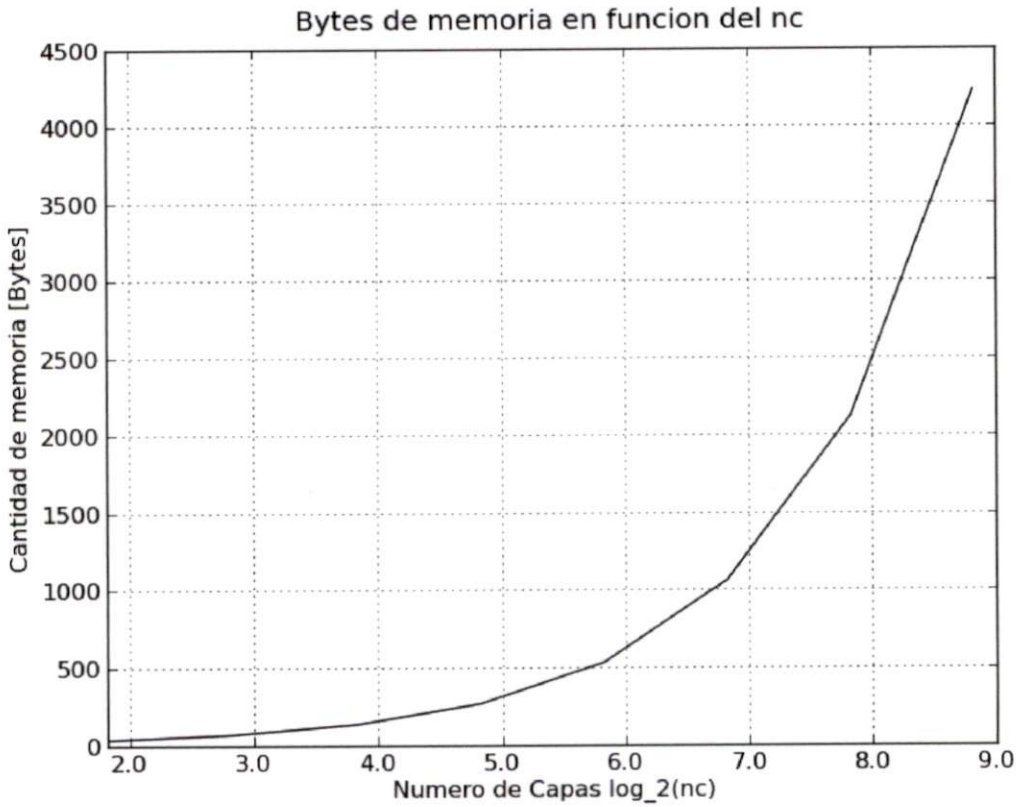


Figura 4.1 – *Muestra la cantidad de bytes requeridos para almacenar los cálculos intermedios en función del número de características nc.*

Graficando la ecuación 4.1 para distintos valores de k se obtiene la figura 4.1 que muestra la cantidad de bytes requeridos para almacenar los cálculos intermedios en función del número de características nc de los vectores. En el eje x se muestran los valores de k y en el eje y su correspondiente cantidad de bytes. Observese que para $k = 9$ se tienen que almacenar aproximadamente 4300 bytes que corresponden a vectores de 512 características. Para el caso de la detección de P300 los valores de k estarán entre 3 y 7, es decir, se ocuparán aproximadamente 1000 bytes como máximo para almacenar los cálculos intermedios.

De la misma figura 4.1 se observa un crecimiento exponencial, la mayor contribución a la cantidad de bytes de almacenamiento se debe a las capas de sumas paralelas, por ello se propone que para $k > 9$ se reutilicen registros o se construya una ALU especializada en esos cálculos.

Para el caso en que nc no sea potencia de 2 los recursos CB se calculan con la ecuación 4.2

$$CB = 32nc + 33 + 4k + 100 \tag{4.2}$$

pero en este caso k es el número de iteraciones necesarias para obtener el acumulado de los productos paralelos. Observese que este crecimiento es lineal comparación con la acumulación cuando nc sí es potencia de 2.

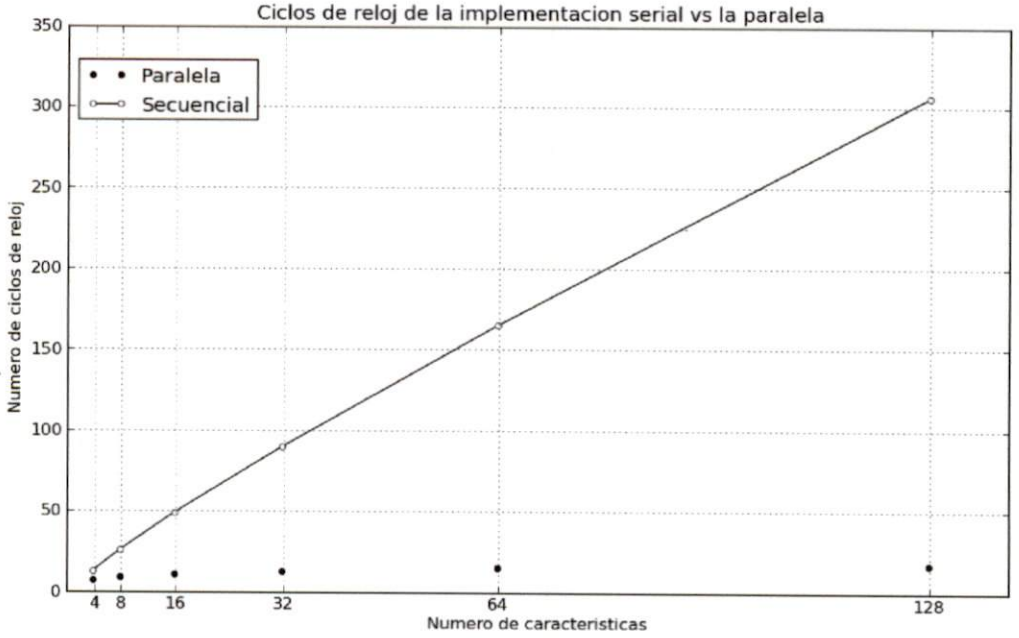


Figura 4.2 – *Comparación entre el número de ciclos de reloj requeridos para que una implementación secuencial clasifique un vector y los requeridos por una implementación paralelizada.*

4.2.2. Aceleración de la clasificación debida a la paralelización

Para saber que tanto se acelera la clasificación debido al paralelismo se hizo el siguiente análisis: a partir de la figura 3.4 se dedujo que el número de operaciones requeridas para clasificar esta dado por la ecuación 4.3 asumiendo que nc es una potencia de 2.

$$\begin{aligned} num\_op &= 2^k + \sum_{i=0}^{k-1} 2^i + 2 \\ &= 2^{k+1} + 1 \end{aligned} \tag{4.3}$$

Por otra parte, una implementación secuencial Handel-C del clasificador requiere de num_op ciclos de reloj para realizar las num_op operaciones necesarias para llevar a cabo la clasificación, esto se debe que se requiere de un ciclo de reloj para realizar cada instrucción en Handel-C.

La implementación paralela que se propuso y que sigue el *datapath* de la figura 3.4 utiliza únicamente $k+3$ ciclos de reloj para concluir una clasificación para el caso donde nc es potencia de 2. En la figura 4.2 se muestra que el número de ciclos de reloj de implementación secuencial crece exponencialmente con respecto al crecimiento lineal de la implementación paralelizada.

En el caso en que nc no es potencia de 2 la implementación paralelizada ocupa $nc+3$ ciclos de reloj para completar la clasificación, esto sigue siendo lineal con respecto a los ciclos ocupados por la implementación secuencial, es decir, la paralelización reduce de exponencial a lineal la rapidez con la que se realizan las operaciones.

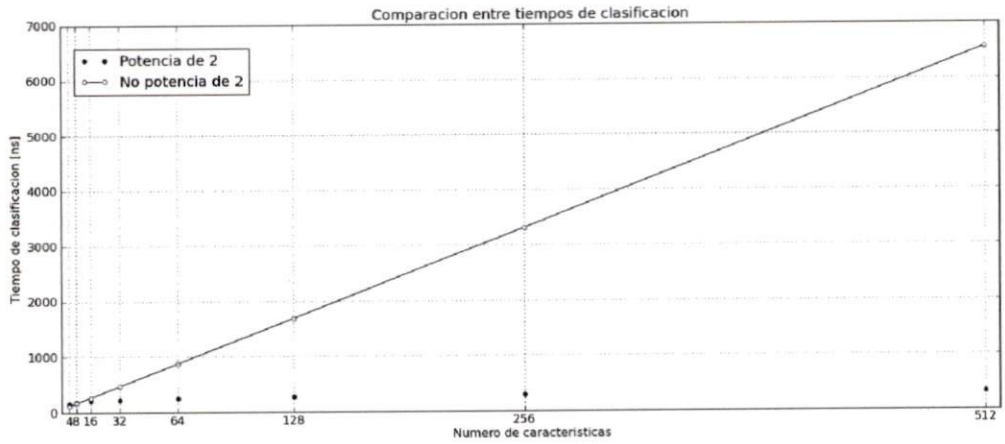


Figura 4.3 – *Gráficas de los comportamientos de los tiempos de clasificación en función del número de características.*

Por los detalles que se analizan en la siguiente subsección la aceleración se ve afectada por que al desglosar las operaciones del clasificador en operaciones sencillas el número de ciclos de reloj aumenta, pero se sigue conservando la aceleración de exponencial a lineal, ya que el aumento del número de ciclos de reloj es constante para cada implementación.

4.2.3. Medición del tiempo de clasificación de las implementaciones FPGA de las SVM lineales

El tiempo de clasificación también depende si nc es o no potencia de 2. El análisis es el siguiente.

4.2.3.1. Tiempo de clasificación para nc potencia de 2.

Para el caso particular de $nc = 8$ se requieren de 6 ciclos para completar la clasificación de acuerdo con el *datapath* de la figura 3.4; estos ya en la implementación se extendieron a 12 ciclos con el fin de hacer las instrucciones Handel-C más sencillas. Por otra parte el retraso máximo obtenido con el *place and route* fue de 12.68 ns. De lo anterior se deduce que el tiempo máximo de clasificación es de 156.16 ns. Es decir, se ocupa un 5.2×10^{-5} % del tiempo fijado como límite (300 ms) y un 5.2×10^{-3} % de que requiere la clasificación original (3.0 ms).

Generalizando para cualquier $nc > 1$ potencia de 2: se requieren $2(4 + log_2(nc))$ ciclos, por lo tanto, asumiendo que el tiempo de retraso se conserva; el tiempo de clasificación es $25.36(4+log_2(nc))$ ns. Obsérvese que el tiempo de clasificación crece logarítmicamente con respecto al crecimiento de nc.

Valdría la pena realizar una prueba para medir directamente el tiempo de clasificación, esta no se realizó ya que por cuestiones presupuestales y administrativas no fue posible renovar la licencia de Handel-C. La metodología propuesta es la siguiente: describir un contador ascendente de 24 bits que funcione concurrentemente durante la clasificación; tiene que iniciar su cuenta una vez recibida la última componente del vector a clasificar y debe detenerla justo antes de construir la palabra que contiene la clase y que va a ser transmitida. El tiempo se calcula multiplicando el entero decimal resultado de la cuenta por el inverso de la frecuencia de reloj. Se consideró que

una resolución de 24 bits para una frecuencia de reloj de 100 MHz es suficiente para medir los tiempos de clasificación ya que se pueden medir hasta aproximadamente 100 ms.

4.2.3.2. Tiempo de clasificación para nc que no son potencia de 2.

En el caso particular $nc = 10$ se requieren de 15 ciclos y por un argumento similar al caso anterior se extendieron a 22 ciclos. Asumiendo el mismo tiempo de retraso; el tiempo de clasificación es de 278.96 ns

Generalizando para cualquier $nc > 2$ que no es potencia de 2: se requieren $5 + nc$ ciclos para realizar una clasificación, es decir, el tiempo de clasificación es $(12.685 + nc)$ ns. Es decir, se ocupa un 9.2×10^{-5} % del tiempo fijado como límite y un 9.2×10^{-3} % de que requiere la clasificación original.

En la figura 4.3 se muestra una comparación entre los tiempos de clasificación dependiendo de la forma de nc. Se observa que para $nc < 10$ da igual que implementación se utilice ambas clasifican aproximadamente en el mismo tiempo. Para $nc > 10$ el tiempo de clasificación es mucho más pequeño para implementaciones nc potencia de dos. En la práctica la gráfica para nc potencia de 2 esta es válida para los valores discretos de nc adecuados.

Únicamente como ejemplo: para $nc = 1024$ con la implementación nc potencia de 2 se requieren de 300 ns mientras que en el otro caso se requieren 13 μ s; a pesar de la diferencia de más de un orden de mil, ambas implementaciones clasifican dentro de la restricción y en menor tiempo que la implementación original.

Por lo anterior, no necesariamente se requiere del *pipeline* para esta aplicación y por ello no se implementó. En aplicaciones como el reconocimiento de patrones en imágenes o en general aplicaciones que requieran de un tiempo de clasificación más pequeño el *pipeline* puede ser una opción.

4.2.4. Análisis de la representación Qn.m

Dado que $\overline{\mu}$ acumula toda la información de las α_i, y_i y $\mathbf{x_i}$ y por ello la representación Qn.m debe ser adecuada para representar los datos.

Se constató empíricamente que la identidad n+m=15 representa adecuadamente los datos de las SVM lineales implementadas; esto según los resultados que se presentan en la siguiente sección. Aunque no se descarta como trabajo futuro un análisis de los parámetros antes mencionados para realizar estimaciones sobre las magnitudes suprema e ínfima y así definir la mejor representación Qn.m.

El peor caso es cuando el supremo se representa con Q15.0 y el ínfimo con Q0.15, lo cual, en términos prácticos es descartar una de las componentes del vector $\overline{\mu}$. Esta situación se agravaría si al menos otra componente del vector debería ser del mismo orden de magnitud del ínfimo.

Para que la representación sea adecuada se requiere de lo siguiente: sabiendo que la representación del supremo es Qn.m, el ínfimo debe ser mayor o igual a 2^{-m}. De esta forma se asegura que no se pierde información ya que en la descripción SVM la precisión siempre aumenta y nunca se trunca.

4.2.5. Proyección para implementaciones SVM de más de 8 características.

Del análisis de recursos se estima que el crecimiento de de los mismos es exponencial dependiendo del número de características, en la figura 4.4 se muestra el comportamiento de la cantidad de recursos del FPGA en

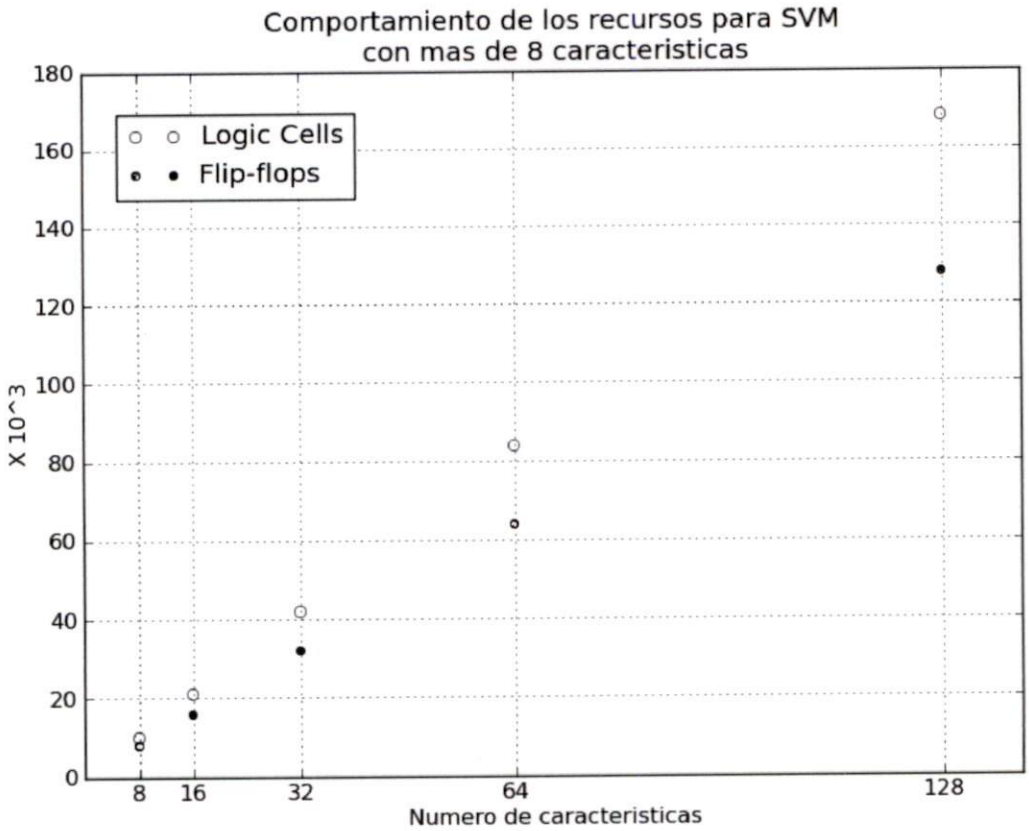


Figura 4.4 – *Comportamiento del número de Slices y de flip-flops requeridos de acuerdo al número de características.*

nc	# Logic Cells	#Flip-flops	Familia FPGA	Modelo FPGA
16	20000	9000	Spartan 3	XC3S1500
64	82000	62000	Spartan 6	XC6SL150
128	197000	191000	Virtex 6	XC6VHX565T

Tabla 4.3 – *Proyección para la implementación FPGA de SVM con más de 8 características.*

función del número de características. De acuerdo a dicha figura se proyecta que dado el aumento del *nc* las implementaciones SVM podrían realizarse de acuerdo a la tabla 4.3 . Las especificaciones de los FPGA fueron obtenidas de [9–11]

Se observa que se pueden implementar SVM de hasta 128 características usando el generador de código. Para utilizar una menor cantidad de recursos se deberá utilizar otra plantilla. En el trabajo futuro queda reutilizar los sumadores de las capas paralelas, de esta forma se reducirá el gasto de recursos, así como el uso de módulos MAC. Se destaca que para este análisis se supuso que las celdas lógicas *Logic Cells* de las familias Spartan 6 y Virtex 6 contienen lo mismo que sus homónimas en el Spartan 3, lo cual no es cierto ya que en ambos casos contienen más elementos. Se reitera que no se pudieron realizar estas implementaciones ya que no se cuenta con una licencia de Handel-C.

4.2.6. Análisis de la descripción Handel-C de la plantilla SVM gaussiana

Se realizó la implementación de CORDIC en modo hiperbólico para calcular e^x para $x < 0$. La descripción se realizó con 18 sistemas digitales cada uno representando una iteración del CORDIC, es decir, se implementó la versión "desenrollada" (*unrolled*) del mismo.

Debido a que en el *kernel* gaussiano sí requiere realizar *nvs* iteraciones para clasificar un vector se usó *pipeline*.

Durante la etapa de pruebas se constató que el intervalo en el que el CORDIC funciona adecuadamente con un error menor al 10 % con respecto a la función exponencial de Octave es entre -0.1 y 1; este intervalo no es de utilidad para las SVM gaussianas a pesar de una precisión Q4.27. Se verificó formalmente usando *model checking* [36] para saber si era problema de la implementación, lo cual no resultó cierto. En 2008 se publicó un articulo que dice que el intervalo para calcular la función exponencial con CORDIC es entre 0 y 1 [41]. Esto aunado al tiempo para concluir el proyecto evitó que se concluyera satisfactoriamente la implementación de SVM gaussianas.

Omitiendo la evaluación de la exponencial, la clasificación de SVM gaussianas se parece mucho a como se realiza con SVM lineales. Es decir, el problema a resolver es como evaluar la función e^{-x}. Dado que en este caso x corresponde a la norma de una diferencia de vectores su magnitud es siempre positiva por lo que el argumento de la exponencial siempre es negativo. Para implementar dicha función en FPGA sería necesario evaluar $\frac{1}{e^{-x}}$ esto tiene como desventaja implementar una división en el FPGA. Actualmente se está trabajando con aproximación de la exponencial con líneas rectas, el problema es encontrar el número óptimo de líneas para: perder poca precisión de la evaluación de la exponencial; utilizar pocos recursos del FPGA; y evaluar rápidamente.

4.2.7. Evaluación de la implementación SVM lineal optimizada

Implementada la SVM optimizada se le enviaron 100 vectores de prueba para que los clasificase y regresara el resultado para ser comparado. Esto se realizó con el gCodeGen con un plugin de evaluar implementación FPGA. De esos vectores, 50 pertenecen a una clase y el resto a la otra; la clasificación patrón se obtuvo con Octave-Shogun. El conjunto de entrenamiento fueron vectores de 8 características generados aleatoriamente que se mezclan poco. La tasa clasificación con Octave-Shogun para los vectores de prueba fue del 100 %. La comparación entre implementaciones resultó de un 100 % de similitud, es decir, que a pesar de la representación punto fijo del FPGA las clasificaciones son completamente equivalentes.

Una afirmación fuerte es que esta descripción SVM lineal se puede ocupar para todas las SVM de 8 características únicamente calculando $\overline{\mu}$; como se mencionó anteriormente. Esto sólo es posible si todas las componentes de dicho vector y el sesgo pueden representarse adecuadamente con 16 bits Qn.m. En caso contrario se requerirá de otra plantilla SVM lineal y otro algoritmo generador de código que ajuste la precisión.

Se realizó la misma evaluación para la implementación SVM lineal de 10 características (recuerdese que es la misma SVM de 8 características con dos adicionales en cero), misma que también tuvo el 100 % de similitud; magnitudes similares tanto en tiempo de clasificación y recursos utilizados a la SVM de 8 características.

Finalmente, se realizó una evaluación y análisis de resultados detallado para cada sistema digital de la implementación FPGA de la SVM lineal plantilla, los cuales se pueden consultar en el Apéndice A.

Por otra parte, no se evalúo la SVM gaussiana ya que la implementación no funcionó adecuadamente.

4.3. Diseño e Implementación de la Aplicación gCodeGen

Se identificaron y definieron los casos de uso de la aplicación los cuales son:

1. Generar Código
2. Guardar Archivos
3. Evaluar Implementaciones FPGA
4. Decodificar Archivos
5. Gestionar Puertos de Comunicación
6. Transferir Datos

Por otra parte, a continuación se muestra la gestión de los 5 principales riesgos para el desarrollo del gCodeGen. Se consideró la probabilidad de que el riesgo ocurra; que tan grave es la afectación de su ocurrencia clasificada alta y media cada una dividida a su vez en alta y media; la prioridad de implementación siendo 1 lo primero que se implementó y 4 lo último; finalmente se propuso un plan de contingencia.

Riesgo 1. Comunicación RS232 no soportada

- *Probabilidad:* 0.50-0.85
- *Gravedad:* media alta
- *Prioridad:* 2
- *Plan de contingencia:*
  - Tratar de implementar otros protocolos compatibles con java.
  - Almacenar los vectores de prueba en el FPGA y mostrar los resultados de la clasificación en los display de la tarjeta.

Riesgo 2. Algoritmo generador de código requiere más parámetros de los inicialmente planteados.

- *Probabilidad:* 0.60-0.90
- *Gravedad:* media baja
- *Prioridad:* 4
- *Plan de contingencia:*
  - Analizar las implementaciones SVM en software para encontrar el modelo correspondiente en hardware.
  - Incluir los parámetros adicionales encontrados en el análisis.

Riesgo 3. RCP de eclipse incompatible con los protocolos de comunicación.

- *Probabilidad:* 0.40-0.70
- *Gravedad:* media baja

- *Prioridad:* 3
- *Plan de contingencia:*
  - Revisar el ligado de las librerías del sistema operativo con el RCP.
  - Implementar un *wrapper* para Java del protocolo de comunicación.

Riesgo 4. Errores con la biblioteca para decodificar archivos MAT

- *Probabilidad:* 0.65-0.85
- *Gravedad:* media alta
- *Prioridad:* 1
- *Plan de contingencia:*
  - Buscar otras bibliotecas para decodificar archivos MAT
  - Guardar los datos con formatos más sencillos como texto o csv.

Riesgo 5. Manipular directamente la creación de instancias de los plugins (necesario para crear menús)

- *Probabilidad:* 0.5-0.70
- *Gravedad:* alta
- *Prioridad:* 1
- *Plan de contingencia:*
  - Utilizar versiones modificadas de los plugins del núcleo del RCP.
  - Crear los plugins desde XML.

La arquitectura de implementación de la aplicación se muestra en la figura 3.8. Se realizaron las pruebas de escritorio sobre la arquitectura y se diseñaron y especificaron las clases e interfaces sobre dicha arquitectura, aunque no se construyeron los diagramas de clases.

En cuanto a los plugins, se diseñaron las interfaces para cada punto de extensibilidad, estas son las siguientes:

1. IGenerar Código
2. IGestionar Puertos
3. IEvaluar Implementación
4. IDecodificar Archivo

Todos los plugins desarrollados para gCodeGen se muestran en la figura 4.5. Los plugins marcados son los que se incluyeron en la versión final de la aplicación.

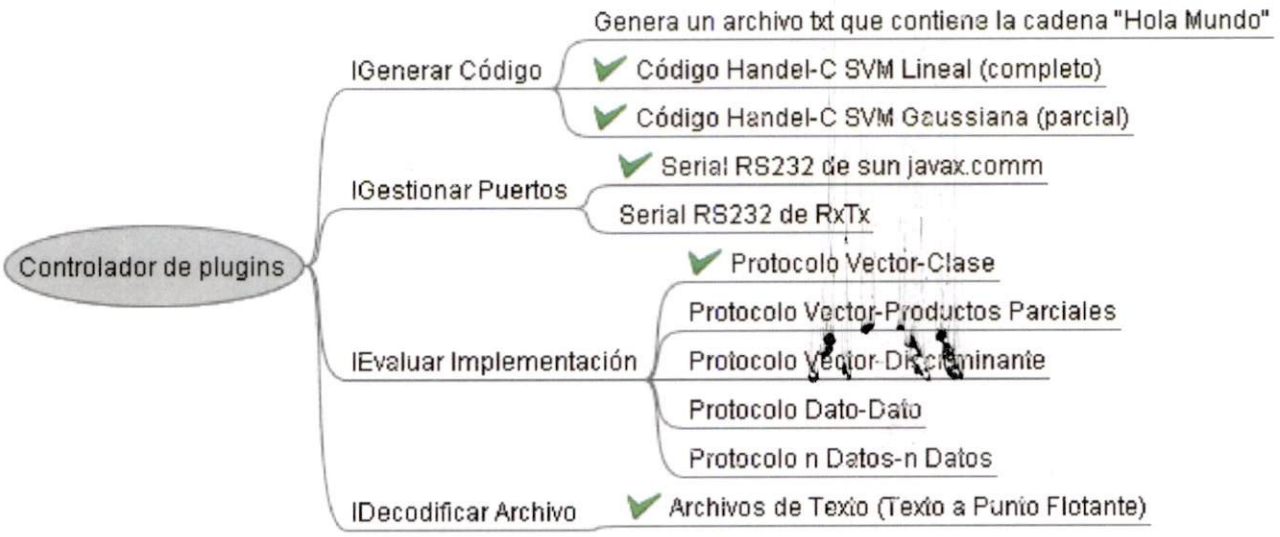


Figura 4.5 – *Muestra los 4 puntos de extensibilidad y todos los plugins que las extienden. Los plugins marcados fueron los que se incluyeron en la versión liberada del gCodeGen. Los que no están marcados sirvieron para la evaluación de la SVM plantilla.*

Figura 4.6 – *Muestra la pantalla de inicio (splash) del gCodeGen ejecutandose en Windows XP.*

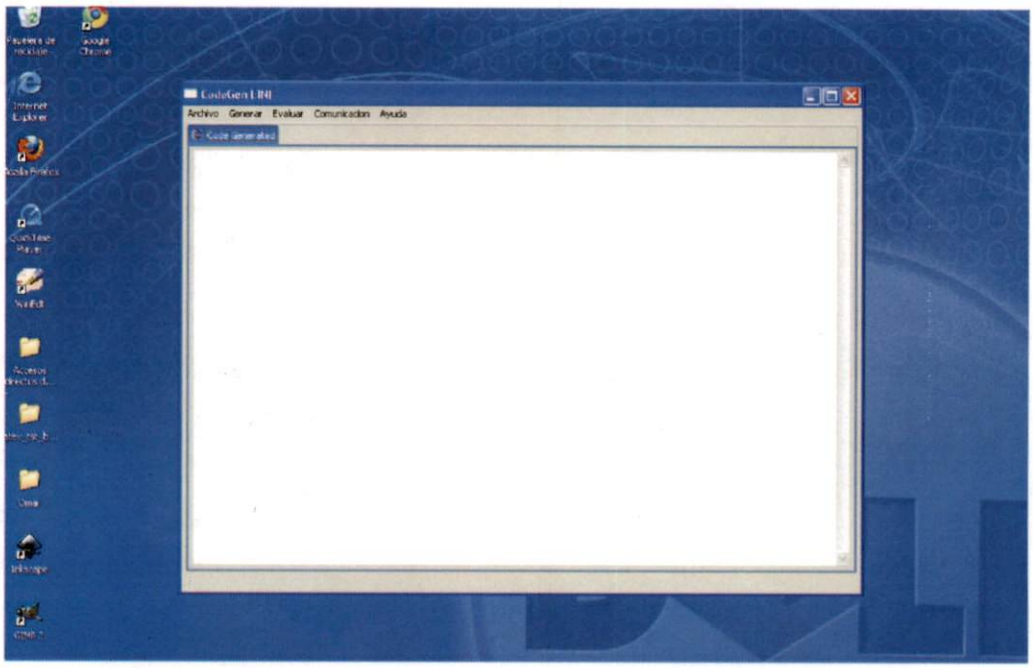


Figura 4.7 – *Muestra la pantalla principal del gCodeGen. Se ve el área para el código generado; y los menús.*

4.3.1. Evaluación del funcionamiento de la aplicación gCodeGen

4.3.1.1. Aplicación stand-alone

La aplicación gCodeGen en versión *stand-alone* se ejecutó en *Windows XP* con *service pack 2*. En la figura 4.6 se muestra la pantalla de inicio de dicha aplicación. Contiene: el logotipo del laboratorio donde se desarrolló la aplicación; el nombre de la aplicación; quien la desarrolló; y la forma de contactarlo.

Por otra parte, mientras se despliega la pantalla de inicio se ejecuta la secuencia de inicio del gCodeGen, que es donde se registran los plugins y se construyen los menús.

Terminado lo anterior se muestra la ventana principal de la aplicación (figura 4.7) donde se ve: el área para el código generado; y los menús.

4.3.1.2. Actualización de los menús dinámicos al agregar o quitar plugins.

Para mostrar como se actualizan los menús dinámicos se muestra en la figura 4.8 el estado inicial del menú generar, observese que hay dos plugins instalados uno para generar código de SVM lineales y otro para SVM gaussianas.

Se creó un plugin llamado *prueba menú tesis* para el punto de extensibilidad IGenerar Código se agregó al registro de plugins del RCP y se volvió a lanzar el gCodeGen (ver creación de plugins Apéndice B), en la implementación de la interfaz se le indicó que crearía el submenú tesis y el nombre del elemento que incluyó en el mismo fue Acción Prueba tesis . En la figura 4.9 se muestra como se actualizó automáticamente menú.

Figura 4.8 – *Muestra el estado inicial del menú Generar del gCodeGen.*

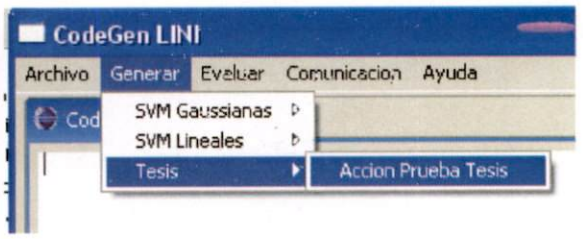


Figura 4.9 – *Muestra la actualización automática del menú dinámico Generar.*

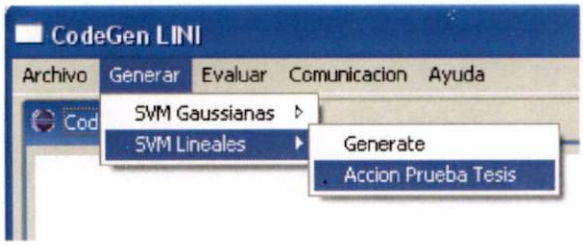


Figura 4.10 – *Muestra el menú Generar después de agregar un elemento a un submenú ya creado por otro plugin.*

Figura 4.11 – *Muestra el menú Generar después de remover dos plugins asociados a su punto de extensibilidad.*

También se realizó la prueba cuando el plugin *prueba menú tesis* agrega un elemento a un submenú creado por un plugin previo figura 4.10 y cuando se eliminan todos los plugins excepto uno figura 4.11.

De esta manera se demostró que el menú Generar se actualiza automáticamente al agregar o quitar plugins. Los gestores de los menús dinámicos Evaluar y Puertos generalizan la misma clase abstracta que el gestor del menú Generar; la única diferencia es que lo hacen utilizando sus correspondientes registros de plugins. Por lo anterior se infiere, y de hecho se probó, que ambos menús también se actualizan adecuadamente; aunque esos resultados no se presentan en esta tesis.

Por otra parte, el tiempo de desarrollo de plugins no se midió ya que depende de muchos factores y también del punto de extensibilidad, algunos ejemplos de estos factores son: si ya ha sido desarrollado el algoritmo de generación de código o apenas se va a desarrollar; si existen bibliotecas y los controladores para Java del protocolo de comunicación que se quiere implementar; entre otros. De lo anterior se pudo constatar que cuando se crean plugins el núcleo de alto nivel se encarga de contectarlos con el resto del gCodeGen, es decir,que el desarrollador de plugins ya no necesita preocuparse por la interacción de sus plugins con el sistema.

Como se muestra mas a detalle en el Apéndice B cuando se requiere crear un nuevo plugin se utilizan los *wizards* del RPC Eclipse y se obtiene una plantilla que al ser llenada correctamente por el desarrollador y luego de ser registrada el plugin queda instalado en el gCodeGen.

4.3.1.3. Generación automática de código Handel-C que describe SVM lineales

Se seleccionó la opción generar código del menú Generar→SVM lineales→ generate figura 4.11; seguidamente aparece la pantalla del navegador de archivos figura 4.12 para seleccionar el archivo fuente. Una vez que este es seleccionado se realiza la generación de código, mismo que se escribe en la ventana principal figura 4.13.

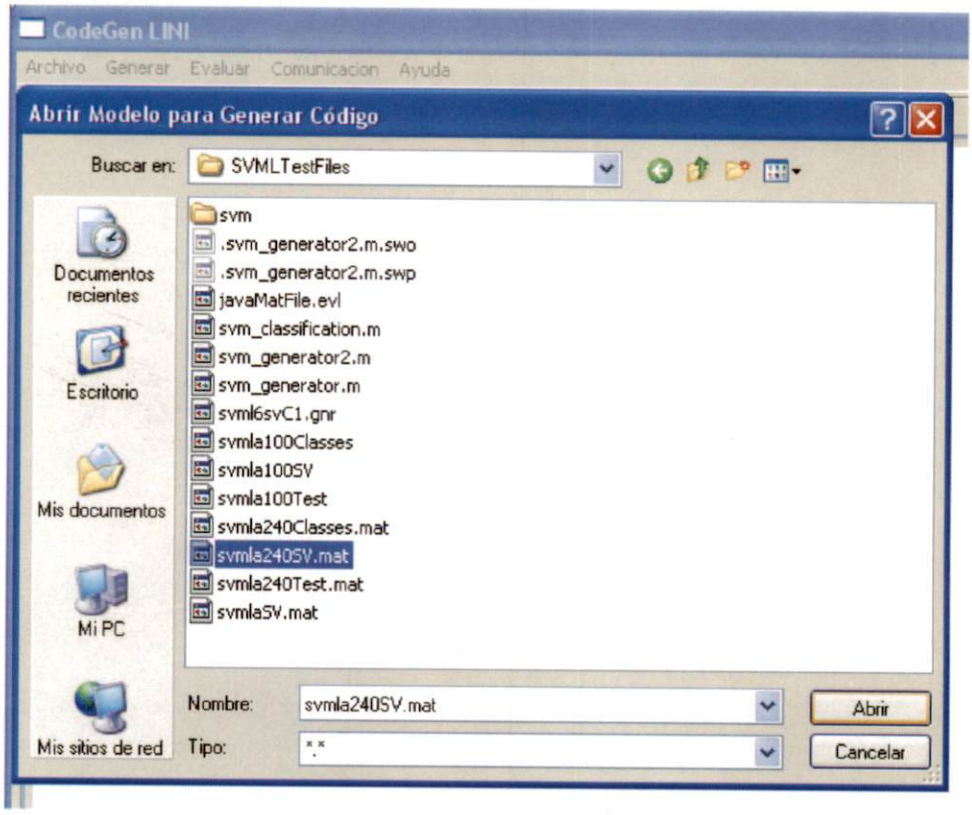


Figura 4.12 – *Navegador de archivos para que el usuario seleccione el archivo fuente para generar SVM lineales.*

Aquí es donde se realizó la primera parte inciso 1 de la evaluación del gCodeGen descrita en el capitulo anterior. Y resultó que el código generado y su respectiva plantilla difieren en un 1 % lo que se debe únicamente a espacios en blanco y saltos de línea. En otras palabras, tanto el código generado como la plantilla describen la misma SVM.

4.3.1.4. Protocolos de comunicación.

Se realizaron pruebas piloto de envío de un conjunto de datos que debía ser reenviado íntegramente a la PC. El resultado de todas ellas fue satisfactorio (Apéndice A).

De las bibliotecas que implementan el protocolo RS232 *javax.comm* de Sun Microsystem y GNU LGPL *RXTX* de Keane Jarvi, únicamente el primero se consideró para la versión final debido a que no se encontró, aunque no se descarta que sí exista, una forma de hacerlo compatible con la biblioteca Handel-C para el manejo del puerto serial RS232 y por ende tampoco el controlador de la tarjeta RC10 de dicho protocolo.

A pesar que el protocolo RS232 ha caído en desuso, las descripciones FPGA de SVM lineales no serán afectadas significativamente ya que en la aplicación final el FPGA no va a comunicarse con la PC porque en él se implementará la BCI completa, y tampoco se requiere de un protocolo demasiado rápido para el envío de los vectores a clasificar debido al protocolo de estimulación de la P300. En el peor caso tendrá que comunicarse con otros dispositivos digitales y esto se haría utilizando protocolos como I^2C o SPI. Lo anteriormente dicho, justifica que

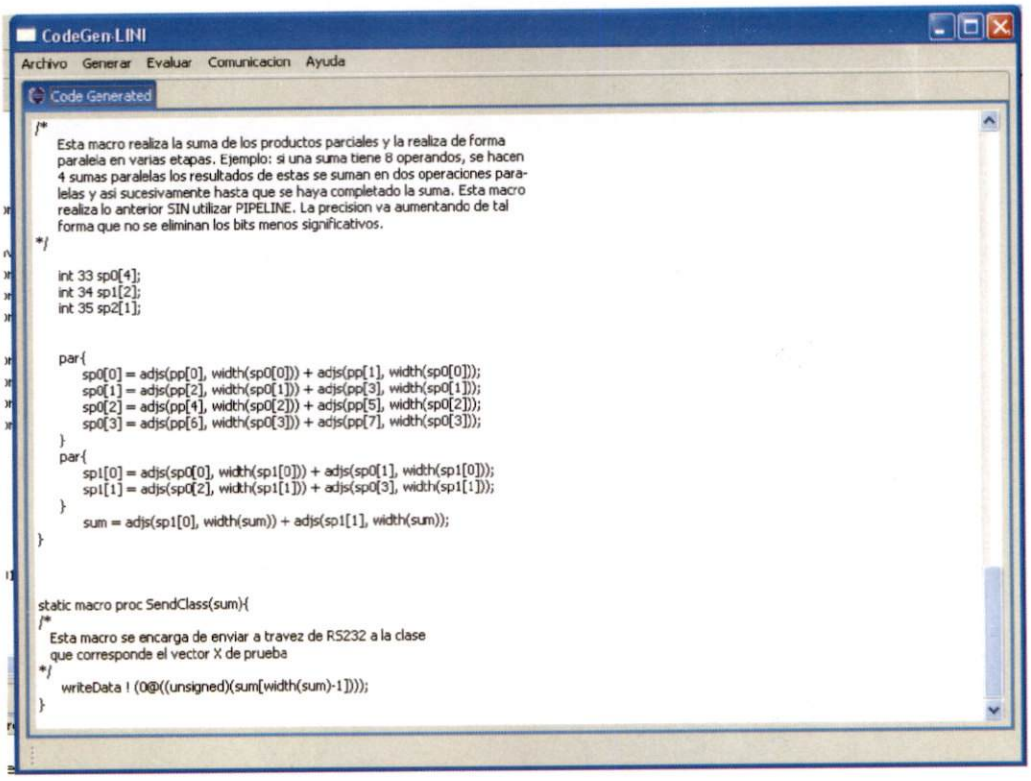


Figura 4.13 – *Código Handel-C generado que describe SVM lineales.*

se haya utilizado el protocolo RS232, además una implementación Java para controlar el protocolo USB debido a que hacerla hubiera implicado mayor complejidad y mayor tiempo de desarrollo; y se habrían subutilizado sus bondades.

Finalmente, la forma en que se diseñó y construyó el controlador del caso de uso Gestionar Puertos de Comunicación desacopla completamente la forma en que se implementa el controlador del protocolo de comunicación de las escrituras y lecturas al mismo que realiza gCodeGen.

4.3.1.5. Evaluación de implementaciones FPGA de SVM

La evaluación de implementaciones se inicia desencadenando las acciones del menú Evaluar figura 4.15. En este caso se evaluaron implementaciones FPGA de SVM lineales.

Si el puerto serial no ha sido configurado, se lanza una ventana de error figura 4.15, para configurarlo se accede al menú Puerto y se selecciona el puerto a configurar; en este caso solo estaba implementado el puerto serial RS232 con la biblioteca *javax.comm*. Cuando se selecciona dicho elemento del menú aparece se lanza la ventana de configuración que se muestra en la figura 4.16

Después, ya se puede evaluar implementaciones, para ello se lanzan dos navegadores de archivos, uno para seleccionar el archivo de prueba y otro para el archivo patrón. Una vez seleccionados se lanza la ventana de selección de protocolo de evaluación figura 4.17; seleccionado el protocolo se ejecuta la evaluación y se muestra el resultado de la misma una vez concluida figura 4.18.

Figura 4.14 – *Muestra el elemento del menú evaluar que inicia la secuencia de evaluación de implementaciones FPGA de SVM lineales.*

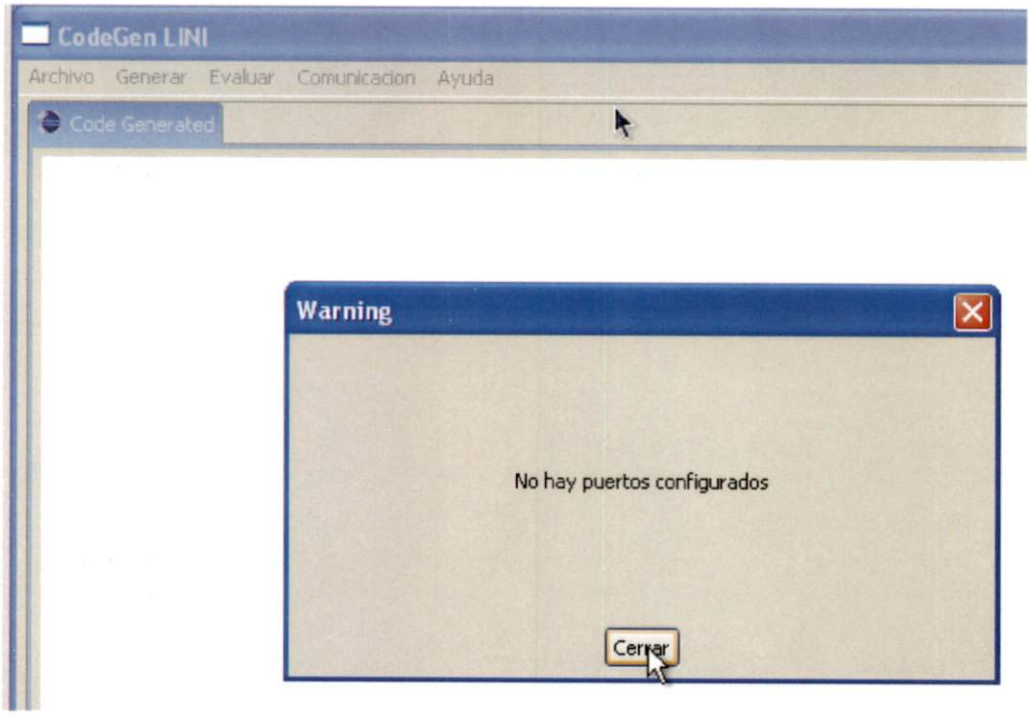


Figura 4.15 – *Error generado por no haber configurado un puerto antes de iniciar la evaluación.*

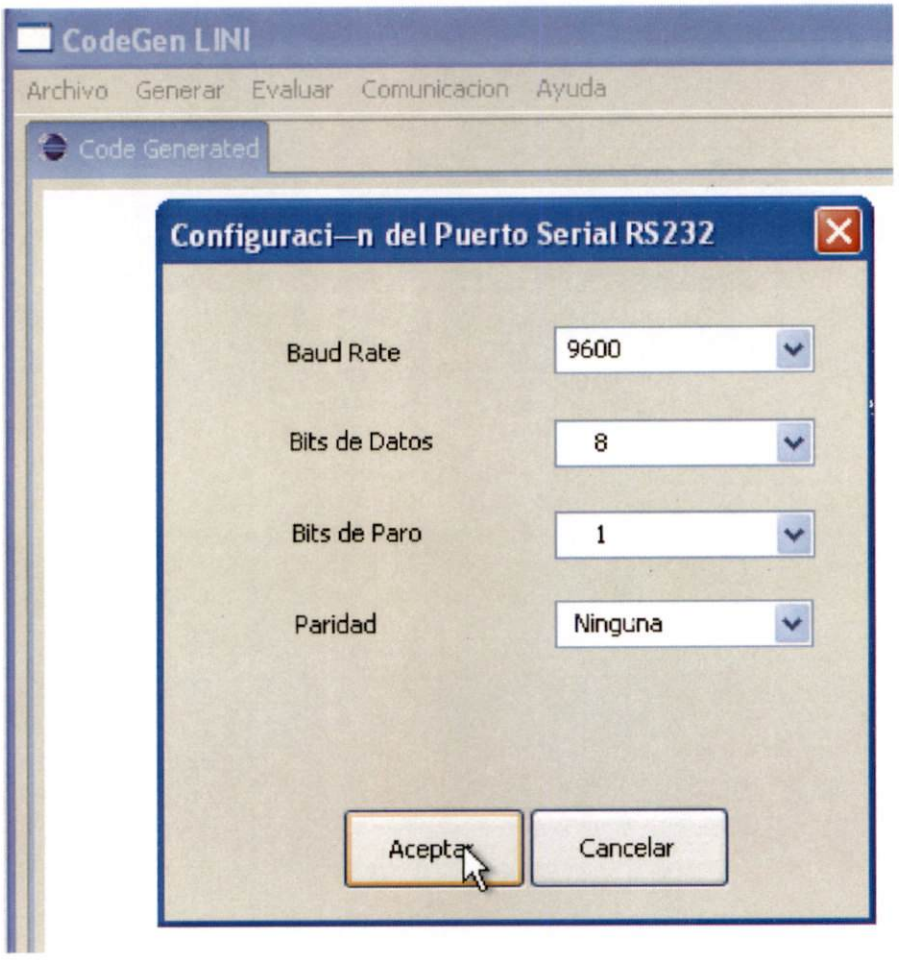


Figura 4.16 – *Ventana de configuración de los parámetros del protocolo RS232; los que aparecen son los valores por defecto.*

El proceso anterior fue el que se realizó para evaluar las implementaciones FPGA de las SVM lineales de código generado, cuyos resultados se describirán posteriormente.

4.3.1.6. Decodificadores de archivo

De las pruebas anteriores se infiere que la cadena de responsabilidad modificada que se implementó para seleccionar automáticamente el decodificador adecuado funciona adecuadamente. Ya que la generación de código y la evaluación requieren de archivos diferentes (al menos 3) lo que implica que se requirió elegir entre diferentes decodificadores.

En el registro de plugins del RCP Eclipse del gCodeGen se cambió el orden de los plugins del punto de extensibilidad IDecodificar Archivos para asegurar que el orden no influye en la cadena de responsabilidad modificada. Lo cual fue verificado.

Por otra parte, cuando se intentó abrir un archivo del cuál no había un decodificador adecuado se lanza la

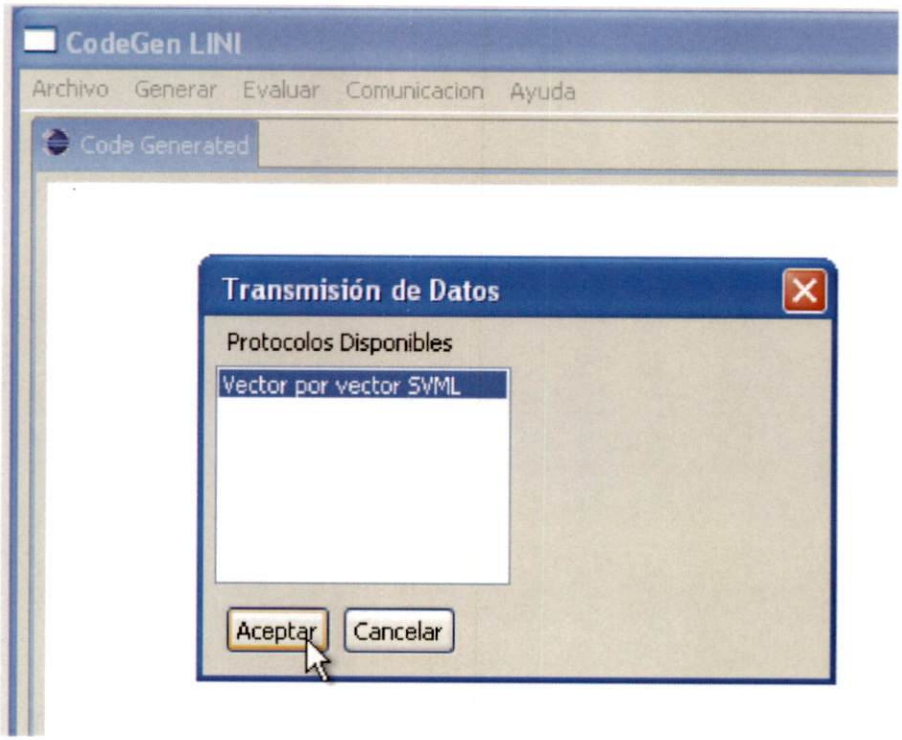


Figura 4.17 – *Ventana de selección del protocolo de evaluación de la implementación FPGA.*

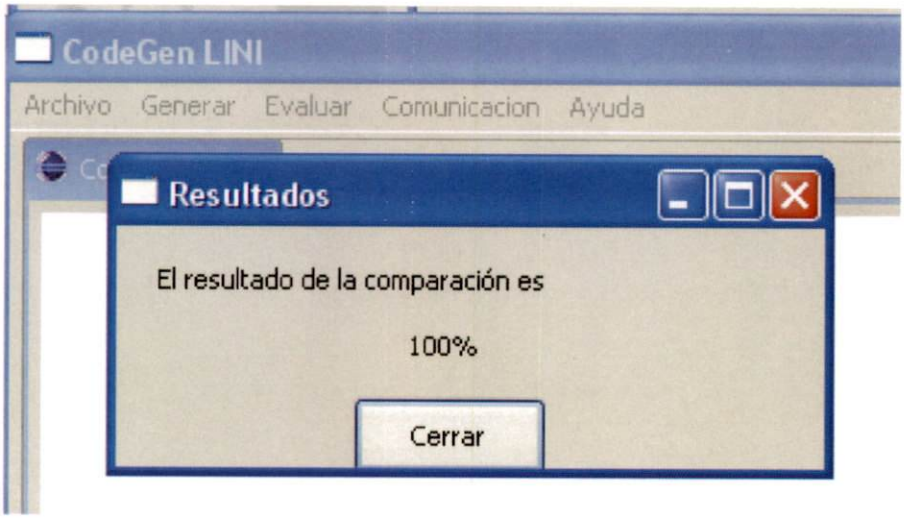


Figura 4.18 – *Ventana donde se muestra el resultado de la evaluación de las implementaciones FPGA.*

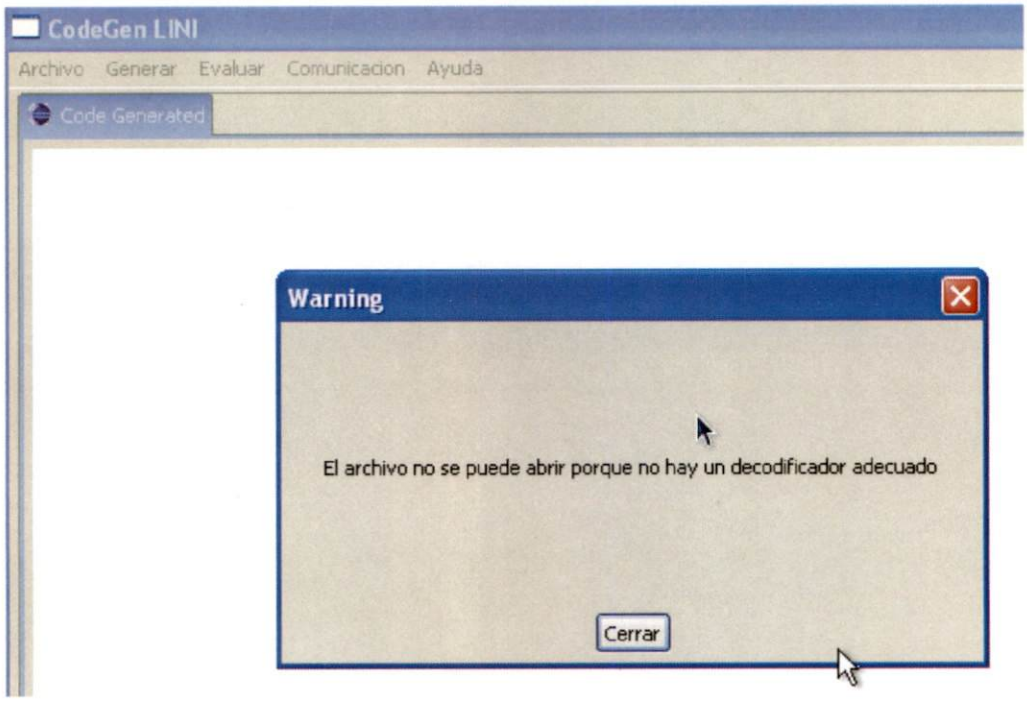


Figura 4.19 – *Ventana de error cuando la cadena de responsabilidad modificada no encuentra un decodificador adecuado para el archivo seleccionado por el usuario.*

ventana de error que se muestra en la figura 4.19.

Se hizo una última prueba del selector automático de decodificadores y consistió en una evaluación de implementaciones FPGA en la que los datos de prueba estuvieron almacenados en un archivo MAT y las clases patrón en un archivo de texto[1]. La evaluación se llevó a cabo de manera normal. Se desarrollo el plugin decodificador de archivos de texto para esta prueba. A pesar de que se usaron dos tipos de archivos diferentes para la evaluación la cadena de responsabilidad modificada encontró el decodificador adecuado y este se encargó de proveer los datos en el formato correcto para la evaluación y esto se realizó sin modificar nada del gCodeGen excepto agregar el plugin para decodificar archivos de texto. Esto demuestra el desacoplamiento de los plugins, siempre y cuando estos implementen adecuadamente los métodos de sus interfaces correspondientes.

De las pruebas realizadas se descubrió que la biblioteca JMatIO en la que se basan los decodificadores de archivos MAT de las utilerias del núcleo de alto nivel; lanza una excepción que interrumpe de forma abrupta tanto el flujo genérico como el extendido del caso de uso que requiera la decodificación. Este error únicamente tiene tres causas que son:

a) La matriz requerida por el decodificador no se encuentre en el archivo.

b) Que el archivo este dañado.

c) La herramienta (Octave o Matlab) no haya almacenado correctamente la información debido al uso incorrecto de los comandos de almacenamiento.

[1]El formato del archivo de texto es una columna de caracteres ASCII que representan número en punto flotante. La primera línea de este archivo contiene la cabecera.

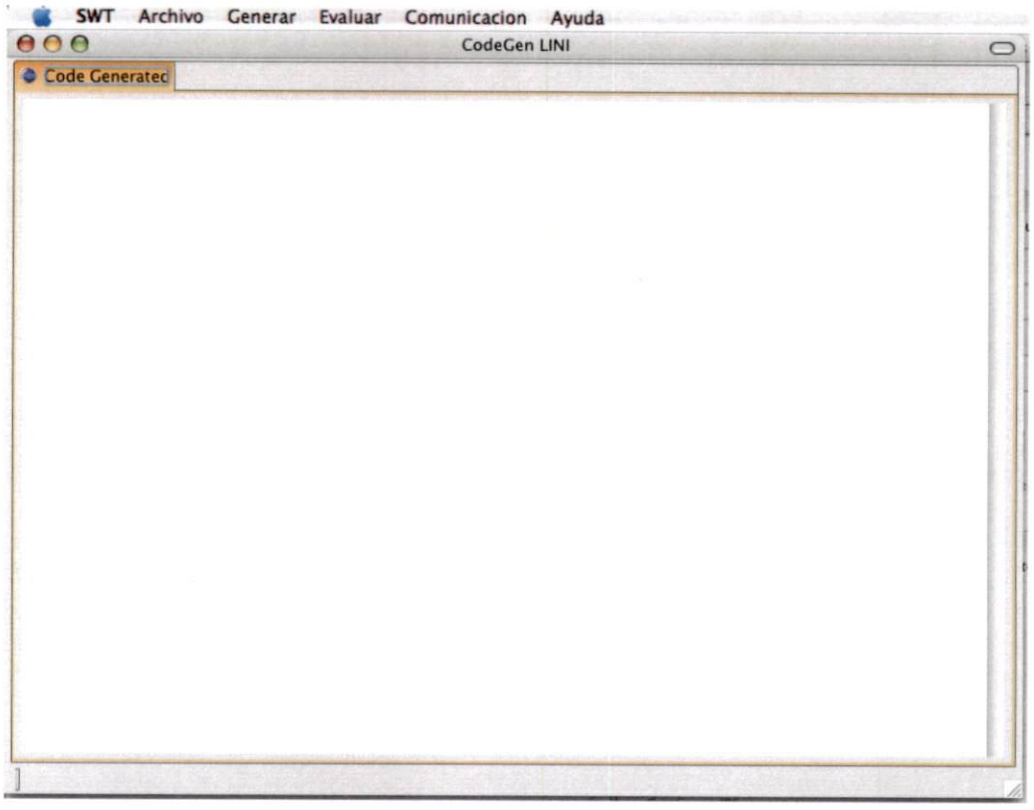


Figura 4.20 – *Ventana principal del gCodeGen para Mac OS 10.4 Tiger.*

Para evitar que la decodificación de archivos MAT interrumpa la selección automática del decodificador adecuado se desarrolló un *script* en Octave-Shogun que realiza el entrenamiento de las SVM y genera los archivos MAT fuente, prueba y evaluación compatibles con el gCodeGen. El *script* requiere que se le indiquen cual es el conjunto de entrenamiento y el de prueba, además del tipo de *kernel* de la SVM.

Como trabajo futuro se pretende realizar modificaciones a los decodificadores basado en JMatIO desarrollados en este proyecto para que capturen las excepciones y las maneje de mejor manera.

4.3.1.7. Portabilidad del gCodeGen

Se realizaron las mismas pruebas descritas anteriormente para la versión del gCodeGen para Mac OS 10.4 *Tiger*. Se enfatiza que el código de esta versión y su contraparte para Windows XP es el mismo, únicamente se recompiló para el sistema operativo.

En forma resumida los resultados fueron los mismos que los obtenidos en la versión de Windows XP, a continuación se muestran una serie de figuras de la versión para Mac OS 10.4 del gCodeGen:

- Ventana principal; figura 4.20.
- Código generado para las mismas SVM lineales que la versión Windows XP; figura 4.21.

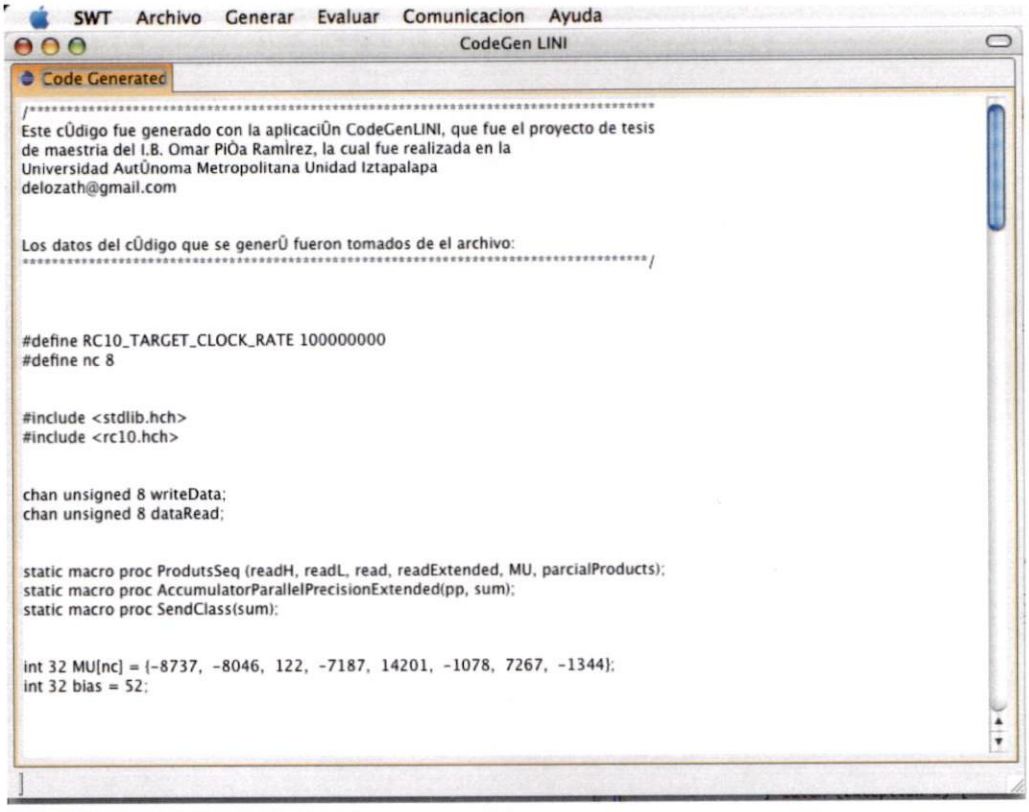


Figura 4.21 – *Código Handel-C generado que describe SVM lineales.*

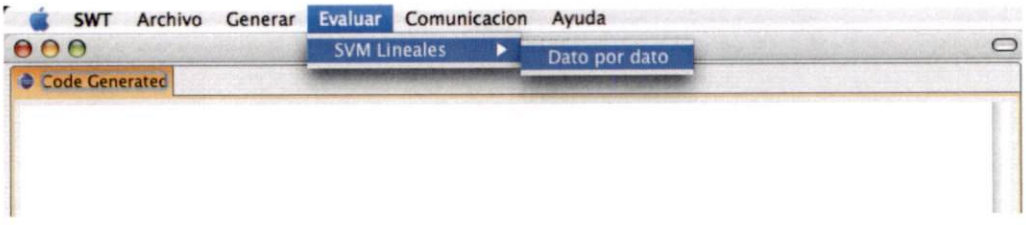


Figura 4.22 – *Muestra los elementos del menú Evaluar y la selección de uno de sus elementos. Observese que los menús son al estilo Mac OS.*

Figura 4.23 – *Ventana de error que aparece cuando se intenta evaluar una implementación sin haber configurado un puerto.*

Figura 4.24 – *Ventana de selección de los parámetros del puerto serial RS232.*

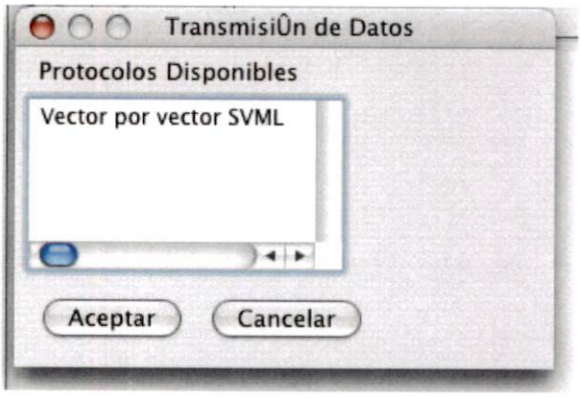


Figura 4.25 – *Ventana de selección del protocolo de evaluación de implementaciones FPGA.*

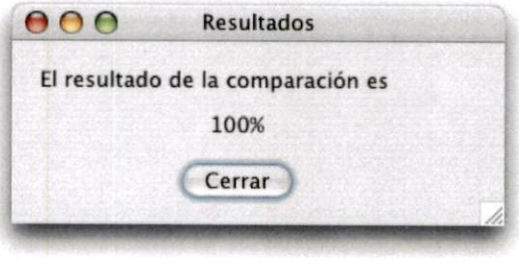


Figura 4.26 – *Ventana que muestra el resultado de la evaluación de una implementación FPGA.*

- Inicia la evaluación de implementaciones FPGA de SVM lineales, notese que los menús son al estilo Mac OS; figura 4.22.
- Error puerto no configurado; figura 4.23
- Ventana de configuración de puertos; figura 4.24. Ya que ni la biblioteca *javax.comm* ni RXTX implementan RS232 para Mac OS, y aprovechando las interfaces de java, se emuló al puerto serial con escrituras y lecturas a un archivo, usando un flujo de entrada y uno de salida del mismo. Esto involucró la creación de un plugin para el punto de extensibilidad IGestionar Puertos.
- Ventana de selección del protocolo de evaluación de implementaciones FPGA; figura 4.25
- Ventana que muestra los resultados de la evaluación de las implementaciones FGPA de SVM lineales figura 4.26.

No se pudo comprobar la portabilidad a Linux debido a que las bibliotecas SWT de Eclipse no eran totalmente compatibles con las usadas en las versiones de Windows y Mac. No se descarta que haya una forma de hacerlos compatibles aunque no se exploró más.

4.3.2. Evaluación de implementaciones FPGA de SVM lineales cuyo código fue generado

Antes de llevar a cabo las evaluaciones de las implementaciones del código generado, se realizaron pruebas piloto para verificar que se llevaran a cabo correctamente los flujos: genérico y extendido del casos de uso Evaluar Implementaciones FPGA. En todas estas pruebas se obtuvo un resultado satisfactorio (Apéndice A).

Pruebas: Primera parte inciso 2 y 3 del plan de evaluación. Comparación entre la clasificación en Octave de la plantilla SVM lineal y su contraparte implementada en FPGA cuyo código fue generado automáticamente. La similitud entre ambas implementaciones fue del 100 % a pesar de la pérdida de precisión debida a la representación Qn.m. Esto se esperaba, pues la plantilla y la plantilla generada solo difieren en un 1 %.

También se realizó una comparación entre la implementación de la plantilla SVM lineal de código generado con la SVM lineal original de donde partió este proyecto, es decir, que los resultados de la plantilla generada son los mismos que los mostrados en la tabla 4.1 obtenidos con el mismo conjunto de prueba.

Rehaciendo el análisis para la cantidad de recursos del Spartan 3 del RC10, de acuerdo a lo que se muestra en la tabla 4.1 se observa que a pesar de que el *nc* de la plantilla es mayor que el de la SVM lineal original, la plantilla redujo 92 % el número de compuertas y el 33 % la cantidad de flip-flops.

Se realizó una evaluación similar para la plantilla SVM lineal generada de 10 características con respecto a la original; y los resultados fueron muy similares a los presentados en la tabla 4.1.

Pruebas: Segunda parte del plan de evaluación. Se les llamó SVM1, SVM2 y SVM3 a las implementaciones FPGA de las SVM lineales cuyo código fue generado. En la tabla 4.4 se muestra un resumen de los resultados obtenidos al evaluar las implementaciones de las SVM antes mencionadas.

Los datos de prueba son mitad de una clase y mitad de la otra para todos los casos. SVM1 y SVM2 fueron generadas a partir de datos sintéticos mientras que SVM3 corresponde a una SVM ocupada para detectar P300.

	SVM1	SVM2	SVM3
nvs	4	12	82
nc	8	8	4
# Datos Prueba	100	3000	100
Tasa Coincidencia	100 %	100 %	100 %
# Compuertas	69000	69000	62000
# Flip-Flops	1200	1200	1200

Tabla 4.4 – *Resumen de los resultados de las implementaciones FPGA de las SVM lineales de código generado por el gCodeGen.*

	SVM1	SVM original
nvs	4	76
nc	8	6
# Datos Prueba	240	100
coincidencia	100 %	100 %
# Compuertas	69000	860000
# Flip-Flops	1200	1800

Tabla 4.5 – *Comparación entre la plantilla SVM lineal y la SVM original.*

La la tasa clasificación de la SVM patrón fue del %100 para todas las SVM. La SVM1 no es la misma que la SVM plantilla, aunque sus parámetros son similares.

De los resultados se puede decir, que las implementaciones SVM de código generado son equivalentes a las implementadas en Octave. Y de acuerdo al análisis del tiempo de clasificación, ninguna de las implementaciones clasifica un vector en más de 200 ns lo cual es lo que se buscaba.

En promedio se ocupó el 4.4 % de las compuertas del FPGA Spartan 3 y ocupó cerca del 13 % de los flip-flops de dicho dispositivo. Es decir, para el caso de compuertas se ocupó únicamente el 22 % de los recursos inicialmente planteados como óptimos y en el caso de los flip-flops se ocupó un 65 % de los límites totales. En otras palabras, se ocuparon muchos menos del 20 % de los recursos totales del FPGA planteados inicialmente como límite.

Pruebas: primera parte inciso 3 del plan de evaluación En la tabla 4.5 se muestra una comparación entre los datos de implementación entre la SVM original y la SVM1, esta última es el código generado correspondiente a la SVM plantilla. De la misma tabla se observa que el proceso de optimización resultó efectivo ya que solo en la optimización se ahorró el 92 % de las compuertas y el 33 % de los flip-flops

De los resultados anteriores obsérvese que el cálculo del vector $\overline{\mu}$ ahorra muchos recursos a las implementaciones y, de hecho, no importa el *nvs* ya que su cálculo siempre se mapea a dicho vector. Esto solo es válido para SVM lineales. Lo anterior presenta un problema cuando los elementos del vector $\overline{\mu}$ no pueden ser representados adecuadamente con 16 bits lo cual se vuelve una limitación del algoritmo generador de código, aunque esto puede resolverse con un plugin cuyo algoritmo de generación de código extienda la precisión.

Conclusiones

Se desarrolló una descripción Handel-C de una SVM lineal que optimiza los recursos de y es apta para aplicaciones BCI. A partir de esta descripción se desarrolló un algoritmo generador de código Handel-C de la SVM lineal requerida por el usuario.

También se desarrollaron los protocolos de evaluación para verificar que las implementaciones FPGA de las SVM fueran similares a sus contrapartes implementadas con Octave.

Se desarrolló una aplicación para facilitar al usuario la generación de código y la evaluación de implementaciones FPGA de SVM.

Por otra parte, la implementación del código generado utiliza los recursos adecuados para dejar recursos en el FPGA para que se implemente el resto de la BCI. Además, clasifica en un tiempo mucho menor a 3 ms por lo que puede aplicarse a otras áreas.

El gCodeGen también fue diseñado para evaluar implementaciones FPGA con respecto a una implementación patrón. De hecho, si se agregan los plugins adecuados se podrían evaluar implementaciones que no fueran en FPGA.

Por otra parte la extensibilidad tardía permite agregar muchas variantes en cada caso de uso. Por ejemplo, el generador de código con una correcta decodificación permitiría generar cualquier tipo de código, inclusive, siendo no numéricos los datos leídos del archivo fuente. La extensibilidad tardía también permite que la evaluación automática pueda extenderse a muchas áreas cuyo paradigma sea: leer datos de prueba, enviarlos a algún dispositivo, y comparar sus resultados con un patrón.

En cuanto al generador de código se comprobó que por medio de transformaciones a una plantilla de código Handel-C que describe SVM lineales se puede generar el código de otra SVM lineal requerida por el usuario. Las transformaciones están regidas por el preprocesamiento que se le hace a los datos contenidos en el archivo fuente que contiene los parámetros que definen a la SVM requerida.

En todas la pruebas realizadas a los códigos generados, la clasificación de sus implementaciones FPGA resultó equivalente a la realizada en software de cálculo numérico a pesar de la pérdida de precisión debida a la representación Qn.m de 16 bits. Esto se atribuye a dos factores: el primero es la precisión extendida, es decir, que cada cálculo necesario para la clasificación extiende adecuadamente el signo para contener los posibles acarreos; el segundo factor es el cálculo del vector $\overline{\mu}$ lo cual condensa la información en un vector que es calculado usando doble precisión en las operaciones. Si se hubiesen almacenado los vectores de soporte en el FPGA como en la implementación original, en cada una de las nvs iteraciones requeridas para clasificar un vector se hubiera propagando el error de discretización de la representación Qn.m tantas veces como el número de iteraciones.

La principal contribución del diseño de la aplicación gCodeGen desde el punto de vista de desarrollo, es la capa intermedia denominada *núcleo de alto nivel* que es la que relaciona el *runtime kernel*, los plugins básicos de Eclipse con los plugins agregados para que la aplicación se convirtiera en el gCodeGen. El desarrollo de esta capa enriqueció mucho en conocimientos ya que se tuvo que aprender: manejo de cargadores de clases en Java, administración de componentes dinámicos, administración y diseño de bases de datos, funcionamiento avanzado de Eclipse IDE, manejo de menús de Eclipse IDE, JNI (*Java Native Interfaces*), por mencionar algunos.

Con respecto al lenguaje de descripción de hardware Handel-C se considera que las descripciones son fáciles de hacer si no se considera ningún método de optimización de recursos. En caso contrario el diseñador del sistema digital requiere de amplios conocimientos sobre el diseño de circuitos digitales para que los métodos de optimización sean transparentes. Además de lo anterior el costo de la licencia de Handel-C renovable anualmente no es muy atractivo. Por lo anterior se contemplan lenguajes como VHDL o verilog para futuros desarrollos o incluso alternativas libres de código abierto y además de alto nivel como *MyHDL* que es un lenguaje de descripción de hardware basado en Python.

A pesar de no haber usado ni formal ni completamente el OpenUP resultó de mucha utilidad para el desarrollo del gCodeGen ya que se pudo tener el modelo de la arquitectura y de esta forma planear exactamente el plan de desarrollo del gCodeGen. Se constató que es necesaria experiencia en su uso para tener la visión y la proyección requerida para ejecutarlo de manera adecuada. Se considera como resultado la experiencia adquirida en el desarrollo de este proyecto para el desarrollo futuro de aplicaciones de software, se adquirió una visión general del proceso de desarrollo de software de calidad así como de sus ventajas y limitaciones. De hecho ya se aplicó a un par de aplicaciones desarrolladas para la industria y para un proyecto en conjunto con el Instituto Nacional de Cardiología Ignacio Chávez para el procesamiento de imágenes de ventriculografía.

Actualmente se han contemplado varias mejoras al gCodeGen antes de ser liberado como una aplicación *open source* entre las que se encuentran: hacer tres selectores automáticos de archivos uno para cada tipo de archivo; registrar los menús en una base de datos para que no se tengan que crear instancias de los plugins cada vez que se inicia el gCodeGen la actualización automática se haría en base a un *checksum* sobre el registro de los plugins y los registrados en la base de datos, si hubiese diferencias entonces se generan los menús, en caso contrario se dejarían los que estuviesen en la base de datos; crear en el núcleo de alto nivel una biblioteca para incluir rutinas en Python lo cual facilitaría el desarrollo de los decodificadores y generadores de código.

En cuanto al ahorro de recursos, se tiene la hipótesis que hay otros factores, además de los mencionados en los resultados, que influyeron en la reducción de los recursos en la plantilla, aunque no fueron corroborados formalmente. Estas hipótesis se enuncian a continuación:

- La disminución de la complejidad del hardware debida a que no se controla explícitamente el número de datos que se reciben ya que son canales los que controlan el inicio de la clasificación una vez que se han recibido todos los componentes del vector a clasificar.
- Disminución de la lógica que controla el almacenamiento y lectura de los datos. Debido a que ya no se requieren almacenar ni leer los parámetros α_i, y_i, $\mathbf{x_i}$ y b ya que se resume en la lectura del vector $\overline{\mu}$
- Evitar el uso de las comparaciones en las estructuras cíclicas.

Las BCI ocupan, además de la P300, otros paradigmas de estimulación muchos de los cuales utilizan SVM como clasificadores. Para estos casos el gCodeGen puede implementar las SVM especificas para cada paradigma sin

realizarle ninguna modificación. Pero en general puede generarse la descripción Handel-C de cualquier SVM lineal, cuyo vector $\overline{\mu}$ pueda ser representado adecuadamente con 16 bits Qn.m.

Quedó pendiente la implementación FPGA de las SVM gaussianas; actualmente se está trabajando en la implementación FPGA de una exponencial decreciente usando segmentos de lineas. La aportación es que se está desarrollando con el paradigma *co-design*, en la parte de software se está desarrollando un algoritmo que analice los datos de las SVM gaussianas para estimar el intervalo en que la exponencial es válida, además del valor mínimo y máximo que serán evaluados; y una vez conocidos utilizar la mejor representación punto fijo. Con esto se pretende utilizar los recursos justos del FPGA.

La arquitectura de plugins del gCodeGen permitirá agregar los algoritmos generadores de código de las descripciones del las partes restantes de la BCI, también permitirá agregar sus correspondientes protocolos de evaluación. Una vez probadas independientemente todas las componentes de la BCI se podrá realizar la descripción total de la misma y con esta se podrá desarrollar el algoritmo generador de código de la BCI completa. Este algoritmo solo requerirá de un conjunto de parámetros que, generalmente, diferirán entre pacientes. De esta forma, se implementaran en FPGA BCI completas, personalizadas, optimizadas e independientes sin necesidad de que el que lo haga requiera de conocimientos de sistemas digitales ni FPGA.

Apéndice A

Evaluación detallada de los sistemas digitales de la implementación FPGA de la SVM lineal

Sistema digital Receptor Su evaluación consistió en implementarlo como sistema digital único en el FPGA. Los parámetros de configuración fueron *baud rate*=9600, 1 bit de paro, sin protocolo de *handshaking* y 8 bits por paquete de datos. Se realizaron dos pruebas que se describen a continuación:

Prueba 1. Se le envió al FPGA un dato único para que desplegara en los *display* de siete segmentos de la tarjeta RC10. El envío se realizó con la herramienta *Hyperterminal* de Windows. Se verificó visualmente para 22 datos distintos, que no hubo perdida de información. Cada envío requirió la reconfiguración del FPGA.

Prueba 2. Consistió en un flujo de 16 datos pseudoaleatorios, a cada uno el FPGA debió sumarle un "1" aritmético y mostrar el resultado con el *display* de 7 segmentos . Esta evaluación fue realizada con el gCodeGen (únicamente el envío de datos). Se utilizaron 4 conjuntos de prueba distintos para los cuales se verificó visualmente el resultado en todos los casos.

Sistema digital Transmisor de Resultados Su evaluación consistió en dos pruebas que se detallan a continuación:

Prueba 1. En esta se describió en Handel-C un ciclo que se repite n veces, cuyo valor de la variable de control se transmitió a la PC en cada iteración, misma que recibió por medio de la *Hyperterminal*. Se probó con $n = \{40, 60, 80\ 100\}$, en todos los casos el resultado fue el esperado.

Prueba 2. Consistió en enviar a la PC un dato previamente recibido por el sistema digital receptor. La implementación se hizo para flujo de datos, y en modo de sincrónico. El control de la evaluación se llevó a cabo con un plugin del punto de extensibilidad IEvaluar Implementaciones este: realiza una comparación uno a uno entre los datos esperados y los datos recibidos del FPGA. Se realizaron 7 evaluaciones

con archivos diferentes y en ninguno de los casos se detectaron errores en el procesamiento y por ende en la transmisión y recepción de datos.

No se realizó una evaluación más exhaustiva ya que se contaba con experiencia previa en la implementación del protocolo RS232 usando la PSL y Handel-C. En la descripción original de los sistemas digitales receptor y transmisor fueron secuenciales, dependientes e indicaban explícitamente el número de veces que clasificarían. La aportación de este proyecto en ese sentido fue que los dichos sistemas digitales se desarrollaron independientes entre ellos, es decir, un circuito de control para cada uno usando el *scope* de Handel-C. Además los sistemas son reactivos, es decir, mientras el FPGA este energizado y configurado con alguna SVM se podrán realizar clasificaciones. Esto disminuyó la cantidad de recursos destinados al control de ciclos.

Sistemas digitales Preprocesamiento y Clasificar. Se realizó una evaluación para cada una de los resultados obtenidos en las distintas etapas del *datapath*. Se crearon varios plugin para automatizar el proceso de evaluación. Los resultados patrón se obtuvieron con la biblioteca *fixed point* de Matlab. No se procedía a la siguiente etapa del *datapath* hasta que la actual no funcionara adecuadamente y cumpliera las restricciones iniciales.

Apéndice B

Tutorial para agregar pluings al gCodeGen

Agregar un plugin al punto de extensibilidad IGenerar Código.

En la figura B.1 se muestra el IDE RCP Eclipse. Se da clic derecho en el *package* donde se quiera agregar la clase del plugin→*new*→*class*. Luego aparece un *wizard* figura B.2 para crear la clase. En el nombre de la clase se puso, para este ejemplo, *TesisPruebaAgregarElementoMenuGenerar*; posteriormente se da clic en *add* para agregar interfaces. En este caso se agrego la interfaz que corresponde al punto de extensibilidad de nombre *ICodeGenerator*[1] para luego dar clic en *finish*.

Al finalizar el *wizard* se genera una plantilla de la clase que contiene los métodos que el desarrollador de plugins debe implementar, en este caso, para generar el código. En la figura B.3 se muestra el resultado de la plantilla; esos métodos que se ven en la figura son los únicos que se deben implementar para generar automáticamente código. Como ya se dijo, la interacción con los demás plugins y con el gCodeGen lo resuelve el núcleo de alto nivel de la aplicación.

Posteriormente se registra el nuevo plugin usando otro *wizard* que permite modificar el archivo *Manifest* generado por el RCP figura B.4 y se le vincula con la clase creada anteriormente usando el botón *browse*.

Observese que hay tres plugins en el registro; dos que ya estaba antes de crear el nuevo plugin y el que se acaba de anexar y vincular.

Terminado lo anterior y lanzar de nuevo el gCodeGen se podrá observar que el plugin ya se ha agregado en el menú y al dar clic en la acción el controlador del caso de uso Generar Código se ejecutará. En otras palabras el plugin ha sido agregado.

[1] Corresponde a la interfaz IGenerador de Código. Se utilizaron nombres en inglés para facilitar el desarrollo de plugins a desarrolladores no hispanohablantes).

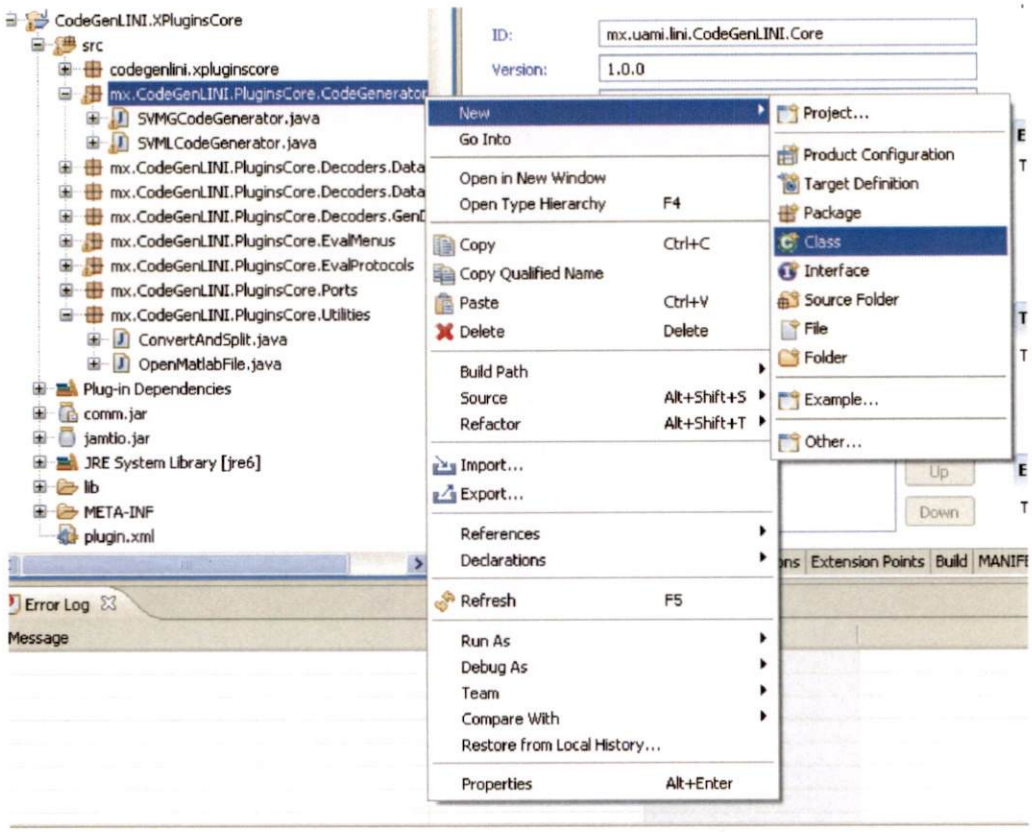


Figura B.1 – *Muestra como se genera una nueva clase en el RCP de eclipse para ser agregada como un plugin del punto de extensibilidad IGenerar Código.*

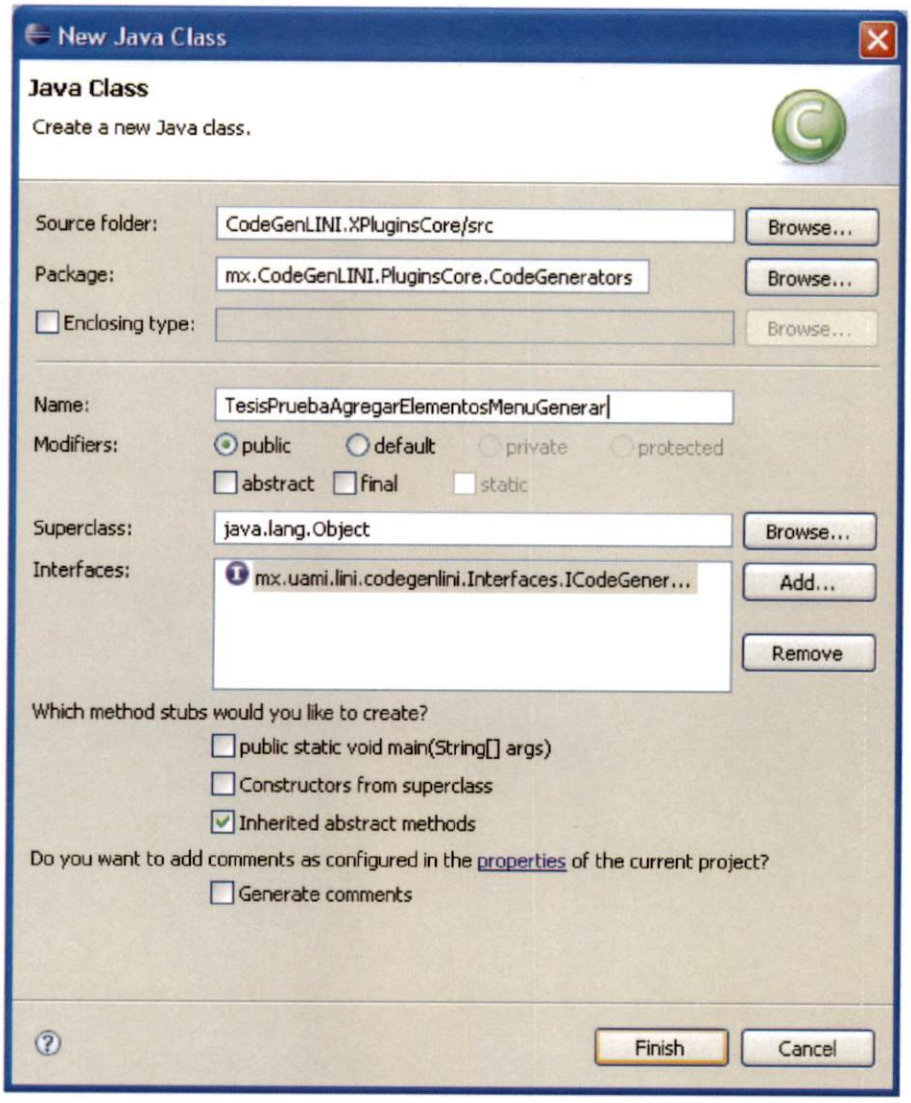


Figura B.2 – *Muestra el wizard para la creación de una clase que será incluida como un plugin del punto de extensibilidad IGenerar Código.*

OpenMatlabFile.java | mx.uami.lini.CodeGenLINI.Core | TesisPruebaAgregarElementosMenuGenerar.java

```
package mx.CodeGenLINI.PluginsCore.CodeGenerators;

import java.util.ArrayList;

public class TesisPruebaAgregarElementosMenuGenerar implements ICodeGenerator {

    public ArrayList<String> generateCode(ArrayList[] param) {
        // TODO Auto-generated method stub
        return null;
    }

    public String getActionName() {
        // TODO Auto-generated method stub
        return null;
    }

    public String getSubMenuType() {
        // TODO Auto-generated method stub
        return null;
    }

    public IPlugin pluginRef() {
        // TODO Auto-generated method stub
        return null;
    }

    public String pluginType() {
        // TODO Auto-generated method stub
        return null;
    }

    public void setPluginManagerRef(IPluginManager ref) {
        // TODO Auto-generated method stub

    }
```

Figura B.3 – *Muestra la plantilla que el desarrollador de plugins tiene que llenar para que se genere automáticamente el código.*

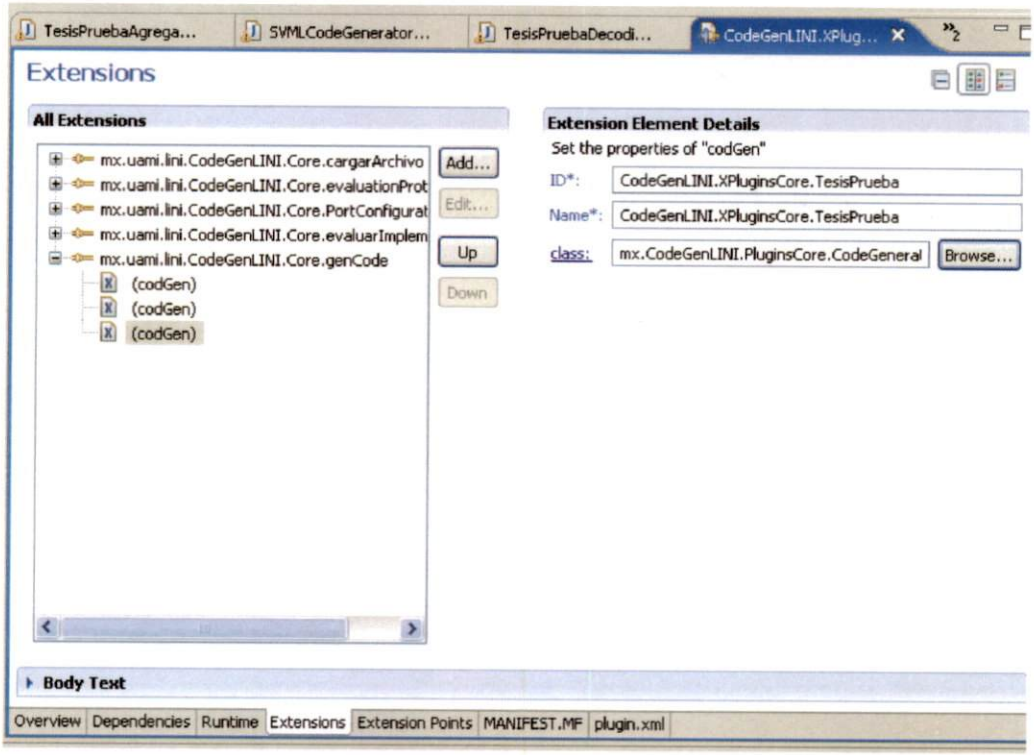


Figura B.4 – *Muestra como se registra un nuevo plugin y como este se vincula a la clase generada en el paso anterior.*

Bibliografía

[1] D. Anguita, A. Boni, and S. Ridella. "The Digital Kernel Perceptron". *Electronics Letters*, 38(10):445–456, 2002.

[2] D. Anguita, A. Boni, and S. Ridella. "A Digital Architecture for Support Vector Machines: Theory, Algorithm, and FPGA Implementation". *IEEE Transactions on Neural Networks*, 14(5):993–1009, September 2003.

[3] Arthur C. Guyton and John E. Hall. *"Tratado de Fisiología Médica"*. Mc Graw Hill Interamericana, 10 edition, 2004.

[4] Centro de Estadísticas Nacionalonales de Lesiones de Méduldulas, Birmingham, Alabamabama. EEUU. "Lesión de Medula Espinal: Datos y Cifras". Technical report, University of Alabama Birmingham, 2000.

[5] Celoxica Co. "Handel-C Code Optimization". `http://www.celoxica.com`.

[6] Celoxica Co. "Handel C Language Reference Manual. `http://www.celoxica.com`.

[7] Celoxica Co. "Pipelined Floating-point Library Manual". `http://babbage.cs.qc.edu/courses/cs345/Manuals/Floating%20Point%20Library%20Manual.pdf`.

[8] Xilinx Co. "HDL Synthesis for FPGAs". `http://www.xilinx.com`.

[9] Xilinx Co. "Spartan-3 FPGA Family". `http://www.xilinx.com`, Diciembre 2009.

[10] Xilinx Co. "Spartan-6 Family Overview". `http://www.xilinx.com`, Noviembre 2010.

[11] Xilinx Co. "Virtex-6 Family Overview". `http://www.xilinx.com`, Enero 2010.

[12] Nello Cristianini and John Shawe Taylor. *"An Introduction to Support Vector Machines"*. Cambridge University, 2000.

[13] Herbert Dawid and Heinrich Meyr. "CORDIC Algorithms and Architectures"'. Technical report, Advanced Topics in CAD Component-Based Design of Electronic Systems. The University of California, 2000.

[14] Emanuel Donchin and M. G. H. coles. "Is the P300 manifestation of context updating? *Behavioural Brain Science*, 11:357–374, 1988.

[15] R.O. Duda, P.E. Hart, and D.G. Stork. *"Pattern Classification"*. Wiley-Interscience, 2000.

[16] E.J. Ciaccio, S.M. Dunn, and M. Akay. "Biosignal Pattern Recognition and Interpretation System: Feature Extraction". *IEEE Engineering in Medicine and Biology*, pages 106–113, December 1993.

[17] E.J. Ciaccio, S.M. Dunn, and M. Akay. "Biosignal Pattern Recognition and Interpretation System: General Concepts". *IEEE Engineering in Medicine and Biology*, pages 89–97, September 1993.

[18] E.J. Ciaccio, S.M. Dunn, and M. Akay. "Biosignal Pattern Recognition and Interpretation System: Methods of Classification". *IEEE Engineering in Medicine and Biology*, pages 129–135, February/March 1994.

[19] E.J. Ciaccio, S.M. Dunn, and M. Akay. "Biosignal Pattern Recognition and Interpretation System: Review of Applications". *IEEE Engineering in Medicine and Biology*, pages 269–283, April/May 1994.

[20] Erich Gamma, Richard Helm, Ralph Jonson, and John Vlissides. *"Design Patterns : Elements of Reusable Object Oriented Software"*. 1995.

[21] David Garlan and Mary Shaw. "An Introduction to Software Architecture". Technical report, SEI Technical Report CMU/SEI-94-TR-21, 1993.

[22] Roman Genov. "Kerneltron: Support Vector Machine in Silicon". *IEEE Transactions on Neural Networks*, 14(5), September 2003.

[23] Wojciech Gradkowski. "JMatIO - Matlab's MAT-file I/O in JAVA". `http://sourceforge.net/projects/jmatio`, 2007.

[24] World Health Organitation. "World Health Organitation, Noncommunicable diseases and mental health". `http://www.who.int/nmh/a5817/en/`, 2000.

[25] John Hopf. "A Parameterizable HandelC Divider Generator for FPGAs with Embedded Hardware Multipliers". *Proceedings. IEEE International Conference on Field-Programmable Technology, 2004*, 2004.

[26] David J. Freedman and John A. Assad. "Experience-dependent Representation of Visual Categories in Parietal Cortex". *Nature*, (443):85–88, September 2006.

[27] Jacques J. Vidal. "Toward Direct Brain-Computer Communication". *Brain Computer Institute, University of California. L.A. EUA*, 1973.

[28] Ivan Jaconson, Grandy Booch, and James Rumbaugh. *"The Unified Software Development Process"*. Addison Wesley, 1999.

[29] Keane Jarvi. "Serial Port RXTX". `http://rxtx.qbang.org/wiki/index.php/Main_Page`, 2008.

[30] Matthias Kaper, Peter Meinicke, Ulf Grossekathoefer, Thomas Lingner, and Helge Ritter. "BCI Competition 2003-Data Set IIb: Support Vector Machines for the P300 Speller Paradigm". *IEEE Transactions On Biomedical Engineering*, 51(6), June 2004.

[31] Brian Kernighan and Dennis Ritchie. *"The C Programming Language"*. Prentince Hall, 1988.

[32] F. Khan, M. Arnold, and W. Pottenger. "Hardware-Based Support Vector Machine Classification in Logarithmic Number Systems". *IEEE Euromicro Symp. Digital System Design (DSD)*, pages 254–261, September 2004.

[33] F. Khan, M. Arnold, and W. Pottenger. "Finite precision analysis of support vector machine classification in logarithmic number systems. *IEEE International Symposium on Circuits and Systems*, pages 1–4, May 2005.

[34] C. Lin. "libSVM: A library for support vector machines.". `http://www.csic.ntu.tw/~cjlin/libsvm/index.html`.

[35] Chistopher M. Bishop. *"Pattern Recognition and Machine Learning"*. Springer, 2006.

[36] Edmund M. Clarke. *"Lecture Notes in Computer Science: Model Checking"*. Springer, 1997.

[37] Winmerge Org. "Winmerge". `http://winmerge.org/`, 2008.

[38] Kroll P. and Kruchten P. *" The Rational Unified Process Made Easy"*. Addison Wesley, 2003.

[39] Omar Piña Ramírez. "Implementación en Hardware de Máquinas de Soporte Vectorial Lineales". Master's thesis, Universidad Autónoma Metropolitana Unidad Iztapalapa, 2005.

[40] Omar Piña-Ramírez, Raquel Valdés-Cristerna, and Oscar Yañez-Suárez. "An FPGA implementation of linear kernel Support Vector Machines". *Proceedings of the 2006 IEEE International Conference on Reconfigurable Computing and FPGA's* , 2006.

[41] Robin Pottathuparambil and Ron Sass. "Implementation of a CORDIC based Double-Precision Exponential Core on an FPGA". *Proceedings. Reconfigurable Systems Summer Institute 2008*, 2008.

[42] Eclipse Project. "Eclipse Process Framework Project (EPF)". `http://www.eclipse.org/epf/`, 2006.

[43] Elipse Project. "OpenUP Eclipse Open Framework". `http://epf.eclipse.org/wikis/openup/`.

[44] Esther R. Marx. "EDIF Standard for Workstation Communication". *IEEE Micro.*, 1985.

[45] Ulrich Rückert. "vMagic". `http://wwwhni.uni-paderborn.de/sct/extern/vmagic/`, 2009.

[46] J. Rumbaugh, I. Jacobson, and G. Booch. *"El lenguaje Unificado de Modelado, Manual de Referencia"*. Pearson, 2000.

[47] Gerwin Schalk, Dennis J. McFarland, Thilo Hinterberger, Niels Birbaumer, and Jonathan R.Wolpaw. "BCI2000: A General-Purpose Brain-Computer Interface (BCI) System". *Transaction on Biomedical Engineering*, 51(6), June 2004.

[48] Konstantinos Seeeigeorgidis and Jan Rabaey. "Ultra Low Power CORDIC Processor for Wireless Communication Algorithms". *Journal of VLSI Signal Processing*, 38, 2004.

[49] IEEE Computer Society. " IEEE Standard Glossary of Software Engineering Terminology". *IEEE STD 610.12-1990*, 1990.

[50] Software Engineering Project 2004 Steering Committee. "Software Engineering 2004. Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering.". *IEEE Computing Curricula Series*, 2004.

[51] S.Sonnenburg, G.Raetsch, C.Schaefer, and B.Schoelkopf. "Shogun - A Large Scale Machine Learning Toolbox". `http://www.fml.tuebingen.mpg.de/raetsch/projects/shogun`, 2008.

[52] Sun Development Network. "Java Communications". `http://java.sun.com/products/javacomm/`, 2000.

[53] The MathWorks, Inc. "Matlab Getting Started". `http://www.mathworks.com/`, 2009.

[54] S. Theodoridis and K. Koutroumbas. *"Pattern Recognition"*. USA: Academic Press, 1999.

[55] Eric W. Sellers a, Dean J. Krusienski, Dennis J. McFarland, Theresa M. Vaughan, and Jonathan R. Wolpawa. "P300 Event-Related Potential Brain-Computer Interface: The Effects of Matrix Size and Inter Stimulus Interval on Performance". *Elsevier Biological Psychology*, 73:242–252, 2006.

[56] Stephen Williams. "Icarus Verilog Develoment Tool". `http://www.icarus.com/eda/verilog/`, 2006.