INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

TRABAJO TERMINAL II

Aplicación Multiplataforma para el Aprendizaje y Depuración de Código HDL sin Requerimiento de Hardware Físico

No. de Registro: TT 2026-B059

PRESENTA:

Ulrich Tamayo Daniel

DIRECTORES:

M. en C. Pastrana Fernández Carlos Jesús

Dr. Cifuentes Álvarez Alejandro Sigrido

Ciudad de México, Junio de 2026

Advertencia

Este trabajo fue realizado en la Escuela Superior de Cómputo del Instituto Politécnico Nacional, como parte de los requisitos para obtener el título de Ingeniero en Sistemas Computacionales. La utilización e implementación de los resultados obtenidos se limita a los alcances definidos en este documento.

El autor autoriza al Instituto Politécnico Nacional para que este documento sea consultado con fines académicos y de investigación, siempre que se cite la fuente correspondiente.

Ciudad de México, a _____ de _____ de _____

_____

Ulrich Tamayo Daniel

Resumen

El presente trabajo terminal aborda el desarrollo de Kmila, una aplicación multiplataforma orientada al aprendizaje y depuración de código en VHDL (VHSIC Hardware Description Language), ejecutada en entorno local y sin requerimiento de hardware especializado (FPGA). La enseñanza del diseño de sistemas digitales enfrenta barreras significativas: el costo de tarjetas FPGA como la Altera Cyclone IV (~$1,866 MXN) o la Basys 3 (~$3,400 MXN), las limitaciones de las versiones gratuitas de herramientas comerciales como ModelSim o Quartus Prime, y la fragmentación del flujo de trabajo en herramientas de código abierto como GHDL y GTKWave. Ante este panorama, se propone una plataforma integrada, gratuita y de código abierto que unifique edición, simulación y visualización de señales digitales en una sola interfaz.

El trabajo comprende el análisis de requerimientos, el diseño arquitectónico y la construcción del sistema completo. La aplicación, desarrollada con .NET 9 mediante MAUI para plataformas nativas (Windows, macOS, Android, iOS) y Blazor para la versión web (Linux), integra un editor de código basado en Monaco Editor con resaltado de sintaxis para VHDL, un sistema de gestión de proyectos y archivos con persistencia en SQLite, un selector de dispositivos FPGA con especificaciones de 34 modelos de fabricantes como AMD/Xilinx, Intel/Altera y Lattice, un panel de configuración de parámetros de simulación, y un visualizador de formas de onda renderizado en SVG en línea. Adicionalmente, la interfaz cuenta con soporte de internacionalización en siete idiomas. El soporte para Verilog se contempla como extensión futura.

El núcleo del sistema es un motor de simulación propio —analizador sintáctico, intérprete de ejecución por ciclos delta y control temporal— cuya salida se validó funcionalmente equivalente a la del simulador de referencia GHDL en 148 de 148 casos de prueba. Sobre dicho motor se incorporaron además un modo de ejecución visual interactivo, la depuración en vivo del diseño y un editor bidireccional VHDL–bloques, junto con los subsistemas pedagógicos de aprendizaje, práctica y conceptos.

Palabras clave: Academia de Ingeniería de Software, Departamento de Sistemas Electrónicos, Educación en Lenguaje Descriptivo de Hardware, Simulación Digital, Interfaz Multiplataforma

Abstract

This thesis addresses the development of Kmila, a cross-platform application aimed at learning and debugging VHDL (VHSIC Hardware Description Language) code, running locally without requiring specialized hardware (FPGA). The teaching of digital systems design faces significant barriers: the cost of FPGA boards such as the Altera Cyclone IV (~$1,866 MXN) or the Basys 3 (~$3,400 MXN), the limitations of free versions of commercial tools like ModelSim or Quartus Prime, and the fragmented workflow of open-source tools such as GHDL and GTKWave. In light of this, an integrated, free, and open-source platform is proposed that unifies code editing, simulation, and digital signal visualization within a single interface.

This work comprises the requirements analysis, the architectural design, and the construction of the complete system. The application, built with .NET 9 using MAUI for native platforms (Windows, macOS, Android, iOS) and Blazor for the web version (Linux), integrates a code editor based on Monaco Editor with VHDL syntax highlighting, a project and file management system with SQLite persistence, an FPGA device selector featuring specifications for 34 models from manufacturers such as AMD/Xilinx, Intel/Altera, and Lattice, a simulation parameter configuration panel, and an inline-SVG-based waveform viewer. Additionally, the interface provides internationalization support for seven languages. Verilog support is planned as a future extension.

The system's core is a custom simulation engine —syntactic analyzer, delta-cycle execution interpreter, and temporal control— whose output was validated as functionally equivalent to that of the reference simulator GHDL across 148 of 148 test cases. On top of this engine, an interactive visual execution mode, live design debugging, and a bidirectional VHDL–blocks editor were added, together with the learning, practice, and concepts pedagogical subsystems.

Keywords: Software Engineering Academy, Electronic Systems Department, Hardware Description Language Education, Digital Simulation, Cross-platform Interface

Índice de Tablas

Índice de Figuras

Glosario y Acrónimos

Este glosario recopila los términos técnicos y acrónimos empleados a lo largo del documento. Cuando un mismo acrónimo admite más de una interpretación (por ejemplo, **RTL**, **DSP** o **ISP**), se incluyen entradas separadas indicando el sentido específico con el que aparece cada uno en el texto.

Término	Definición
Amaranth HDL	Lenguaje de descripción de hardware embebido en Python, sucesor conceptual de MyHDL. Genera Verilog como formato de salida para síntesis.
ASIC	*Application-Specific Integrated Circuit* — circuito integrado de aplicación específica, fabricado para una función concreta y no reconfigurable una vez manufacturado.
AST	*Abstract Syntax Tree* — árbol de sintaxis abstracta; representación intermedia del código fuente tras el análisis sintáctico.
Bitstream	Archivo binario resultado del proceso de síntesis, place-and-route y generación, que configura físicamente la lógica interna de un FPGA.
Blazor	Framework de Microsoft para construir interfaces web interactivas con C# sobre WebAssembly o servidor.
BRAM	*Block RAM* — bloque de memoria RAM dedicado embebido en la matriz lógica del FPGA, empleado para implementar memorias síncronas de hasta decenas de Kbits.
BSV	*Bluespec SystemVerilog* — lenguaje de descripción de hardware con paradigma de reglas guardadas (*guarded atomic actions*) desarrollado en el MIT.
C4	Modelo de arquitectura de software propuesto por Simon Brown con cuatro niveles jerárquicos: contexto, contenedores, componentes y código.
Chisel	*Constructing Hardware In a Scala Embedded Language* — DSL embebido en Scala para generar hardware parametrizable; base del ecosistema RISC-V y del procesador Rocket Chip.
CLB	*Configurable Logic Block* — bloque lógico configurable; unidad básica de la arquitectura de FPGAs de Xilinx, compuesta por LUTs, flip-flops y lógica de acarreo.
CPLD	*Complex Programmable Logic Device* — dispositivo lógico programable complejo; antecesor arquitectónico del FPGA, basado en matrices de términos producto y memoria no volátil.
DCM	*Digital Clock Manager* — bloque de gestión de reloj en FPGAs de Xilinx (familia Spartan-6), capaz de realizar multiplicación, división y desfase de frecuencia.
DSL	*Domain-Specific Language* — lenguaje de dominio específico; lenguaje diseñado para un problema o dominio particular (por ejemplo, Chisel o SpinalHDL para descripción de hardware).
DSP (disciplina)	*Digital Signal Processing* — procesamiento digital de señales; área del tratamiento matemático de señales muestreadas.

Término	Definición
DSP (bloque)	Bloque de hardware dedicado al procesamiento aritmético integrado en los FPGAs. Recibe distintos nombres según el fabricante: *DSP48A1*/*DSP48E1* en Xilinx, *multiplicador 18x18* en Intel/Altera, y *MAC 16x16* o *multiplicador 18x18* en Lattice. Es el sentido empleado en las tablas de dispositivos.
E^{2}PROM	*Electrically Erasable Programmable Read-Only Memory* — memoria no volátil borrable eléctricamente. Tecnología de base de los dispositivos GAL y de la configuración no volátil en familias como iCE40 e ispLSI.
EBR	*Embedded Block RAM* — denominación de Lattice para los bloques de memoria RAM dedicada embebidos en sus FPGAs (equivalente a BRAM en Xilinx y M9K en Intel/Altera Cyclone IV).
ESCOM	Escuela Superior de Cómputo del Instituto Politécnico Nacional.
ESL	*Electronic System Level* — nivel de abstracción de sistema electrónico, empleado por lenguajes como SystemC para modelado arquitectónico temprano.
FF	*Flip-Flop* — elemento de memoria básico biestable utilizado como unidad de almacenamiento síncrono en el hardware digital. Una columna dedicada en los catálogos de FPGAs reporta el número disponible.
FIRRTL	*Flexible Intermediate Representation for RTL* — representación intermedia utilizada por Chisel como paso previo a la generación de Verilog.
FPGA	*Field-Programmable Gate Array* — dispositivo de lógica programable en campo; circuito integrado reconfigurable por el usuario final después de su fabricación.
GAL	*Generic Array Logic* — familia de dispositivos lógicos programables no volátiles basados en E^{2}PROM, comercializada originalmente por Lattice a partir de 1985.
GHDL	Analizador, simulador y compilador de código abierto para VHDL.
GTKWave	Visor de formas de onda electrónicas de código abierto, ampliamente utilizado con archivos VCD y FST.
HDL	*Hardware Description Language* — lenguaje de descripción de hardware; categoría genérica que incluye VHDL, Verilog y SystemVerilog.
HLS	*High-Level Synthesis* — síntesis de alto nivel; traducción de código C/C++ o SystemC a RTL sintetizable en VHDL o Verilog.
IDE	*Integrated Development Environment* — entorno integrado de desarrollo que agrupa editor, compilador, depurador y otras herramientas en una sola aplicación.
IPN	Instituto Politécnico Nacional.
ISP (arquitectura)	*Instruction Set Processor* — formalismo propuesto por Bell y Newell en 1971 para describir la arquitectura de procesadores a nivel de transferencia de registros. Aparece en la Sección 2.1.1.
ISP (electrónica)	*In-System Programming* — técnica que permite reprogramar dispositivos lógicos no volátiles ya montados en la placa del circuito. Es el origen del nombre comercial de la familia *ispLSI* de Lattice.
Kmila	Nombre propio del software de simulación y depuración de código HDL desarrollado en el presente Trabajo Terminal.

Término	Definición
LE	*Logic Element* — elemento lógico; unidad básica de la arquitectura de FPGAs de Intel/Altera, compuesta por una LUT de 4 entradas, un registro y lógica auxiliar.
LUT	*Look-Up Table* — tabla de consulta; elemento lógico configurable fundamental de todo FPGA. El número de entradas varía según el fabricante y la familia: 4 entradas en Intel/Altera Cyclone IV y Lattice iCE40; 6 entradas en AMD/Xilinx series 6 y 7 y en Lattice ECP5.
MAC	*Multiply-Accumulate* — operación de multiplicación y acumulación fusionada; en los FPGAs Lattice iCE40 UltraPlus designa el tipo de bloque DSP disponible (MAC 16×16).
MAUI	*.NET Multi-platform App UI* — framework de Microsoft para construir aplicaciones nativas multiplataforma (Windows, macOS, Android, iOS) sobre .NET.
MMCM	*Mixed-Mode Clock Manager* — gestor mixto de reloj; evolución del DCM en FPGAs de Xilinx (serie 7 y posteriores) con capacidad de síntesis de frecuencia de precisión fraccionaria.
Monaco Editor	Editor de código embebible desarrollado por Microsoft, núcleo de Visual Studio Code, integrado en Kmila para la edición de archivos HDL.
MyHDL	Biblioteca de Python para descripción de hardware digital con capacidad de conversión a Verilog y VHDL.
Netlist	Lista de componentes lógicos primitivos y sus interconexiones, producto de la etapa de síntesis.
Place-and-Route	Etapa del flujo de diseño FPGA que asigna los elementos del netlist a recursos físicos (LUTs, flip-flops, BRAM) y determina el enrutamiento de las interconexiones.
PLL	*Phase-Locked Loop* — lazo de seguimiento de fase; circuito integrado de los FPGAs usado para generación y síntesis de frecuencias de reloj.
PSL	*Property Specification Language* — lenguaje de especificación de propiedades, estandarizado como IEEE 1850 e integrado en VHDL 2008.
RTL (abstracción)	*Register-Transfer Level* — nivel de transferencia de registros; nivel de abstracción en diseño digital que describe el circuito en términos de registros y la lógica combinacional que transfiere datos entre ellos en cada ciclo de reloj. Es el nivel empleado para síntesis.
RTL (escritura)	*Right-to-Left* — dirección de escritura de derecha a izquierda, empleada por idiomas como el árabe y el hebreo. En Kmila se aplica al idioma árabe mediante el atributo HTML `dir=rtl"`.
SerDes	*Serializer/Deserializer* — bloque transceptor de alta velocidad que convierte entre flujos de datos serie y paralelo, integrado en familias FPGA de gama media y alta (Artix-7, ECP5).
SoC	*System-on-Chip* — sistema en chip; circuito integrado que combina procesador, memoria, periféricos y eventualmente lógica programable en un único dispositivo (por ejemplo, la familia Xilinx Zynq).
SpinalHDL	Lenguaje de descripción de hardware embebido en Scala, alternativa a Chisel con énfasis en la legibilidad. Compila a Verilog y VHDL.

Término	Definición
SQLite	Motor de base de datos relacional embebido, sin servidor, utilizado en Kmila para la persistencia local de proyectos y configuración.
SVA	*SystemVerilog Assertions* — subconjunto de SystemVerilog orientado a la verificación formal y funcional mediante aserciones.
Síntesis lógica	Proceso de traducción de código RTL (en VHDL o Verilog) a un netlist de compuertas lógicas a partir de una biblioteca tecnológica específica.
SystemC	Biblioteca de C++ estandarizada como IEEE 1666 para el modelado de hardware a nivel de sistema (ESL).
SystemVerilog	Extensión de Verilog que añade programación orientada a objetos, aserciones (SVA), cobertura funcional y verificación con restricciones aleatorias; estandarizado como IEEE 1800.
Testbench	Banco de pruebas; código HDL (no sintetizable) que genera estímulos de entrada y verifica las salidas del circuito bajo prueba durante la simulación.
TT	Trabajo Terminal; proyecto de titulación en la ESCOM dividido en dos fases, TT1 y TT2.
UVM	*Universal Verification Methodology* — metodología estándar de verificación para ASIC y FPGA basada en SystemVerilog.
VCD	*Value Change Dump* — formato estándar ASCII para registrar cambios de valor en señales durante una simulación; soportado por GTKWave y por Kmila.
Verilog	Lenguaje de descripción de hardware con sintaxis inspirada en C, estandarizado como IEEE 1364 y fusionado con SystemVerilog en IEEE 1800 a partir de 2009.
VHDL	*VHSIC Hardware Description Language* — lenguaje de descripción de hardware desarrollado por el Departamento de Defensa de los Estados Unidos y estandarizado como IEEE 1076. Es el lenguaje primario soportado por Kmila.
VHSIC	*Very High Speed Integrated Circuit* — programa del Departamento de Defensa de los Estados Unidos (1980–1990) que financió el desarrollo de VHDL.
WASM	*WebAssembly* — formato de instrucciones binarias portable diseñado para la ejecución eficiente de código en navegadores web; empleado por Blazor.
Yosys	Suite de síntesis lógica de código abierto con soporte para Verilog y SystemVerilog, ampliamente utilizada con FPGAs Lattice iCE40 y ECP5.

Capítulo 1

Fundamentos del Proyecto

1.1. Introducción

La enseñanza del diseño de sistemas digitales y su implementación en dispositivos lógicos programables, como las Field Programmable Gate Arrays (FPGAs), enfrenta barreras significativas que limitan la formación práctica de los estudiantes. Aunque el dominio de lenguajes de descripción de hardware (HDLs) como VHDL es esencial, la comprensión profunda de la lógica subyacente exige una experiencia de depuración que va más allá de la simple escritura de código.

Las herramientas educativas actuales presentan limitaciones en dos dimensiones principales: económicas y pedagógicas. Por un lado, la dependencia de hardware especializado (tarjetas FPGA) constituye un obstáculo financiero relevante para estudiantes e instituciones. Una tarjeta FPGA Altera Cyclone IV puede costar alrededor de $1,866.00 MXN, mientras que la popular Basys 3 alcanza los $3,400.00 MXN, sin contar accesorios [1], [2]. A ello se suma el precio de cursos especializados, que en plataformas como Coursera puede superar los $1,000 MXN mensuales [3]. Por otro lado, existe un vacío pedagógico en la oferta de plataformas virtuales. Aunque fabricantes como Intel y AMD/Xilinx ofrecen entornos de diseño (Quartus Prime, Vivado), sus versiones gratuitas poseen limitaciones técnicas importantes [4], [5].

En cuanto a herramientas de código abierto, GHDL es un simulador potente, pero su uso está limitado a la línea de comandos y depende de la integración con GTKWave para visualizar señales, lo que genera un flujo de trabajo fragmentado y poco intuitivo para principiantes [6], [7]. En contraste, soluciones comerciales como ModelSim o QuestaSim ofrecen entornos robustos, pero sus licencias educativas restringidas impiden un uso completo en entornos académicos [8].

En la revisión de Trabajos Terminales (TTs) en la ESCOM, no se han encontrado propuestas previas dedicadas a la creación de un depurador interactivo de HDL. Sin embargo, existen antecedentes de proyectos relacionados con simulación, control lógico y sistemas educativos interactivos [9], lo que evidencia una línea institucional que puede ser ampliada con la presente propuesta.

Ante este panorama, el presente trabajo terminal propone el desarrollo de Kmila, una plataforma académica multiplataforma ejecutada como aplicación local, gratuita y de código abierto, enfocada en la depuración lógica y funcional de código VHDL. A diferencia de los simuladores tradicionales, Kmila propone una experiencia integrada con interfaz gráfica intuitiva, visualización de señales digitales y controles de simulación en una sola aplicación, lo que la convierte en una herramienta complementaria para fortalecer la enseñanza del diseño digital en la ESCOM, permitiendo a los estudiantes preparar y validar sus diseños antes de acceder al laboratorio físico, o como alternativa cuando el hardware no esté disponible.

El presente documento abarca el análisis de requerimientos del sistema, el diseño de su arquitectura, la construcción de la interfaz gráfica unificada y la implementación y validación del motor de simulación, junto con los subsistemas de depuración en vivo, ejecución visual y apoyo pedagógico que completan la plataforma.

1.2. Objetivo General

Desarrollar una aplicación multiplataforma, gratuita y de código abierto, que permita la simulación y depuración interactiva de código en lenguajes de descripción de hardware (HDL) en un entorno local, facilitando el aprendizaje de diseño digital sin requerir hardware especializado (FPGA).

1.3. Objetivos Específicos

1. Analizar los requerimientos funcionales y no funcionales del sistema, identificando las necesidades de los usuarios objetivo (estudiantes y docentes de ESCOM) en el contexto de la enseñanza del diseño digital.
2. Diseñar la arquitectura del sistema considerando una estructura modular que contemple los componentes de edición de código, simulación, visualización de señales y control temporal, utilizando .NET 9 con MAUI y Blazor como frameworks base.
3. Construir una interfaz gráfica unificada que integre un editor de código basado en Monaco Editor con resaltado de sintaxis para VHDL, un sistema de gestión de proyectos y archivos, y un panel de configuración de parámetros de simulación.
4. Implementar un selector de dispositivos FPGA que permita al usuario consultar las especificaciones de recursos de hardware (LUTs, flip-flops, pines I/O, bloques de memoria, DSPs) de múltiples modelos comerciales.
5. Desarrollar un visualizador de formas de onda que permita la representación gráfica de señales digitales, preparado para su integración futura con el motor de simulación.
6. Validar el funcionamiento de la interfaz en su versión web orientada a Linux, verificando la correcta renderización y funcionalidad de los componentes mediante Blazor.

1.4. Antecedentes y Estado del Arte

1.4.1. Contexto Histórico

El diseño digital ha evolucionado significativamente desde la introducción de los primeros lenguajes de descripción de hardware en la década de 1980. VHDL, desarrollado originalmente por el Departamento de Defensa de los Estados Unidos y estandarizado por IEEE en 1987 (IEEE Std 1076) [10], se consolidó como el lenguaje predominante en entornos académicos de Europa y Latinoamérica [11]. Por su parte, Verilog, estandarizado como IEEE Std 1364 [12], encontró mayor adopción en la industria norteamericana y asiática [13].

La enseñanza de estos lenguajes en instituciones como la ESCOM se ha apoyado históricamente en el uso de tarjetas FPGA físicas, principalmente modelos de la familia Cyclone de Intel/Altera y Spartan/Artix de AMD/Xilinx. Sin embargo, la disponibilidad limitada de este hardware en los laboratorios, combinada con su costo, ha restringido la práctica individual de los estudiantes fuera del aula.

En respuesta a estas limitaciones, ha surgido un ecosistema diverso de herramientas de simulación y desarrollo que puede clasificarse en cuatro categorías: herramientas de código abierto, herramientas comerciales, plataformas web y herramientas académicas especializadas. A continuación se analiza cada categoría en detalle, evaluando sus capacidades, limitaciones y pertinencia para el contexto educativo que motiva el presente proyecto.

1.4.2. Herramientas de Código Abierto

GHDL es un analizador, compilador y simulador de código abierto para VHDL, desarrollado principalmente por Tristan Gingold bajo licencia GPLv2 [6]. A diferencia de los simuladores interpretados, GHDL compila el código VHDL a código máquina nativo mediante backends seleccionables (GCC, LLVM o un generador mcode propio), lo que le confiere velocidades de simulación significativamente superiores. GHDL soporta los estándares IEEE 1076-1987, 1076-1993 y 1076-2002 de forma completa, con soporte parcial para IEEE 1076-2008 y 1076-2019. Adicionalmente, puede actuar como frontend de VHDL para el sintetizador Yosys mediante el plugin ghdl-yosys-plugin, habilitando un flujo de diseño completamente abierto desde RTL hasta bitstream. Sin embargo, GHDL opera exclusivamente por línea de comandos y no incluye visor de formas de onda, por lo que requiere la integración con una herramienta externa para la visualización de señales [14].

GTKWave es un visor de formas de onda de código abierto basado en GTK+, desarrollado por Anthony Bybell bajo licencia GPL [7]. Soporta múltiples formatos de archivo de volcado de simulación, incluyendo VCD, FST, LXT, LXT2, VZT y GHW (formato nativo de GHDL). GTKWave es la herramienta estándar de visualización en el ecosistema EDA de código abierto y es compatible con la salida de GHDL, Icarus Verilog y Verilator. Sus limitaciones principales son una interfaz gráfica que puede resultar poco intuitiva para principiantes y la ausencia de capacidad de simulación propia, lo que obliga al usuario a gestionar un flujo de trabajo fragmentado entre el simulador y el visor.

El flujo de trabajo combinado **GHDL + GTKWave** representa la alternativa de código abierto más completa para simulación de VHDL, pero exige que el estudiante instale, configure y opere dos herramientas independientes con interfaces y paradigmas distintos, generando una curva de aprendizaje que desvía la atención del objetivo pedagógico principal: comprender el comportamiento del hardware descrito en HDL.

Icarus Verilog (iverilog) es un compilador y simulador de código abierto para Verilog, creado por Stephen Williams bajo licencia GPL [15]. Compila código Verilog en un formato intermedio que es ejecutado por un motor de simulación (vvp), generando volcados de señales en formato VCD o FST. Icarus Verilog soporta los estándares IEEE 1364-1995, 1364-2001 y 1364-2005, con soporte parcial para SystemVerilog. Es ampliamente utilizado como backend de simulación en plataformas educativas como EDA Playground y HDLBits. Al igual que GHDL, opera exclusivamente por línea de comandos y depende de GTKWave para la visualización de señales.

Verilator es un simulador de Verilog/SystemVerilog de alto rendimiento que compila HDL sintetizable en modelos optimizados de C++ o SystemC, desarrollado por Wilson Snyder bajo licencia LGPLv3 [16]. Reporta velocidades hasta 100 veces superiores a las de simuladores interpretados como Icarus Verilog en un solo hilo, con aceleraciones adicionales de 2x a 10x mediante multithreading. Sin embargo, Verilator presenta limitaciones importantes para uso educativo: opera únicamente con simulación de dos estados (sin soporte para valores desconocidos “x” ni alta impedancia “z”), está restringido a diseños sintetizables (sin soporte para retardos `#delay` ni bloques `initial` con temporización), y requiere conocimientos de C++ para la escritura de testbenches.

Yosys (Yosys Open SYnthesis Suite) es un framework de síntesis RTL de código abierto desarrollado por Claire Xenia Wolf y mantenido por YosysHQ bajo licencia ISC [17]. Yosys procesa Verilog-2005 sintetizable y genera netlists para FPGAs de familias como Lattice iCE40, ECP5, Xilinx 7-Series y Gowin. Aunque no es un simulador en sí mismo, su backend CXXRTL permite simulación post-síntesis y, combinado con nextpnr, habilita un

flujo completo RTL-to-bitstream de código abierto. Yosys es principalmente una herramienta de síntesis y no aborda las necesidades de simulación comportamental ni visualización interactiva que requiere un entorno educativo integrado.

1.4.3. Herramientas Comerciales

ModelSim / QuestaSim (Siemens EDA) es el simulador HDL comercial de referencia en la industria, basado en tecnología Single Kernel Simulator (SKS) que permite simulación mixta transparente de VHDL, Verilog y SystemVerilog [8]. Ofrece depuración avanzada, cobertura de código (sentencias, ramas, condiciones, toggle, FSM) y un visor de formas de onda profesional con búsqueda de señales, marcadores y comparación. QuestaSim, su versión superior, añade soporte completo para UVM, aserciones SystemVerilog y mayor rendimiento. Sin embargo, la licencia completa de ModelSim/QuestaSim es comercial y su costo resulta prohibitivo para muchas instituciones educativas en Latinoamérica. Existen versiones gratuitas limitadas: la Questa-Intel FPGA Starter Edition impone un límite de 10,000 líneas de código y está restringida a bibliotecas de FPGAs Intel [8]. Cabe señalar que Siemens EDA ha anunciado la discontinuación de nuevas ventas de ModelSim a partir de 2024, migrando a la línea Questa One Sim.

Intel Quartus Prime es el entorno de diseño integral para FPGAs y CPLDs de Intel (anteriormente Altera), que proporciona un flujo completo desde la entrada RTL hasta la programación del dispositivo [4]. Su edición Lite es gratuita y no requiere archivo de licencia, pero está limitada a un subconjunto de familias de FPGAs Intel. Incluye la Questa-Intel FPGA Starter Edition para simulación, sujeta a las mismas restricciones de 10,000 líneas. Quartus Prime es ampliamente utilizado en laboratorios universitarios, incluyendo instituciones mexicanas, especialmente con tarjetas de la familia Cyclone. Sus principales limitaciones para uso educativo son su gran tamaño de instalación (decenas de GB), su restricción a Windows y Linux (sin soporte para macOS), y su vinculación exclusiva al ecosistema Intel.

AMD Vivado Design Suite (anteriormente Xilinx Vivado) es el entorno de diseño para FPGAs y SoCs adaptativos de AMD, que incluye XSIM como simulador HDL integrado [5]. La edición ML Standard (anteriormente WebPACK) es gratuita y soporta las familias FPGA más populares en educación, incluyendo Artix-7 (utilizada en la Basys 3), Spartan-7 y Zynq-7000. XSIM permite simulación comportamental, post-síntesis y post-implementación con soporte para VHDL, Verilog y SystemVerilog. Las limitaciones de Vivado incluyen un tamaño de instalación extremo (40–80+ GB según las familias seleccionadas), requisitos elevados de RAM y almacenamiento, soporte limitado a Windows y Linux, y una curva de aprendizaje pronunciada para principiantes. Además, no soporta familias FPGA legadas (Spartan-3, Virtex-5), para las cuales se requiere el discontinuado ISE Design Suite (v14.7, 2013).

1.4.4. Herramientas Web y Académicas

EDA Playground es un entorno de desarrollo integrado basado en web, mantenido por Doulos Ltd., que permite editar, simular y sintetizar código HDL directamente desde un navegador sin instalación alguna [18]. Ofrece múltiples backends de simulación, incluyendo Icarus Verilog y Aldec Riviera Pro (gratuitos) y Synopsys VCS y Cadence Incisive (con validación institucional). Integra EPWave, un visor de formas de onda en navegador, y permite compartir código mediante URLs. Sus limitaciones incluyen la dependencia de conexión a internet, un límite de 1,000,000 caracteres por proyecto, la imposibilidad de simular diseños grandes o multi-archivo de forma efectiva, y la ausencia de persistencia garantizada de los proyectos.

HDLBits es una plataforma web de ejercicios progresivos para la práctica de Verilog HDL, creada por Henry Wong [19]. Funciona como un "juez en línea" para diseño de hardware: el estudiante escribe código Verilog, el sistema lo simula con Icarus Verilog y compara automáticamente las formas de onda resultantes con las esperadas, proporcionando retroalimentación inmediata. HDLBits cubre lógica combinacional, secuencial, máquinas de estados finitos y diseños jerárquicos. Sus limitaciones principales son el soporte exclusivo para Verilog (sin VHDL), la restricción a módulos pequeños y aislados (sin proyectos multi-archivo), y la ausencia de panel de administración para docentes.

DigitalJS es un simulador de circuitos digitales orientado a la enseñanza que se ejecuta completamente en el navegador, desarrollado por Marek Materzok en la Universidad de Wroclaw, Polonia [20]. DigitalJS visualiza circuitos sintetizados por Yosys, permitiendo a los estudiantes interactuar con representaciones a nivel de compuertas de su código Verilog. Ofrece simulación interactiva con visualización de formas de onda en tiempo real y soporte para subcircuitos jerárquicos. Su enfoque es complementario al de un simulador comportamental, ya que requiere la síntesis previa del código y no soporta testbenches ni construcciones no sintetizables.

Logisim Evolution es una herramienta gráfica educativa para el diseño y simulación de circuitos lógicos digitales, mantenida como fork comunitario del Logisim original de Carl Burch [21], [22]. Permite diseñar circuitos mediante una interfaz de arrastrar y soltar con una biblioteca extensa de componentes (compuertas, multiplexores, registros, RAM, ROM), simulación interactiva en tiempo real, y generación de código VHDL y Verilog a partir de esquemáticos. Logisim Evolution es ampliamente utilizado en cursos introductorios de lógica digital a nivel mundial. Sin embargo, opera exclusivamente a nivel de esquemáticos y compuertas, no es un simulador de HDL, y no es adecuado para cursos avanzados de diseño digital o verificación.

1.4.5. Trabajos Académicos y Proyectos Previos

En la revisión de Trabajos Terminales de la ESCOM, se identificaron proyectos que involucran simulación y control de lógica digital de manera indirecta [9]:

- **TT0004 (ARACNOTRÓN):** Robot experimental en forma de araña con control electrónico y remoto que implicó uso de lógica digital simulada.
- **TT0007 (Control de brazo artificial):** Prototipo de mano artificial controlada por PC con lógica digital y control interactivo de hardware.
- **TT0012 (Control de brazo pintor):** Proyecto de control preciso de actuadores que requirió validación previa de lógica digital.
- **TT0006 (Sistema de electrocardiografía):** Adquisición y análisis de señales ECG con diseño de front-end analógico y validación SPICE.
- **TT0022 (Fonocardiógrafo digital):** Captura y análisis de sonidos cardíacos con uso implícito de simulación analógica y digital.
- **TT0030 HM1 (Electroencefalógrafo digital):** Adquisición y transmisión de señales EEG que requirió simulación mixta analógica (SPICE) y digital (HDL).
- **TT0024 (Especificación de protocolos con PROMELA):** Validación formal de protocolos de red con el simulador SPIN, conceptualmente paralelo a la simulación de hardware.

Ninguno de estos trabajos aborda directamente la creación de un entorno integrado de edición, simulación y depuración de código HDL. Esta ausencia evidencia un área de oportunidad institucional que el presente proyecto busca cubrir.

1.4.6. Análisis Comparativo

La Tabla 1.1 presenta un análisis comparativo de las herramientas existentes evaluadas en las secciones anteriores, contrastando sus características con las de la solución propuesta (Kmila). Los criterios de evaluación se seleccionaron con base en las necesidades identificadas en el planteamiento del problema: accesibilidad económica, integración del flujo de trabajo, compatibilidad multiplataforma y orientación educativa.

Tabla 1.1: *Análisis comparativo de herramientas existentes vs. Kmila.*

Herramienta	**Tipo**	**HDL**	**Plataformas**	**Costo**	**Editor integr.**	**Visor ondas integr.**	**GUI**	**Enfoque educativo**
GHDL + GTKWave	Open-source	VHDL	Linux, Win, macOS	Gratuito	No	Separado (GTKWave)	Parcial	Bajo
Icarus Verilog	Open-source	Verilog	Linux, Win, macOS	Gratuito	No	No (requiere GTKWave)	No (CLI)	Bajo
Verilator	Open-source	Verilog/SV (sint.)	Linux, Win, macOS	Gratuito	No	No	No (CLI)	Bajo
Yosys	Open-source (síntesis)	Verilog	Linux, Win, macOS	Gratuito	No	No	No (CLI)	Bajo
ModelSim / Questa	Comercial	VHDL/ Verilog/SV	Win, Linux	Comercial (Starter limitada)	No	Sí	Sí	Medio
Quartus Prime Lite	Comercial	VHDL/ Verilog/SV	Win, Linux	Gratuito (limitado)	Sí	Vía Questa Starter	Sí	Medio
Vivado ML Std.	Comercial	VHDL/ Verilog/SV	Win, Linux	Gratuito (limitado)	Sí	Sí (XSIM)	Sí	Medio
EDA Playground	Web	VHDL/ Verilog/SV	Navegador	Gratuito	Sí	Sí (EPWave)	Sí	Medio
HDLBits	Web	Verilog	Navegador	Gratuito	Sí	Sí	Sí	Alto
DigitalJS	Web (síntesis)	Verilog (vía Yosys)	Navegador	Gratuito	Sí	Sí	Sí	Alto
Logisim Evolution	Académica	Genera VHDL/ Verilog	Win, Linux, macOS	Gratuito	No (esquemático)	Cronograma	Sí	Alto
Kmila	**Open-source**	**VHDL**	**Win, macOS, Linux, Android, iOS, Web**	**Gratuito**	**Sí (Monaco)**	**Sí (SVG)**	**Sí**	**Alto**

Del análisis comparativo se identifican las siguientes brechas que la solución propuesta busca cubrir:

1. **Fragmentación del flujo de trabajo abierto.** Las herramientas de código abierto (GHDL, Icarus Verilog, Verilator) carecen de interfaz gráfica y requieren la combinación de múltiples programas independientes para completar el ciclo edición-simulación-visualización.
2. **Restricciones de las herramientas comerciales.** ModelSim, Quartus Prime y Vivado ofrecen entornos integrados pero con limitaciones significativas en sus versiones gratuitas (líneas de código, familias de dispositivos), grandes requisitos de almacenamiento, y soporte limitado a Windows y Linux.
3. **Limitaciones de las plataformas web.** EDA Playground y HDLBits resuelven el problema de instalación pero dependen de conexión a internet, presentan restricciones de tamaño de proyecto, y no permiten trabajo offline ni persistencia local garantizada.
4. **Ausencia de herramientas académicas para HDL.** Logisim Evolution es excelente para lógica combinacional y secuencial a nivel de esquemáticos, pero no aborda la simulación de código HDL. DigitalJS visualiza síntesis pero no soporta simulación comportamental.
5. **Cobertura multiplataforma insuficiente.** Ninguna de las herramientas analizadas ofrece una aplicación nativa para las seis plataformas principales (Windows, macOS, Linux, Android, iOS y navegadores web) desde un único código base.

Kmila se posiciona como una herramienta complementaria que integra en una sola aplicación local, gratuita y multiplataforma las funcionalidades que actualmente requieren múltiples herramientas o dependen de licencias comerciales restrictivas, con un enfoque explícito en el aprendizaje de diseño digital en el contexto académico de la ESCOM.

1.5. Planteamiento del Problema

Los estudiantes de la Escuela Superior de Cómputo que cursan asignaturas relacionadas con diseño digital, tales como Arquitectura de Computadoras, Fundamentos de Diseño Digital, Diseño de Sistemas Digitales y Sistemas en Chip, enfrentan una brecha entre el conocimiento teórico adquirido en el aula y la práctica efectiva con lenguajes de descripción de hardware.

Esta brecha se origina en tres factores principales:

Barrera económica. El acceso a hardware FPGA representa un costo significativo tanto para los estudiantes como para la institución. Una tarjeta Altera Cyclone IV tiene un costo aproximado de $1,866 MXN y una Basys 3 de $3,400 MXN [1], [2]. Para un grupo de 30 estudiantes, equipar un laboratorio con tarjetas individuales superaría los $55,000 MXN, sin considerar mantenimiento ni reposición. Esto obliga a que los estudiantes trabajen en equipos numerosos con acceso limitado al hardware, reduciendo la experiencia individual de aprendizaje.

Fragmentación de herramientas. Las alternativas de software gratuitas existentes no ofrecen una experiencia integrada. Un estudiante que utilice herramientas de código abierto debe instalar y configurar por separado un editor de texto, un compilador/simulador (GHDL) y un visualizador de señales (GTKWave), cada uno con su propia curva de aprendizaje y flujo de operación [6], [7]. Las herramientas comerciales como ModelSim o Quartus Prime, si bien integran más funcionalidades, restringen sus versiones gratuitas y están vinculadas a familias de hardware específicas [4], [5], [8].

Ausencia de una herramienta institucional. En la revisión de Trabajos Terminales de la ESCOM no se encontró ningún proyecto previo que aborde directamente la creación de un entorno integrado de depuración de HDL [9]. Esto implica que la institución carece de una herramienta propia, gratuita y adaptada a sus necesidades curriculares, que pueda ser distribuida libremente entre estudiantes y docentes.

En consecuencia, se identifica la necesidad de una plataforma unificada que reduzca la dependencia exclusiva de hardware especializado (FPGA), integre las funcionalidades de edición, simulación y visualización en una sola interfaz, y sea accesible sin costo desde múltiples dispositivos, complementando la formación práctica en el laboratorio.

1.6. Público Objetivo

El presente proyecto está dirigido a:

- **Estudiantes de ESCOM** que cursan asignaturas relacionadas con diseño digital, HDL y FPGA, tales como Arquitectura de Computadoras, Fundamentos de Diseño Digital, Diseño de Sistemas Digitales y Sistemas en Chip. Kmila les permite preparar y validar sus diseños en VHDL antes de acceder al laboratorio físico, funcionando como herramienta de pre-laboratorio que optimiza el tiempo limitado con el hardware real. En casos donde el acceso a tarjetas FPGA no sea posible por restricciones económicas o de disponibilidad, la plataforma puede servir como alternativa para completar la práctica de manera individual.
- **Docentes de ESCOM** que imparten las asignaturas mencionadas y requieren una herramienta gratuita y unificada para demostrar conceptos de diseño digital en el aula, asignar prácticas preparatorias que los estudiantes puedan realizar desde cualquier dispositivo previo a las sesiones de laboratorio, y reforzar la comprensión de la lógica HDL sin depender exclusivamente de la disponibilidad de hardware especializado (tarjetas FPGA).
- **Investigadores y desarrolladores** que necesiten un entorno de prototipado rápido para verificar diseños en VHDL de forma preliminar antes de su implementación en hardware, reduciendo tiempos de iteración en proyectos académicos y de investigación.

1.7. Propuesta de Solución

La solución planteada consiste en el desarrollo de Kmila, una plataforma académica multiplataforma ejecutada como aplicación local, gratuita y de código abierto, enfocada en la edición, simulación y depuración de código VHDL. El sistema se diseña como un entorno unificado que integra en una sola interfaz las funcionalidades que actualmente requieren múltiples herramientas independientes.

La arquitectura del sistema se organiza en dos capas principales:

Capa de presentación (alcance de TT1). Comprende la interfaz gráfica unificada, desarrollada con .NET 9 utilizando MAUI para plataformas nativas (Windows, macOS, Android, iOS) y Blazor para la versión web (Linux) [23], [24]. Esta capa integra los siguientes componentes:

- *Editor de código*: basado en Monaco Editor (el mismo motor de Visual Studio Code), proporciona edición de archivos VHDL con resaltado de sintaxis, numeración de líneas y atajos de teclado.
- *Sistema de gestión de proyectos*: permite crear, organizar y administrar proyectos y archivos VHDL con persistencia local mediante SQLite.
- *Selector de dispositivos FPGA*: ofrece un catálogo de 34 modelos de FPGAs de los fabricantes AMD/Xilinx (Spartan-6, Artix-7), Intel/Altera (Cyclone IV E) y Lattice (iCE40 HX, iCE40 UltraPlus, ECP5), con visualización detallada de recursos de hardware (LUTs, flip-flops, pines I/O, bloques de memoria, DSPs).
- *Panel de configuración de simulación*: permite definir parámetros temporales y valores de puertos de entrada para la ejecución futura de simulaciones.
- *Visualizador de formas de onda*: componente gráfico basado en SVG en línea, preparado para representar señales digitales con escala temporal y codificación por colores según el tipo de señal.
- *Internacionalización*: soporte para siete idiomas (español, inglés, francés, alemán, chino simplificado, japonés y árabe), incluyendo dirección de escritura de derecha a izquierda (RTL, *Right-to-Left*) para este último.

Capa de simulación (alcance de TT2). Comprenderá el motor de ejecución del sistema, organizado en tres módulos: un analizador sintáctico (Parser) para procesar código VHDL y generar un Árbol de Sintaxis Abstracta (AST), un intérprete (Interpreter) para ejecutar la lógica descrita mediante ciclos delta y gestión de señales, y un módulo de control temporal (TimeMachine) para administrar el tiempo discreto de simulación y los ciclos de reloj.

A diferencia de las herramientas existentes, Kmila no requiere la instalación de múltiples programas ni depende de línea de comandos. Su enfoque integrado permite que el usuario edite, configure y —en la segunda fase— simule y visualice resultados dentro de la misma aplicación, reduciendo la curva de aprendizaje y complementando la práctica en el laboratorio físico.

1.8. Alcances del Proyecto

1.8.1. Alcances de TT1

- Análisis de requerimientos funcionales y no funcionales del sistema.
- Diseño de la arquitectura modular del sistema utilizando el modelo C4 (contexto, contenedores, componentes, despliegue).
- Construcción de la interfaz gráfica unificada con editor de código Monaco Editor, gestión de proyectos/archivos, y panel de configuración de simulación.
- Implementación del selector de dispositivos FPGA con especificaciones de 34 modelos comerciales.
- Desarrollo del visualizador de formas de onda preparado para integración con el motor de simulación.
- Validación del funcionamiento de la interfaz en su versión web orientada a Linux mediante Blazor.
- Soporte de internacionalización en siete idiomas.
- Documentación completa de análisis y diseño: diagramas UML (clases, secuencia, casos de uso, actividades), historias de usuario, casos de uso detallados y marco teórico.

1.8.2. Alcances de TT2

- Implementación del analizador sintáctico (Parser) para código VHDL con generación de AST.
- Implementación del intérprete de ejecución (Interpreter) con soporte de ciclos delta y gestión de señales.
- Implementación del módulo de control temporal (TimeMachine) para simulación en tiempo discreto.
- Integración del motor de simulación con la interfaz gráfica construida en TT1.
- Estimación de recursos de hardware (flip-flops, LUTs, pines I/O, dominios de reloj) contra especificaciones de dispositivos FPGA.
- Pruebas unitarias, de integración y de aceptación.
- Validación académica con usuarios reales (estudiantes y docentes).
- Documentación técnica final y manual de usuario.
- Publicación del código fuente en repositorio público.

1.8.3. Limitaciones

- El sistema no realiza síntesis de hardware ni genera archivos de configuración (bitstreams) para tarjetas FPGA reales.
- El soporte de HDL se limita a VHDL en su versión inicial; Verilog se contempla como extensión futura.
- La simulación se enfoca en el nivel comportamental (behavioral) y RTL, sin simulación a nivel de compuertas con retardos físicos.
- La aplicación opera exclusivamente en entorno local, sin componentes de servidor ni almacenamiento en la nube.
- La estimación de recursos FPGA es aproximada y no sustituye las herramientas de síntesis de los fabricantes.

1.9. Justificación

1.9.1. Justificación Técnica

Las herramientas actuales para la simulación de HDL presentan una fragmentación significativa en su flujo de trabajo. Un estudiante que utilice herramientas de código abierto debe configurar por separado un compilador (GHDL), un visualizador de señales (GTKWave) y un editor de texto genérico [6], [7]. Las soluciones comerciales como ModelSim integran estas funcionalidades pero dependen de licencias restrictivas y están vinculadas a familias de hardware específicas [8]. Kmila aborda esta fragmentación al ofrecer un entorno unificado donde edición, configuración y visualización coexisten en una sola aplicación multiplataforma sin requerir dependencias externas en la versión web [23], [24].

1.9.2. Justificación Educativa

La formación en diseño digital requiere que los estudiantes no solo escriban código HDL, sino que comprendan el comportamiento temporal de las señales que describen [25]. Sin una herramienta accesible para visualizar y experimentar con estos conceptos, la enseñanza queda condicionada al tiempo de acceso al laboratorio. Kmila funciona como herramienta de pre-laboratorio: permite a los estudiantes preparar y verificar sus diseños antes de la sesión con hardware real, optimizando el tiempo presencial y fortaleciendo la comprensión individual.

1.9.3. Justificación Académica

En la revisión de Trabajos Terminales de la ESCOM, no se identificó ningún proyecto previo dedicado a la creación de un entorno integrado de depuración de HDL [9]. El presente proyecto contribuye a la línea institucional de herramientas educativas para diseño digital, generando un recurso de código abierto que puede ser adoptado y extendido por la comunidad académica de ESCOM e IPN.

1.9.4. Justificación Económica

El costo de equipar un laboratorio con tarjetas FPGA individuales para un grupo de 30 estudiantes supera los $55,000 MXN considerando únicamente el hardware [1], [2]. Kmila se distribuye de forma gratuita y no requiere hardware adicional: cualquier computadora o dispositivo móvil con navegador web puede ejecutar la aplicación, reduciendo la barrera de entrada para la práctica individual y complementando la inversión institucional en infraestructura de laboratorio.

1.10. Factibilidad

1.10.1. Factibilidad Técnica

El desarrollo de Kmila se sustenta en tecnologías maduras, ampliamente documentadas y de uso libre. La Tabla 1.2 presenta el stack tecnológico seleccionado para el proyecto.

Tabla 1.2: *Stack tecnológico del proyecto Kmila.*

Componente	Tecnología	Versión	Justificación
Framework nativo	.NET MAUI	9.0	Aplicaciones nativas para Windows, macOS, Android e iOS desde un único código base [23]
Framework web	Blazor WebAssembly	9.0	Ejecución en navegador sin servidor, cobertura de Linux [24]
Lenguaje principal	C#	13.0	Ecosistema unificado con .NET, tipado fuerte, orientado a objetos
Editor de código	Monaco Editor (BlazorMonaco)	3.3.0	Motor del editor Visual Studio Code, resaltado de sintaxis y autocompletado
Visualización de ondas	SVG en línea (nativo)	—	Trazas escalonadas interactivas para representación de formas de onda
Persistencia local	SQLite (sqlite-net-pcl)	1.9.172	Base de datos embebida, sin servidor, multiplataforma
Motor de simulación	Motor propio en C#	—	Control total del ciclo de simulación HDL (alcance de TT2)

Todas las tecnologías listadas son de código abierto o cuentan con licencias gratuitas para uso académico. El entorno de desarrollo requiere únicamente el SDK de .NET 9 (gratuito), un editor de código como Visual Studio Code o Visual Studio Community (ambos gratuitos), y un dispositivo para pruebas. No se requiere infraestructura de servidor ni servicios en la nube.

1.10.2. Factibilidad Económica

1.10.2.1. Metodología de estimación de costos

El análisis económico del proyecto se estructura mediante tres enfoques complementarios. La Cost Breakdown Structure (CBS) permite descomponer el costo total en categorías identificables (recursos humanos, equipo, licencias y distribución), facilitando la trazabilidad de cada partida. El Total Cost of Ownership (TCO) extiende el análisis al considerar no solo el desarrollo inicial sino también los costos recurrentes asociados a la distribución y mantenimiento del software. Finalmente, se realiza un análisis costo-beneficio en contexto educativo para contrastar la inversión requerida con el valor que la solución aporta frente a las alternativas existentes.

1.10.2.2. Estimación del esfuerzo de desarrollo

El costo principal del proyecto se deriva del esfuerzo de desarrollo, estimado en horas de trabajo. El proyecto abarca 10 meses de esfuerzo de desarrollo dentro del calendario de febrero a diciembre de 2026, con una dedicación aproximada de 3.5 horas diarias, 6 días por semana. La Tabla 1.3 desglosa la estimación por módulo y fase.

Tabla 1.3: *Estimación de horas de desarrollo por módulo.*

Fase	Módulo	Horas estimadas
TT1	Investigación, documentación y protocolo	80
TT1	Arquitectura y diseño del sistema	60
TT1	Interfaz de usuario (Monaco Editor, pestañas, explorador de archivos)	100
TT1	Gestión de proyectos y persistencia (CRUD, SQLite)	60
TT1	Catálogo FPGA (carga JSON, 34 dispositivos)	40
TT1	Internacionalización (7 idiomas, soporte RTL)	40
TT1	Configuración (temas, eventos reactivos, .k-config)	30
TT1	Visor de formas de onda (SVG en línea)	30
TT1	Pruebas de integración y ajustes	20
	Subtotal TT1	**460**
TT2	Analizador léxico y sintáctico (Parser VHDL)	120
TT2	Generación de AST y modelo intermedio	80
TT2	Motor de simulación (DeltaCycleEngine)	100
TT2	Depuración paso a paso (TimeMachine)	60
TT2	Construcción de estímulos (StimulusBuilder)	40
TT2	Integración motor-UI y visualización de resultados	50
TT2	Pruebas, depuración y optimización	60
TT2	Documentación de TT2	40
	Subtotal TT2	**550**
	Total del proyecto	**1,010**

1.10.2.3. Desglose de costos del proyecto

La Tabla 1.4 presenta el desglose conforme a la CBS del proyecto. Los costos se organizan en cinco categorías: recursos humanos (valuados a una tarifa de referencia de $200 MXN/hora para un desarrollador junior en el mercado mexicano), equipo de desarrollo, equipo de pruebas, licencias de software y costos de distribución. Los costos de equipo se prorratean a 10 meses sobre una vida útil estimada de 48 meses.

Tabla 1.4: *Desglose de costos del proyecto.*

Categoría	Concepto	Costo estimado
Recursos humanos	1,010 horas de desarrollo ($\times$ $200 MXN/hr)	$202,000 MXN
Equipo de desarrollo	Laptop Lenovo IdeaPad L340 — prorrateo 10/48 meses sobre valor de $15,000 MXN	$3,125 MXN
Equipo de pruebas	Smartphone POCO F5 Pro (Android) — prorrateo 10/48 meses sobre valor de $8,500 MXN	$1,771 MXN
Licencias de software	.NET 9 SDK, Visual Studio Code, Git, Android SDK	$0 MXN
Distribución	Google Play Store — registro de desarrollador (pago único)	$500 MXN
Distribución	Apple Developer Program (suscripción anual)	$2,000 MXN
Distribución	Dominio web (registro anual)	$250 MXN
Distribución	Hosting web — Netlify (nivel gratuito)	$0 MXN
Distribución	Repositorio de código — GitHub (público, gratuito)	$0 MXN
	Costo total estimado (TCO)	**$209,646 MXN**

1.10.2.4. Costos reales de implementación

El costo predominante es el recurso humano (96.3 %), lo cual es característico de proyectos de software cuyas herramientas de desarrollo son de código abierto. En el contexto académico del Trabajo Terminal, este costo constituye una estimación teórica que permite dimensionar el valor del esfuerzo invertido, pero no representa un desembolso monetario real dado que el desarrollo se realiza como parte de la formación del estudiante. Los costos efectivos del proyecto, es decir, aquellos que requieren un desembolso real para la publicación y distribución del software, ascienden a $2,750 MXN, distribuidos en el registro de las tiendas de aplicaciones ($2,500 MXN) y el registro de dominio web ($250 MXN).

1.10.2.5. Beneficio económico de la solución

Desde la perspectiva del análisis costo-beneficio, Kmila aborda directamente el problema económico identificado en la Sección 1.5: el costo de equipar un laboratorio con tarjetas FPGA individuales para un grupo de 30 estudiantes supera los $55,000 MXN considerando únicamente el hardware [1], [2], y las herramientas comerciales como ModelSim requieren licencias cuyo costo las hace inaccesibles para uso individual [8]. Kmila se distribuye de forma gratuita, se ejecuta en cualquier computadora o dispositivo móvil sin requerir hardware especializado (FPGA), y su código fuente permanece disponible en un repositorio público. Esto permite que cada estudiante acceda a un entorno de simulación completo sin costo alguno, reduciendo la dependencia del hardware especializado del laboratorio.

1.10.2.6. Conclusión de viabilidad económica

El proyecto es económicamente viable debido a que los costos operativos son prácticamente nulos (todas las tecnologías de desarrollo son de código abierto), el desembolso real para distribución es de $2,750 MXN, y la solución elimina las barreras económicas que actualmente limitan el acceso de los estudiantes a herramientas de simulación de circuitos digitales.

1.10.3. Factibilidad Operativa

El proyecto es desarrollado por un alumno (Ulrich Tamayo Daniel) bajo la supervisión de dos directores: M. en C. Pastrana Fernández Carlos Jesús y Dr. Cifuentes Álvarez Alejandro Sigrido. El cronograma se divide en dos fases: TT1 (febrero–junio 2026) y TT2 (julio–diciembre 2026), conforme al calendario académico de la ESCOM y al protocolo registrado TT 2026-B059.

El desarrollo se realiza de forma local en el equipo de cómputo personal del alumno, sin requerir permisos institucionales especiales, acceso a servidores, ni infraestructura adicional. Las pruebas de la versión web se ejecutan en un navegador estándar sobre Linux, y las pruebas de la versión nativa se realizan en el mismo equipo de desarrollo.

No se identifican dependencias externas que pongan en riesgo la viabilidad operativa del proyecto. El código fuente se gestiona mediante Git con respaldo en repositorio remoto, lo que garantiza la trazabilidad y recuperación ante pérdida de datos.

Capítulo 2

Marco Teórico

2.1. Lenguajes de Descripción de Hardware (HDL)

2.1.1. Historia y Evolución

Los lenguajes de descripción de hardware (HDL, por sus siglas en inglés) surgen como respuesta a la creciente complejidad del diseño de circuitos integrados. Durante las décadas de 1960 y 1970, los primeros esfuerzos formales incluyeron lenguajes como ISP (Instruction Set Processor), propuesto por Bell y Newell en 1971, que introdujo el concepto de nivel de transferencia de registros (RTL) para describir el comportamiento de procesadores [26]. Sin embargo, el diseño de hardware continuó dominado por la captura esquemática manual hasta mediados de la década de 1980.

El punto de inflexión se produjo con dos desarrollos paralelos. Por un lado, el Departamento de Defensa de los Estados Unidos, a través de su programa VHSIC (Very High Speed Integrated Circuit), financió en 1983 la creación de VHDL como un lenguaje estándar para documentar y simular diseños de circuitos integrados [10], [11]. Por otro lado, en 1984, Phil Moorby y Prabhu Goel crearon Verilog en Gateway Design Automation como una herramienta propietaria de simulación con sintaxis inspirada en el lenguaje C [12], [13].

La introducción de la síntesis lógica a finales de la década de 1980, que permitió compilar código HDL en netlists a nivel de compuertas, transformó radicalmente la industria. Antes de 1990, la mayoría de los diseños se iniciaban mediante captura esquemática; para mediados de los años noventa, aproximadamente la mitad de los nuevos diseños utilizaban HDLs [26]. En la actualidad, prácticamente todos los diseños digitales de complejidad media o alta emplean métodos basados en HDL.

La evolución continuó con SystemVerilog (IEEE Std 1800), publicado inicialmente en 2005 y fusionado con el estándar Verilog en 2009, que incorporó capacidades de verificación orientada a objetos, restricciones aleatorias y aserciones [27]. En paralelo, VHDL mantuvo su evolución con revisiones en 2008 y 2019 que integraron mejoras como genéricos en paquetes y soporte completo de PSL (Property Specification Language) [10].

En cuanto a la adopción regional, estudios de la industria indican que VHDL predomina en Europa, sectores aeroespaciales y de defensa, y en la academia latinoamericana, mientras que Verilog y SystemVerilog dominan en la industria de semiconductores de Estados Unidos y Asia [28]. En el contexto del diseño con FPGAs, VHDL se mantiene como el lenguaje más utilizado a nivel global según encuestas recientes del Wilson Research Group [28].

2.1.2. VHDL

VHDL (VHSIC Hardware Description Language) fue desarrollado entre 1983 y 1985 bajo contrato del Departamento de Defensa de los Estados Unidos (contrato F33615-83-C-1003), adjudicado a un equipo conformado por Intermetrics, Inc. como contratista principal, Texas Instruments como experto en diseño de circuitos integrados, e IBM como especialista en diseño de sistemas computacionales [11]. El lenguaje fue fuertemente influenciado por Ada, del cual heredó su sintaxis verbosa, su sistema de tipos estricto y sus mecanismos de modularidad.

La primera versión pública (VHDL 7.2) se liberó en 1985, y en 1987 fue aprobado como estándar IEEE bajo la designación IEEE Std 1076-1987. Desde entonces, el estándar ha sido revisado en múltiples ocasiones, como se muestra en la Tabla 2.1.

Tabla 2.1: *Revisiones del estándar IEEE 1076 (VHDL).*

Estándar	Año	Cambios principales
IEEE 1076-1987	1987	Primera estandarización; definición base del lenguaje
IEEE 1076-1993	1993	Revisión mayor: caracteres ISO-8859-1, operador xnor, sintaxis más consistente
IEEE 1076-2000	2000	Tipos protegidos para control de concurrencia en variables compartidas
IEEE 1076-2008	2008	Revisión mayor: subconjunto PSL, genéricos en paquetes y subprogramas; absorbe IEEE 1164 y 1076.3
IEEE 1076-2019	2019	Revisión vigente: integración completa de PSL, mejoras diversas

Las características fundamentales de VHDL incluyen:

- **Tipado fuerte:** el compilador verifica estrictamente la compatibilidad de tipos, previniendo errores comunes en tiempo de compilación.
- **Modelo de ejecución concurrente:** las sentencias dentro de una arquitectura se ejecutan en paralelo, reflejando el comportamiento real del hardware.
- **Paquetes y bibliotecas:** permiten encapsular declaraciones reutilizables (tipos, funciones, constantes) para su uso en múltiples diseños.
- **Genéricos:** habilitan diseños parametrizados (por ejemplo, el ancho de un bus como parámetro configurable).
- **Configuraciones:** permiten vincular entidades con arquitecturas y componentes con implementaciones sin modificar el código fuente.

Los elementos estructurales básicos de VHDL son la *entity* (interfaz externa del módulo), la *architecture* (descripción interna del comportamiento o estructura), el *process* (bloque de ejecución secuencial activado por una lista de sensibilidad), la *signal* (representación de conexiones físicas entre componentes) y el *component* (mecanismo de instanciación jerárquica) [11], [29].

La preferencia por VHDL en la academia europea y latinoamericana se explica por factores pedagógicos e históricos: su tipado fuerte impone disciplina en el modelado de hardware, su sintaxis explícita favorece la comprensión en entornos de aprendizaje, y la tradición académica europea, donde VHDL fue adoptado tempranamente, se extendió a las instituciones latinoamericanas que siguen modelos curriculares similares [25].

2.1.3. Verilog

Verilog fue creado entre finales de 1983 e inicios de 1984 por Phil Moorby, Prabhu Goel y Chi-Lai Huang en Automated Integrated Design Systems, empresa posteriormente renombrada como Gateway Design Automation en 1985 [13]. Moorby es reconocido como el diseñador principal del lenguaje y también desarrolló Verilog-XL, el primer simulador de referencia industrial para Verilog.

Inicialmente un lenguaje propietario, Verilog dio un giro decisivo cuando Cadence Design Systems adquirió Gateway Design Automation en noviembre de 1989. En 1990, Cadence liberó la especificación del lenguaje al dominio público, lo que condujo a la formación de Open Verilog International (OVI) y, eventualmente, a su estandarización por IEEE como IEEE Std 1364-1995 [12]. La Tabla 2.2 resume las revisiones del estándar.

Tabla 2.2: *Revisiones del estándar IEEE 1364 (Verilog) e IEEE 1800 (SystemVerilog).*

Estándar	Año	Descripción
IEEE 1364-1995	1995	Primera estandarización (Verilog-95)
IEEE 1364-2001	2001	Revisión mayor: sentencias generate, declaración de puertos mejorada, aritmética con signo
IEEE 1364-2005	2005	Último estándar independiente de Verilog
IEEE 1800-2005	2005	Primer estándar de SystemVerilog
IEEE 1800-2009	2009	Fusión de Verilog (1364) con SystemVerilog; fin del estándar Verilog independiente
IEEE 1800-2017	2017	Correcciones y clarificaciones

Las características principales de Verilog incluyen:

- **Sintaxis similar a C:** resulta familiar para ingenieros de software, con una notación concisa y menos verbosa que VHDL.
- **Tipado débil:** un sistema de tipos más permisivo con conversiones implícitas, lo que acelera la escritura pero puede favorecer errores sutiles.
- **Lógica de cuatro valores:** los valores 0, 1, x (desconocido) y z (alta impedancia) están integrados en el lenguaje base.
- **Modelado comportamental y estructural:** soporta tanto bloques `always` (descripción de comportamiento) como instanciación de compuertas (descripción estructural).
- **Directivas de preprocesador:** similares a las de C (`` `define ``, `` `include ``, `` `ifdef ``).

SystemVerilog, su sucesor, se originó a partir del lenguaje Superlog desarrollado por Co-Design Automation en 1997 y de las capacidades de verificación de OpenVera de Synopsys [27]. Definido como un "lenguaje unificado de diseño, especificación y verificación de hardware," SystemVerilog añade programación orientada a objetos, verificación con restricciones aleatorias, aserciones (SVA) y cobertura funcional, convirtiéndose en el estándar dominante para verificación de ASICs modernos.

El dominio de Verilog y SystemVerilog en la industria estadounidense y asiática se atribuye a sus raíces en Silicon Valley (donde Gateway Design Automation operaba), a la posición de mercado de Cadence con Verilog-XL, y a la adopción por parte de las fundiciones y casas de diseño asiáticas que operan como proveedores de la industria

fabless norteamericana [13], [28].

2.1.4. Comparativa VHDL vs. Verilog

La Tabla 2.3 presenta una comparativa de las características principales de ambos lenguajes.

Tabla 2.3: *Comparativa entre VHDL y Verilog.*

Característica	**VHDL**	**Verilog**
Origen	DoD (EE.UU.), 1983	Gateway Design Automation, 1984
Estándar IEEE	1076 (1987–2019)	1364 (1995–2005), fusionado en 1800
Influencia sintáctica	Ada	C
Tipado	Fuertemente tipado	Débilmente tipado
Verbosidad	Alta	Baja
Modelo de ejecución	Concurrente	Concurrente
Lógica multivaluada	Vía paquete IEEE 1164 (`std_logic`: 9 valores)	Nativo (4 valores: 0, 1, x, z)
Verificación avanzada	Limitada (PSL externo)	SystemVerilog (UVM, SVA, cobertura)
Predominio académico	Europa, Latinoamérica	EE.UU., Asia
Predominio industrial	Aeroespacial, defensa, FPGA	Semiconductores, ASIC

Ambos lenguajes son funcionalmente equivalentes para la descripción de hardware digital y comparten el paradigma de ejecución concurrente. La elección entre uno u otro depende principalmente del contexto institucional, regional y del dominio de aplicación.

2.1.5. Otros Lenguajes de Descripción de Hardware

Aunque VHDL y Verilog (junto con su extensión SystemVerilog) concentran la práctica totalidad del diseño profesional de hardware digital, en las últimas dos décadas han surgido diversos lenguajes de más alto nivel orientados a mitigar la verbosidad y la rigidez sintáctica de los HDL clásicos. A continuación se describen los más relevantes en la literatura académica e industrial contemporánea.

SystemC no es estrictamente un HDL, sino una biblioteca de C++ con soporte para modelado concurrente de hardware a nivel de sistema (ESL, *Electronic System Level*). Fue introducido por la Open SystemC Initiative en 1999 y estandarizado como IEEE 1666 (última revisión en 2023) [30]. Su caso de uso principal es el modelado arquitectónico de SoCs y aceleradores en fases tempranas de diseño, donde la velocidad de simulación y la interoperabilidad con código C++ son prioritarias. SystemC no es sintetizable de forma directa: para obtener un *netlist* es necesario traducir los modelos a RTL en VHDL o Verilog mediante herramientas de síntesis de alto nivel (HLS).

MyHDL, creado por Jan Decaluwe en 2003, es una biblioteca de Python que permite describir hardware digital empleando la sintaxis del lenguaje. MyHDL incluye un simulador propio y la capacidad de convertir los modelos a Verilog y VHDL para su síntesis posterior [31]. Su adopción se ha mantenido circunscrita a proyectos académicos y a comunidades de hardware abierto.

Amaranth HDL (originalmente nMigen), surgido en 2019 en el ecosistema del proyecto SymbiFlow y mantenido actualmente como proyecto independiente, es una evolución conceptual de MyHDL construida sobre Python moderno. A diferencia de MyHDL, Amaranth integra de forma nativa el manejo de dominios de reloj, transiciones síncronas explícitas y un subsistema de verificación formal [32]. Amaranth genera Verilog como formato de salida para síntesis con herramientas como Yosys [17], consolidándose como el lenguaje preferido en la comunidad de hardware libre y en proyectos abiertos basados en FPGAs Lattice.

Chisel (*Constructing Hardware in a Scala Embedded Language*), desarrollado por el equipo de UC Berkeley liderado por Krste Asanović desde 2012, es un lenguaje de dominio específico (DSL) embebido en Scala diseñado para construir generadores de hardware parametrizables [33]. Chisel se ha consolidado como el lenguaje de referencia en el ecosistema RISC-V y es la base del procesador Rocket Chip. Genera Verilog como salida mediante su representación intermedia FIRRTL.

SpinalHDL, de Charles Papon, es otro DSL embebido en Scala aparecido en 2015, con enfoque en la legibilidad sintáctica y la facilidad de integración con proyectos Java [34]. Compila tanto a Verilog como a VHDL, y privilegia una descripción estructural explícita del circuito.

Bluespec SystemVerilog (BSV), desarrollado por Arvind en el MIT y comercializado desde 2000, adopta un paradigma de *reglas guardadas* (*guarded atomic actions*) que describe el comportamiento del hardware en términos de transiciones atómicas con condiciones de habilitación [35]. Esta semántica se traduce internamente a Verilog sintetizable. BSV ha tenido mayor penetración en el ámbito académico y en diseños de aceleradores de alto rendimiento que en la industria de propósito general.

Es importante destacar un patrón común en todas estas alternativas: ninguna reemplaza a VHDL ni a Verilog en el flujo de síntesis. Todas ellas generan, como formato final, código RTL en VHDL o Verilog que posteriormen-

te es procesado por las herramientas de síntesis convencionales de los fabricantes de FPGA. En consecuencia, el conocimiento de VHDL o Verilog permanece como requisito fundamental para el diseño digital profesional, independientemente de la capa de abstracción adicional que se emplee durante el desarrollo.

2.1.6. Justificación de la Elección de VHDL para Kmila

La selección de VHDL como lenguaje primario soportado por Kmila, con Verilog contemplado como extensión prevista para TT2, se fundamenta en cinco consideraciones alineadas con el propósito pedagógico del proyecto:

Cobertura curricular. Los planes de estudio de Ingeniería en Sistemas Computacionales de la ESCOM incorporan VHDL como lenguaje primario en las asignaturas de Fundamentos de Diseño Digital, Arquitectura de Computadoras, Diseño de Sistemas Digitales y Sistemas en Chip. El mismo patrón se observa en la mayor parte de las instituciones latinoamericanas y europeas, como se documentó en la Sección 2.1.1. Kmila está orientado explícitamente al estudiante en formación inicial dentro de este contexto institucional.

Dominancia industrial y vigencia. De acuerdo con las encuestas periódicas del Wilson Research Group [28], VHDL se mantiene como el lenguaje de mayor adopción en diseño con FPGA a nivel global, mientras que Verilog y SystemVerilog dominan el segmento ASIC. Ambos lenguajes cuentan con estándares IEEE vigentes (1076-2019 para VHDL; 1800-2017 para SystemVerilog) y respaldo institucional de los tres fabricantes de FPGA tratados en la Sección 2.3.

Dependencia de los lenguajes de alto nivel. Como se estableció en la Sección 2.1.5, los lenguajes alternativos (SystemC, MyHDL, Amaranth, Chisel, SpinalHDL, Bluespec) no reemplazan a VHDL ni a Verilog: su flujo de síntesis desemboca invariablemente en código RTL en uno de estos dos lenguajes. Enseñar únicamente un lenguaje de alto nivel dejaría al estudiante sin las bases necesarias para depurar la salida sintetizada, interpretar un *netlist* o integrar módulos en proyectos heterogéneos donde coexisten componentes escritos en VHDL o Verilog por terceros.

Madurez del ecosistema. VHDL y Verilog disponen de un ecosistema consolidado de simuladores (GHDL, Icarus Verilog, ModelSim, Verilator), visores de formas de onda (GTKWave) y herramientas de análisis estático. Este ecosistema facilita la implementación del simulador de Kmila, ya que permite adoptar formatos estándar (por ejemplo, VCD para el visor de formas de onda) y validar los resultados del simulador propio frente a herramientas de referencia ampliamente utilizadas [6], [7].

Nivel de abstracción apropiado al propósito pedagógico. El objetivo de Kmila, como se detalla en la Sección 2.4, es reducir la barrera de entrada al aprendizaje del diseño digital. El nivel de abstracción adecuado para un primer contacto con HDL es el nivel comportamental y RTL, donde el estudiante puede observar directamente la correspondencia entre las estructuras sintácticas del lenguaje y los elementos físicos del circuito (señales, registros, procesos). Los lenguajes de más alto nivel (Chisel, Amaranth, SystemC) introducen capas de abstracción que ocultan esta correspondencia y resultan más apropiados para etapas posteriores de formación.

En consecuencia, la versión actual de Kmila implementa soporte completo para VHDL como lenguaje primario, contempla Verilog como extensión prevista para TT2 y descarta explícitamente del alcance del proyecto el soporte nativo para lenguajes de alto nivel, sin perjuicio de que los archivos generados por dichos lenguajes (Verilog o

VHDL sintetizable) puedan analizarse en Kmila una vez integrado el soporte para Verilog.

2.2. Simulación Digital

Esta sección presenta los fundamentos teóricos de la simulación de circuitos digitales descritos en HDL, abarcando el modelo de eventos discretos, la semántica de señales y procesos, la temporización basada en ciclos de reloj y los niveles de abstracción empleados en el flujo de diseño.

2.2.1. Modelo de Eventos Discretos

La simulación de circuitos digitales descritos en HDL se fundamenta en el modelo de simulación dirigida por eventos discretos (DESM, Discrete Event Simulation Model), formalizado en el estándar IEEE 1076 para VHDL y en IEEE 1364/1800 para Verilog y SystemVerilog [10], [12]. A diferencia de la simulación analógica, donde las magnitudes varían de forma continua en el tiempo, la simulación digital opera sobre un conjunto finito de valores lógicos que cambian en instantes discretos.

El principio fundamental del modelo es que el simulador mantiene una cola de eventos ordenados cronológicamente. Un evento se define como un cambio de valor en una señal en un instante de tiempo determinado. En cada paso de simulación, el simulador extrae los eventos correspondientes al tiempo actual, actualiza las señales afectadas y evalúa los procesos sensibles a dichas señales. Si la evaluación produce nuevos cambios, estos se insertan en la cola como eventos futuros [29].

Un concepto central en la simulación VHDL es el **ciclo delta**. Cuando una asignación de señal se ejecuta sin un retardo explícito (es decir, con retardo cero), el nuevo valor no se aplica inmediatamente sino que queda programado como pendiente. Solo al finalizar el ciclo delta actual se aplican todos los valores pendientes de forma simultánea. Este paso infinitesimal de tiempo preserva el orden causal sin avanzar el tiempo de simulación. Dentro de un mismo instante de tiempo pueden ocurrir múltiples ciclos delta sucesivos hasta que el sistema alcanza un estado estable, es decir, hasta que ningún proceso genera nuevos eventos [11], [29].

Este mecanismo es la clave del determinismo en la simulación concurrente. Dado que los nuevos valores de las señales no se hacen visibles hasta el final del ciclo delta, todos los procesos que se ejecutan dentro de un mismo ciclo leen los mismos valores (los del ciclo anterior), independientemente del orden en que el simulador los evalúe internamente. Así, el resultado de la simulación es siempre el mismo sin importar el orden de ejecución de los procesos, análogamente a lo que ocurre en hardware real, donde todas las compuertas operan simultáneamente sobre las señales presentes en sus entradas.

Para ilustrar el mecanismo, considérese un diseño con tres procesos (A, B y C), donde A modifica una señal presente en la lista de sensibilidad de B, y B modifica una señal sensible de C:

1. **Ciclo delta 1:** los tres procesos se activan y ejecutan su lógica interna, leyendo los valores del estado inicial. A genera una asignación pendiente sobre una señal que B observa; B genera una asignación sobre una señal que C observa. Al finalizar el ciclo, se aplican todos los valores pendientes.
2. **Ciclo delta 2:** solo B y C se reactivan, porque las señales en sus listas de sensibilidad cambiaron. A permanece suspendido, ya que ninguna de sus señales sensibles fue modificada. B y C ejecutan y generan nuevas asignaciones. Se aplican los cambios.

3. **Ciclos delta sucesivos:** en cada ciclo, únicamente los procesos cuyas señales sensibles hayan cambiado se reactivan. El resto permanece suspendido.
4. **Estado estable:** cuando ningún proceso genera nuevos eventos al finalizar un ciclo delta, el sistema ha alcanzado un estado estable para el instante de tiempo actual. Solo entonces el simulador avanza al siguiente instante de tiempo con eventos programados (por ejemplo, el siguiente flanco de reloj).

El flujo de ejecución de cada ciclo delta sigue tres fases:

1. **Actualización de señales:** se aplican los valores programados durante el ciclo delta anterior (o los valores iniciales, en el primer ciclo).
2. **Evaluación de procesos:** se ejecutan todos los procesos cuyas señales de la lista de sensibilidad hayan cambiado de valor. Los procesos cuyas señales no fueron modificadas permanecen suspendidos.
3. **Programación de eventos:** las asignaciones realizadas por los procesos generan nuevos eventos, ya sea con retardo explícito (avanzan el tiempo de simulación) o con retardo delta (permanecen en el mismo instante).

Este ciclo de tres fases se repite hasta que no quedan eventos pendientes en el tiempo actual, momento en que el simulador avanza al siguiente instante con eventos programados [10].

2.2.2. Señales y Procesos

En el modelo de ejecución HDL existen dos paradigmas fundamentales de asignación: la asignación concurrente y la asignación secuencial. Comprender su diferencia es esencial para interpretar correctamente los resultados de simulación.

Las **señales** (*signals* en VHDL, *wires* y *regs* en Verilog) representan conexiones físicas entre componentes del circuito. En VHDL, las señales se actualizan al finalizar el ciclo delta actual, no en el momento de la asignación. Esto implica que múltiples asignaciones a la misma señal dentro de un mismo proceso resultan en que solo la última asignación tenga efecto, ya que las anteriores son sobrescritas antes de aplicarse [11].

Las **variables** (*variables* en VHDL) son elementos de almacenamiento local dentro de un proceso que se actualizan inmediatamente al ejecutar la asignación. A diferencia de las señales, las variables no tienen representación física directa en el hardware sintetizado y su uso principal es como almacenamiento temporal para cálculos intermedios dentro de un proceso [29].

Un **proceso** (*process* en VHDL, bloque *always* en Verilog) es la unidad fundamental de ejecución secuencial. Cada proceso se activa cuando alguna señal de su lista de sensibilidad cambia de valor. Una vez activado, el cuerpo del proceso se ejecuta secuencialmente de principio a fin, y al terminar, el proceso se suspende hasta la siguiente activación. Aunque la ejecución interna de un proceso es secuencial, todos los procesos de un diseño se ejecutan de forma concurrente entre sí, reflejando el paralelismo inherente del hardware [10].

Las **sentencias concurrentes** en VHDL (asignaciones de señal fuera de un proceso, instanciaciones de componentes, sentencias *generate*) son equivalentes a procesos implícitos con listas de sensibilidad derivadas automáticamente de las señales leídas en la expresión. Por ejemplo, la asignación concurrente `y <= a and b;` equivale a un proceso sensible a las señales `a` y `b` [11].

Esta distinción entre concurrencia y secuencialidad tiene implicaciones directas en la simulación: el simulador debe garantizar que el resultado sea independiente del orden en que se evalúan los procesos concurrentes, lo cual se logra mediante el mecanismo de ciclos delta descrito anteriormente.

2.2.3. Ciclos de Reloj y Temporización

En la simulación de circuitos digitales síncronos, la señal de reloj es el elemento temporal fundamental que gobierna la evolución del sistema. Un circuito síncrono organiza sus operaciones en torno a los flancos (transiciones) de una señal de reloj periódica, de modo que los elementos de memoria (flip-flops, registros) capturan nuevos valores únicamente en el instante del flanco activo [29].

La **resolución temporal** del simulador define la granularidad mínima con la que se representan los instantes de tiempo. En VHDL, la resolución base es el femtosegundo (fs, 10^{-15} s), aunque en la práctica las simulaciones emplean unidades mayores como nanosegundos (ns) o picosegundos (ps). En Verilog, la resolución se establece mediante la directiva `` `timescale ``, que especifica la unidad de tiempo y la precisión (por ejemplo, `` `timescale 1ns/1ps ``) [12].

El modelado de retardos en simulación se clasifica en tres categorías:

- **Retardo inercial** (*inertial delay*): modelo por defecto en VHDL, donde un pulso más corto que el retardo especificado es filtrado (absorbido) por el elemento. Este comportamiento modela la inercia eléctrica real de las compuertas, que requieren un tiempo mínimo para conmutar.
- **Retardo de transporte** (*transport delay*): modela una línea de transmisión ideal que propaga cualquier pulso, independientemente de su duración, con un retardo fijo. En VHDL se indica con la palabra reservada `transport`.
- **Retardo delta** (*delta delay*): retardo infinitesimal sin avance del tiempo de simulación, utilizado para resolver dependencias causales entre asignaciones concurrentes, como se describió en la Sección 2.2.1.

En un testbench típico, la señal de reloj se genera mediante un proceso que alterna el valor de la señal con retardos explícitos. La frecuencia del reloj define el periodo, y la relación entre el tiempo en alto y el periodo total determina el ciclo de trabajo (*duty cycle*). Los estímulos de entrada se sincronizan típicamente con los flancos de reloj mediante sentencias `wait until` (VHDL) o `@(posedge clk)` (Verilog), garantizando que el circuito bajo prueba reciba datos estables respecto a los tiempos de setup y hold de los elementos secuenciales [11].

2.2.4. Niveles de Abstracción

El diseño y la simulación de hardware digital operan en múltiples niveles de abstracción, cada uno con un grado diferente de detalle y un propósito distinto en el flujo de desarrollo. Estos niveles, de mayor a menor abstracción, son:

Nivel comportamental (*behavioral*). Describe la funcionalidad del circuito en términos algorítmicos, sin referencia a la estructura de hardware subyacente. En VHDL, este nivel se expresa mediante procesos con sentencias secuenciales (if-then-else, case, loops) y asignaciones de señal con retardos explícitos. La simulación comportamental verifica que el algoritmo produce los resultados correctos, sin considerar las restricciones de temporización del dispositivo físico. Es el nivel adecuado para la fase inicial de diseño y para la especificación de testbenches [29].

Nivel de transferencia de registros (*Register Transfer Level*, RTL). Describe el circuito en términos de registros (elementos de almacenamiento) y la lógica combinacional que transforma datos entre ellos en cada ciclo de reloj. Este es el nivel de abstracción principal para la síntesis: las herramientas de síntesis RTL traducen la descripción en HDL a una netlist de compuertas lógicas. En VHDL, el nivel RTL se caracteriza por el uso de procesos sensibles a flancos de reloj para lógica secuencial y asignaciones concurrentes o procesos combinacionales para la lógica entre registros. La simulación RTL verifica la funcionalidad ciclo a ciclo del diseño sintetizable [26].

Nivel de compuertas (*gate level*). Describe el circuito como una interconexión de compuertas lógicas primitivas (AND, OR, NOT, flip-flops) provenientes de una biblioteca tecnológica específica. Este nivel resulta de la síntesis del código RTL y se utiliza para simulación post-síntesis, donde se incorporan los retardos reales de las compuertas y las interconexiones. La simulación a nivel de compuertas es significativamente más lenta que la simulación RTL o comportamental, pero permite verificar que el circuito sintetizado cumple con las restricciones temporales del dispositivo objetivo [29].

Para el presente proyecto, Kmila implementa simulación a nivel comportamental, que es el nivel de abstracción más relevante para el aprendizaje de HDL en entornos académicos. Este nivel permite al estudiante concentrarse en la lógica y el comportamiento del diseño sin la complejidad adicional de las restricciones tecnológicas de un dispositivo específico. La extensión a simulación RTL y post-síntesis se contempla como trabajo futuro.

2.3. Fabricantes y Dispositivos FPGA Soportados

La industria de dispositivos lógicos programables (FPGA) está dominada globalmente por tres fabricantes principales: AMD (antes Xilinx), Intel (a través de Altera) y Lattice Semiconductor. En conjunto, estos tres proveedores concentran la mayor parte del mercado mundial de FPGAs, tanto en segmentos de alto desempeño como en nichos de bajo consumo y aplicaciones educativas [36]. Kmila toma como universo de dispositivos soportados un catálogo representativo de las familias más empleadas en el ámbito académico y en proyectos de hardware de gama media, cubriendo los tres fabricantes mencionados.

2.3.1. Historia de los Fabricantes

2.3.1.1. AMD (Xilinx)

Xilinx fue fundada en 1984 en Silicon Valley por Ross Freeman, Bernard Vonderschmitt y James V. Barnett II, con la premisa de que el costo de los transistores decrecería lo suficiente como para justificar arquitecturas lógicas configurables por el usuario final [36]. En 1985 introdujo el *XC2064*, reconocido como el primer FPGA comercial, con apenas 64 bloques lógicos configurables (CLBs) y 85 000 transistores. Durante las décadas siguientes, la compañía consolidó la arquitectura basada en bloques configurables, interconexión segmentada y memoria de configuración SRAM, que se mantiene vigente en la mayoría de FPGAs actuales.

La evolución tecnológica se materializó en líneas diferenciadas: la familia **Spartan** (introducida en 1998) orientada al segmento de bajo costo y uso educativo; la familia **Virtex** (1998) para aplicaciones de alto desempeño; y la familia **Artix-7** (2010), perteneciente a la serie 7 junto con *Kintex-7* y *Virtex-7*, que consolidó el uso del nodo de fabricación de 28 nm. Posteriormente se introdujeron la generación *UltraScale* (2013) y la plataforma *Zynq* (2011), que integra un procesador ARM Cortex-A9 con lógica programable en el mismo chip [37].

En febrero de 2022, Advanced Micro Devices (AMD) completó la adquisición de Xilinx por aproximadamente 49 000 millones de dólares estadounidenses, en lo que constituyó la mayor operación de la industria de semiconductores hasta esa fecha. A partir de ese momento, la línea de FPGAs pasó a operar bajo el *AMD Adaptive Computing Group*, manteniendo la nomenclatura histórica de sus familias [5], [38].

2.3.1.2. Intel (Altera)

Altera Corporation fue fundada en 1983 en San José, California, por Robert Hartmann, James Sansbury, Paul Newhagen y Michael Magranet, inicialmente como fabricante de dispositivos lógicos programables borrables (EPLDs). Su primer producto comercial, el *EP300*, apareció en 1984 y consolidó a la empresa como pionera en el segmento de lógica reconfigurable no volátil [36]. Durante los años noventa, Altera consolidó las familias **MAX** (dispositivos lógicos complejos, CPLDs) y **FLEX**, esta última siendo su primera arquitectura propiamente FPGA basada en LUTs.

A partir del año 2002, Altera reorganizó su oferta en tres líneas diferenciadas por costo y desempeño: **Cyclone** (bajo costo, orientada a aplicaciones industriales y académicas), **Arria** (gama media con transceptores integrados) y **Stratix** (alto desempeño). La familia *Cyclone IV*, introducida en 2009 en tecnología de 60 nm y utilizada ampliamente en tarjetas de desarrollo educativas como la DE0-Nano y la DE2-115, es particularmente relevante para el presente proyecto [39].

En diciembre de 2015, Intel Corporation completó la adquisición de Altera por 16 700 millones de dólares estadounidenses, integrándola como *Programmable Solutions Group* (PSG). Posteriormente, en 2024, Intel anunció la separación del grupo en una subsidiaria independiente que recuperó el nombre *Altera Corporation*, con un proceso de desinversión gradual formalizado a lo largo de 2025 [40], [41]. Para efectos de este documento, se emplea la denominación *Intel (Altera)* por ser la vigente en la mayor parte de la literatura y en las herramientas de desarrollo (Intel Quartus Prime).

2.3.1.3. Lattice Semiconductor

Lattice Semiconductor Corporation fue fundada en 1983 en Hillsboro, Oregón, con un enfoque inicial en dispositivos GAL (*Generic Array Logic*) basados en tecnología E^2PROM, alternativa no volátil a los productos de Altera y Xilinx en el segmento de lógica programable. A lo largo de los años noventa y dos mil consolidó las familias **ispLSI** (CPLDs) y **MachXO** (FPGAs no volátiles de bajo consumo), diferenciándose estratégicamente del segmento de alto desempeño dominado por Xilinx y Altera para concentrarse en aplicaciones de bajo consumo, automoción, comunicaciones e IoT [36].

El evento más relevante en su portafolio reciente fue la adquisición de SiliconBlue Technologies en diciembre de 2011, operación mediante la cual Lattice incorporó la familia **iCE40**, caracterizada por su consumo extremadamente bajo (del orden de microvatios en modo reposo) y por su baja densidad, lo que la hace idónea para dispositivos portátiles, sensores y hardware educativo [42], [43]. En 2014 la compañía introdujo la familia **ECP5**, orientada a aplicaciones de gama media con transceptores SerDes integrados, y en años posteriores las familias *MachXO3*, *CrossLink* y *CrossLink-NX*.

Un aspecto distintivo de Lattice, particularmente relevante para el contexto académico y de software libre, es que las familias iCE40 y ECP5 cuentan con flujos de síntesis y *place-and-route* completamente abiertos (*Project IceStorm*, *nextpnr* y *Yosys*), lo que las convierte en la opción preferida en entornos de investigación, *hardware hacking* y educación con herramientas de código abierto [17], [44].

2.3.2. Catálogo de Dispositivos Soportados por Kmila

Kmila incorpora un catálogo de 34 dispositivos FPGA distribuidos entre los tres fabricantes descritos, seleccionados por su presencia en el ámbito académico, su disponibilidad comercial en México y su cobertura del espectro de capacidades (desde dispositivos de entrada con menos de 2 000 LUTs hasta dispositivos de gama media-alta con más de 200 000 celdas lógicas). La distribución por fabricante es la siguiente:

- **AMD (Xilinx):** 16 dispositivos, distribuidos en 8 modelos de la familia Spartan-6 y 8 modelos de la familia Artix-7.
- **Intel (Altera):** 9 dispositivos de la familia Cyclone IV E.
- **Lattice:** 9 dispositivos, distribuidos en 3 modelos iCE40 HX, 2 modelos iCE40 UltraPlus y 4 modelos ECP5.

Las Tablas 2.4, 2.5 y 2.6 presentan una síntesis de los recursos principales de cada dispositivo, enfocada en las métricas pedagógicamente más relevantes: unidades lógicas (LUTs, celdas lógicas o elementos lógicos, según la terminología de cada fabricante), registros (flip-flops), memoria interna en bloques BRAM (expresada en Kbits totales), bloques DSP, PLLs y número máximo de pines de entrada/salida de usuario. La especificación completa de cada dispositivo, incluyendo el tipo de bloque BRAM, el tipo de multiplicador DSP, la frecuencia máxima de PLL, el voltaje del núcleo y el número de bancos de I/O, se presenta en el Anexo A.4.

Tabla 2.4: *Dispositivos FPGA de AMD (Xilinx) soportados por Kmila.*

Modelo	Familia	Celdas lóg.	FFs	BRAM (Kb)	DSP	PLLs	I/O máx.
XC6SLX4	Spartan-6	3 840	4 800	216	8	2	132
XC6SLX9	Spartan-6	9 152	11 440	576	16	2	200
XC6SLX16	Spartan-6	14 579	18 224	576	32	2	232
XC6SLX25	Spartan-6	24 051	30 064	936	38	4	266
XC6SLX45	Spartan-6	43 661	54 576	2 088	116	4	358
XC6SLX75	Spartan-6	74 637	93 296	3 096	180	6	498
XC6SLX100	Spartan-6	101 261	126 576	4 824	180	6	498
XC6SLX150	Spartan-6	147 443	184 304	4 824	180	6	576
XC7A12T	Artix-7	12 800	16 000	720	40	5	170
XC7A15T	Artix-7	16 640	20 800	900	45	5	250
XC7A25T	Artix-7	23 360	29 200	1 170	80	5	250
XC7A35T	Artix-7	33 280	41 600	1 800	90	5	250
XC7A50T	Artix-7	52 160	65 200	2 700	120	5	250
XC7A75T	Artix-7	75 520	94 400	3 780	180	6	300
XC7A100T	Artix-7	101 440	126 800	4 860	240	6	500
XC7A200T	Artix-7	215 360	269 200	13 140	740	10	500

Tabla 2.5: *Dispositivos FPGA de Intel (Altera) soportados por Kmila.*

Modelo	Familia	LEs	FFs	BRAM (Kb)	DSP	PLLs	I/O máx.
EP4CE6	Cyclone IV E	6 272	6 272	270	15	2	179
EP4CE10	Cyclone IV E	10 320	10 320	414	23	2	179
EP4CE15	Cyclone IV E	15 408	15 408	504	56	4	343
EP4CE22	Cyclone IV E	22 320	22 320	594	66	4	153
EP4CE30	Cyclone IV E	28 848	28 848	594	66	4	532
EP4CE40	Cyclone IV E	39 600	39 600	1 134	116	4	532
EP4CE55	Cyclone IV E	55 856	55 856	2 340	154	4	374
EP4CE75	Cyclone IV E	75 408	75 408	2 745	200	4	426
EP4CE115	Cyclone IV E	114 480	114 480	3 888	266	4	528

Tabla 2.6: *Dispositivos FPGA de Lattice Semiconductor soportados por Kmila.*

Modelo	Familia	LUTs	FFs	BRAM (Kb)	DSP	PLLs	I/O máx.
iCE40HX1K	iCE40 HX	1 280	1 280	64	0	1	96
iCE40HX4K[†]	iCE40 HX	—	—	—	0	—	178
iCE40HX8K	iCE40 HX	7 680	7 680	128	0	1	206
iCE40UP3K	iCE40 UltraPlus	2 800	2 800	80	4	1	39
iCE40UP5K	iCE40 UltraPlus	5 280	5 280	120	8	1	39
LFE5U-12	ECP5	12 000	12 000	576	12	2	197
LFE5U-25	ECP5	24 000	24 000	1 008	28	2	245
LFE5U-45	ECP5	44 000	44 000	1 944	72	4	259
LFE5U-85	ECP5	84 000	84 000	3 744	156	4	365

[†] El dispositivo iCE40HX4K comparte silicio con el iCE40HX8K, con recursos lógicos parcialmente deshabilitados; Lattice no publica cifras oficiales para LUTs, flip-flops ni capacidad de BRAM efectiva, por lo que el catálogo del software omite esos valores.

Las terminologías empleadas por cada fabricante para las unidades lógicas básicas no son equivalentes en sentido estricto: Xilinx denomina *celdas lógicas* (*logic cells*) a la suma ponderada de LUTs y slices, Altera utiliza *elementos lógicos* (*logic elements*, LEs) que contienen una LUT de 4 entradas y un registro, y Lattice emplea *LUTs de 4 entradas* como unidad base en iCE40 y ECP5. Esta heterogeneidad implica que la comparación directa del conteo de unidades lógicas entre fabricantes no refleja fielmente la capacidad real del dispositivo; para comparaciones precisas es necesario considerar también el tipo de LUT, la presencia de cadenas de acarreo dedicadas y la estructura del bloque DSP. Para efectos de Kmila, esta información se presenta al usuario como metadato informativo del dispositivo, sin que el simulador comportamental dependa de ella, ya que la simulación se realiza a nivel funcional e independiente de la tecnología objetivo.

2.4. Educación en Diseño Digital

2.4.1. Problemática Actual

La enseñanza del diseño digital en instituciones de educación superior enfrenta una tensión fundamental entre la naturaleza práctica de la disciplina y las restricciones materiales de los entornos académicos. El diseño de hardware, a diferencia de la programación de software, requiere en última instancia la validación sobre dispositivos físicos (FPGAs, CPLDs o ASICs) cuya adquisición, mantenimiento y disponibilidad están sujetos a limitaciones presupuestales y logísticas [25].

En el caso particular de la ESCOM, las asignaturas de Fundamentos de Diseño Digital, Arquitectura de Computadoras, Diseño de Sistemas Digitales y Sistemas en Chip conforman la columna vertebral del área de hardware dentro del plan de estudios de la carrera de Ingeniería en Sistemas Computacionales. Estas materias requieren que el estudiante adquiera competencia tanto en la descripción de hardware mediante HDL como en la verificación del comportamiento de los diseños mediante simulación.

Sin embargo, la práctica individual se ve restringida por varios factores concurrentes:

- **Disponibilidad limitada de laboratorios.** Los laboratorios de hardware son recursos compartidos entre múltiples grupos y asignaturas, lo que reduce el tiempo efectivo de práctica por estudiante a las horas asignadas en el horario curricular.
- **Dependencia de herramientas comerciales pesadas.** Las herramientas estándar de la industria (Vivado, Quartus Prime) requieren instalaciones de decenas de gigabytes, hardware con especificaciones elevadas y están restringidas a Windows y Linux, excluyendo a estudiantes con macOS o equipos de gama baja.
- **Curva de aprendizaje fragmentada.** Las alternativas de código abierto (GHDL, Icarus Verilog, GTKWave) requieren la instalación, configuración y coordinación de múltiples herramientas independientes operadas por línea de comandos, lo que desvía el esfuerzo del estudiante hacia problemas de infraestructura en lugar de concentrarse en el aprendizaje del diseño digital.
- **Ausencia de retroalimentación inmediata.** A diferencia de los entornos de desarrollo de software, donde el ciclo editar-compilar-ejecutar es inmediato, el flujo de trabajo típico en diseño digital involucra múltiples pasos manuales (edición, compilación, elaboración, simulación, visualización) que ralentizan la experimentación.

2.4.2. Limitaciones del Hardware en el Aula

El acceso a tarjetas FPGA representa la barrera económica más significativa para la práctica individual de diseño digital. Como se documentó en la Sección 1.5, una tarjeta Altera Cyclone IV tiene un costo aproximado de $1,866 MXN y una Digilent Basys 3 (Xilinx Artix-7) de $3,400 MXN [3], [4]. Estos costos resultan prohibitivos para la mayoría de los estudiantes de instituciones públicas como la ESCOM, donde la adquisición de hardware queda fuera del presupuesto personal típico.

A nivel institucional, equipar un laboratorio con tarjetas suficientes para trabajo individual implica una inversión considerable, a la que se suman los costos de mantenimiento, reposición de unidades dañadas y actualización tecnológica conforme los fabricantes introducen nuevas familias de dispositivos. En la práctica, esto resulta en que los estudiantes trabajan en equipos de tres o más personas por tarjeta, lo que reduce drásticamente la experiencia individual de aprendizaje.

Adicionalmente, el hardware FPGA presenta limitaciones inherentes para el aprendizaje:

- **Depuración opaca.** Las señales internas de un diseño implementado en FPGA no son directamente observables sin herramientas especializadas como analizadores lógicos integrados (ChipScope en Xilinx, SignalTap en Intel), cuyo uso requiere conocimientos avanzados que exceden el nivel de un curso introductorio.
- **Ciclo de retroalimentación lento.** El flujo completo desde la modificación del código hasta la verificación en hardware (síntesis, place-and-route, generación de bitstream, programación del dispositivo) puede tomar varios minutos incluso para diseños pequeños, lo que desincentiva la experimentación iterativa.
- **Portabilidad nula.** El hardware de laboratorio no puede trasladarse al domicilio del estudiante, limitando la práctica al horario y espacio físico del laboratorio.

Estas limitaciones no implican que el hardware FPGA sea prescindible en la formación del ingeniero. Por el contrario, la validación sobre hardware real es una competencia indispensable. Sin embargo, la simulación por software puede cubrir las etapas iniciales del aprendizaje, donde el objetivo es comprender la semántica del lenguaje HDL y el comportamiento lógico del diseño, reservando el hardware para las etapas posteriores de validación e implementación.

2.4.3. Simulación como Complemento Pedagógico

La simulación de código HDL por software constituye una herramienta pedagógica que complementa, sin reemplazar, la práctica con hardware real. Su valor educativo radica en varios aspectos:

Accesibilidad universal. Un simulador por software se ejecuta en cualquier dispositivo personal sin requerir hardware especializado. Esto permite que cada estudiante practique de forma individual, en cualquier momento y lugar, eliminando las restricciones de horario y espacio del laboratorio.

Ciclo de retroalimentación inmediato. La simulación comportamental omite las etapas de síntesis, place-and-route y programación del dispositivo, reduciendo el ciclo de iteración de minutos a segundos. Esta inmediatez favorece la experimentación y el aprendizaje por prueba y error, que es el mecanismo natural de adquisición de habilidades de programación [25].

Observabilidad completa. A diferencia del hardware, donde las señales internas son opacas, la simulación permite inspeccionar el valor de cualquier señal en cualquier instante de tiempo. Los visores de formas de onda muestran la evolución temporal de todas las señales del diseño, facilitando la comprensión del comportamiento concurrente y la identificación de errores lógicos.

Entorno seguro para el error. En simulación, un diseño incorrecto no daña ningún dispositivo ni consume recursos irrecuperables. El estudiante puede cometer errores, analizar sus consecuencias y corregirlos sin costo material, lo que fomenta una actitud exploratoria frente al aprendizaje.

Preparación para el laboratorio. La simulación funciona como etapa de pre-laboratorio: el estudiante verifica la corrección lógica de su diseño antes de implementarlo en hardware, optimizando el tiempo efectivo de laboratorio y reduciendo los ciclos de depuración sobre la tarjeta FPGA.

En el contexto del presente proyecto, Kmila se concibe como una herramienta que integra el ciclo completo de edición, análisis y simulación de código VHDL en una sola aplicación multiplataforma, con el objetivo de reducir las barreras de entrada al aprendizaje de diseño digital y maximizar el tiempo que el estudiante dedica a comprender el comportamiento del hardware descrito, en lugar de gestionar la infraestructura de herramientas necesaria para simularlo.

Capítulo 3

Marco Conceptual

Este capítulo presenta las tecnologías, metodologías y herramientas seleccionadas para el desarrollo del sistema, así como la descripción de la solución propuesta. A diferencia del Capítulo 2, que aborda los fundamentos teóricos del dominio de aplicación (diseño digital, HDL, simulación), este capítulo se centra en las decisiones de ingeniería de software que sustentan la implementación de Kmila.

3.1. Tecnologías de Desarrollo

3.1.1. .NET MAUI

.NET Multi-platform App UI (.NET MAUI) es un framework de código abierto desarrollado por Microsoft para la creación de aplicaciones nativas multiplataforma a partir de un único código base en C# y XAML [23]. MAUI es el sucesor de Xamarin.Forms y forma parte integral del ecosistema .NET desde la versión 6. El presente proyecto utiliza la versión incluida en .NET 9; si bien .NET 10 ya se encuentra disponible, se mantiene la versión 9 para garantizar la compatibilidad total del ecosistema, dado que los releases iniciales de una versión mayor pueden presentar incompatibilidades con bibliotecas de terceros. La migración a .NET 10 se realizará una vez que se confirme la estabilidad del nuevo release.

La arquitectura de .NET MAUI se fundamenta en una capa de abstracción que mapea controles de interfaz de usuario a sus equivalentes nativos en cada plataforma objetivo: Android (vistas Android nativas), iOS (UIKit), macOS (Mac Catalyst) y Windows (WinUI 3). Esto permite que una aplicación MAUI se compile en un binario nativo para cada plataforma, aprovechando las APIs y el rendimiento del sistema operativo subyacente, a diferencia de los enfoques basados en contenedores web que ejecutan la interfaz dentro de un navegador embebido.

Un componente clave de .NET MAUI para el presente proyecto es **BlazorWebView**, que permite alojar componentes de Blazor dentro de una aplicación MAUI nativa. Este modelo, denominado **Blazor Hybrid**, combina las ventajas de ambos frameworks: la distribución nativa de MAUI (acceso a APIs del dispositivo, instalación desde stores, ejecución offline) con el modelo de componentes de Blazor (interfaz definida en Razor/HTML/CSS, reutilización de componentes entre plataformas). En Kmila, la aplicación MAUI actúa como contenedor nativo que aloja la totalidad de la interfaz de usuario mediante BlazorWebView, delegando la lógica de presentación a componentes Razor compartidos [23], [24].

La selección de .NET MAUI se justifica por tres factores: (1) permite cubrir cuatro plataformas nativas (Windows, macOS, Android, iOS) desde un único proyecto, reduciendo drásticamente el esfuerzo de desarrollo y mantenimiento; (2) la integración nativa con Blazor mediante BlazorWebView habilita la reutilización completa de la interfaz de usuario con el proyecto web; y (3) el acceso a APIs nativas del dispositivo (sistema de archivos, almacenamiento

local, notificaciones) es directo, sin necesidad de puentes o plugins adicionales.

3.1.2. Blazor

Blazor es un framework de Microsoft para la construcción de interfaces de usuario web interactivas utilizando C# en lugar de JavaScript [24]. Forma parte de ASP.NET Core y permite definir componentes de interfaz mediante la sintaxis Razor, que combina marcado HTML con lógica en C# dentro de archivos `.razor`.

Blazor ofrece múltiples modelos de hospedaje, de los cuales dos son relevantes para Kmila:

- **Blazor Hybrid** (utilizado en la aplicación MAUI). Los componentes Razor se ejecutan directamente en el proceso nativo de la aplicación, con acceso completo a los recursos del dispositivo. La interfaz se renderiza en un control WebView embebido, pero la lógica de negocio y el acceso a datos se ejecutan de forma nativa, sin comunicación con un servidor remoto. Este es el modelo empleado por Kmila en sus versiones para Windows, macOS, Android e iOS.
- **Blazor Server** (utilizado en el proyecto web). Los componentes Razor se ejecutan en el servidor ASP.NET Core, y la interfaz se actualiza en el navegador del cliente mediante una conexión SignalR en tiempo real. Cada interacción del usuario se transmite al servidor, que procesa el evento, calcula el diferencial del DOM y lo envía de vuelta al navegador. Este modelo permite que la aplicación web funcione en cualquier navegador moderno sin requerir la descarga de un runtime adicional, lo que resulta adecuado para cubrir la plataforma Linux y para usuarios que prefieren no instalar una aplicación nativa.

La arquitectura de Kmila se estructura en torno a una **biblioteca de clases Razor compartida** (Kmila.Shared) que contiene la totalidad de los componentes de interfaz, páginas, servicios y modelos de datos. Tanto el proyecto MAUI como el proyecto web referencian esta biblioteca, de modo que la interfaz de usuario se define una sola vez y se ejecuta en ambos contextos sin duplicación de código. Esta estrategia de código compartido es posible gracias a que los componentes Razor son agnósticos respecto al modelo de hospedaje: el mismo componente funciona tanto en BlazorWebView (MAUI) como en Blazor Server (web), y la inyección de dependencias permite sustituir las implementaciones específicas de cada plataforma cuando es necesario.

3.1.3. C# y .NET 9

C# es un lenguaje de programación multiparadigma (orientado a objetos, funcional, genérico) desarrollado por Microsoft como parte de la plataforma .NET [45]. Desde su primera versión en 2002, C# ha evolucionado de forma continua, incorporando características modernas como expresiones lambda, LINQ, programación asíncrona (async/await), pattern matching, records y tipos de referencia anulables. La versión 13 del lenguaje, incluida en .NET 9, es la empleada en el presente proyecto.

.NET 9 es la plataforma unificada de desarrollo de Microsoft utilizada en el presente proyecto, publicada en noviembre de 2024 [46]. Consolida en un único SDK los frameworks previamente separados (.NET Framework, .NET Core, Mono, Xamarin) bajo una implementación común del runtime (CoreCLR para servidor y escritorio, Mono para móviles y WebAssembly). Cabe señalar que .NET 10 fue publicado posteriormente; sin embargo, se mantiene .NET 9 como plataforma objetivo para garantizar la estabilidad y compatibilidad plena con todas las dependencias del proyecto (BlazorMonaco, sqlite-net-pcl, Blazor.Bootstrap, entre otras). La migración a .NET 10 se contempla una vez que el ecosistema de bibliotecas confirme su compatibilidad con la nueva versión. Las características relevantes de .NET 9 para el proyecto incluyen:

- **Compilación AOT (Ahead-of-Time):** permite compilar la aplicación directamente a código nativo, reduciendo el tiempo de inicio y el consumo de memoria en dispositivos móviles.
- **Rendimiento mejorado del garbage collector:** optimizaciones en el recolector de basura que reducen las pausas durante la ejecución, relevante para la simulación en tiempo real.
- **Soporte multiplataforma nativo:** un mismo proyecto puede compilarse para Windows (x64/ARM64), macOS (x64/ARM64), Linux (x64/ARM64), Android e iOS sin modificaciones al código fuente.
- **Hot Reload:** permite aplicar cambios en el código C# y Razor durante la ejecución sin reiniciar la aplicación, acelerando el ciclo de desarrollo.

La selección de C# y .NET como plataforma de desarrollo se justifica por la unificación del ecosistema: el mismo lenguaje y las mismas bibliotecas se utilizan en la aplicación nativa (MAUI), la aplicación web (Blazor Server), el motor de simulación (Parser, Interpreter, Debugger) y la capa de persistencia (SQLite). Esta uniformidad elimina las fricciones de interoperabilidad entre lenguajes y reduce la complejidad del proyecto.

3.1.4. SQLite

SQLite es un motor de base de datos relacional embebido, implementado como una biblioteca en C sin dependencias externas, que almacena la base de datos completa en un único archivo [47]. A diferencia de los sistemas de gestión de bases de datos cliente-servidor (PostgreSQL, MySQL, SQL Server), SQLite no requiere un proceso separado ni configuración de red: la aplicación accede directamente al archivo de base de datos mediante llamadas a la biblioteca.

En Kmila, SQLite se emplea mediante la biblioteca **sqlite-net-pcl** (v1.9.172), un micro-ORM para .NET que proporciona acceso asíncrono a SQLite con un API basado en atributos y expresiones LINQ [48]. La clase `ApplicationDbContext` del proyecto extiende `SQLiteAsyncConnection` y define las tablas para las entidades del dominio (proyectos, archivos de proyecto, configuraciones).

La selección de SQLite se fundamenta en tres criterios: (1) es la base de datos embebida más desplegada del mundo, con soporte nativo en Android, iOS, macOS, Windows y Linux, lo que garantiza compatibilidad multiplataforma sin configuración; (2) su naturaleza embebida elimina la necesidad de infraestructura de servidor, alineándose con el requisito de que Kmila funcione completamente offline; y (3) su bajo consumo de recursos la hace adecuada para dispositivos móviles con memoria y almacenamiento limitados.

3.2. Plataformas Soportadas y Requisitos del Sistema

Kmila se distribuye en dos modalidades complementarias que cubren el conjunto completo de plataformas objetivo descritas en los requerimientos del proyecto: una **aplicación nativa multiplataforma** construida con .NET MAUI (Android, iOS, iPadOS, Windows y macOS) y una **aplicación web** construida con Blazor Server (acceso desde cualquier navegador moderno; despliegue en servidor Linux). Ambas modalidades comparten la misma biblioteca de lógica y vistas (`Kmila.Shared`), por lo que la experiencia funcional es equivalente entre ellas; las diferencias se concentran en el mecanismo de despliegue, el ciclo de vida de inyección de dependencias y los requisitos de conexión.

La Tabla 3.1 resume las versiones mínimas y recomendadas de cada plataforma soportada, junto con los requisitos de hardware. Los valores mínimos se extrajeron directamente de los descriptores de proyecto (`Kmila.csproj`, `Package.appxmanifest`, `Info.plist`); las recomendaciones reflejan las versiones bajo las cuales se ha verificado el funcionamiento durante el desarrollo.

3.2.1. Modalidades de Despliegue

Modalidad nativa (MAUI). La aplicación se empaqueta como ejecutable nativo de cada sistema operativo (APK en Android, IPA en iOS/iPadOS, MSIX o paquete autónomo en Windows, App Bundle en macOS). Toda la lógica, los recursos (catálogo FPGA, diccionarios de idioma, lecciones del módulo *Aprender*) y la base de datos SQLite se incluyen en el binario o se inicializan localmente al primer arranque. Esta modalidad funciona **completamente sin conexión a internet** tras la instalación.

Modalidad web (Blazor Server). La aplicación se ejecuta en un servidor con .NET 9 (típicamente un servidor Linux), y el cliente accede desde cualquier navegador moderno. La interfaz se renderiza en el servidor y se sincroniza con el navegador mediante una conexión persistente *WebSocket* sobre el protocolo SignalR. Esta modalidad requiere conexión continua entre cliente y servidor, pero elimina por completo los requisitos de instalación en el cliente: basta con abrir una URL.

Ambas modalidades comparten el mismo motor de simulación, el mismo catálogo de dispositivos FPGA, el mismo conjunto de idiomas y el mismo módulo de aprendizaje. La modalidad web es la vía de acceso recomendada para sistemas Linux, ya que .NET MAUI no compila para Linux como objetivo nativo.

3.2.2. Versiones Soportadas

Tabla 3.1: *Plataformas soportadas: versiones mínimas, recomendadas y arquitecturas de procesador.*

Plataforma	Versión mínima	Recomendada	Arquitecturas
Android	7.0 Nougat (API 24)	12 Snow Cone (API 31)+	ARMv7, ARM64, x86_64
iOS	iOS 15	iOS 17+	ARM64 (iPhone 6s+)
iPadOS	iPadOS 15	iPadOS 17+	ARM64 (iPad 5ª gen.+)
macOS	macOS 12 Monterey	macOS 14 Sonoma+	x64 (Intel), ARM64 (Apple Silicon)
Windows	Windows 10 v1809 (build 17763)	Windows 11 22H2+	x64
Linux (vía web)	Ubuntu 20.04 o equivalente†	Ubuntu 22.04 LTS+	x64, ARM64

† Distribuciones equivalentes oficialmente soportadas por el runtime de .NET 9: Debian 11, RHEL 8, Fedora 39, openSUSE 15, Alpine 3.18 (servidor).

Las versiones mínimas para Android y iOS quedan condicionadas por las restricciones de `SupportedOSPlatformVersion` declaradas en `Kmila.csproj`; la versión mínima de Windows está fijada por el rango `MinVersion=10.0.17763.0`, `MaxVersionTested=10.0.19041.0` declarado en `Package.appxmanifest`; macOS hereda la restricción de MacCatalyst 15.0, equivalente a macOS 12 Monterey. Linux no es un objetivo nativo de MAUI, por lo que el soporte en distribuciones Linux se obtiene exclusivamente a través de la modalidad web; las distribuciones listadas corresponden al conjunto soportado oficialmente por el runtime de .NET 9 [46].

3.2.2.1. Navegadores soportados (modalidad web)

Para la modalidad web, el cliente requiere un navegador con soporte completo de WebAssembly, WebSocket y módulos ES2020. La Tabla 3.2 resume los navegadores y versiones mínimas verificados.

Tabla 3.2: *Navegadores soportados para la modalidad web.*

Navegador	Versión mínima	Notas
Google Chrome	90 (abril 2021)	Plataforma de referencia
Microsoft Edge	90 (abril 2021)	Basado en Chromium
Mozilla Firefox	88 (abril 2021)	Soporte completo
Apple Safari	14 (sept. 2020)	macOS 11 / iOS 14
Chromium derivados (Brave, Opera, Vivaldi)	equivalente a Chrome 90+	Funcionalidad completa

3.2.3. Requisitos de Hardware

La Tabla 3.3 establece los requisitos mínimos de memoria RAM y almacenamiento por plataforma. Estos valores incluyen la huella del runtime `.NET` 9, los recursos empaquetados de Kmila (catálogo de 34 dispositivos FPGA, diccionarios de los 7 idiomas, manifiesto y bodies del módulo *Aprender*) y una holgura razonable para datos del usuario (proyectos, archivos VHDL, configuraciones guardadas).

Tabla 3.3: *Requisitos mínimos de hardware por plataforma.*

Plataforma	RAM	Almacenamiento	Procesador
Android	2 GB	150 MB libres	ARMv7-A o ARM64, doble núcleo $\geq$ 1.5 GHz
iOS / iPadOS	2 GB	150 MB libres	ARM64 (Apple A9 o posterior)
macOS	4 GB	400 MB libres	Intel x64 doble núcleo o Apple Silicon
Windows	4 GB	400 MB libres	x64 doble núcleo $\geq$ 1.8 GHz
Cliente web (cualquier OS)	2 GB	50 MB caché de navegador	Cualquiera con navegador moderno
Servidor web (Linux)	1 GB libre	200 MB libres	x64 o ARM64, 1 vCPU base

Los valores de RAM y almacenamiento son indicativos para el funcionamiento de la aplicación en estado de reposo y bajo cargas de trabajo típicas (un proyecto activo con un archivo VHDL de complejidad media, un dispositivo FPGA seleccionado y una simulación con 10^4 a 10^5 muestras de forma de onda). Para sesiones con múltiples proyectos abiertos simultáneamente, simulaciones con duraciones extensas o números elevados de señales observadas, se recomienda doblar los valores indicados.

3.2.4. Pesos de los artefactos de distribución

Como verificación empírica del requerimiento no funcional de distribución ligera, se ejecutó la compilación en modo `Release` de Kmila para cada plataforma objetivo mediante `dotnet publish -c Release`. Los pesos resultantes de los archivos entregables se midieron con la utilidad `du -sh` sobre los directorios y archivos generados en mayo de 2026 (Tabla 3.4).

Tabla 3.4: *Pesos reales de los artefactos de distribución, medidos sobre las salidas de `dotnet publish -c Release`.*

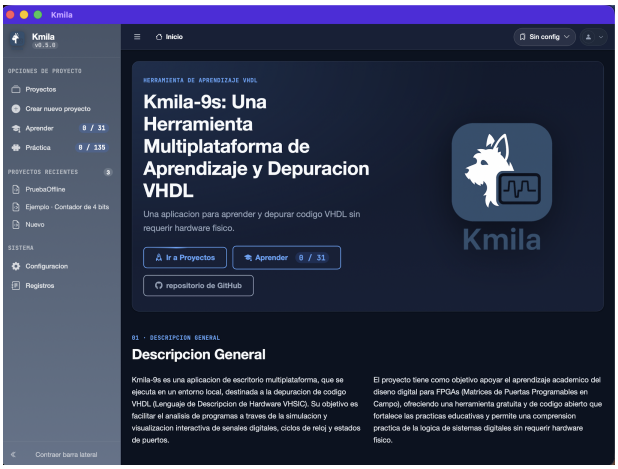
Plataforma	Archivo entregable	Peso	Build
Android	`kmila.kmila-Signed.apk`	72 MB	182.5 s
Android (Play)	`kmila.kmila-Signed.aab`	72 MB	—
iOS	`Kmila.ipa`	43 MB	258.7 s
macOS	`Kmila-1.0.pkg` (instalador)	68 MB	197.9 s
Linux / Web	Publicación Blazor completa	124 MB	60.4 s
Linux / Web	Contenido servido al cliente (`_content`)	76 MB	—
Windows	Pendiente (requiere host Windows o CI)	—	—

El conjunto de artefactos se ubica en un rango de 43–147 MB según la plataforma y la forma de empaquetado (instalador comprimido frente a *bundle* sin comprimir). Esta huella contrasta directamente con la magnitud de las herramientas comerciales equivalentes documentadas en la Sección 1.5: AMD Vivado ML Standard ocupa 40–80 GB según las familias seleccionadas [5], Intel Quartus Prime supera los 20 GB [4] y Siemens ModelSim requiere más de 1.5 GB [8]. En consecuencia, el factor de reducción de Kmila frente a Vivado se sitúa entre 500 y 1000 veces, y frente a Quartus en aproximadamente 250 veces.

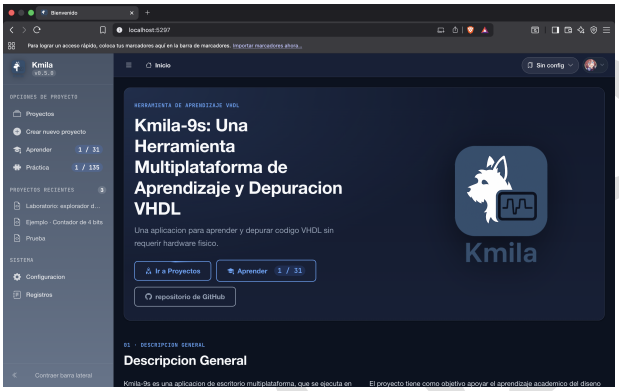
Esta reducción de huella tiene tres implicaciones pedagógicas relevantes: (i) hace viable la distribución a estudiantes con equipos de cómputo modestos y restricciones de almacenamiento, (ii) permite la instalación en dispositivos móviles personales como apoyo de pre-laboratorio (`43 MB` en iOS y `72 MB` en Android entran cómodamente sobre redes celulares), y (iii) elimina la barrera operativa de instalaciones prolongadas, que en Vivado pueden superar la hora en redes institucionales.

3.2.5. Evidencia visual de ejecución multiplataforma

Complementando la verificación cuantitativa de los pesos, se documenta la ejecución real de Kmila en cuatro de las plataformas objetivo (macOS, iOS, Android y Web sobre Chrome), capturada el 10 de mayo de 2026. La Figura 3.1 muestra la pantalla de bienvenida y la Figura 3.2 muestra el editor de código VHDL en cada plataforma. El target de Windows queda pendiente al requerir un host Windows o pipeline de integración continua.

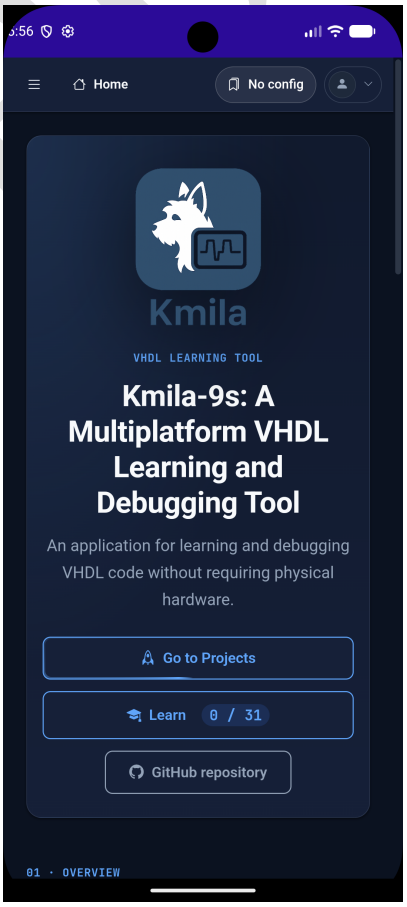


(a) *macOS (MAUI MacCatalyst, nativo).*



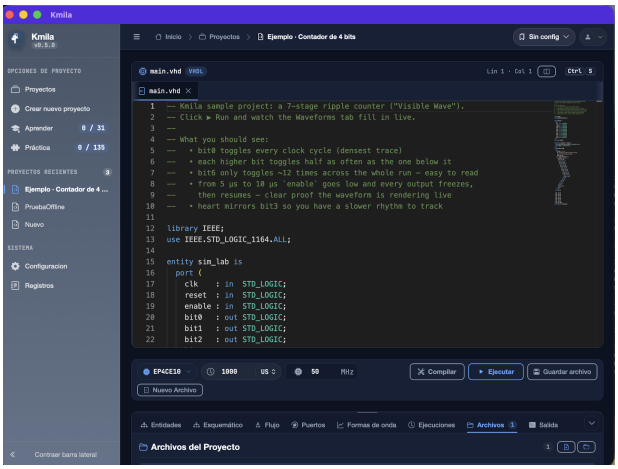
(b) *Web (Blazor Server, Chrome).*

(c) *iOS (MAUI nativo).*

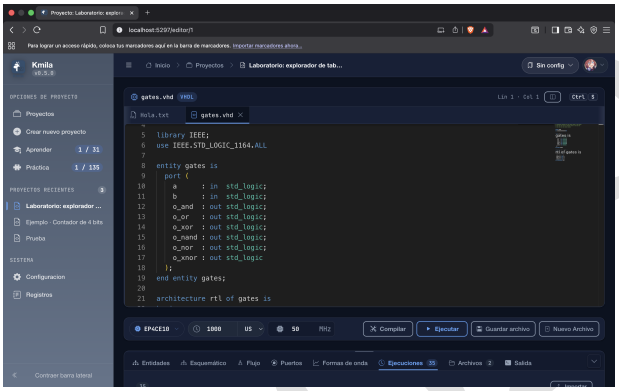


(d) *Android (MAUI nativo, idioma inglés).*

Figura 3.1: *Pantalla de bienvenida de Kmila ejecutándose en cuatro plataformas. El layout (sidebar, área de bienvenida con descripción y botones de navegación) se adapta de forma responsiva desde una ventana de escritorio hasta orientación vertical en teléfono. Los semáforos de macOS y la barra de estado de iOS/Android confirman que se trata de aplicaciones nativas, no de contenedores web embebidos.*

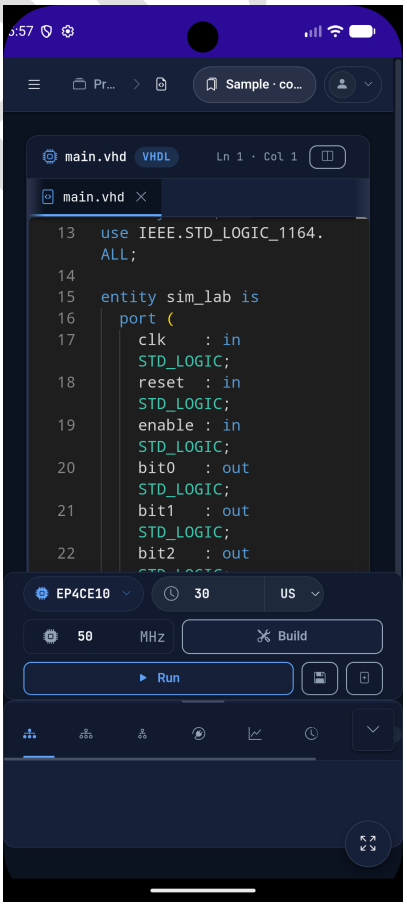


(a) *Editor en macOS.*



(b) *Editor en Web (Chrome).*

(c) *Editor en iOS.*



(d) *Editor en Android (UI en inglés).*

Figura 3.2: *Editor de código VHDL de Kmila operando sobre el mismo proyecto en cuatro plataformas distintas. La pestaña de archivo, el resaltado de sintaxis Monaco, la barra de configuración inferior (FPGA, frecuencia, escala temporal) y los controles de acción (Compilar/Ejecutar, equivalentes a Build/Run en la versión Android en inglés) se preservan íntegramente, validando empíricamente la promesa de* un solo código base *(`Kmila.Shared`) compartido entre todos los destinos.*

Verificación del modo offline (RNF-02)

La Figura 3.3 muestra a Kmila ejecutándose en macOS con la conexión de red deshabilitada (modo avión activado en la barra de estado del sistema). Toda la funcionalidad de edición, gestión de archivos y navegación de proyectos se mantiene disponible, confirmando empíricamente el requerimiento no funcional RNF-02 (Disponibilidad 100 % *offline*) para la modalidad nativa.

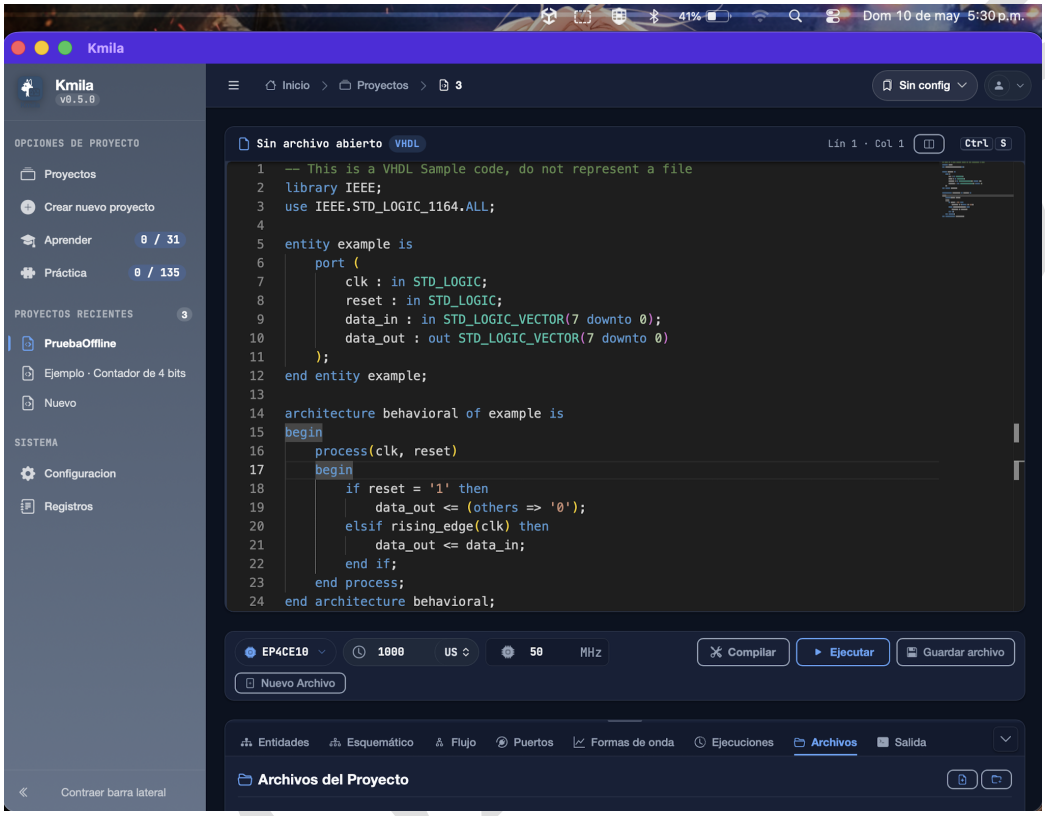


Figura 3.3: *Kmila operando con la conexión a red deshabilitada (modo avión) en macOS. El editor mantiene resaltado de sintaxis, navegación por archivos y persistencia local, demostrando que la modalidad nativa no depende de servicios externos.*

Verificación del tiempo de respuesta de UI (RNF-03)

La Figura 3.4 muestra el panel *Performance* de Chrome DevTools instrumentando la versión web de Kmila durante una sesión típica de uso. La métrica *Largest Contentful Paint* (LCP) reportada por el navegador es de **45 ms**, holgadamente por debajo del umbral de 200 ms establecido en RNF-03 y del umbral de 2.5 s recomendado por Google para una experiencia "buena". El análisis programático del *trace* subyacente (39 904 eventos sobre 9.01 s de sesión) confirma además que: (i) no se registró ninguna tarea de procesamiento superior a 200 ms, (ii) los *clicks* de usuario se procesaron en un máximo de 1.56 ms, y (iii) el promedio de `EventDispatch` sobre 532 eventos fue de 0.05 ms.

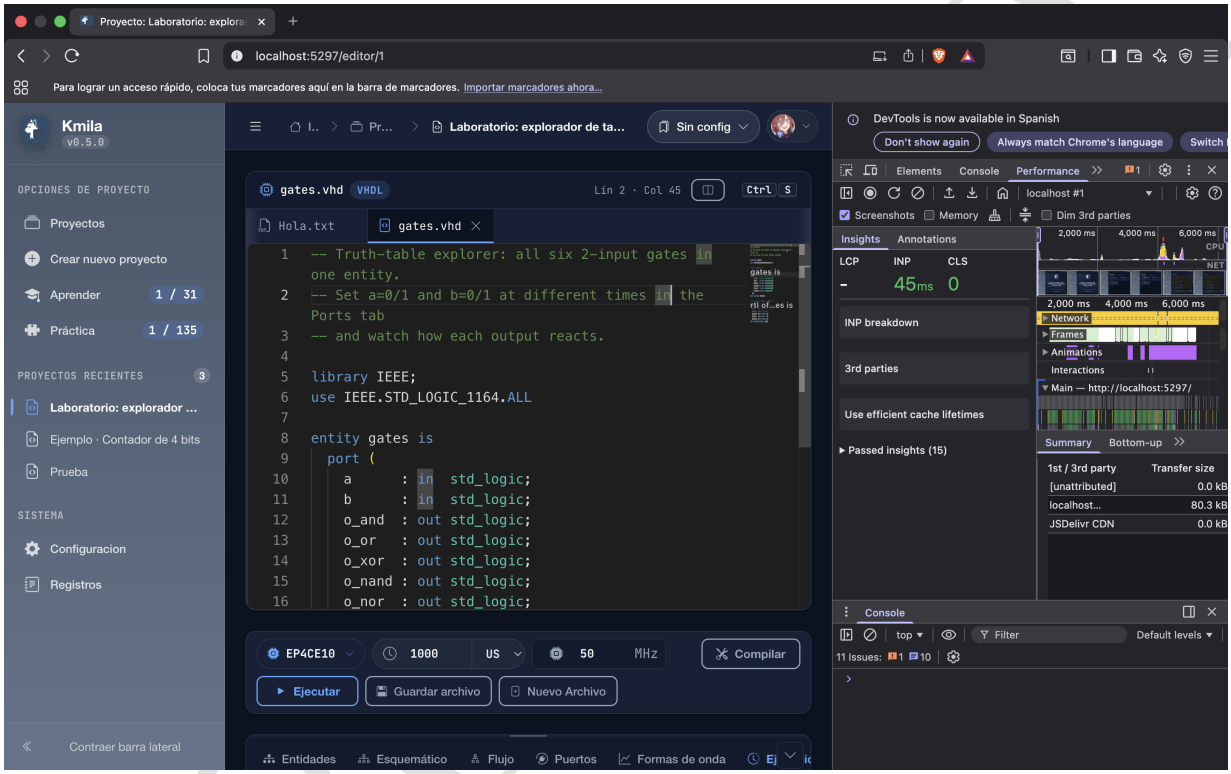


Figura 3.4: *Panel* Performance Insights *de Chrome DevTools sobre Kmila Web (`localhost:5297`). La métrica LCP reportada por el navegador es de 45 ms, ∼4.4 veces menor al umbral de RNF-03 (200 ms) y dos órdenes de magnitud bajo el umbral recomendado por Google (2.5 s).*

3.2.6. Huella de memoria en ejecución

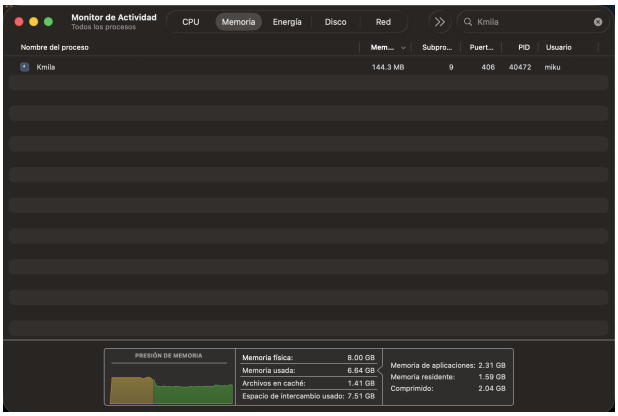
Como complemento al análisis de pesos en disco, se midió la huella de memoria RAM de Kmila en ejecución sobre cuatro plataformas. Las mediciones se obtuvieron con el Monitor de Actividad de macOS para los procesos de la modalidad nativa (`Kmila` sobre macOS y sobre el iOS Simulator) y para el proceso de servidor de la modalidad web (`Kmila.Web`), y con el comando `top` sobre un dispositivo Android físico para la versión móvil. La Tabla 3.5 resume los valores observados; las capturas de soporte se muestran en la Figura 3.5.

Tabla 3.5: *Huella de memoria RAM medida por plataforma (build `Debug`, sin optimización AOT completa).*

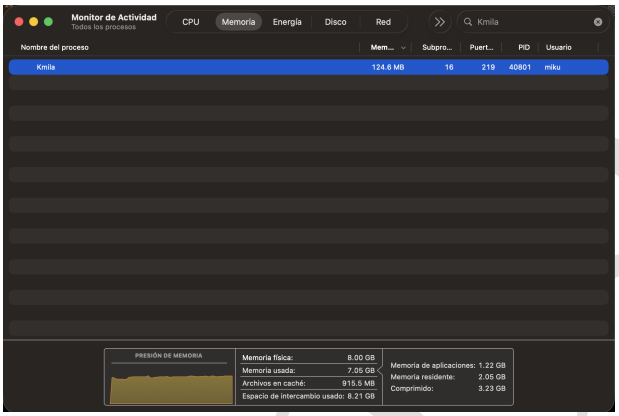
Plataforma	Memoria reportada	Privada / residente	Herramienta
macOS (MacCatalyst)	144.3 MB	70.2 MB privada	Monitor de Actividad
iOS (Simulator)	124.6 MB	30.0 MB residente	Monitor de Actividad
Android (físico)	348 MB residente	5 MB SHR	`top` sobre adb shell
Web (Blazor Server, Debug)	947.3 MB	37 subprocs	Monitor de Actividad

Las tres modalidades nativas se ubican en un rango de 124 MB a 348 MB, valores comparables a los de un navegador web ligero. La modalidad web presenta un consumo significativamente mayor (947 MB) por dos razones técnicas verificables: (i) Blazor Server mantiene el estado de cada sesión cliente en el servidor mediante un circuito SignalR persistente, lo que implica retener el modelo de componentes en memoria; y (ii) la medición se realizó sobre un *build* en modo `Debug` con depurador adjunto, lo que típicamente añade entre 200 y 400 MB sobre la huella en `Release` debido a símbolos de depuración, instrumentación de *hot reload* y desactivación de la compilación AOT.

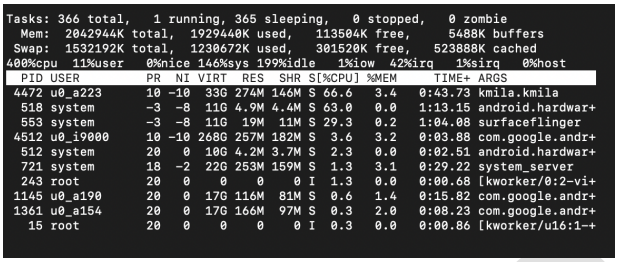
Incluso bajo el peor caso medido (947 MB en modo `Debug`), Kmila se sitúa significativamente por debajo de los requisitos de memoria de las herramientas comerciales equivalentes documentadas en el Sección 1.5: AMD Vivado recomienda un mínimo de 8 GB de RAM y comúnmente excede 16 GB en flujos de implementación reales [5], e Intel Quartus Prime requiere de manera similar entre 4 y 8 GB para operación típica [4]. El factor de reducción frente a Vivado es de aproximadamente 8× en el escenario web y de 55× a 64× en las modalidades nativas, lo que valida empíricamente la viabilidad de Kmila como herramienta de pre-laboratorio sobre equipos modestos.



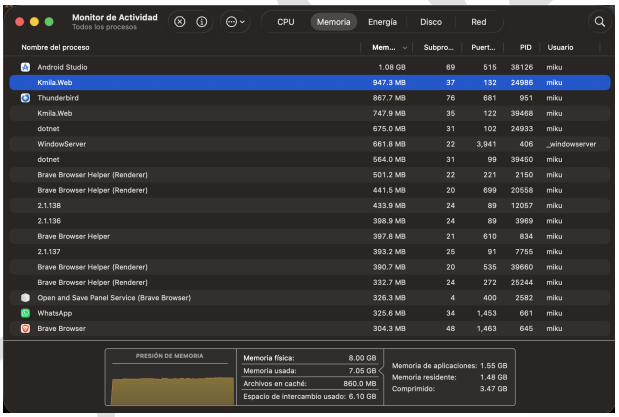
(a) *macOS — 144.3 MB.*



(b) *iOS Simulator — 124.6 MB.*



(c) *Android (físico) — 348 MB residente.*



(d) *Web (Blazor Server, Debug) — 947.3 MB.*

Figura 3.5: *Capturas del Monitor de Actividad de macOS y `top` sobre Android mostrando el consumo de memoria RAM del proceso `Kmila` en cada plataforma objetivo. Las tres modalidades nativas se ubican entre 124 MB y 348 MB; la modalidad web (build `Debug`) reporta 947 MB por la arquitectura Blazor Server.*

3.2.7. Requisitos de Conexión y Permisos

Conectividad.

- **Modalidad nativa:** no requiere conexión a internet tras la instalación. Toda la funcionalidad (edición, análisis, simulación, visualización, persistencia) opera localmente.
- **Modalidad web:** requiere conexión persistente con el servidor durante toda la sesión. El protocolo subyacente es WebSocket (con fallback a long-polling sobre HTTP). Se recomienda un ancho de banda mínimo de 256 kbps y una latencia inferior a 200 ms para una experiencia fluida.

Permisos por plataforma.

- **Android:** `INTERNET` y `ACCESS_NETWORK_STATE` (utilizadas únicamente para descargar plantillas o componentes a futuro), `READ_EXTERNAL_STORAGE` y `WRITE_EXTERNAL_STORAGE` (para abrir y guardar archivos VHDL fuera del directorio de la aplicación).
- **iOS / iPadOS:** acceso al *sandbox* de la aplicación (estándar). El sistema operativo gestiona automáticamente los permisos de almacenamiento dentro del directorio asignado.
- **macOS / Windows / Linux (servidor):** acceso de lectura/escritura al directorio de instalación o al directorio del usuario donde se almacenan los proyectos (`./Projects/` por convención).

Pantalla mínima. El diseño responsivo se adapta desde 360 px de ancho efectivo (teléfonos en orientación vertical) hasta resoluciones de escritorio. Para la pantalla del editor con todas las herramientas visibles simultáneamente se recomienda un ancho mínimo de 1280 px.

3.3. Ingeniería de Software

3.3.1. Patrones de Diseño

La arquitectura de Kmila emplea varios patrones de diseño que promueven la separación de responsabilidades, la testabilidad y la reutilización de código:

Inyección de dependencias (DI). Todas las dependencias entre componentes se resuelven mediante el contenedor de inyección de dependencias integrado en .NET (`Microsoft.Extensions.DependencyInjection`). Los servicios se registran en el punto de entrada de cada aplicación (`MauiProgram.cs` para la versión nativa, `Program.cs` para la versión web) con el ciclo de vida apropiado (Singleton para servicios compartidos como `Traductor` y `ConfigurationService`, Scoped para contextos de base de datos). Este patrón permite que los componentes dependan de abstracciones (interfaces) en lugar de implementaciones concretas, facilitando la sustitución de servicios entre plataformas.

Patrón repositorio. El acceso a datos se encapsula detrás de la interfaz genérica `IRepository<T>`, que define operaciones CRUD asíncronas (`CreateAsync`, `UpdateAsync`, `RemoveAsync`, `GetListAsync`, `GetByAsync`). Las implementaciones concretas (`Projects`, `ProjectFiles`) encapsulan las consultas SQLite, de modo que los componentes de interfaz y los servicios de negocio no dependen directamente del mecanismo de persistencia. Este patrón facilita la migración futura a otros motores de almacenamiento sin modificar la lógica de negocio.

Arquitectura basada en eventos. El servicio central `ProgramBuilder`, que orquesta el flujo de compilación y simulación de código VHDL, emplea un modelo de comunicación basado en eventos (`OnSimulationEnded`, `OnProgressAdvanced`, `OnBuildProgressChanged`, `OnBuildError`). Los componentes de interfaz se suscriben a estos eventos para actualizar la visualización en tiempo real sin acoplamiento directo con la lógica del motor. Este patrón desacopla el motor de simulación de la capa de presentación y permite que múltiples componentes reaccionen a un mismo evento de forma independiente.

Biblioteca de clases compartida. Como se describió en la Sección 3.1.2, el proyecto Kmila.Shared centraliza los componentes Razor, servicios, repositorios y modelos de datos en una biblioteca de clases referenciada tanto por el proyecto MAUI como por el proyecto web. Este patrón de código compartido maximiza la reutilización y garantiza la consistencia funcional entre plataformas.

3.3.2. Herramientas de Gestión

El desarrollo del proyecto emplea las siguientes herramientas:

- **Git y GitHub.** Sistema de control de versiones distribuido utilizado para el seguimiento de cambios, la gestión de ramas y el respaldo del código fuente. El repositorio principal se aloja en GitHub, lo que habilita la distribución del código como proyecto de código abierto.
- **Visual Studio Code.** Editor de código ligero y extensible utilizado como entorno de desarrollo principal. Dado que el desarrollo se realiza íntegramente en Linux, Visual Studio Code es la herramienta más adecuada por su compatibilidad nativa con la plataforma, su integración con el SDK de .NET mediante la extensión C# Dev Kit, y su soporte para depuración, terminal integrada y control de versiones. No se utiliza Visual Studio 2022, ya que no está disponible de forma nativa en Linux.
- **Android Debug Bridge (ADB).** Herramienta de línea de comandos del SDK de Android utilizada para la depuración y despliegue de la aplicación en dispositivos Android físicos durante el desarrollo.
- **SDK de .NET 9.** Kit de desarrollo que incluye el compilador de C#, el runtime, las herramientas de compilación (MSBuild) y las plantillas de proyecto. Es el único requisito de infraestructura para compilar el proyecto completo.

3.4. Solución Propuesta

Kmila es una aplicación multiplataforma de código abierto diseñada para el aprendizaje, la práctica y la depuración de código VHDL, orientada a estudiantes de ingeniería que cursan asignaturas de diseño digital. La aplicación integra en un solo entorno las funcionalidades que actualmente requieren la combinación de múltiples herramientas independientes: edición de código con resaltado de sintaxis, análisis léxico y sintáctico, simulación comportamental, visualización de formas de onda, vista esquemática del netlist sintetizado y, en una contribución original del proyecto, un componente pedagógico completo organizado en tres subsistemas didácticos.

La solución se estructura en torno a cinco pilares:

Entorno de edición integrado (*workbench*). Kmila incorpora el editor Monaco (el mismo motor que impulsa Visual Studio Code) mediante la biblioteca BlazorMonaco, proporcionando al estudiante un editor de código con resaltado de sintaxis para VHDL, numeración de líneas, plegado de bloques y búsqueda integrada. El editor se acompaña de un *dock* inferior de nueve pestañas que centraliza todas las vistas del diseño en curso: **Entities** (estructura jerárquica de entidades, arquitecturas y puertos), **Schematic** (visualizador de circuitos a nivel de compuertas con tres modos: *Naive*, *Synthesis* y *Debug* con cursor sincronizado), **Flow** (diagrama de flujo del control extraído del AST mediante el servicio `PlantUmlFlowEmitter`), **Ports** (panel de estímulos por momento de tiempo), **Waveforms** (formas de onda con modo *compare* para superponer múltiples corridas), **Runs** (historial de simulaciones con etiquetado, anclado y comparación), **Replay** (reproducción paso a paso post-simulación que reutiliza el componente `ExecutionPlayer`), **Files** (gestor de archivos del proyecto) y **Output** (consola de *build* y simulación). Una barra superior `RunBar` concentra los controles *Run/Pause/Stop/Restart* y la barra inferior `StatusBar` reporta utilización de recursos del FPGA seleccionado y estado de la compilación en tiempo real.

Motor de análisis y simulación. El núcleo computacional de Kmila se compone de cinco módulos independientes desarrollados en C#: el **Parser** (análisis léxico y sintáctico de código VHDL, generación del diccionario de *scopes* y diagnósticos estructurados), el **Interpreter** (motor de eventos discretos `DeltaCycleEngine` con `ProcessScheduler`, `SignalScheduler` y `SignalHistory`; resolución de atributos VHDL `'event`, `'last_value`, `'stable` y `'active`), el **Debugger** (fachada de *lint* resiliente que sobrevive a fallos parciales de parseo), el **TimeMachine** (*timer* interactivo para ejecución paso a paso con doble `CancellationTokenSource` para pausa/parada) y el **LibraryCompiler** (resolución topológica del DAG de dependencias `use` con caché por *hash* SHA-256 y cascada de fuentes *host*–embebida–*fallback*). Estos módulos implementan el modelo de eventos discretos y ciclos delta descrito en la Sección 2.2.1, permitiendo la simulación comportamental de diseños VHDL conforme al estándar IEEE 1076.

Subsistemas pedagógicos. Kmila incorpora tres subsistemas pedagógicos complementarios al editor: **Learn** (catálogo estructurado de lecciones de VHDL con animaciones de compuertas y FSMs, alineado con los programas oficiales de las asignaturas ESCOM de *Fundamentos de Diseño Digital*, *Diseño de Sistemas Digitales*, *Arquitectura de Computadoras* y *Sistemas en Chip*), **Practice** (catálogo de ejercicios autoevaluables organizados en tres modalidades inspiradas en plataformas como *CodeForces*, *HackerRank* y *LeetCode*: *FromZero*, *FixBug* y *FillGap*) y **Concepts** (módulo mini con reproductor de ejecución paso a paso y editor visual *Blockly* para enseñar el modelo mental de ejecución secuencial vs. concurrente en VHDL). El detalle de estos subsistemas se desarrolla en la Sección 5.8.

Cobertura multiplataforma desde un único código base. Mediante la combinación de .NET MAUI (Blazor Hybrid) y Blazor Server, Kmila se distribuye como aplicación nativa para Windows, macOS, Android e iOS, y como aplicación web accesible desde cualquier navegador moderno. La biblioteca compartida Kmila.Shared centraliza 12 páginas Razor, 30 componentes Razor y 40 servicios inyectables, garantizando una experiencia de usuario idéntica en las seis plataformas objetivo. Esta arquitectura resuelve directamente la brecha de cobertura multiplataforma identificada en el análisis comparativo de la Sección 1.4.6.

Productividad y servicios transversales. Adicionalmente, Kmila incorpora un catálogo de 34 dispositivos FPGA (familias AMD/Xilinx Spartan-6 y Artix-7, Intel/Altera Cyclone IV E, Lattice iCE40 HX/UltraPlus y ECP5) que permite al estudiante contextualizar su diseño respecto a los recursos disponibles en un dispositivo real, un sistema de internacionalización con soporte para siete idiomas (español, inglés, francés, alemán, chino, japonés y árabe con escritura de derecha a izquierda), exportación a múltiples formatos (VCD compatible con GTKWave, SVG vectorial del esquemático y de las formas de onda, PlantUML del flujo de control, *bundle* comprimido del proyecto completo), un sistema opcional de reporte automático de errores a GitHub mediante OAuth, persistencia de *snapshots* de simulación con comparación entre corridas y un almacén de *assets* de usuario (`UserAssetStore`) que permite añadir dispositivos FPGA personalizados, paquetes de idioma y librerías VHDL propias en tiempo de ejecución.

El desarrollo se organiza en dos fases académicas: TT1 abarcó la interfaz de usuario, la gestión de proyectos, el editor de código, la arquitectura del sistema y el prototipo inicial del motor de simulación; TT2 consolida el motor de análisis y simulación, la visualización de formas de onda y del esquemático, los subsistemas pedagógicos, la pestaña de *replay* interactivo y la publicación en las tiendas de aplicaciones.

3.5. Metodología de Desarrollo

La selección de una metodología de desarrollo de software apropiada constituye una decisión de ingeniería que condiciona la organización del trabajo, las prácticas técnicas adoptadas, la cadencia de las entregas y, en última instancia, la calidad del producto final [49]. Para un proyecto académico como el presente Trabajo Terminal, esta decisión adquiere especial relevancia por tres motivos: (i) el equipo de desarrollo está conformado por una sola persona, lo que invalida marcos diseñados para equipos multidisciplinarios; (ii) el horizonte temporal está fijado por el calendario académico (~11 meses divididos en TT1 y TT2), con hitos de revisión predefinidos; y (iii) el dominio de aplicación (simulación de hardware descrito en VHDL) presenta un fuerte componente experimental que requiere iteración y refactorización frecuente.

Esta sección revisa las metodologías de desarrollo de software más relevantes para el contexto del proyecto, presenta una comparativa estructurada y justifica la metodología seleccionada para el desarrollo de Kmila.

Aclaración previa: reclasificación de Kanban

El documento de TT1 presentaba a *Kanban* como la metodología de desarrollo adoptada por el proyecto. Tras revisión de la literatura especializada y la retroalimentación recibida durante la evaluación, se reconoce que dicha clasificación es imprecisa: Kanban es un **método de gestión visual del flujo de trabajo** derivado del Sistema de Producción Toyota [50] y adaptado al desarrollo de software por Anderson [51], pero no prescribe prácticas de ingeniería de software (pruebas, integración, diseño, refactorización), por lo que no constituye una metodología de desarrollo en sentido estricto [52]. En esta versión del documento, Kanban se reposiciona correctamente como una herramienta complementaria de gestión del trabajo, mientras que la metodología de desarrollo seleccionada se justifica con base en los criterios técnicos del proyecto.

3.5.1. Extreme Programming (XP)

Extreme Programming (XP) fue formulada por Kent Beck a finales de la década de 1990 durante el proyecto Chrysler Comprehensive Compensation (C3), y publicada formalmente en *Extreme Programming Explained: Embrace Change* [53], [54]. XP se distingue del resto de los marcos ágiles por su énfasis en las *prácticas de ingeniería* más que en los procesos de gestión, llevando al extremo (de ahí su nombre) un conjunto de prácticas reconocidas como beneficiosas para el desarrollo de software.

XP se articula en torno a cinco valores fundamentales: **comunicación**, **simplicidad**, **retroalimentación**, **valentía** y **respeto**. Estos valores se materializan en doce prácticas concretas, agrupadas tradicionalmente en cuatro categorías:

- **Retroalimentación continua:** desarrollo guiado por pruebas (TDD) [55], juego de planificación, cliente *in situ* y programación en parejas (*pair programming*).
- **Proceso continuo:** integración continua, refactorización constante y entregas pequeñas y frecuentes (*small releases*).
- **Comprensión compartida:** diseño simple, metáfora del sistema, propiedad colectiva del código y estándar de codificación.
- **Bienestar del programador:** ritmo sostenible (originalmente *40-hour week*).

Las fortalezas de XP incluyen su alto nivel de prescripción técnica, su énfasis empírico en pruebas automatizadas que producen una red de seguridad para refactorizaciones agresivas, y su capacidad probada para producir código de alta calidad en proyectos con requerimientos volátiles [56]. Sus limitaciones principales son la dependencia de prácticas que requieren disciplina individual sostenida (TDD, refactorización), la suposición de disponibilidad continua del cliente, y la dificultad de aplicar literalmente algunas prácticas (programación en parejas) en equipos pequeños o unipersonales.

3.5.2. Scrum

Scrum fue propuesto por Ken Schwaber y Jeff Sutherland a mediados de la década de 1990 y formalizado en el libro *Agile Software Development with Scrum* [57]. El marco se mantiene actualizado mediante la *Scrum Guide* oficial, cuya versión vigente data de 2020 [58]. Scrum es el marco ágil de mayor adopción industrial y se centra en la gestión del proceso de desarrollo más que en las prácticas técnicas.

Scrum prescribe tres **roles**: el *Product Owner* (responsable de maximizar el valor del producto y gestionar el Product Backlog), el *Scrum Master* (facilitador del proceso y eliminador de impedimentos) y el *Equipo de Desarrollo* (típicamente entre tres y nueve personas multifuncionales). El trabajo se organiza en iteraciones de duración fija denominadas **Sprints** (típicamente de dos a cuatro semanas), durante las cuales el equipo se compromete a entregar un incremento de producto potencialmente liberable.

Cada Sprint contiene cinco **eventos formales**: Planificación del Sprint, Scrum Diario (*Daily Scrum*, reunión de quince minutos), Revisión del Sprint, Retrospectiva del Sprint y Refinamiento del Backlog (proceso continuo). Los **artefactos** principales son el Product Backlog (lista priorizada de funcionalidades), el Sprint Backlog (subconjunto de elementos seleccionados para el Sprint actual) y el Incremento (suma de todos los elementos completados).

Las fortalezas de Scrum residen en su claridad estructural, su amplia adopción (lo que facilita la incorporación de personal y la comunicación con otros equipos) y su énfasis en la inspección y adaptación periódicas. Sus limitaciones para el contexto del presente proyecto son significativas: Scrum asume un equipo multifuncional de al menos tres personas, los roles diferenciados (Product Owner, Scrum Master) carecen de sentido en un proyecto unipersonal, y la ceremonia formal de los cinco eventos por Sprint introduce sobrecarga sin beneficio proporcional [59].

3.5.3. Crystal Clear

La familia de metodologías *Crystal* fue desarrollada por Alistair Cockburn a partir de su experiencia consultiva con IBM en la década de 1990, y formalizada en *Crystal Clear: A Human-Powered Methodology for Small Teams* [60]. Crystal es en realidad una familia escalada por color según el tamaño del equipo y la criticidad del sistema: *Clear* (1–6 personas), *Yellow* (7–20), *Orange* (21–40), *Red* (41–80) y *Maroon* (más de 80).

Crystal Clear, la variante más ligera, se enfoca en equipos pequeños desarrollando sistemas no críticos y propone siete **propiedades** que el proceso debe exhibir, ordenadas por importancia: (i) entrega frecuente (cada dos o tres meses), (ii) mejora reflexiva (retrospectivas periódicas), (iii) comunicación osmótica (información que fluye por proximidad), (iv) seguridad personal (libertad para expresar desacuerdos), (v) foco (claridad sobre prioridades y tiempo ininterrumpido), (vi) acceso fácil a usuarios expertos, y (vii) ambiente técnico con pruebas automatizadas, integración continua y gestión de configuración.

A diferencia de XP o Scrum, Crystal Clear no prescribe prácticas concretas, sino que confía en la capacidad del equipo para auto-organizarse y descubrir las prácticas adecuadas a su contexto, siempre que se respeten las siete propiedades. Las fortalezas de Crystal Clear son su ligereza (mínima sobrecarga ceremonial), su énfasis explícito en las personas sobre los procesos, y su escalabilidad. Sus limitaciones son la baja prescripción técnica (que requiere mayor madurez del equipo), la dependencia de la comunicación informal (que se diluye en equipos distribuidos) y la dificultad de evaluar objetivamente su correcta aplicación [49].

3.5.4. Lean Software Development

Lean Software Development fue formulada por Mary y Tom Poppendieck en *Lean Software Development: An Agile Toolkit* [61], como adaptación al dominio del software de los principios del Sistema de Producción Toyota originalmente descritos por Taiichi Ohno [50]. Más que una metodología prescriptiva, Lean es un conjunto de principios y herramientas filosóficas que orientan la toma de decisiones a lo largo del proceso de desarrollo.

Los siete principios de Lean aplicados al software son:

1. **Eliminar el desperdicio** (todo aquello que no agrega valor para el cliente: código no usado, defectos, espera, transferencia de tareas, sobreproducción).
2. **Amplificar el aprendizaje** (mediante iteraciones cortas, prototipos y pruebas tempranas).
3. **Decidir lo más tarde posible** (mantener opciones abiertas hasta que la información sea suficiente).
4. **Entregar lo más rápido posible** (ciclos cortos de desarrollo y retroalimentación).
5. **Empoderar al equipo** (delegar las decisiones técnicas en quienes ejecutan el trabajo).
6. **Construir la integridad incorporada** (calidad como propiedad inherente, no añadida al final).
7. **Ver el todo** (optimizar el sistema completo, no las partes individuales).

Las fortalezas de Lean son su naturaleza universal (los principios son aplicables a cualquier escala y dominio) y su énfasis en la eliminación sistemática del desperdicio. Sus limitaciones son su carácter filosófico antes que prescriptivo: Lean no especifica cómo hacer pruebas, cómo gestionar el código fuente o cómo estructurar el equipo, por lo que requiere combinarse con una metodología prescriptiva (típicamente XP o Scrum) para obtener un marco completo de desarrollo.

3.5.5. Feature-Driven Development (FDD)

Feature-Driven Development (FDD) fue desarrollada por Jeff De Luca y Peter Coad para un proyecto bancario en Singapur en 1997, y posteriormente formalizada por Stephen R. Palmer y John M. Felsing en *A Practical Guide to Feature-Driven Development* [62]. FDD se distingue por organizar el desarrollo en torno a *features* (funcionalidades) granularmente especificadas y por mantener un modelo de dominio explícito durante todo el ciclo de vida.

FDD prescribe cinco **procesos secuenciales**: (1) Desarrollar un Modelo Global (*Develop an Overall Model*), donde el equipo construye un modelo de dominio del problema mediante UML; (2) Construir una Lista de Funcionalidades (*Build a Features List*), que descompone el sistema en funcionalidades expresadas como "`<acción>el <objeto>`" (p. ej. "calcular el total de una orden"); (3) Planificar por Funcionalidad (*Plan by Feature*), asignando funcionalidades a desarrolladores y fechas; (4) Diseñar por Funcionalidad (*Design by Feature*), realizando diagramas de secuencia y diseño detallado; y (5) Construir por Funcionalidad (*Build by Feature*), implementando, probando e integrando cada funcionalidad por separado.

Las fortalezas de FDD son la **trazabilidad** (cada funcionalidad es identificable desde el modelo hasta el código), la **escalabilidad** (es de las pocas metodologías ágiles diseñadas explícitamente para equipos grandes, de cientos de desarrolladores), y su **alineación con UML** (lo que facilita la documentación formal). Sus limitaciones son el peso del proceso de modelado inicial (impráctico para proyectos pequeños o dominios poco conocidos) y la rigidez de la jerarquía de roles (Arquitecto Jefe, Programador Jefe, Programador Propietario de Clase, etc.).

3.5.6. Modelo en Cascada (Waterfall)

El modelo en cascada es el referente tradicional de los métodos secuenciales de desarrollo y fue descrito por primera vez por Winston W. Royce en su artículo *Managing the Development of Large Software Systems* [63]. Es relevante señalar una ironía histórica: Royce presentó el modelo lineal de cascada para *criticarlo* y proponer un modelo iterativo con retroalimentación entre fases, pero la versión simplificada (sin la retroalimentación) se popularizó como la formulación canónica.

El modelo en cascada divide el desarrollo en fases secuenciales, donde cada fase debe completarse antes de iniciar la siguiente:

1. **Análisis de requerimientos:** captura formal de las necesidades del cliente.
2. **Diseño del sistema:** elaboración de la arquitectura y los diagramas de diseño.
3. **Implementación:** escritura del código fuente.
4. **Integración y pruebas:** verificación del sistema completo.
5. **Despliegue y mantenimiento:** entrega al cliente y soporte posterior.

Las fortalezas del modelo en cascada son su simplicidad conceptual, su énfasis en la documentación exhaustiva por fase (lo que facilita auditorías) y su trazabilidad estricta. Sus limitaciones son severas en proyectos con requerimientos volátiles: el costo del cambio crece exponencialmente con la fase en que se detecta, los defectos descubiertos durante la integración son extremadamente costosos de corregir, y el cliente no obtiene valor utilizable hasta el final

del proceso [49], [59]. En la práctica industrial moderna, el modelo en cascada se reserva para proyectos con requerimientos extremadamente estables y dominios de regulación estricta (sistemas críticos de seguridad, certificación aeronáutica).

3.5.7. Proceso Unificado (RUP)

El *Rational Unified Process* (RUP) fue desarrollado por Rational Software Corporation (más tarde adquirida por IBM) a partir del trabajo de Grady Booch, Ivar Jacobson y James Rumbaugh —los tres creadores de UML— y formalizado por Philippe Kruchten en *The Rational Unified Process: An Introduction* [64]. RUP es un marco iterativo e incremental que combina la rigurosidad documental de los métodos tradicionales con elementos de iteración cortos.

RUP organiza el ciclo de vida del proyecto en cuatro **fases** temporales:

1. **Inception (Concepción):** establecimiento del caso de negocio, identificación del alcance y estimación inicial.
2. **Elaboration (Elaboración):** construcción de la arquitectura ejecutable, mitigación de riesgos técnicos principales.
3. **Construction (Construcción):** desarrollo del grueso de funcionalidad, iteraciones de implementación.
4. **Transition (Transición):** despliegue, capacitación y aceptación del usuario final.

Atravesando estas fases, RUP define nueve **disciplinas** (*workflows*): seis de ingeniería (Modelado de Negocio, Requerimientos, Análisis y Diseño, Implementación, Pruebas, Despliegue) y tres de soporte (Gestión de Configuración, Gestión de Proyecto, Entorno). RUP se caracteriza por cuatro principios rectores: ser *dirigido por casos de uso*, *centrado en la arquitectura*, *iterativo e incremental* y *enfocado a riesgos*.

Las fortalezas de RUP son su rigurosidad documental (con plantillas formales para cada artefacto), su énfasis en la arquitectura como artefacto central y su capacidad para coordinar proyectos grandes con múltiples equipos. Sus limitaciones para el contexto del presente proyecto son su **peso documental** (30 roles, 100 artefactos en su versión completa), la **dependencia de herramientas comerciales** (originalmente IBM Rational Suite) y la **baja agilidad** para incorporar cambios tardíos en los requerimientos [56].

3.5.8. Kanban (método de gestión complementario)

Kanban (literalmente, "tablero visual" en japonés) tiene su origen en el Sistema de Producción Toyota descrito por Taiichi Ohno [50], donde fue utilizado para gestionar el flujo de partes en líneas de manufactura mediante señales visuales. Su adaptación al desarrollo de software fue formalizada por David J. Anderson en *Kanban: Successful Evolutionary Change for Your Technology Business* [51], y su relación con Scrum y otros marcos ágiles fue analizada por Henrik Kniberg y Mattias Skarin [52].

Es necesario subrayar que **Kanban no es una metodología de desarrollo de software**, sino un *método de gestión visual del flujo de trabajo*. Kanban no prescribe cómo escribir código, cómo realizar pruebas, cómo refactorizar ni cómo desplegar; se limita a gestionar el flujo de tareas a través del proceso de desarrollo existente. Por esta razón, Kanban se combina típicamente con una metodología prescriptiva (XP, Scrum) que aporte las prácticas de ingeniería [52].

Kanban se articula en torno a cuatro **principios fundamentales** y seis **prácticas básicas** [51]:

- **Principios:** comenzar con lo que se hace actualmente; acordar perseguir el cambio evolutivo e incremental; respetar los procesos, roles y responsabilidades existentes; alentar el liderazgo en todos los niveles.
- **Prácticas:** visualizar el flujo de trabajo (típicamente mediante un tablero con columnas); limitar el trabajo en progreso (WIP); gestionar el flujo; hacer explícitas las políticas; implementar ciclos de retroalimentación; mejorar colaborativamente.

En el contexto del presente proyecto, Kanban se adopta como **método complementario** para la gestión visual de las tareas y la limitación del trabajo en progreso (con WIP máximo de dos tareas simultáneas), pero la metodología de desarrollo propiamente dicha es Extreme Programming, como se justifica en la Sección 3.5.10.

3.5.9. Comparativa de Metodologías

La Tabla 3.6 presenta una comparativa estructurada de las siete metodologías (más Kanban como método complementario) descritas en las subsecciones anteriores, evaluadas respecto a criterios relevantes para el contexto del Trabajo Terminal.

Tabla 3.6: *Comparativa de metodologías de desarrollo de software frente a los criterios del proyecto Kmila.*

Metodología	Tipo	Equipo	Prescr. téc.	Pruebas	Doc. formal	1 dev
Extreme Programming	Ágil prescriptiva	2–10	Alta	Núcleo (TDD)	Baja	Adaptable
Scrum	Ágil ceremonial	3–9	Baja	Media	Media	Forzado
Crystal Clear	Ágil ligera	1–6	Baja	Media	Baja	**Sí**
Lean SD	Filosofía	–	Nula	Media	Baja	Complemento
Feature-Driven Dev.	Ágil estructurada	5–∞	Media	Media	Alta	No
Cascada	Tradicional	–	Baja	Tardía	Alta	Posible
Proceso Unificado	Iterativo formal	10–∞	Media	Media	Muy alta	No
Kanban*	Método de flujo	1–∞	Nula	Nula	Baja	**Sí**

*Kanban no es una metodología de desarrollo; se incluye en la comparativa por su uso frecuente en el sector y por su papel como método complementario.

Los criterios de evaluación corresponden a: **Tipo** (categoría general de la metodología), **Equipo** (rango de tamaño recomendado por sus autores), **Prescr. téc.** (grado de prescripción de prácticas técnicas concretas), **Pruebas** (énfasis explícito en pruebas automatizadas), **Doc. formal** (volumen de documentación requerida) y **1 dev** (idoneidad para un proyecto unipersonal con revisor externo).

3.5.10. Metodología Seleccionada: Extreme Programming

Con base en la comparativa anterior y los criterios específicos del proyecto Kmila, se selecciona **Extreme Programming (XP)** como metodología de desarrollo principal, complementada por **Kanban** como método de gestión visual del trabajo. La justificación se estructura en torno a tres argumentos: la adecuación al contexto del proyecto, la evidencia empírica de su aplicación efectiva a lo largo de TT1, y las adaptaciones necesarias para un equipo unipersonal.

Adecuación al contexto del proyecto

XP es la metodología que mejor encaja con las restricciones y oportunidades del proyecto Kmila por las siguientes razones:

- **Frente a Scrum:** Scrum requiere un equipo de tres a nueve personas y roles diferenciados (Product Owner, Scrum Master), por lo que su adopción en un proyecto unipersonal resultaría en una ceremonialidad vacía sin valor práctico [58].
- **Frente al modelo en cascada:** los requerimientos del dominio (simulación de hardware VHDL) se han refinado progresivamente conforme se ha profundizado en el estándar IEEE 1076, lo que hace inviable un modelo

lineal sin retroalimentación; el modelo en cascada también desincentiva la entrega temprana de valor, en contradicción con la práctica académica de revisiones periódicas con los directores [63].

- **Frente a RUP:** el peso documental de RUP (decenas de artefactos formales, plantillas, roles) es desproporcionado para un proyecto unipersonal de horizonte académico de once meses [64].
- **Frente a Crystal Clear:** aunque Crystal Clear es la metodología más ligera y técnicamente compatible con un proyecto pequeño, su baja prescripción técnica deja la responsabilidad de definir las prácticas en el equipo, lo que en un contexto académico de un solo desarrollador puede llevar a omisiones en disciplina (p. ej. ausencia de pruebas automatizadas). XP, en cambio, prescribe explícitamente TDD, integración continua y refactorización [60].
- **Frente a FDD:** FDD requiere construir un modelo global del dominio en su primera fase, lo que no es viable cuando el dominio (semántica de simulación VHDL) se está aprendiendo progresivamente durante el propio desarrollo [62].
- **Frente a Lean:** Lean es una filosofía universalmente aplicable pero no prescriptiva, por lo que no constituye una metodología completa por sí sola; se incorporan sus principios (eliminación del desperdicio, decidir lo más tarde posible, entrega rápida) como guía transversal, pero no como reemplazo de XP [61].

Evidencia empírica de aplicación durante TT1

La fase de TT1 del proyecto Kmila ya ha aplicado *de facto* prácticas centrales de XP, lo que valida empíricamente su idoneidad. La Tabla 3.7 documenta esta correspondencia.

Tabla 3.7: *Aplicación verificable de prácticas XP durante TT1 del proyecto Kmila.*

Práctica XP	**Evidencia en el proyecto**
Desarrollo guiado por pruebas (TDD)	230/230 pruebas pasando en los módulos Parser, Interpreter, Debugger y TimeMachine al cierre de TT1.
Integración continua	Repositorio Git con commits frecuentes y verificación manual previa a cada *merge* a la rama principal.
Entregas pequeñas (*small releases*)	Versiones incrementales etiquetadas `v1.15`, `v1.16`, `v1.17`, `v1.18`, `v1.19` entregadas durante TT1, cada una con un conjunto coherente de funcionalidades.
Refactorización constante	Reorganización progresiva del Parser (de `Parse()` legado a `ParseV2()`), consolidación del módulo `LibraryCompiler`, separación de responsabilidades en `Kmila.Shared`.
Diseño simple	Arquitectura modular con biblioteca compartida `Kmila.Shared` y separación estricta entre capas (Parser → Interpreter → Debugger → TimeMachine).
Cliente *in situ* (adaptado)	Reuniones recurrentes con los directores del proyecto cumplen el rol del cliente disponible.
Estándar de codificación	Uso uniforme de las convenciones de .NET y C# verificadas mediante el analizador integrado del SDK.

Adaptaciones al contexto unipersonal

Tres prácticas de XP requieren adaptación dado que el equipo de desarrollo es unipersonal. La Tabla 3.8 documenta las adaptaciones adoptadas.

Tabla 3.8: *Adaptaciones de prácticas XP al contexto de un proyecto unipersonal.*

Práctica original	**Adaptación**
Programación en parejas (*pair programming*)	Sustituida por revisiones formales con los directores del proyecto y por la técnica del *rubber duck debugging* para razonamiento estructurado individual.
Cliente *in situ*	Sustituido por reuniones quincenales con los directores; los protocolos de TT y los entregables formales fijan el conjunto de requerimientos.
Propiedad colectiva del código	Autor único, pero el código se mantiene en GitHub como repositorio público a partir de la entrega de TT2, lo que habilita la propiedad colectiva por parte de la comunidad académica futura.

Kanban como complemento

Kanban se adopta como herramienta de gestión visual del flujo de trabajo, complementando a XP. Esta combinación no es novedosa: la literatura especializada documenta híbridos similares (*Scrumban*, XP + Kanban) con resultados favorables para equipos pequeños [52]. En el contexto de Kmila, Kanban aporta:

- **Tablero visual** con columnas *Backlog*, *En progreso*, *En revisión*, *Completado*.
- **Límite de trabajo en progreso (WIP)** de dos tareas simultáneas, para evitar dispersión.
- **Priorización continua** de tareas según el valor pedagógico para el estudiante (interfaz y gestión de proyectos en TT1; motor de simulación y visualización en TT2).
- **Flujo continuo** sin iteraciones de duración fija; los hitos académicos (entregas de TT1 y TT2) regulan naturalmente la cadencia de entrega.

En síntesis, el desarrollo del proyecto Kmila adopta **Extreme Programming como metodología de desarrollo** (que prescribe las prácticas técnicas de pruebas, refactorización, integración continua, diseño simple y entregas pequeñas) y **Kanban como método de gestión visual del flujo de trabajo** (que organiza la priorización y limita el trabajo en progreso). Esta combinación responde a la naturaleza unipersonal, académica e iterativa del proyecto, y se alinea con los principios del *Manifiesto Ágil* [65].

3.6. Cronograma de Actividades

El cronograma del proyecto abarca 11 meses, de febrero a diciembre de 2026, divididos en dos fases correspondientes a Trabajo Terminal 1 (febrero–junio) y Trabajo Terminal 2 (julio–diciembre). Las actividades y su distribución temporal se derivan del protocolo registrado TT 2026-B059 y se gestionan mediante el método Kanban descrito en la Sección 3.5.8, con priorización continua de las tareas según el valor pedagógico aportado al estudiante.

3.6.1. Cronograma General

Tabla 3.9: *Cronograma de actividades del proyecto (febrero – diciembre 2026).*

#	Actividad	FEB	MAR	ABR	MAY	JUN	JUL	AGO	SEP	OCT	NOV	DIC
1	Investigación del estado del arte	•	•									
2	Definición de requerimientos		•									
3	Análisis del alcance y planeación de módulos		•	•								
4	Definición de requerimientos técnicos y versiones compatibles		•	•								
5	Diseño de la arquitectura del sistema (MAUI/.NET, ASP.NET Blazor)	•	•	•								
6	Diseño de la interfaz y editor de código (Monaco)	•	•	•								
7	Desarrollo inicial del motor de simulación HDL			•	•							
8	Documentación preliminar y entrega para **Evaluación TT1**				•	•						
9	Desarrollo del motor de simulación avanzado			•	•		•					
10	Implementación de la visualización de señales y ciclos de reloj						•	•	•			
11	Generación de ejecutables multiplataforma compatibles							•	•	•		
12	Pruebas de validación (compatibilidad, rendimiento, usabilidad)										•	•
13	Elaboración del manual de usuario y documentación técnica final										•	•
14	Publicación del código fuente en GitHub y entrega para **Evaluación TT2**											•

Las actividades 1 a 8 corresponden a la fase de Trabajo Terminal 1 y las actividades 9 a 14 a la fase de Trabajo Terminal 2. La actividad 9 (desarrollo del motor de simulación avanzado) inicia en la fase TT1 con un prototipo inicial y se extiende a TT2 con la implementación completa.

3.6.2. Avance Actual

Al inicio del segundo periodo (Trabajo Terminal 2, julio de 2026), la fase de TT1 se considera completada con la entrega y evaluación favorable del documento correspondiente. El avance del proyecto respecto al cronograma se resume en la Tabla 3.10.

Tabla 3.10: *Estado de avance del cronograma al inicio de TT2 (julio de 2026).*

#	Actividad	Estado
1	Investigación del estado del arte	Completada
2	Definición de requerimientos	Completada
3	Análisis del alcance y planeación de módulos	Completada
4	Definición de requerimientos técnicos y versiones compatibles	Completada
5	Diseño de la arquitectura del sistema (MAUI/.NET, Blazor)	Completada
6	Diseño de la interfaz y editor de código (Monaco)	Completada
7	Desarrollo inicial del motor de simulación HDL	Completada
8	Documentación preliminar y entrega para Evaluación TT1	Completada
9	Desarrollo del motor de simulación avanzado	En curso (transición TT1→TT2)
10	Implementación de la visualización de señales y ciclos de reloj	Iniciada anticipadamente
11	Generación de ejecutables multiplataforma compatibles	Iniciada anticipadamente
12	Pruebas de validación (compatibilidad, rendimiento, usabilidad)	Pendiente
13	Manual de usuario y documentación técnica final	Pendiente
14	Publicación del código fuente y entrega para Evaluación TT2	Pendiente

Las actividades 1 a 8 de la fase de TT1 se completaron dentro del calendario establecido, con el siguiente conjunto de entregables verificables:

- **Documento de TT1 entregado a sinodales** en el formato institucional (LaTeX, ∼159 páginas) en abril–mayo de 2026, con dictamen favorable.
- **Siete versiones incrementales** del prototipo etiquetadas `v1.15` a `v1.19`, con un *changelog* estructurado por versión.
- **Suite de pruebas automatizadas** con 230 pruebas pasando sobre los módulos Parser, Interpreter, Debugger y TimeMachine, lo que constituye la red de seguridad para la fase de refactorización y extensión de TT2.
- **Ejecución verificada en cuatro plataformas** (macOS, iOS, Android y web sobre Chrome), documentada con capturas y métricas en la Sección 3.2.5.

Adicionalmente, varias actividades originalmente previstas para TT2 se han iniciado anticipadamente. El motor de simulación avanzado (actividad 9) ya cuenta con un prototipo funcional integrado en las versiones `v1.18` y `v1.19`, e incluye el módulo `LibraryCompiler` y el subsistema *Concepts* (un módulo pedagógico complementario para enseñar las diferencias entre ejecución secuencial y concurrente en VHDL). La visualización de formas de onda (actividad 10) cuenta también con un componente funcional integrado en el panel de simulación. Esta anticipación permite reasignar margen de tiempo en TT2 a las actividades de validación (actividad 12) y a la fase de publicación (actividad 14), que históricamente concentran la mayor parte de los riesgos de cierre.

El proyecto se encuentra en línea con los compromisos del protocolo registrado y con un margen de holgura

favorable para abordar el alcance restante de TT2.

Capítulo 4

Análisis del Sistema

4.1. Modelo del Dominio

El modelo del dominio identifica los conceptos fundamentales del sistema y las relaciones entre ellos. Estos conceptos se derivan del análisis del problema descrito en la sección 1.5 y del alcance definido en la sección 1.8. Las Tablas 4.1 a 4.5 presentan las entidades del dominio agrupadas por categoría.

Tabla 4.1: *Entidades del dominio: gestión de proyecto.*

Entidad	Descripción
Proyecto	Contenedor lógico que agrupa archivos HDL, configuración de simulación y metadatos del usuario.
Archivo HDL	Archivo fuente en VHDL (`.vhd`) que contiene la descripción de un circuito digital. Un proyecto puede contener múltiples archivos.
Sesión	Estado persistente del espacio de trabajo del usuario, incluyendo archivos abiertos, posición del cursor y configuración activa.

Tabla 4.2: *Entidades del dominio: modelo HDL.*

Entidad	Descripción
Entidad (Entity)	Declaración VHDL que define la interfaz externa de un módulo: su nombre y sus puertos de entrada/salida.
Arquitectura (Architecture)	Cuerpo VHDL asociado a una entidad que describe su comportamiento o estructura interna.
Puerto (Port)	Punto de conexión de entrada (`in`), salida (`out`) o bidireccional (`inout`) declarado en una entidad.
Señal (Signal)	Elemento que transporta valores lógicos entre componentes dentro de una arquitectura.
Proceso (Process)	Bloque de ejecución secuencial dentro de una arquitectura, activado por una lista de sensibilidad.
Banco de pruebas (Testbench)	Archivo HDL especial que instancia la entidad bajo prueba y genera estímulos de entrada para la simulación.

Tabla 4.3: *Entidades del dominio: análisis de código.*

Entidad	Descripción
AST (Abstract Syntax Tree)	Representación jerárquica intermedia del código HDL generada tras el análisis sintáctico.
Diagnóstico	Mensaje de error, advertencia o información generado durante el análisis del código, asociado a una línea y columna específicas.
Punto de interrupción (Breakpoint)	Marca insertada por el usuario en una línea del código que detiene la simulación al ser alcanzada.
Punto de interrupción de condición	Punto de interrupción sobre una rama `if`/`elsif`/`case` que detiene la simulación cuando esa rama es tomada, aunque no cambie ninguna señal.
Punto de interrupción no rastreable	Punto de interrupción colocado en una línea sin asignación de señal ni condición (declaración, comentario, línea en blanco), que nunca detendría la ejecución; la interfaz lo señala con un glifo atenuado.

Tabla 4.4: *Entidades del dominio: simulación.*

Entidad	Descripción
Configuración de simulación	Conjunto de parámetros que definen la ejecución: tiempo máximo, resolución temporal, dispositivo FPGA objetivo y señales a observar.
Dispositivo FPGA	Modelo de referencia de un dispositivo FPGA comercial (familia, fabricante, cantidad de celdas lógicas, bloques de memoria) utilizado como contexto de la simulación.
Ciclo de simulación	Unidad discreta de avance temporal compuesta por una fase de actualización de señales y una fase de ejecución de procesos.
Forma de onda (Waveform)	Serie temporal de valores lógicos de una señal a lo largo de los ciclos de simulación.
Runtime visual (Visual Runtime)	Modo de simulación interactiva de corrida infinita en el que los puertos del diseño se enlazan a componentes virtuales sobre un lienzo, reutilizando el motor de simulación.
Componente virtual (Widget)	Elemento gráfico del *Visual Runtime* (LED, interruptor, botón, siete segmentos, motor paso a paso) que lee o impone el valor de un puerto del diseño en tiempo de ejecución.
Archivo `.kviz`	Archivo *sidecar* en formato JSON que persiste el estado del lienzo del *Visual Runtime* (componentes, posiciones y enlaces pin→puerto), independiente del código VHDL.

Tabla 4.5: *Entidades del dominio: interfaz de usuario.*

Entidad	Descripción
Editor de código	Componente central de la interfaz basado en Monaco Editor que permite la escritura y edición de código HDL con resaltado de sintaxis.
Visualizador de ondas	Componente gráfico basado en SVG en línea que presenta las formas de onda de las señales seleccionadas en un diagrama temporal.
Panel de diagnóstico	Componente que muestra la lista de errores, advertencias e información generados por el análisis del código.

Relaciones principales. Un **Proyecto** contiene uno o más **Archivos HDL**, cada uno de los cuales puede declarar una o más **Entidades**. Cada Entidad posee una **Arquitectura** asociada que contiene **Señales** y **Procesos**. Un **Banco de pruebas** instancia una Entidad y define los estímulos para la simulación. El análisis de un Archivo HDL produce un **AST** y, en caso de errores, uno o más **Diagnósticos**. La ejecución de la simulación, controlada por una **Configuración de simulación**, genera **Formas de onda** para cada señal observada, las cuales se presentan al usuario a través del **Visualizador de ondas**.

La Figura 4.1 presenta el diagrama del modelo del dominio con las entidades y relaciones descritas.

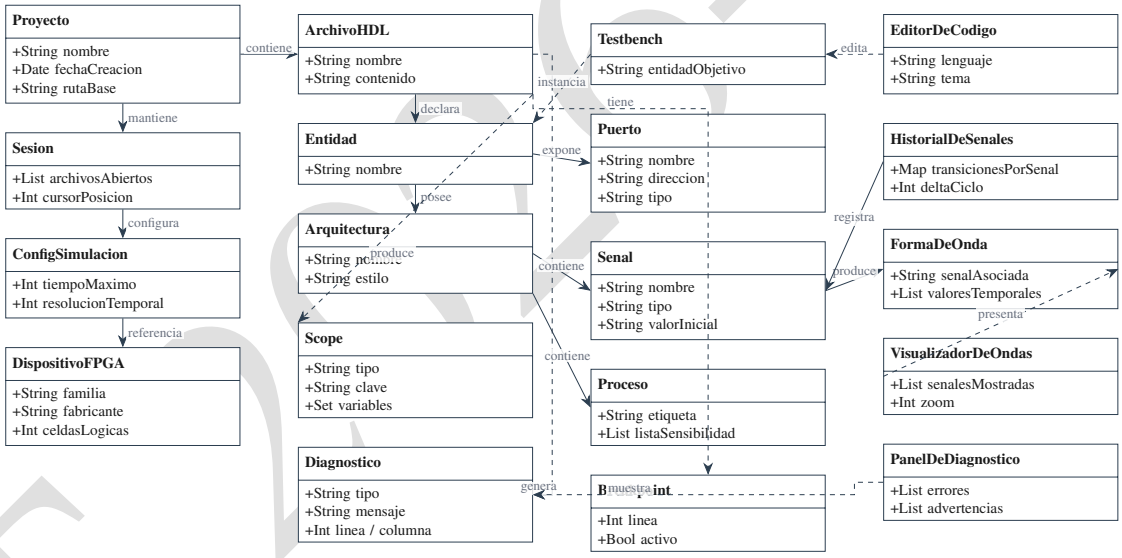


Figura 4.1: *Modelo del dominio completo del sistema, integrando la gestión de proyectos, el modelo HDL (entidad, arquitectura, puertos, señales, procesos), el análisis de código (`Scope`, `Variable`, `Diagnostico`) y la simulación (`HistorialDeSeñales` y ciclo delta).*

Las Figuras 4.2 a 4.5 presentan vistas segmentadas del modelo del dominio por categoría, facilitando su lectura individual.

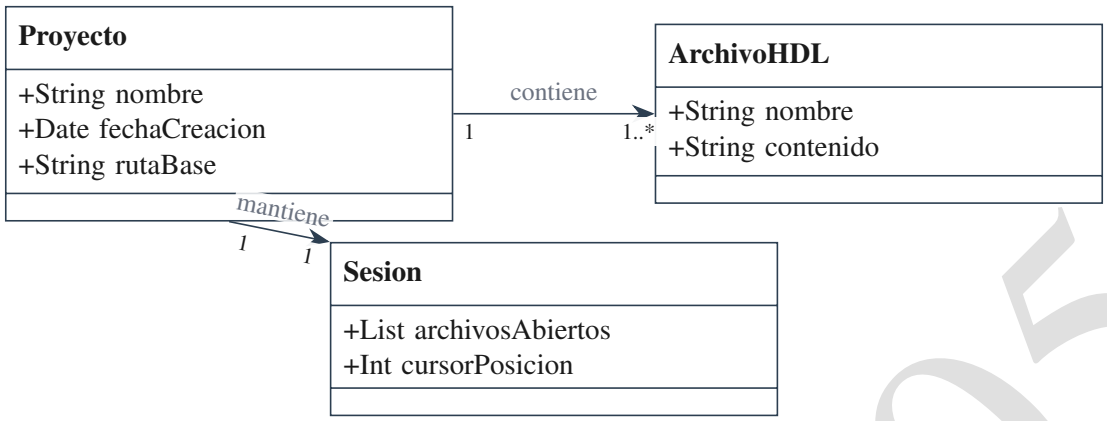


Figura 4.2: *Modelo del dominio — Gestión de proyecto.*

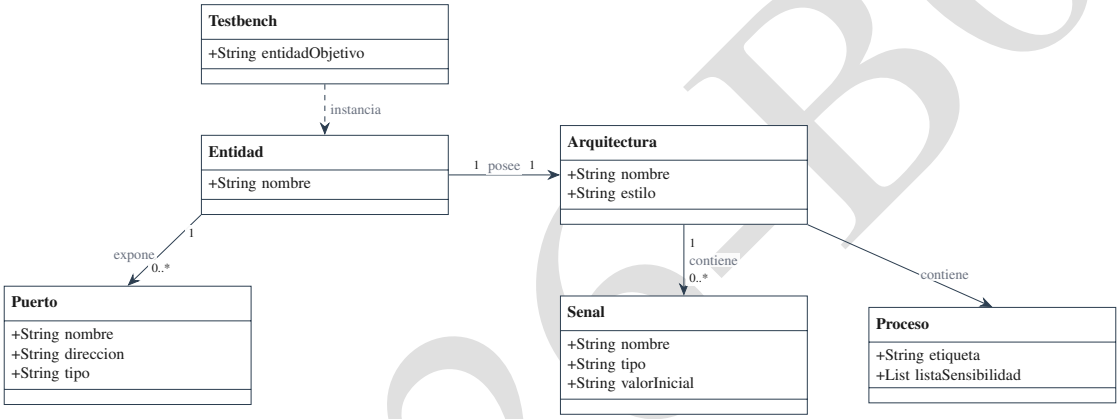


Figura 4.3: *Modelo del dominio — Modelo HDL.*

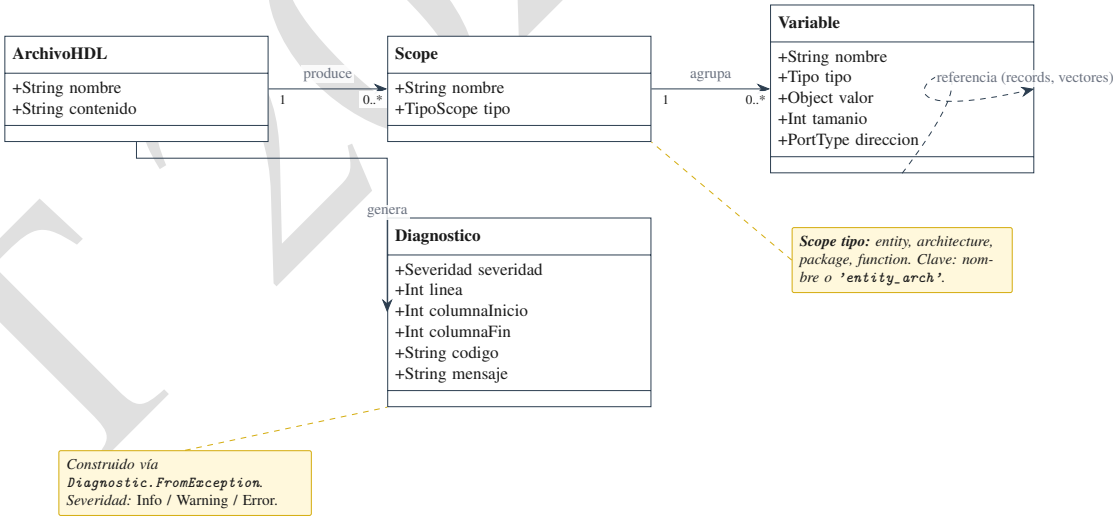


Figura 4.4: *Modelo del dominio — Análisis de código. El analisis sintactico actual produce un diccionario plano de Scopes (entidades, arquitecturas, paquetes, funciones), cada uno asociado a un conjunto de Variables; los Diagnosticos son objetos estructurados independientes del arbol sintactico.*

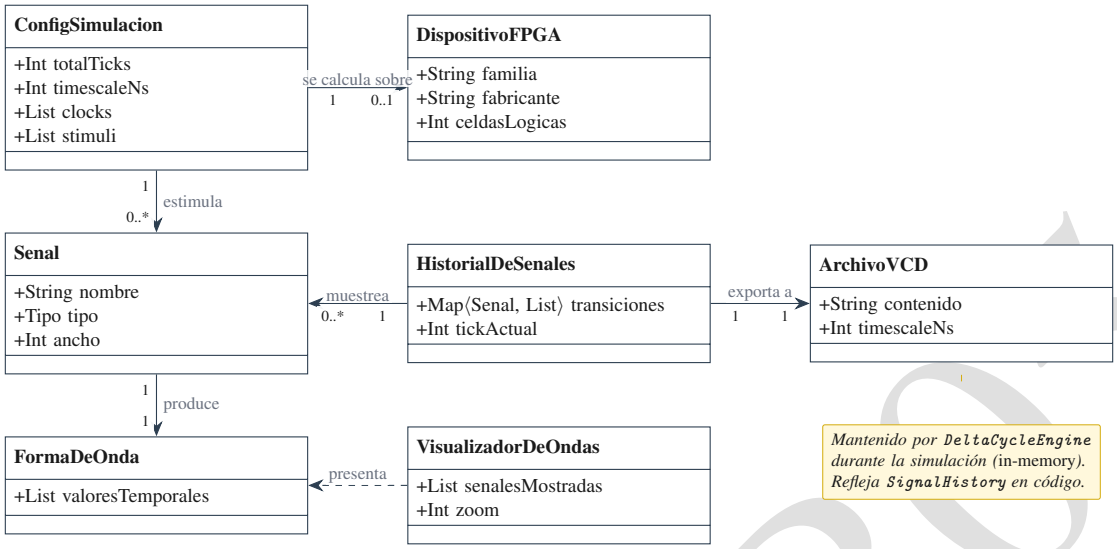


Figura 4.5: *Modelo del dominio — Simulación y visualización. El HistorialDeSenales (corresponde a SignalHistory en codigo) es muestreado por el motor de simulacion durante la ejecucion y exportado a ArchivoVCD para visores externos como GTKWave.*

4.2. Requerimientos Funcionales

Los requerimientos funcionales se organizan en dos grupos correspondientes a las fases del proyecto. Los requerimientos de TT1 (interfaz de usuario y gestión) se detallan en su totalidad, dado que constituyen el alcance del presente trabajo. Los requerimientos de TT2 (motor de simulación) se enuncian para completitud del análisis.

Tabla 4.6: *Resumen de requerimientos funcionales — TT1 (interfaz de usuario y gestión).*

ID	Nombre	Descripción	Prioridad
RF-01	Gestionar proyectos	Crear, abrir, eliminar y listar proyectos que agrupen archivos HDL y configuración	Alta
RF-02	Gestionar archivos HDL	Crear, cargar y guardar archivos `.vhd` dentro de un proyecto con respaldo en base de datos	Alta
RF-03	Editar código HDL	Editar archivos VHDL con resaltado de sintaxis, seguimiento de posición del cursor e indicador de cambios sin guardar	Alta
RF-04	Navegar estructura del proyecto	Explorar archivos del proyecto mediante un panel lateral y una barra de navegación superior	Alta
RF-05	Seleccionar dispositivo FPGA	Elegir un modelo de FPGA de referencia de un catálogo de 34 dispositivos de 3 fabricantes	Media
RF-06	Configurar parámetros de simulación	Definir duración, escala temporal, frecuencia de reloj y estímulos por puerto	Media
RF-07	Visualizar diagnósticos y recursos	Mostrar entidades analizadas, puertos, señales y utilización de recursos del FPGA seleccionado	Alta
RF-08	Visualizar formas de onda	Presentar señales en un visor de ondas con codificación por color, zoom y exportación VCD	Alta
RF-09	Cambiar idioma de la interfaz	Seleccionar entre 7 idiomas (ES, EN, FR, DE, ZH, JA, AR) con dirección de escritura derecha-a-izquierda (*Right-to-Left*) para AR	Baja
RF-10	Cambiar tema visual	Alternar entre tema claro y tema oscuro	Baja
RF-11	Persistir datos localmente	Almacenar proyectos, archivos, configuraciones y preferencias del usuario en SQLite	Media

Tabla 4.7: *Resumen de requerimientos funcionales — TT2 (motor de simulación).*

ID	Nombre	Descripción	Prioridad
RF-12	Analizar sintaxis VHDL	Verificar la corrección sintáctica del código y generar diagnósticos estructurados (severidad, línea, columna, código)	Alta
RF-13	Generar representación intermedia	Construir el diccionario de *scopes* con entidades, arquitecturas, paquetes y variables a partir del código VHDL válido	Alta
RF-14	Ejecutar simulación	Simular la ejecución del circuito mediante el modelo de eventos discretos con ciclos delta	Alta
RF-15	Controlar simulación	Iniciar, pausar, reanudar y detener la ejecución de la simulación	Alta
RF-16	Ejecutar paso a paso	Avanzar la simulación ciclo por ciclo para depuración interactiva mediante el módulo `TimeMachine`	Media
RF-17	Insertar puntos de interrupción	Marcar líneas en el editor donde la simulación debe detenerse durante la repro-	Media

A continuación se detalla cada requerimiento funcional de TT1.

4.2.1. RF-01: Gestionar Proyectos

Campo	Descripción
ID	RF-01
Nombre	Gestionar proyectos
Descripción	El sistema debe permitir al usuario crear un nuevo proyecto (con nombre, descripción y ruta base), abrir un proyecto existente, eliminarlo y consultar la lista de proyectos registrados. La creación de un proyecto genera un directorio en el sistema de archivos y un registro en la base de datos local.
Prioridad	Alta
Entrada	Nombre, descripción y ruta de almacenamiento (creación); identificador del proyecto (apertura/eliminación)
Salida	Proyecto cargado en la interfaz con su estructura de archivos visible en el editor
Precondiciones	La aplicación se encuentra en ejecución
Postcondiciones	El proyecto queda registrado en la tabla `Projects` de SQLite y su directorio existe en el sistema de archivos

4.2.2. RF-02: Gestionar Archivos HDL

Campo	Descripción
ID	RF-02
Nombre	Gestionar archivos HDL
Descripción	El sistema debe permitir crear nuevos archivos `.vhd` dentro de un proyecto, cargar archivos existentes desde el directorio del proyecto y guardar los cambios realizados. El contenido se almacena tanto en disco como en la base de datos SQLite como respaldo, con mecanismo de recuperación ante pérdida del archivo en disco.
Prioridad	Alta
Entrada	Nombre del archivo (creación); contenido editado (guardado)
Salida	Archivo registrado en la tabla `ProjectFiles` y escrito en el directorio del proyecto
Precondiciones	Existe un proyecto abierto
Postcondiciones	El archivo es accesible desde el panel de archivos del editor y su contenido persiste en disco y base de datos

4.2.3. RF-03: Editar Código HDL

Campo	Descripción
ID	RF-03
Nombre	Editar código HDL
Descripción	El sistema debe integrar el editor Monaco (BlazorMonaco 3.3.0) para la edición de archivos VHDL, proporcionando resaltado de sintaxis específico para VHDL, seguimiento de la posición del cursor (línea y columna), indicador visual de cambios sin guardar, detección del tipo de archivo por extensión y atajo de teclado Ctrl+S para guardar. Para archivos Markdown, el editor ofrece vista dividida con previsualización.
Prioridad	Alta
Entrada	Contenido del archivo HDL seleccionado
Salida	Contenido editado con resaltado de sintaxis y metadatos de posición
Precondiciones	Existe al menos un archivo abierto en el proyecto activo
Postcondiciones	Los cambios se reflejan en el editor; el indicador de cambios sin guardar se activa hasta que el usuario guarde

4.2.4. RF-04: Navegar Estructura del Proyecto

Campo	Descripción
ID	RF-04
Nombre	Navegar estructura del proyecto
Descripción	El sistema debe proporcionar un panel de navegación lateral (sidebar) con acceso a las secciones principales (Inicio, Proyectos, Configuración) y, dentro del editor, un listado de los archivos del proyecto activo con iconos diferenciados por tipo de archivo. La barra superior (TopBar) muestra el dispositivo FPGA seleccionado y el selector de idioma.
Prioridad	Alta
Entrada	Interacción del usuario con los elementos de navegación
Salida	Vista correspondiente a la sección o archivo seleccionado
Precondiciones	La aplicación se encuentra en ejecución
Postcondiciones	La interfaz muestra el contenido solicitado sin pérdida del estado de las demás secciones

4.2.5. RF-05: Seleccionar Dispositivo FPGA

Campo	Descripción
ID	RF-05
Nombre	Seleccionar dispositivo FPGA
Descripción	El sistema debe permitir al usuario seleccionar un modelo de FPGA de referencia a partir de un catálogo de 34 dispositivos de tres fabricantes (Intel/Altera Cyclone IV E, Xilinx Spartan-6 y Artix-7, Lattice iCE40 y ECP5). La información de cada dispositivo se carga desde archivos JSON e incluye: modelo, familia, fabricante, recursos lógicos (LUTs, bloques de memoria, flip-flops, DSPs), especificaciones de reloj (PLLs, frecuencia máxima) e información de E/S.
Prioridad	Media
Entrada	Selección del dispositivo mediante menú desplegable en la barra superior
Salida	Dispositivo activo actualizado en el servicio SelectedFPGA; vista de detalle disponible en la pestaña FPGA
Precondiciones	Los archivos JSON de dispositivos se encuentran disponibles en el directorio Devices
Postcondiciones	El dispositivo seleccionado se utiliza como referencia para el cálculo de utilización de recursos

4.2.6. RF-06: Configurar Parámetros de Simulación

Campo	Descripción
ID	RF-06
Nombre	Configurar parámetros de simulación
Descripción	El sistema debe permitir al usuario definir los parámetros de simulación mediante un diálogo modal que incluye: duración de la simulación, escala temporal (NS, US, MS, S, M, H), frecuencia de reloj en MHz para señales de tipo CLK, y configuración de estímulos por puerto (pares tiempo-valor). Las configuraciones se almacenan en formato JSON en la tabla `AppSettings` de SQLite con extensión `.k-config`.
Prioridad	Media
Entrada	Valores de duración, escala, frecuencia y estímulos ingresados por el usuario
Salida	Objeto SimulationConfig construido por StimulusBuilder con los parámetros convertidos a ticks internos
Precondiciones	Existe un archivo HDL analizado con entidades y puertos identificados
Postcondiciones	La configuración persiste en SQLite y se puede recuperar en sesiones futuras

4.2.7. RF-07: Visualizar Diagnósticos y Recursos

Campo	Descripción
ID	RF-07
Nombre	Visualizar diagnósticos y recursos
Descripción	El sistema debe mostrar la información de las entidades VHDL analizadas (arquitectura, puertos, señales internas) y la utilización estimada de recursos respecto al FPGA seleccionado. La utilización se presenta mediante barras de progreso con codificación por color (verde: uso aceptable, amarillo: advertencia, rojo: capacidad excedida) para LUTs, bloques de memoria, flip-flops y DSPs. El componente de progreso de compilación muestra el avance en tiempo real con mensajes de estado y reporte de errores.
Prioridad	Alta
Entrada	Resultado del análisis de una entidad VHDL y dispositivo FPGA seleccionado
Salida	Panel con información de la entidad y porcentajes de utilización de recursos
Precondiciones	Se ha realizado el análisis de al menos un archivo HDL del proyecto
Postcondiciones	El usuario puede identificar si su diseño cabe en el dispositivo FPGA seleccionado

4.2.8. RF-08: Visualizar Formas de Onda

Campo	Descripción
ID	RF-08
Nombre	Visualizar formas de onda
Descripción	El sistema debe presentar las señales del circuito simulado en un visor de ondas basado en SVG en línea con trazas de línea escalonada. Cada señal se muestra en un gráfico independiente con codificación por color según su tipo: verde para señales de reloj, azul para entradas, naranja para salidas y morado para vectores. El visor soporta tipos `std_logic`, `std_logic_vector` e `integer`, e incluye controles de zoom (acercar/alejar), desplazamiento horizontal, etiquetado inteligente del eje temporal (ns, us, ms, s) y botón de exportación a formato VCD.
Prioridad	Alta
Entrada	Historial de señales (SignalHistory) generado por el motor de simulación
Salida	Conjunto de gráficos escalonados con las formas de onda de las señales observadas
Precondiciones	Se ha ejecutado una simulación que produjo datos de señales
Postcondiciones	Las formas de onda se presentan al usuario y pueden ser exportadas en formato VCD

4.2.9. RF-09: Cambiar Idioma de la Interfaz

Campo	Descripción
ID	RF-09
Nombre	Cambiar idioma de la interfaz
Descripción	El sistema debe permitir al usuario seleccionar entre 7 idiomas: español (ES), inglés (EN), francés (FR), alemán (DE), chino simplificado (ZH), japonés (JA) y árabe (AR). El cambio de idioma se realiza de forma reactiva sin recarga de página, mediante el servicio Traductor que carga diccionarios JSON con más de 40 claves de traducción. Para el idioma árabe, la interfaz activa el atributo `dir=rtl"` para soporte de escritura de derecha a izquierda. La preferencia se almacena en SQLite.
Prioridad	Baja
Entrada	Código de idioma seleccionado por el usuario
Salida	Interfaz actualizada con las traducciones del idioma seleccionado
Precondiciones	Los archivos de diccionario JSON se encuentran disponibles en el directorio Dictionaries
Postcondiciones	La preferencia persiste en la tabla `AppSettings` y se aplica automáticamente en sesiones futuras

4.2.10. RF-10: Cambiar Tema Visual

Campo	Descripción
ID	RF-10
Nombre	Cambiar tema visual
Descripción	El sistema debe permitir al usuario alternar entre un tema claro y un tema oscuro. La infraestructura de temas se implementa mediante propiedades CSS personalizadas (`--kmila-primary`, `--kmila-primary-light`, `--kmila-primary-dark`, `--kmila-accent`) definidas en la hoja de estilos global. La preferencia se almacena en SQLite mediante la clave `KEY_THEME` de `AppSettings`.
Prioridad	Baja
Entrada	Selección del tema por el usuario
Salida	Interfaz actualizada con la paleta de colores correspondiente
Precondiciones	La aplicación se encuentra en ejecución
Postcondiciones	La preferencia persiste en SQLite y se aplica automáticamente en sesiones futuras

4.2.11. RF-11: Persistir Datos Localmente

Campo	Descripción
ID	RF-11
Nombre	Persistir datos localmente
Descripción	El sistema debe almacenar de forma persistente toda la información del usuario mediante SQLite (sqlite-net-pcl 1.9.172). La base de datos contiene tres tablas: `Projects` (metadatos de proyectos), `ProjectFiles` (archivos con respaldo de contenido) y `AppSettings` (pares clave-valor para configuración, idioma, tema y parámetros de simulación). El sistema de eventos `StaticOnValueChanged` notifica a los componentes suscritos cuando un valor de configuración cambia, permitiendo actualizaciones reactivas sin recarga.
Prioridad	Media
Entrada	Datos generados por las operaciones del usuario (proyectos, archivos, configuraciones)
Salida	Registros almacenados en la base de datos SQLite local (`kmila.db`)
Precondiciones	La base de datos ha sido inicializada mediante `InitDBAsync` al inicio de la aplicación
Postcondiciones	Los datos persisten entre sesiones y son recuperables ante cierre inesperado de la aplicación

4.3. Requerimientos No Funcionales

Tabla 4.8: *Requerimientos no funcionales: plataforma y experiencia de usuario.*

ID	Categoría	Descripción	Métrica / Criterio
RNF-01	Multiplataforma	El sistema debe ejecutarse en al menos dos modalidades de despliegue: aplicación nativa mediante .NET MAUI (Android API 24+) y aplicación web mediante ASP.NET Core accesible desde cualquier navegador moderno.	Ejecución verificada en Android (MAUI) y navegador web (ASP.NET Core)
RNF-02	Usabilidad	La interfaz debe ser comprensible para alumnos de ISC que cursan asignaturas de diseño digital sin experiencia previa en simuladores HDL profesionales. La disposición debe seguir convenciones de IDEs.	Un alumno de ISC puede crear un proyecto, abrir un archivo `.vhd` y navegar la interfaz en menos de 10 minutos sin asistencia
RNF-03	Rendimiento	Las operaciones de interfaz deben completarse sin retrasos perceptibles. La carga del catálogo de 34 dispositivos FPGA desde archivos JSON no debe bloquear la interfaz.	Tiempo de respuesta de operaciones de UI <1 segundo
RNF-04	Portabilidad web	La versión web no debe requerir plugins, extensiones ni dependencias externas. Toda la funcionalidad debe ejecutarse directamente en el navegador.	Ejecución completa en Chrome, Firefox y Edge sin software adicional

Tabla 4.9: *Requerimientos no funcionales: internacionalización y confiabilidad.*

ID	Categoría	Descripción	Métrica / Criterio
RNF-05	Internacionalización	La interfaz debe soportar múltiples idiomas con cambio reactivo (sin recarga) y soporte RTL. Los textos deben obtenerse de diccionarios JSON externos.	7 idiomas funcionales (ES, EN, FR, DE, ZH, JA, AR) con RTL verificado en AR

ID	Categoría	Descripción	Métrica / Criterio
RNF-06	Confiabilidad	El sistema debe manejar errores de entrada sin interrumpir la ejecución. El contenido debe almacenarse con respaldo dual (disco + SQLite).	0 cierres inesperados por errores de entrada; recuperación verificada
RNF-07	Mantenibilidad	El código debe organizarse en una biblioteca compartida (Kmila.Shared) consumida por ambos hosts, evitando duplicación. Servicios registrados mediante inyección de dependencias.	Una modificación en Kmila.Shared se refleja simultáneamente en ambas plataformas

Tabla 4.10: *Requerimientos no funcionales: extensibilidad y compatibilidad.*

ID	Categoría	Descripción	Métrica / Criterio
RNF-08	Extensibilidad	El catálogo FPGA debe ser extensible mediante archivos JSON en el directorio Devices, sin requerir recompilación.	Agregar un dispositivo requiere solo crear un archivo JSON
RNF-09	Persistencia	Toda la información del usuario debe persistir entre sesiones mediante SQLite local. Los cambios en configuración deben notificarse mediante eventos reactivos.	Datos persisten tras cierre y reapertura; cambios de idioma/tema sin recarga
RNF-10	Compatibilidad	El sistema debe construirse sobre .NET 9.0 con Blazor.Bootstrap 3.4.0, BlazorMonaco 3.3.0 y sqlite-net-pcl 1.9.172.	Compilación exitosa con las versiones especificadas en `.csproj`

4.4. Historias de Usuario

Las historias de usuario se redactan desde la perspectiva de un alumno de ISC que cursa asignaturas de diseño digital (Fundamentos de Diseño Digital, Diseño de Sistemas Digitales, Arquitectura de Computadoras, Sistemas en Chip). Se organizan en dos grupos: TT1 (interfaz y gestión) y TT2 (simulación y depuración).

4.4.1. Historias de Usuario — TT1

4.4.1.1. HU-01: Crear Proyecto de Práctica

Campo	Descripción
ID	HU-01
RF relacionado	RF-01
Rol	Alumno de ISC
Historia	Como alumno, quiero crear un proyecto nuevo para organizar los archivos VHDL de mi práctica de laboratorio, para tener mi trabajo separado por asignatura y ejercicio.
Criterios de aceptación	▪ Puedo asignar un nombre y descripción al proyecto ▪ Se crea un directorio en mi equipo con la estructura del proyecto ▪ El proyecto aparece en la lista de proyectos recientes

4.4.1.2. HU-02: Escribir Código VHDL

Campo	Descripción
ID	HU-02
RF relacionado	RF-02, RF-03
Rol	Alumno de ISC
Historia	Como alumno, quiero crear un archivo `.vhd` dentro de mi proyecto y escribir código VHDL con resaltado de sintaxis, para desarrollar mi diseño digital con asistencia visual.
Criterios de aceptación	▪ Puedo crear un nuevo archivo `.vhd` desde la interfaz ▪ El editor resalta las palabras reservadas de VHDL (`entity`, `architecture`, `signal`, `process`, etc.) ▪ Puedo ver la línea y columna del cursor en todo momento ▪ Se indica visualmente cuando hay cambios sin guardar

4.4.1.3. HU-03: Abrir Práctica Anterior

Campo	Descripción
ID	HU-03
RF relacionado	RF-01, RF-02
Rol	Alumno de ISC
Historia	Como alumno, quiero abrir un proyecto existente y recuperar los archivos que ya escribí, para continuar una práctica de una sesión anterior.
Criterios de aceptación	▪ La lista de proyectos recientes muestra mis proyectos anteriores con fecha de creación ▪ Al abrir un proyecto, los archivos se cargan desde disco ▪ Si un archivo se perdió del disco, se recupera desde el respaldo en base de datos

4.4.1.4. HU-04: Navegar Archivos del Proyecto

Campo	Descripción
ID	HU-04
RF relacionado	RF-04
Rol	Alumno de ISC
Historia	Como alumno, quiero ver todos los archivos de mi proyecto en un panel lateral y cambiar entre ellos, para trabajar con múltiples módulos VHDL (entidad principal, testbench, componentes).
Criterios de aceptación	▪ Los archivos se listan con iconos diferenciados por tipo ▪ Puedo cambiar de archivo con un clic ▪ La barra lateral muestra las secciones principales de la aplicación (Inicio, Proyectos, Configuración)

4.4.1.5. HU-05: Seleccionar FPGA del Laboratorio

Campo	Descripción
ID	HU-05
RF relacionado	RF-05
Rol	Alumno de ISC
Historia	Como alumno, quiero seleccionar el modelo de FPGA que usamos en el laboratorio (por ejemplo, Cyclone IV EP4CE6 o Artix-7 XC7A35T), para verificar que mi diseño es compatible con los recursos disponibles en la tarjeta.
Criterios de aceptación	▪ Puedo elegir entre dispositivos de Intel/Altera, Xilinx y Lattice desde un menú desplegable ▪ Se muestran las especificaciones del dispositivo (LUTs, memoria, flip-flops, DSPs) ▪ La selección se mantiene mientras trabajo en el proyecto

4.4.1.6. HU-06: Configurar Estímulos de Entrada

Campo	Descripción
ID	HU-06
RF relacionado	RF-06
Rol	Alumno de ISC
Historia	Como alumno, quiero configurar la frecuencia del reloj y los valores de las señales de entrada en distintos momentos de tiempo, para definir el escenario de prueba de mi circuito antes de simularlo.
Criterios de aceptación	▪ Puedo definir la frecuencia del reloj en MHz ▪ Puedo agregar pares tiempo-valor para cada puerto de entrada ▪ Puedo elegir la escala temporal (ns, us, ms, s) ▪ La configuración se guarda y puedo recuperarla después

4.4.1.7. HU-07: Revisar Diagnósticos y Utilización

Campo	Descripción
ID	HU-07
RF relacionado	RF-07
Rol	Alumno de ISC
Historia	Como alumno, quiero ver la información de las entidades de mi diseño (puertos, señales, arquitectura) y el porcentaje de recursos que consume respecto al FPGA seleccionado, para saber si mi diseño cabe en la tarjeta antes de probarlo en el laboratorio.
Criterios de aceptación	▪ Se listan las entidades, puertos y señales internas del diseño ▪ Las barras de utilización muestran el porcentaje de LUTs, memoria, flip-flops y DSPs consumidos ▪ Se indica con color si la utilización es aceptable (verde), cercana al límite (amarillo) o excedida (rojo)

4.4.1.8. HU-08: Ver Formas de Onda

Campo	Descripción
ID	HU-08
RF relacionado	RF-08
Rol	Alumno de ISC
Historia	Como alumno, quiero ver las formas de onda de las señales de mi circuito tras la simulación, para verificar visualmente que mi diseño se comporta como esperaba en la práctica.
Criterios de aceptación	▪ Cada señal se muestra en un gráfico de línea escalonada independiente ▪ Las señales se distinguen por color (verde=reloj, azul=entrada, naranja=salida, morado=vector) ▪ Puedo acercar y alejar el eje temporal ▪ Puedo exportar las formas de onda en formato VCD

4.4.1.9. HU-09: Usar la Aplicación en mi Idioma

Campo	Descripción
ID	HU-09
RF relacionado	RF-09
Rol	Alumno de ISC
Historia	Como alumno, quiero cambiar el idioma de la interfaz al español u otro idioma de mi preferencia, para entender todas las opciones y mensajes de la aplicación.
Criterios de aceptación	▪ Puedo seleccionar entre 7 idiomas disponibles ▪ La interfaz se actualiza inmediatamente sin recargar la página ▪ Mi preferencia se recuerda para la siguiente sesión

4.4.1.10. HU-10: Personalizar el Tema Visual

Campo	Descripción
ID	HU-10
RF relacionado	RF-10
Rol	Alumno de ISC
Historia	Como alumno, quiero alternar entre un tema claro y uno oscuro, para adaptar la interfaz a mi preferencia visual o a las condiciones de iluminación del laboratorio.
Criterios de aceptación	▪ Puedo cambiar entre tema claro y oscuro ▪ El cambio se aplica a toda la interfaz ▪ Mi preferencia se recuerda para la siguiente sesión

4.4.2. Historias de Usuario — TT2

Las siguientes historias de usuario corresponden a funcionalidades del motor de simulación, cuya implementación se contempla en la segunda fase del proyecto. Se enuncian para completitud del análisis.

Tabla 4.11: *Resumen de historias de usuario de TT2.*

ID	Historia resumida	RF relacionado
HU-11	Como alumno, quiero que la aplicación detecte errores de sintaxis en mi código VHDL antes de simular, para corregirlos a tiempo.	RF-12, RF-13
HU-12	Como alumno, quiero ejecutar la simulación de mi circuito y ver cómo se propagan las señales, para verificar mi diseño sin necesidad de la tarjeta FPGA.	RF-14, RF-15
HU-13	Como alumno, quiero avanzar la simulación ciclo por ciclo, para entender exactamente cómo cambian las señales en cada paso de tiempo.	RF-16
HU-14	Como alumno, quiero pausar y reanudar la simulación, para analizar el estado del circuito en un momento específico.	RF-15
HU-15	Como alumno, quiero insertar puntos de interrupción en mi código, para que la simulación se detenga automáticamente en las líneas que quiero depurar.	RF-17
HU-16	Como alumno, quiero ver mi diseño como un esquemático de compuertas con tres modos (*naive*, sintético y *debug*), para complementar la vista RTL del editor y entender la traducción del código al circuito.	RF-19
HU-17	Como alumno, quiero ver el diagrama de flujo de mi código (procesos, ramas, asignaciones), para visualizar la estructura de control antes de simular.	RF-20
HU-18	Como alumno, quiero reproducir una corrida completada paso a paso con resaltado de líneas, para revisar después de la simulación qué hizo cada señal y en qué momento del código.	RF-21
HU-19	Como alumno, quiero comparar dos corridas distintas de mi proyecto, para entender el efecto de un cambio en el código o en los estímulos.	RF-22, RF-23
HU-20	Como alumno, quiero estudiar VHDL a través de un catálogo de lecciones (*Learn*), para aprender los fundamentos de forma estructurada y alineada con mis materias de la ESCOM.	RF-24
HU-21	Como alumno, quiero practicar con ejercicios autoevaluables (*Practice*) en tres modalidades —partir de cero, arreglar un *bug* o completar una función—, para ganar fluidez resolviendo problemas representativos.	RF-25
HU-22	Como alumno, quiero comprender por qué el código secuencial dentro de un `process` no es lo mismo que las asignaciones concurrentes, mediante un módulo guiado (*Concepts*) con reproductor paso a paso.	RF-26
HU-23	Como alumno, quiero construir circuitos sencillos arrastrando bloques visuales (Blockly), para que la sintaxis no sea una barrera al inicio del aprendizaje.	RF-27
HU-24	Como alumno, quiero que la aplicación recuerde qué lecciones y ejercicios he completado, para retomar mi progreso entre sesiones.	RF-28
HU-25	Como alumno, quiero exportar mi proyecto como un archivo único (*bundle* `.kmila`), para compartirlo con un compañero o entregar la práctica al profesor.	RF-30
HU-26	Como alumno avanzado, quiero añadir mi propio dispositivo FPGA y mis propias librerías VHDL sin recompilar la aplicación, para experimentar con configuraciones que no vienen de fábrica.	RF-31
HU-27	Como alumno, quiero que el cursor temporal del visor de ondas resalte automáticamente las conexiones activas en el esquemático, para correlacionar comportamiento eléctrico con código.	RF-32
HU-28	Como alumno, quiero que la aplicación reporte automáticamente los errores que	RF-29

4.5. Reglas de Negocio

Tabla 4.12: *Reglas de negocio: gestión y persistencia.*

ID	Regla	Descripción	Justificación
RN-01	Proyecto obligatorio para simulación	Todo archivo HDL debe pertenecer a un proyecto creado en Kmila antes de poder ser simulado o analizado.	La persistencia en SQLite (tabla `Projects`) requiere un contexto de proyecto para asociar archivos y configuraciones.
RN-02	Un dispositivo FPGA por proyecto	Cada proyecto debe tener asociado exactamente un modelo de dispositivo FPGA del catálogo de 34 modelos disponibles.	Los recursos del dispositivo (LUTs, flip-flops, bloques DSP, frecuencia máxima) determinan las restricciones de síntesis y simulación.
RN-03	Extensiones de archivo permitidas	Solo se permiten archivos con extensión `.vhd` o `.vhdl`. El soporte para otros lenguajes HDL (Verilog, SystemVerilog) queda fuera del alcance del presente proyecto y se considera trabajo futuro.	Evita que el usuario agregue archivos no compatibles con el motor de análisis VHDL.
RN-04	Persistencia dual de archivos	Todo archivo de proyecto se almacena en disco (sistema de archivos local) y simultáneamente en SQLite (tabla `ProjectFiles`, campo `Content`) como respaldo.	Garantiza recuperación ante pérdida del archivo en disco y permite portabilidad del proyecto.

Tabla 4.13: *Reglas de negocio: configuración y simulación.*

ID	Regla	Descripción	Justificación
RN-05	Idioma por defecto	El idioma inicial de la interfaz es español (ES). El usuario puede cambiarlo a cualquiera de los 7 idiomas disponibles desde el menú de configuración.	Orientado al contexto académico de la ESCOM donde el idioma principal es español.
RN-06	Configuración por proyecto	Los parámetros de simulación (frecuencia de reloj, tiempo de ejecución, formato de exportación) se almacenan por proyecto en formato `.k-config`.	Permite que cada proyecto tenga configuraciones independientes sin afectar otros proyectos.
RN-07	Validación previa a simulación (TT2)	Antes de ejecutar la simulación, el código debe pasar un análisis sintáctico completo sin errores.	Previene la ejecución de código malformado que podría generar resultados incorrectos o excepciones en el motor de simulación.
RN-08	Límite de tiempo de simulación (TT2)	La simulación se detiene automáticamente al alcanzar el tiempo máximo configurado en el proyecto.	Evita bucles infinitos y consumo excesivo de recursos en simulaciones mal definidas.

4.6. Análisis de Riesgos

Tabla 4.14: *Análisis de riesgos del proyecto.*

ID	Riesgo	Prob.	Impacto	Estrategia de Mitigación
R-01	Complejidad del análisis sintáctico de VHDL excede lo estimado	Alta	Alto	Limitar el subconjunto soportado de VHDL (entidades, arquitecturas, procesos, señales) y usar gramáticas de referencia del estándar IEEE 1076-2008 [10].
R-02	Rendimiento insuficiente de la simulación en WebAssembly	Media	Alto	Realizar benchmarks tempranos comparando ejecución nativa (MAUI) vs. WebAssembly (Blazor). Optimizar estructuras de datos del motor de simulación en TT2.
R-03	Incompatibilidad visual entre plataformas MAUI y Blazor	Media	Medio	Concentrar toda la UI en Kmila.Shared como componentes Razor reutilizables. Probar en Android y Web desde el inicio del desarrollo.
R-04	Retraso en el cronograma por alcance del motor de simulación	Media	Alto	Definir un subconjunto mínimo viable de instrucciones VHDL para TT2. Priorizar la simulación de entidades simples (compuertas, flip-flops, contadores).
R-05	Pérdida de datos del usuario por fallos en persistencia dual	Baja	Alto	Implementar escritura transaccional en SQLite y verificar integridad disco-BD al abrir un proyecto. Incluir pruebas unitarias de persistencia.
R-06	Obsolescencia de dependencias (.NET 9, BlazorMonaco 3.3.0)	Baja	Medio	Monitorear el ciclo de soporte de .NET 9 (versión STS cuyo soporte estándar concluyó en mayo de 2026) y prever la migración a una versión LTS. Encapsular las dependencias externas tras interfaces para facilitar la actualización.
R-07	Curva de aprendizaje del usuario objetivo	Media	Medio	Diseñar la interfaz siguiendo convenciones de IDEs conocidos (VS Code). Incluir tooltips, mensajes de error descriptivos y documentación integrada.

Capítulo 5

Diseño del Sistema

5.1. Arquitectura del Sistema

La arquitectura de Kmila se describe mediante el modelo C4 (Context, Containers, Components, Code) propuesto por Simon Brown, que permite representar el sistema en cuatro niveles de abstracción progresiva. Cada nivel aumenta el detalle técnico, facilitando la comprensión tanto para directores del proyecto como para desarrolladores que deseen extender la plataforma.

5.1.1. Nivel 1 — Diagrama de Contexto

El nivel de contexto muestra a Kmila como una caja negra y sus interacciones con actores y sistemas externos. Kmila opera de forma completamente local: no requiere conexión a internet ni depende de servicios en la nube. El único actor es el estudiante de ISC que interactúa con la aplicación para editar código VHDL, configurar simulaciones y visualizar formas de onda.

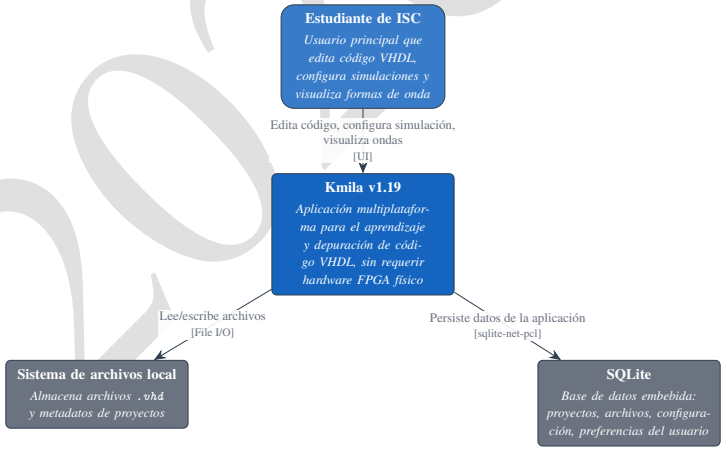


Figura 5.1: *Diagrama de contexto del sistema Kmila (C4 Nivel 1).*

El estudiante interactúa con Kmila a través de la interfaz gráfica unificada. La aplicación persiste los datos del usuario en dos destinos complementarios: el sistema de archivos local (donde residen los archivos `.vhd` del proyecto) y una base de datos SQLite embebida (donde se almacenan metadatos de proyectos, respaldos de archivos, configuraciones de simulación e idioma seleccionado).

5.1.2. Nivel 2 — Diagrama de Contenedores

El nivel de contenedores descompone Kmila en sus proyectos ejecutables y bibliotecas. La solución (`Kmila.sln`) se compone de siete proyectos .NET organizados en dos capas: dos hosts de presentación que consumen una biblioteca compartida, y cuatro módulos de backend que implementan el motor de análisis y simulación.

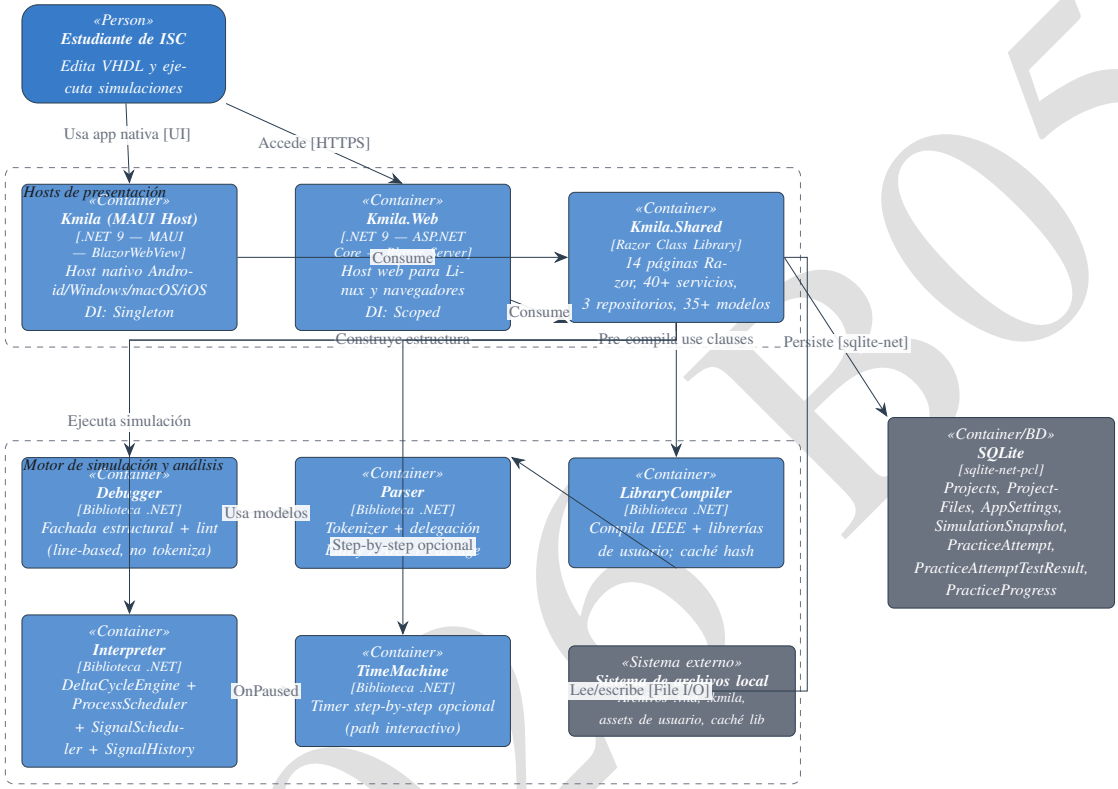


Figura 5.2: *Diagrama de contenedores del sistema Kmila (C4 Nivel 2). Incluye los hosts de presentacion (Kmila MAUI, Kmila.Web, Kmila.Shared) y el motor de simulacion y analisis (Debugger, Parser, LibraryCompiler, Interpreter, TimeMachine).*

Tabla 5.1: *Descripción de contenedores del sistema.*

Contenedor	Tecnología	Responsabilidad
Kmila (MAUI)	.NET 9, MAUI, BlazorWebView	Host nativo para Android, Windows, macOS e iOS. Registra servicios como Singleton. Carga recursos FPGA desde `FileSystem.OpenAppPackageFileAsync`.
Kmila.Web	.NET 9, ASP.NET Core, Blazor Server	Host web para Linux y navegadores. Registra servicios como Scoped. Sirve recursos FPGA desde `_content/Kmila.Shared/`.
Kmila.Shared	Razor Class Library (net9.0 + net9.0-android)	Contiene más de 30 componentes Razor, más de 25 servicios, 3 repositorios y más de 20 modelos. Concentra toda la lógica de UI y negocio.
Debugger	Biblioteca de clases .NET	Orquesta el análisis de archivos VHDL: extrae entidades y arquitecturas mediante una máquina de estados de 3 fases.
Parser	Biblioteca de clases .NET	Tokenización y análisis sintáctico de código VHDL. Genera tokens a partir del código fuente.
Interpreter	Biblioteca de clases .NET	Motor de simulación basado en `DeltaCycleEngine`. Ejecuta ciclos delta, gestiona señales y genera `SignalHistory` para visualización.
TimeMachine	Biblioteca de clases .NET	Framework de depuración paso a paso. Permite avanzar la simulación ciclo por ciclo con control temporal.
SQLite	sqlite-net-pcl 1.9.172	Base de datos embebida con 3 tablas: `Projects` (metadatos), `ProjectFiles` (respaldo de contenido), `AppSettings` (configuración clave-valor).

5.1.3. Nivel 3 — Diagrama de Componentes

El diagrama de componentes detalla la estructura interna de Kmila.Shared, que concentra la lógica de la aplicación. Los componentes se organizan en cuatro capas: páginas (interfaz de usuario), servicios (lógica de negocio), repositorios (acceso a datos) y datos (persistencia).

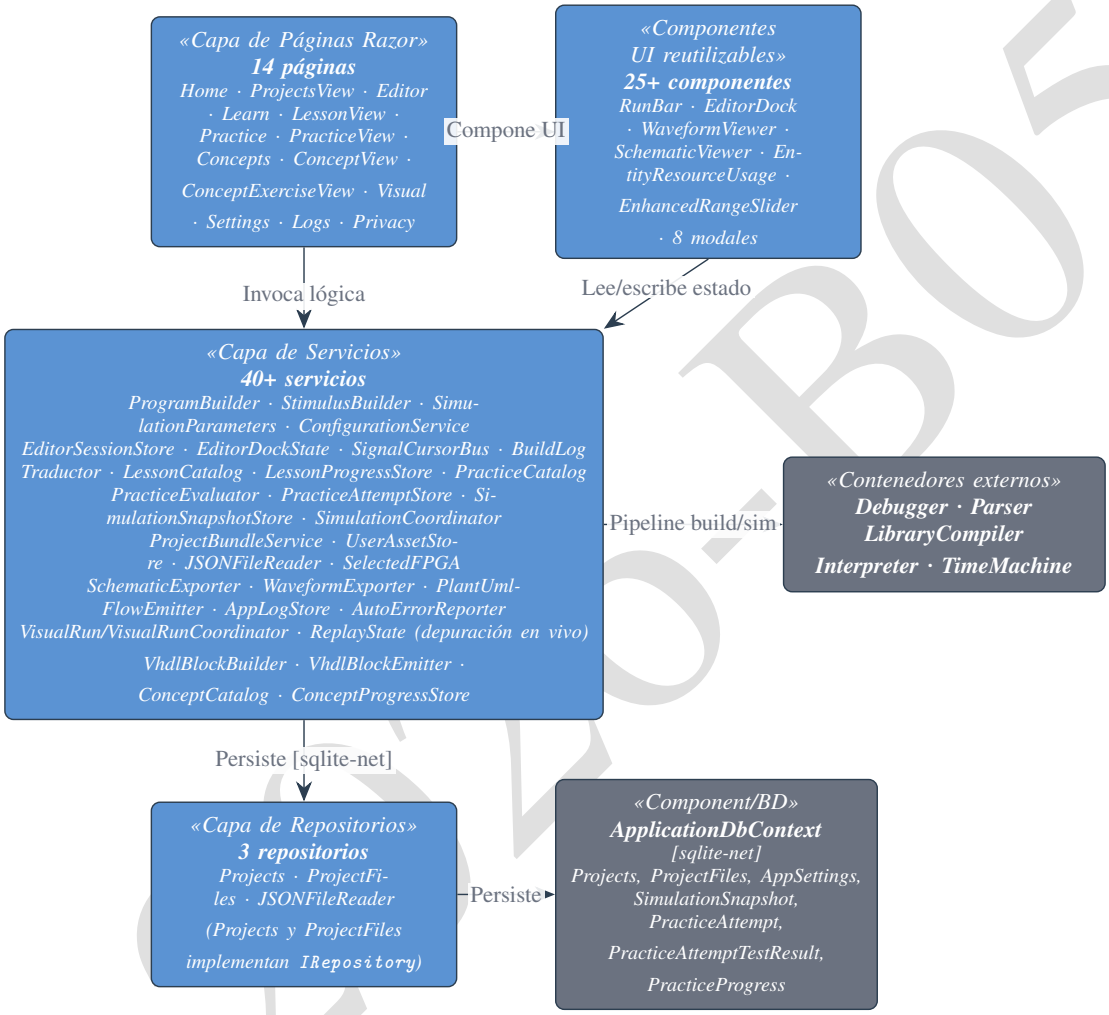


Figura 5.3: *Diagrama de componentes de Kmila.Shared (C4 Nivel 3). Cubre las 9 paginas Razor, los 25+ servicios inyectables, los 25+ componentes UI reutilizables y los 3 repositorios; muestra ademas las dependencias hacia los contenedores externos Debugger, Parser, LibraryCompiler, Interpreter y TimeMachine.*

La página `Editor.razor` (549 líneas) constituye el componente central de la aplicación: integra el editor Monaco, el explorador de archivos (`FilesCard`), el selector de dispositivos FPGA (`FPGATabContent`), el panel de configuración de simulación (`SimulationParametersModal`), el visualizador de formas de onda (`WaveformViewer`) y los controles de compilación (`ButtonCollapse`). El servicio `ProgramBuilder` orquesta el flujo de compilación y simulación: invoca al módulo `Debugger` para el análisis sintáctico y al `Interpreter` para la ejecución de la simulación, delegando al `StimulusBuilder` la conversión de los parámetros de la interfaz al formato `SimulationConfig` requerido por el motor.

A continuación se presentan las vistas segmentadas por capa para facilitar el análisis individual de cada grupo de

componentes.

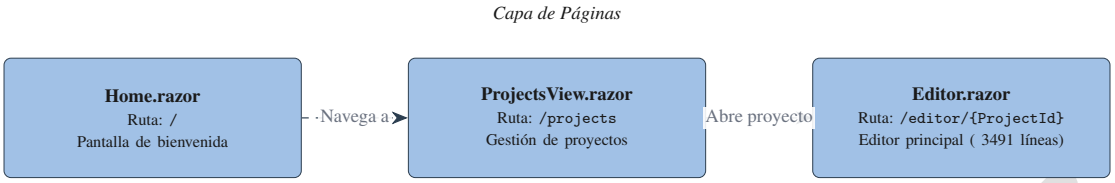


Figura 5.4: *Vista segmentada: capa de paginas. Las 3 paginas Razor definen las rutas principales de la aplicacion.*

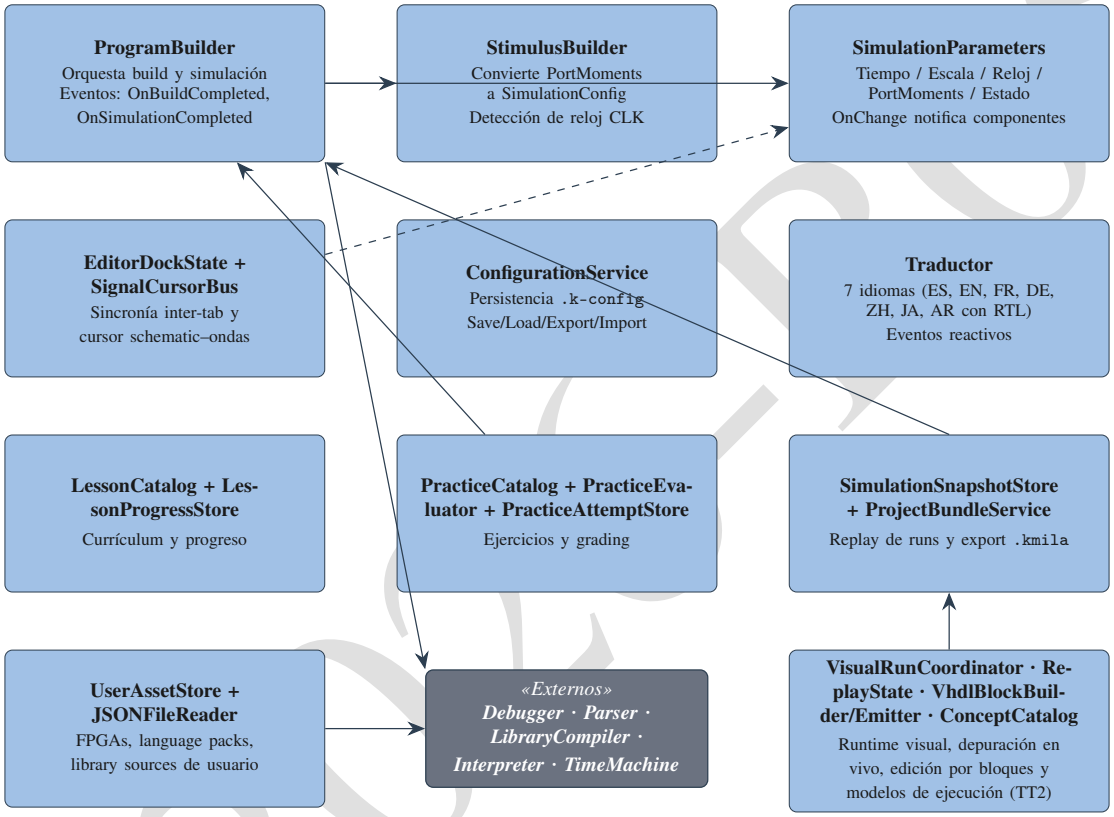


Figura 5.5: *Vista segmentada: capa de servicios (subset representativo de los 25+ servicios). ProgramBuilder es el orquestador central que conecta la UI con los modulos de backend (Debugger, Parser, LibraryCompiler, Interpreter, TimeMachine); SimulationParameters mantiene el estado coarse-grained del editor; EditorDockState y SignalCursorBus sincronizan los siete tabs del workbench; LessonCatalog y la familia Practice* gestionan el subsistema pedagogico; SimulationSnapshotStore y ProjectBundleService persisten corridas y exportaciones; UserAssetStore expone los assets editables por el usuario.*

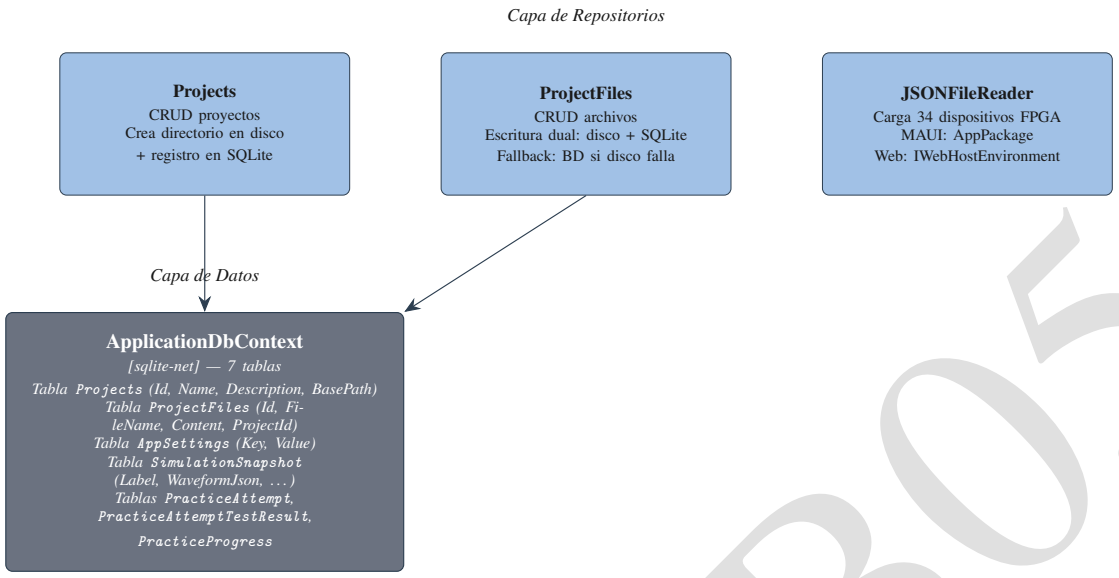


Figura 5.6: *Vista segmentada: capas de repositorios y datos. La persistencia dual (disco + SQLite) garantiza la recuperacion de archivos ante fallos.*

5.1.4. Diagrama de Despliegue

El diagrama de despliegue muestra cómo se distribuye Kmila en las distintas plataformas objetivo. Ambos hosts empaquetan Kmila.Shared dentro de su binario, pero difieren en el mecanismo de carga de recursos y el ciclo de vida de inyección de dependencias.

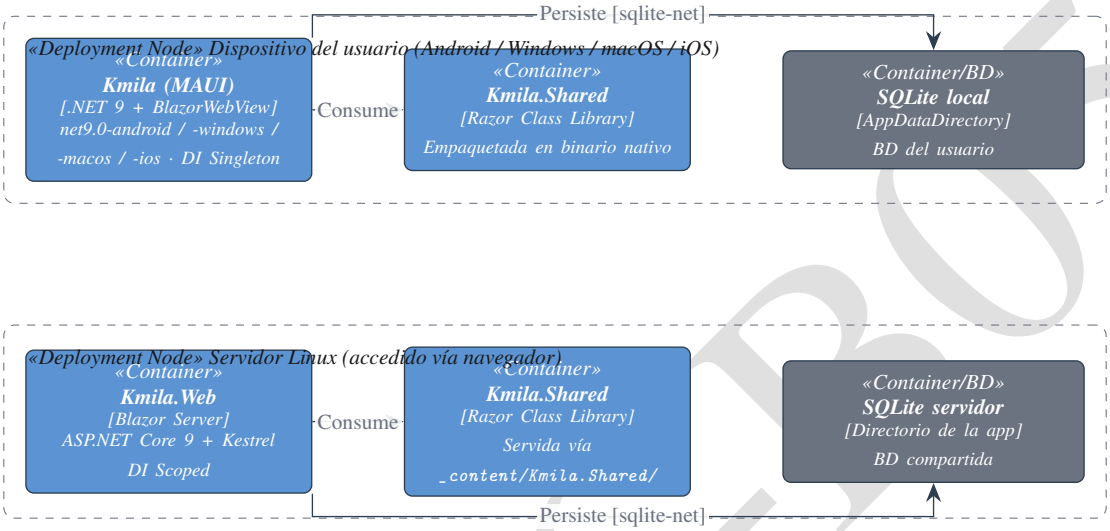


Figura 5.7: *Diagrama de despliegue del sistema Kmila (C4 Nivel 4).*

Tabla 5.2: *Mecanismos de despliegue por plataforma.*

Plataforma	Mecanismo de despliegue	Ciclo de vida DI	Carga de recursos FPGA
Android	APK via .NET MAUI (min API 24)	Singleton	`FileSystem.OpenAppPackageFileAsync`
Windows	MSIX o publicación directa	Singleton	`FileSystem.OpenAppPackageFileAsync`
macOS	App Bundle	Singleton	`FileSystem.OpenAppPackageFileAsync`
iOS	App Bundle	Singleton	`FileSystem.OpenAppPackageFileAsync`
Linux / Web	ASP.NET Core + Blazor Server	Scoped	`IWebHostEnvironment` + `_content/Kmila.Shared/Devices/`

5.2. Diagrama de Clases

El diagrama de clases describe las clases principales del sistema, sus atributos, métodos y relaciones. Debido a la extensión del sistema (más de 20 modelos, más de 25 servicios, 3 repositorios y 4 módulos de backend), se presenta primero un diagrama general con las relaciones entre clases y posteriormente vistas segmentadas por módulo.

5.2.1. Diagrama General

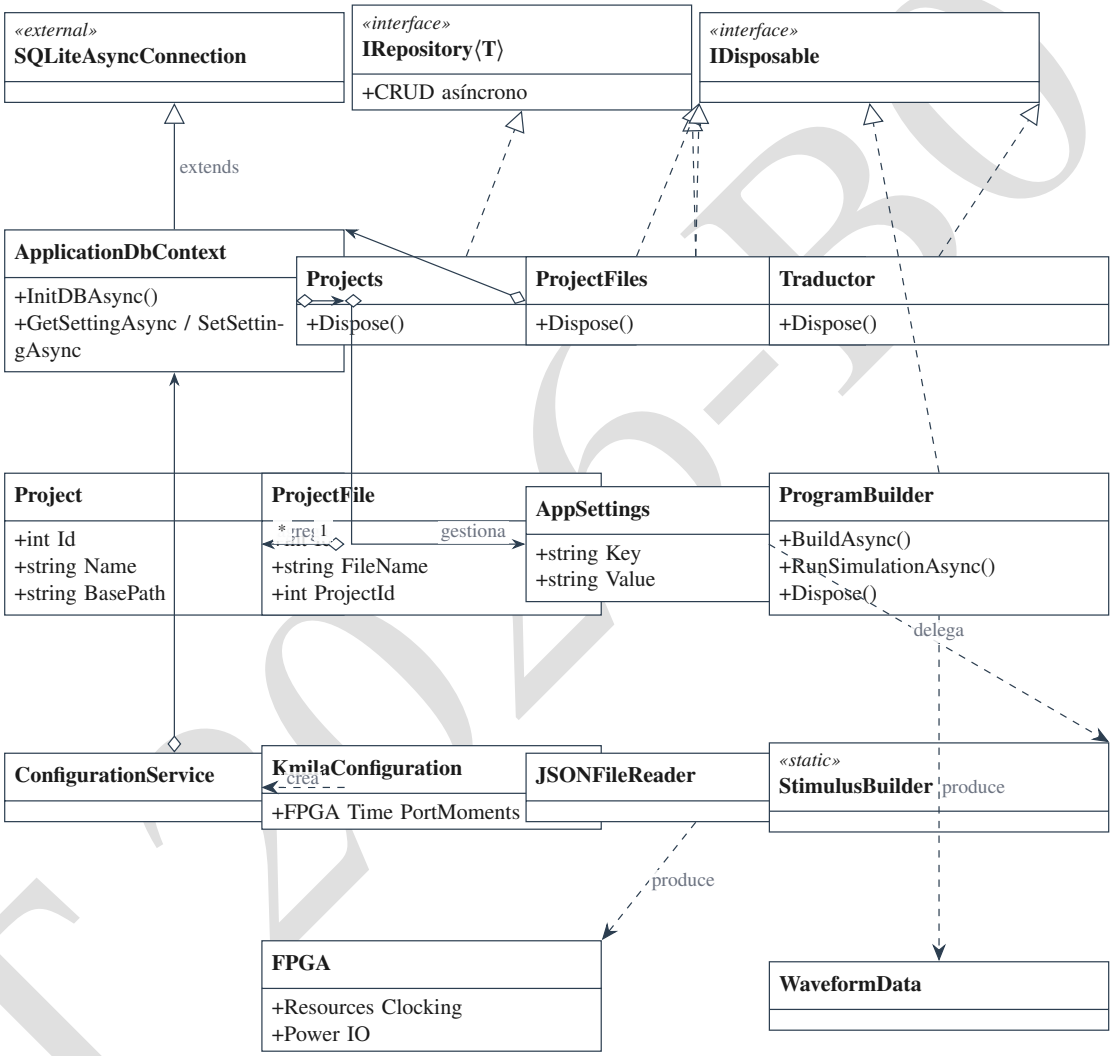


Figura 5.8: *Diagrama general de clases del sistema Kmila.*

5.2.2. Modelos de Datos

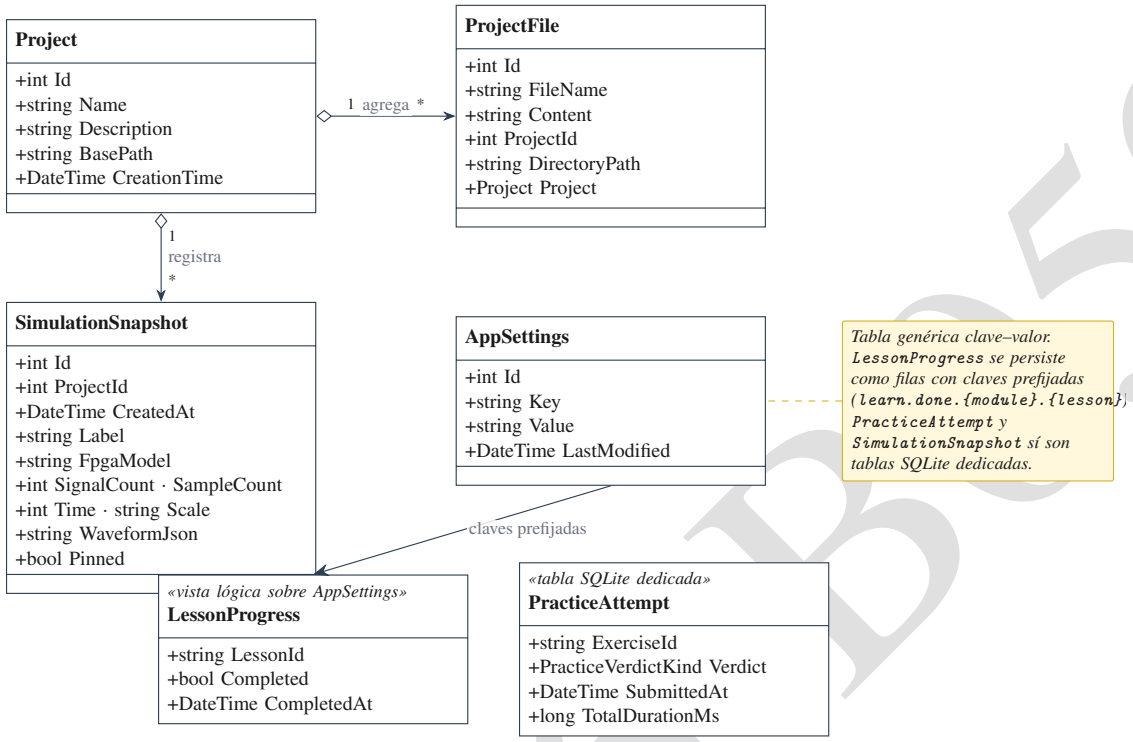


Figura 5.9: *Modelos de persistencia. Project y ProjectFile persisten proyectos y archivos en SQLite; SimulationSnapshot guarda corridas completas con WaveformData serializada para replay; AppSettings es una tabla generica clave-valor donde se materializan ademas LessonProgress (claves `lesson_*`) y PracticeAttempt (claves `practice_*`).*

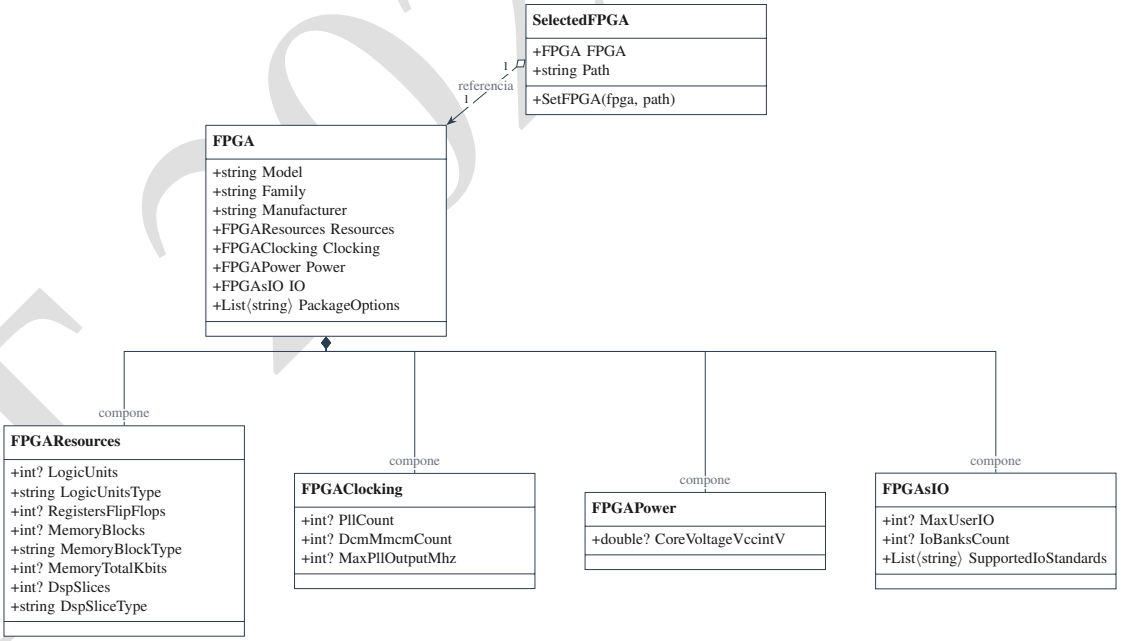


Figura 5.10: *Modelo del catalogo FPGA. FPGA modela las especificaciones de los 34 dispositivos; FPGAResources y FPGAClocking descomponen los recursos logicos y de reloj.*

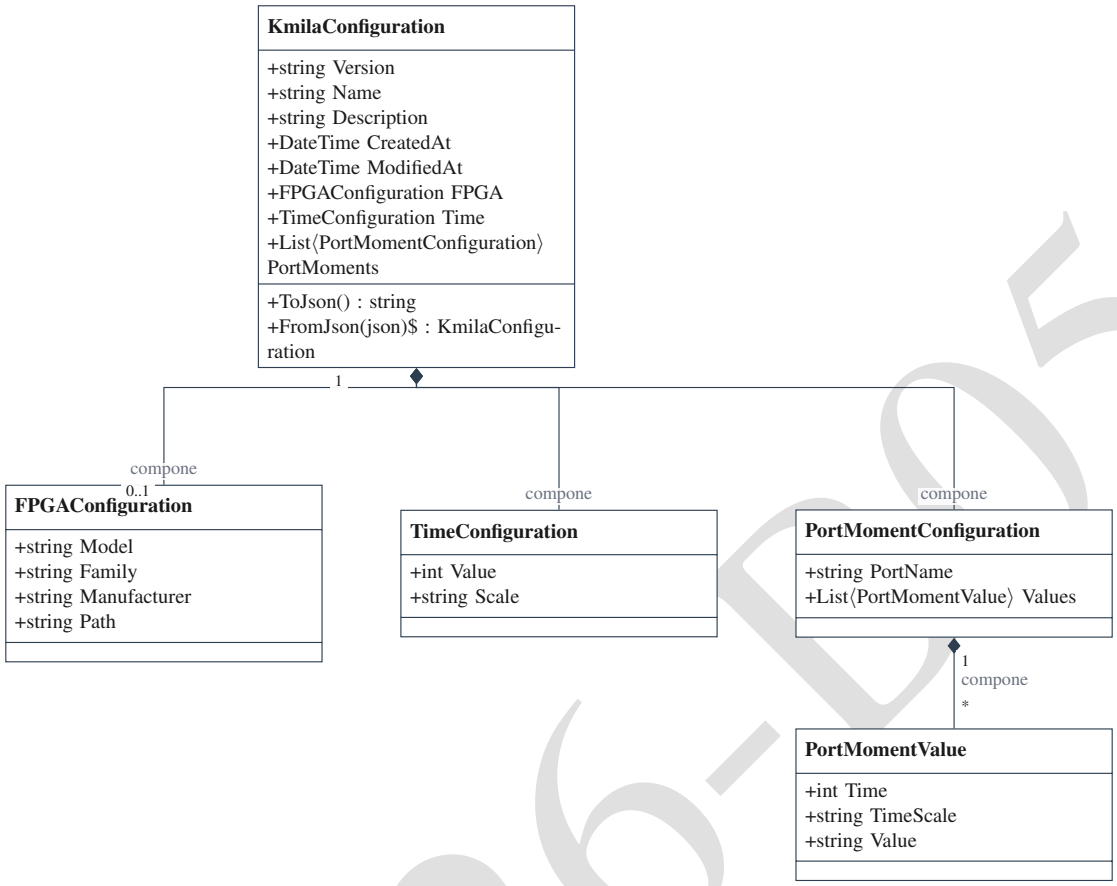


Figura 5.11: *Modelo de configuracion de proyecto. KmilaConfiguration define el formato .k-config; cada PortMoment especifica los estimulos de un puerto.*

5.2.3. Servicios y Repositorios

El siguiente diagrama muestra la vista general de las capas de servicios y repositorios, seguido de vistas particulares de cada clase.

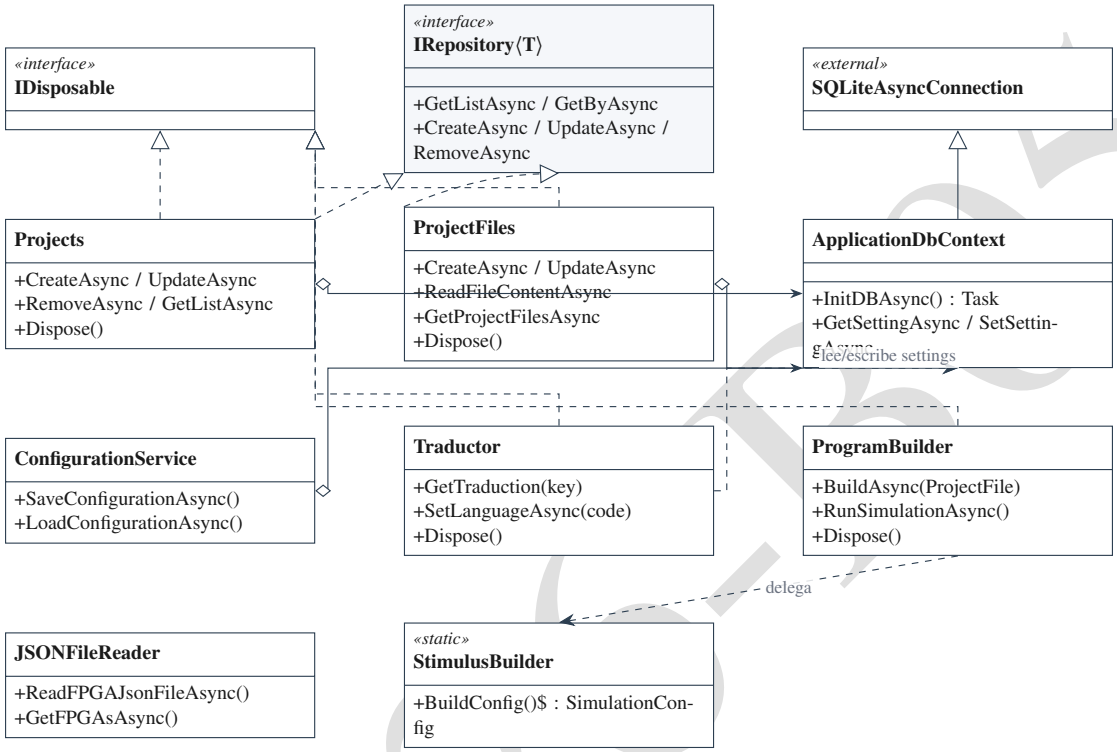


Figura 5.12: *Vista general: servicios y repositorios. Todos los repositorios y servicios de persistencia dependen de ApplicationDbContext.*

A continuación se presenta cada clase en detalle.

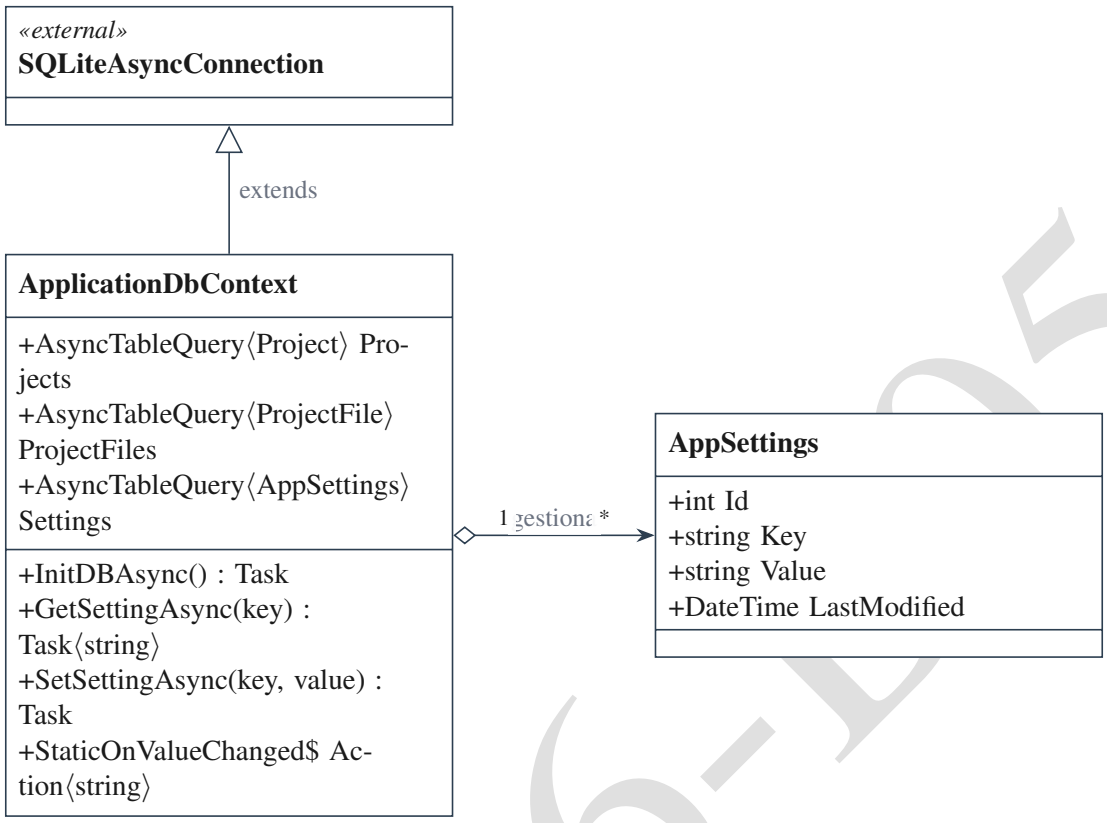


Figura 5.13: *Detalle: ApplicationDbContext. Gestiona la conexion SQLite y expone un evento estatico para notificar cambios en configuracion.*

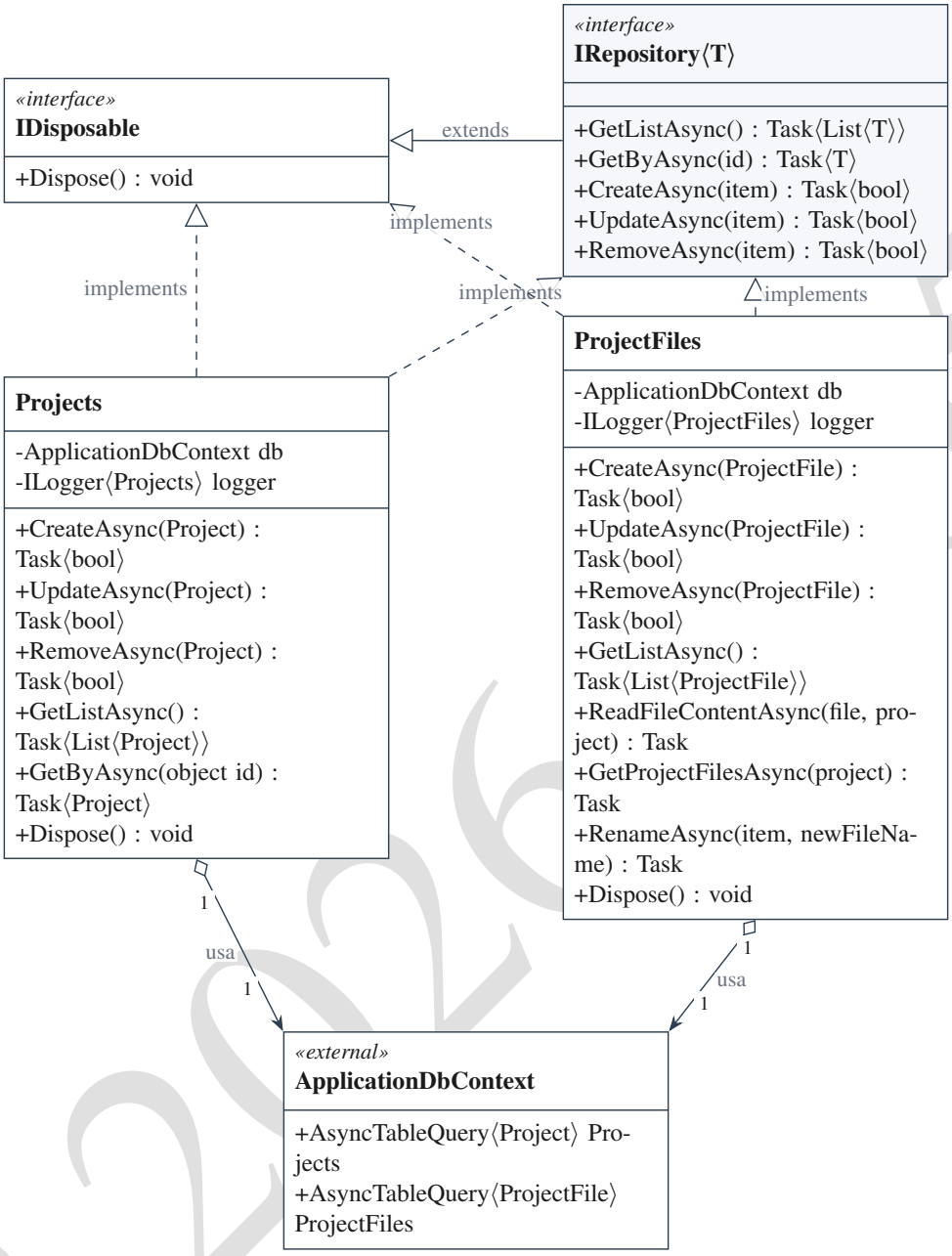


Figura 5.14: *Detalle: repositorios Projects y ProjectFiles. Ambos implementan IRepository y gestionan la sincronizacion entre disco y base de datos.*

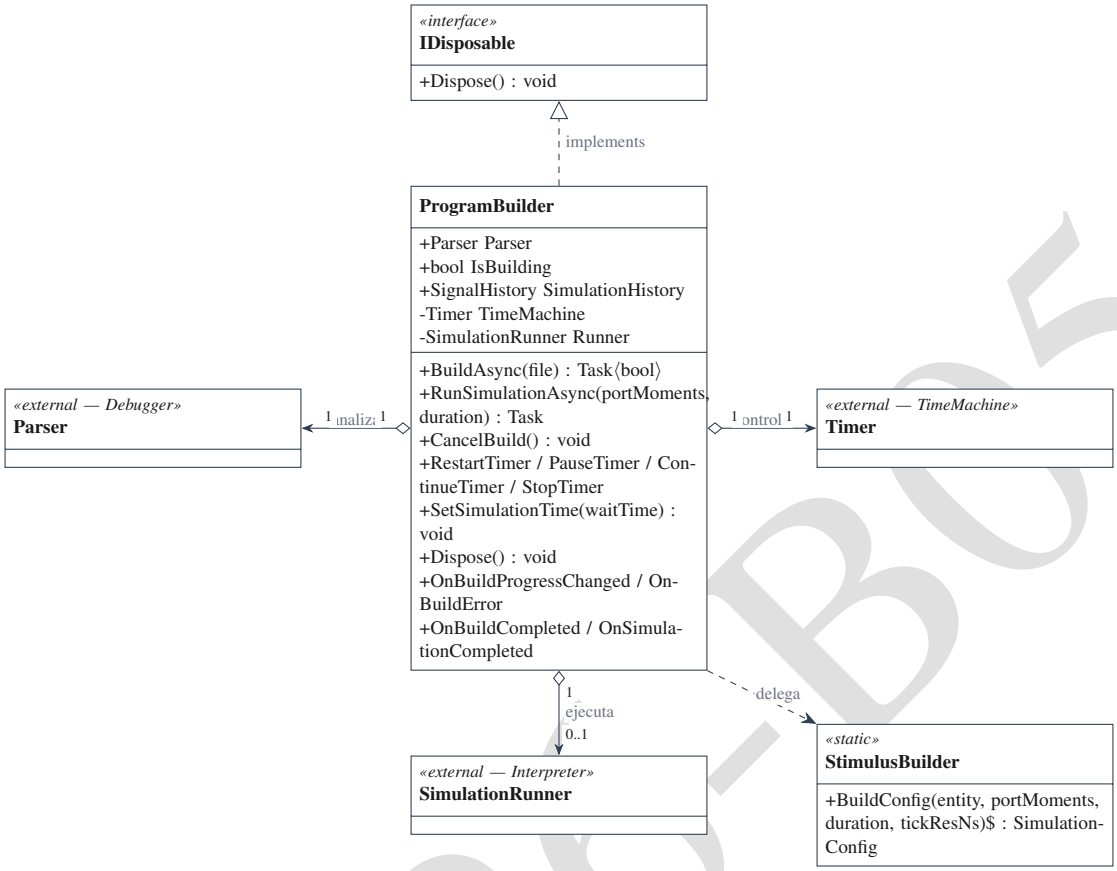


Figura 5.15: *Detalle: ProgramBuilder y StimulusBuilder. ProgramBuilder orquesta compilacion y simulacion; StimulusBuilder convierte la configuracion de la UI al formato SimulationConfig del motor.*

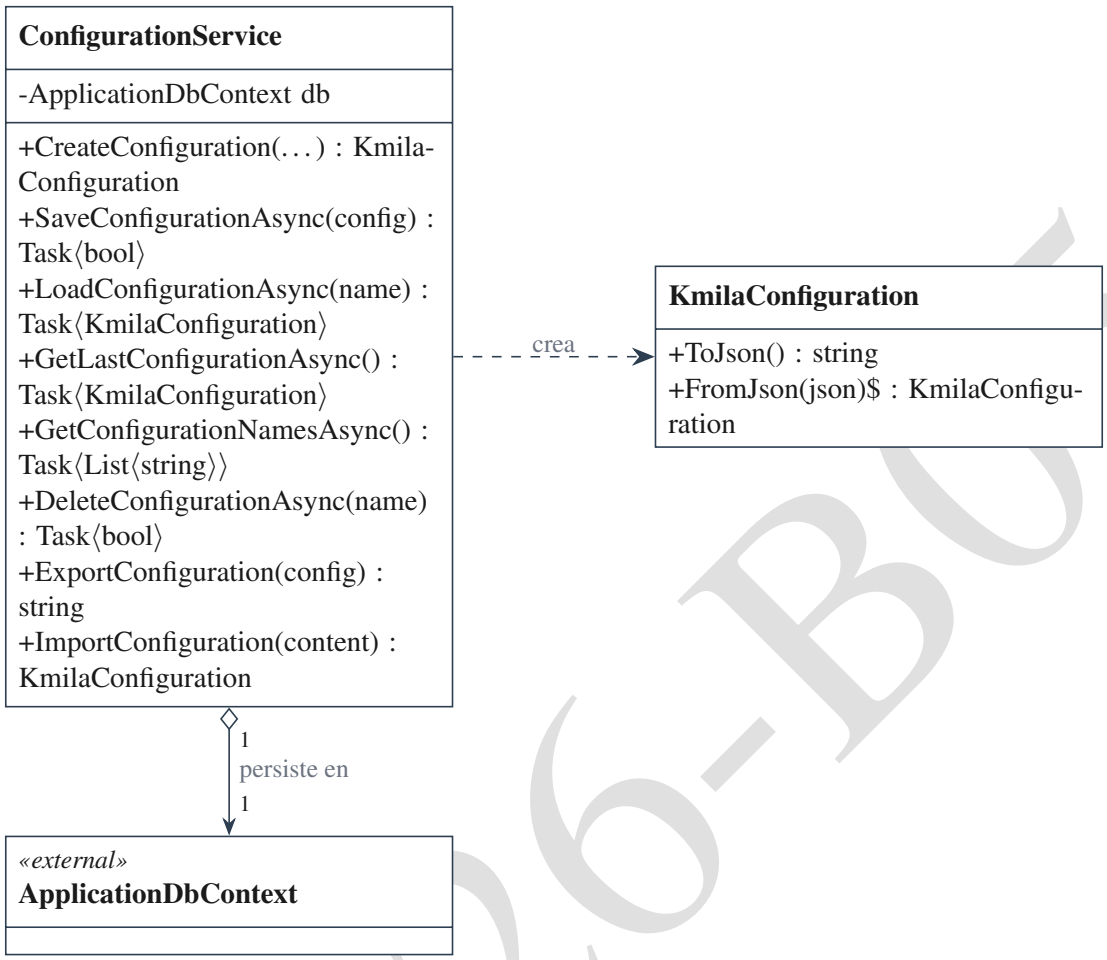


Figura 5.16: *Detalle: ConfigurationService. Gestiona la persistencia de configuraciones de simulacion en formato .k-config (JSON serializado en SQLite).*

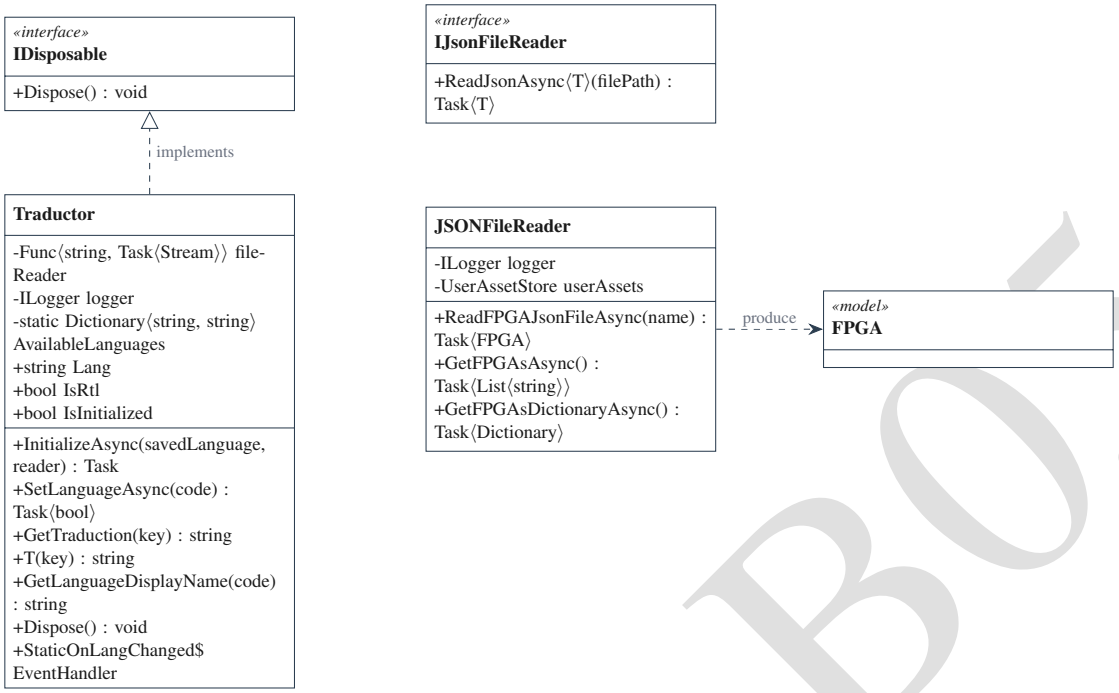


Figura 5.17: *Detalle: Traductor y JSONFileReader. Traductor gestiona la internacionalizacion con eventos reactivos; JSONFileReader carga el catalogo de 34 dispositivos FPGA con estrategias distintas segun la plataforma.*

5.2.4. Motor de Simulación

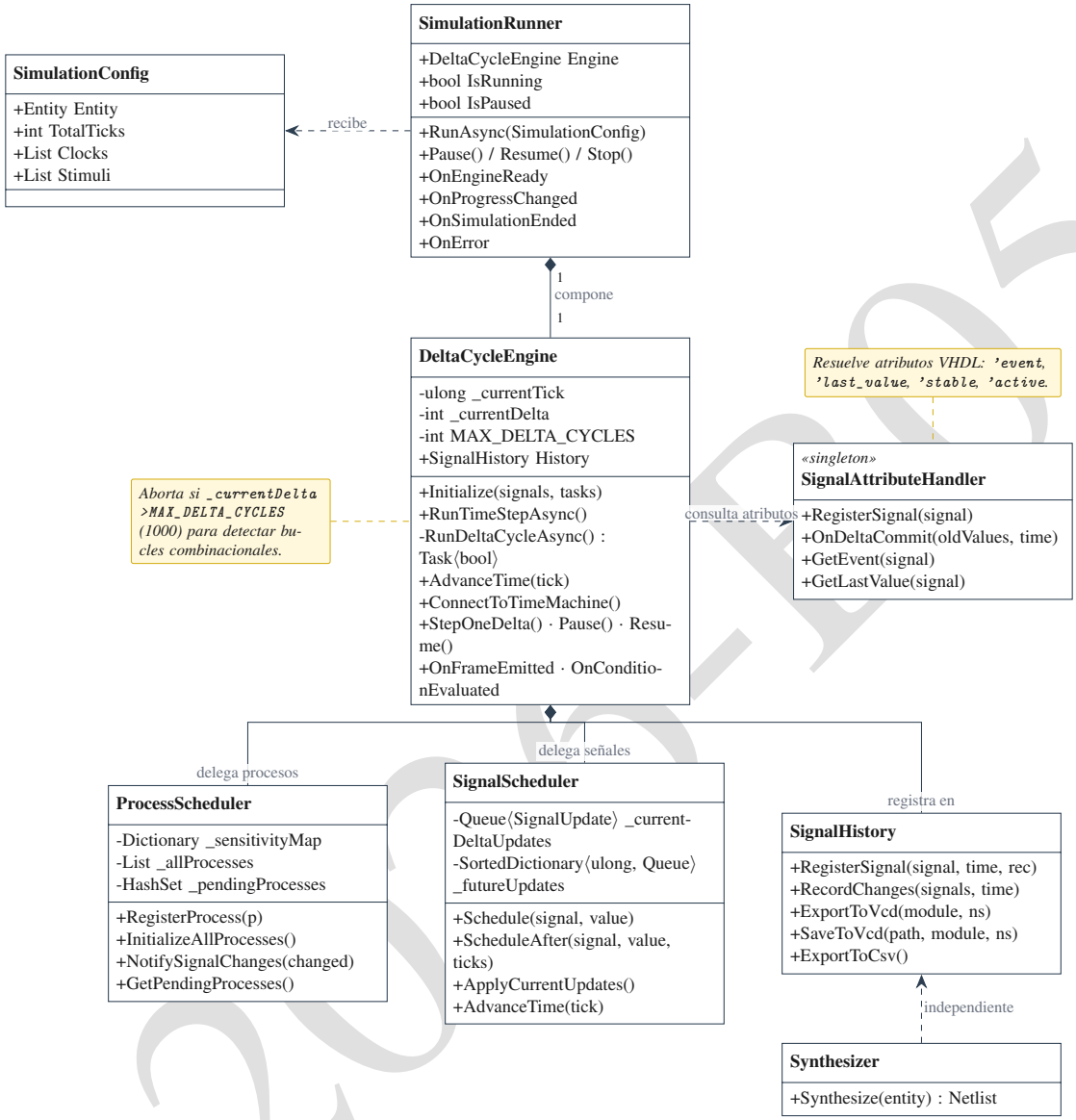


Figura 5.18: *Vista segmentada: motor de simulacion. SimulationRunner orquesta la ejecucion sobre DeltaCycleEngine, que delega la planificacion de procesos a ProcessScheduler y la de senales a SignalScheduler; SignalHistory registra los cambios y produce VCD. SignalAttributeHandler resuelve atributos VHDL (`'event`, `'last_value`, `'stable`, `'active`). El motor aborta cuando el contador de delta cycles supera `MAX_DELTA_CYCLES` para detectar bucles combinacionales.*

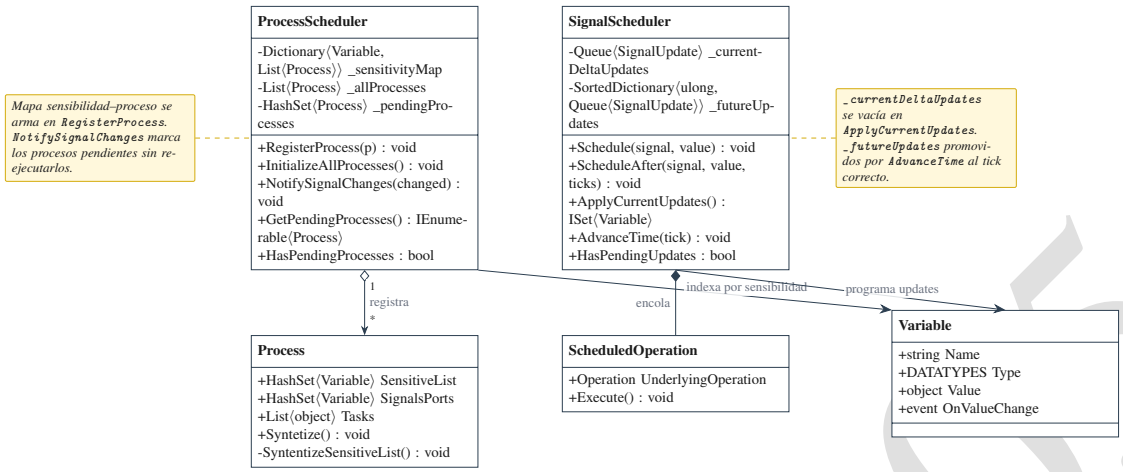


Figura 5.19: *Detalle: planificadores del motor.* `ProcessScheduler` *mantiene la cola de procesos pendientes de evaluación;* `SignalScheduler` *gestiona las actualizaciones de señales por delta cycle. Ambos cooperan para implementar la semántica de eventos discretos del estándar IEEE 1076.*

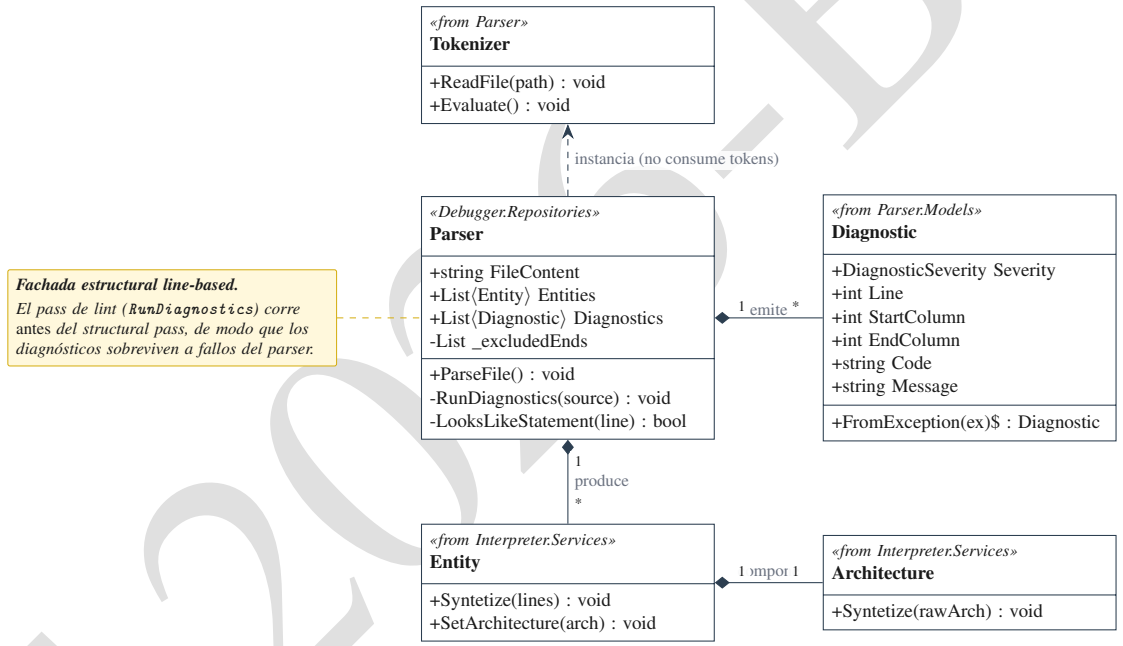


Figura 5.20: *Detalle:* `Debugger`*. Implementa una máquina de estados de tres fases (*lex, lint, validate*) para extraer entidades, arquitecturas y paquetes del archivo VHDL antes de la simulación; registra diagnósticos no fatales sin abortar el análisis completo (véase Figura 5.65).*

5.2.5. Subsistema Concepts

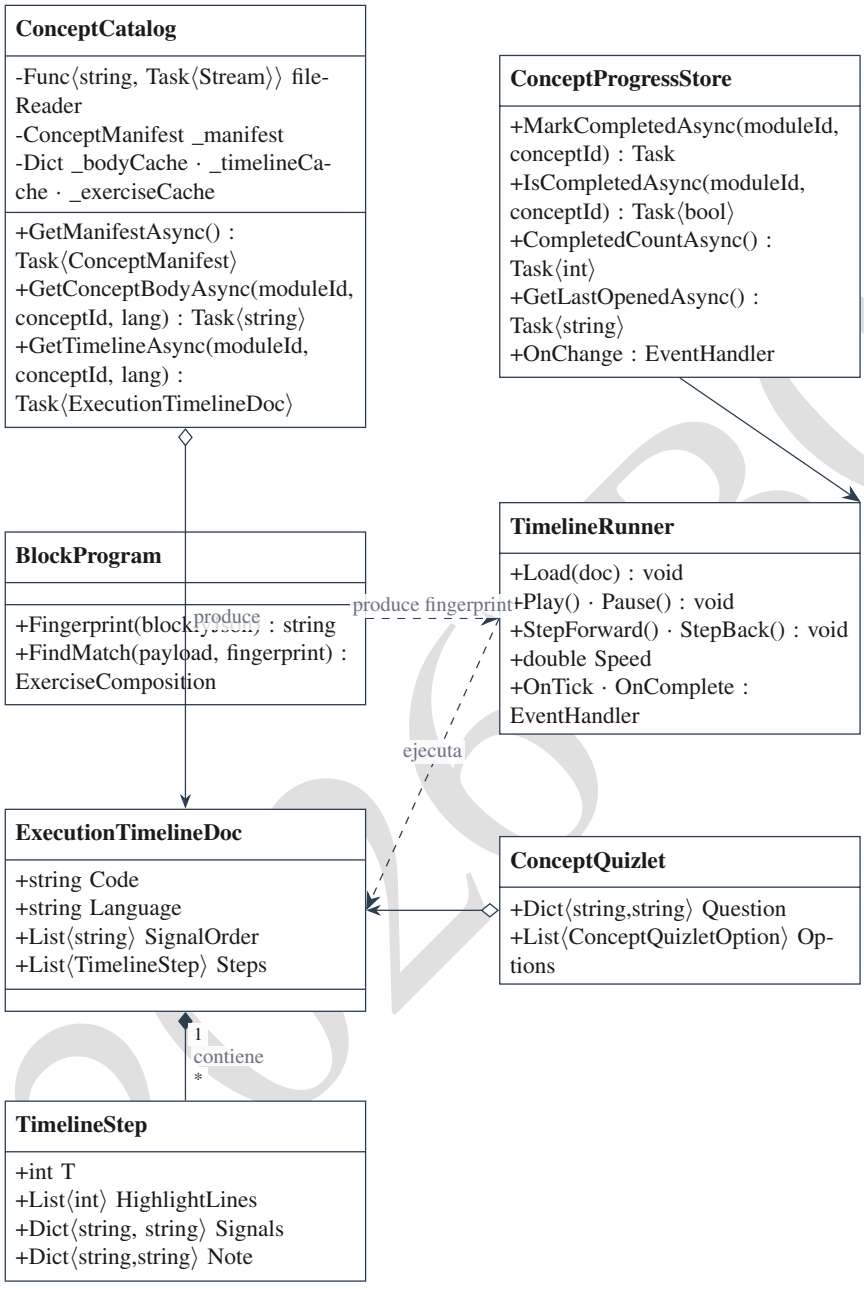


Figura 5.21: *Diagrama de clases del subsistema* Concepts. `ConceptCatalog` *hidrata las lecciones desde el manifiesto JSON y produce un* `ExecutionTimelineDoc` *pre-baked por cada lección con reproductor;* `TimelineRunner` *ejecuta el documento como un bucle* `Task.Delay` *cancelable;* `BlockProgram` *reduce el* workspace *Blockly a un* fingerprint *canónico que se compara contra el catálogo pre-baked;* `ConceptProgressStore` *persiste el avance del estudiante en* `AppSettings` *con claves* `concept.done.*`.

5.2.6. Pipeline de Replay

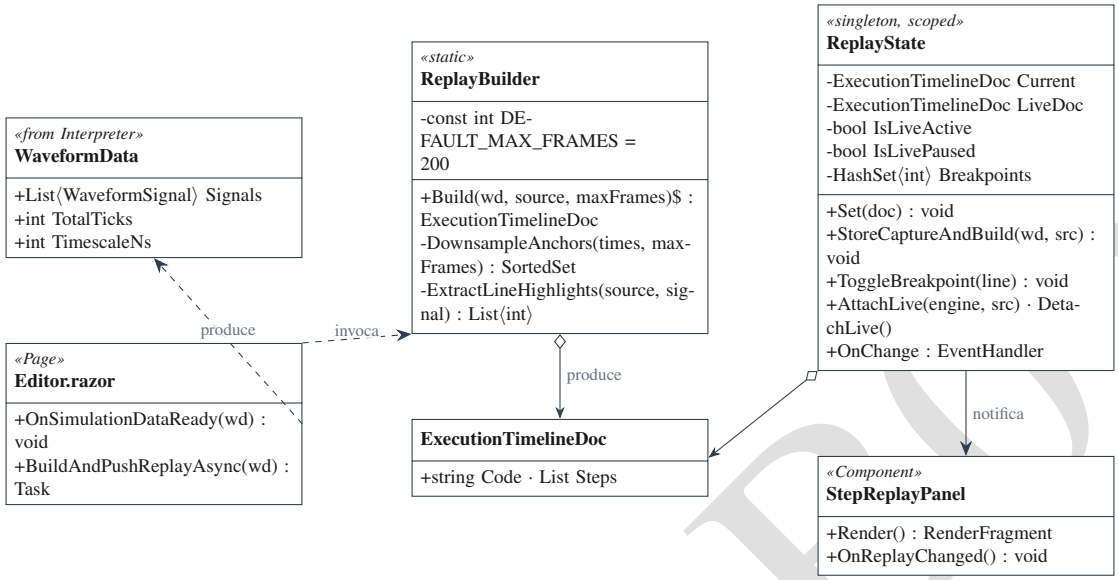


Figura 5.22: *Diagrama de clases del pipeline* Replay. `ReplayBuilder` *(estático y puro) convierte una* `WaveformData` *en un* `ExecutionTimelineDoc` *con* highlights *de línea derivados del análisis del fuente;* `ReplayState` (singleton scoped*) almacena el documento actual y los* breakpoints*;* `StepReplayPanel` *se suscribe a* `ReplayState.OnChange` *y renderiza el* `ExecutionPlayer` *(el mismo componente que usa el subsistema* Concepts*).*

5.2.7. Subsistema Practice

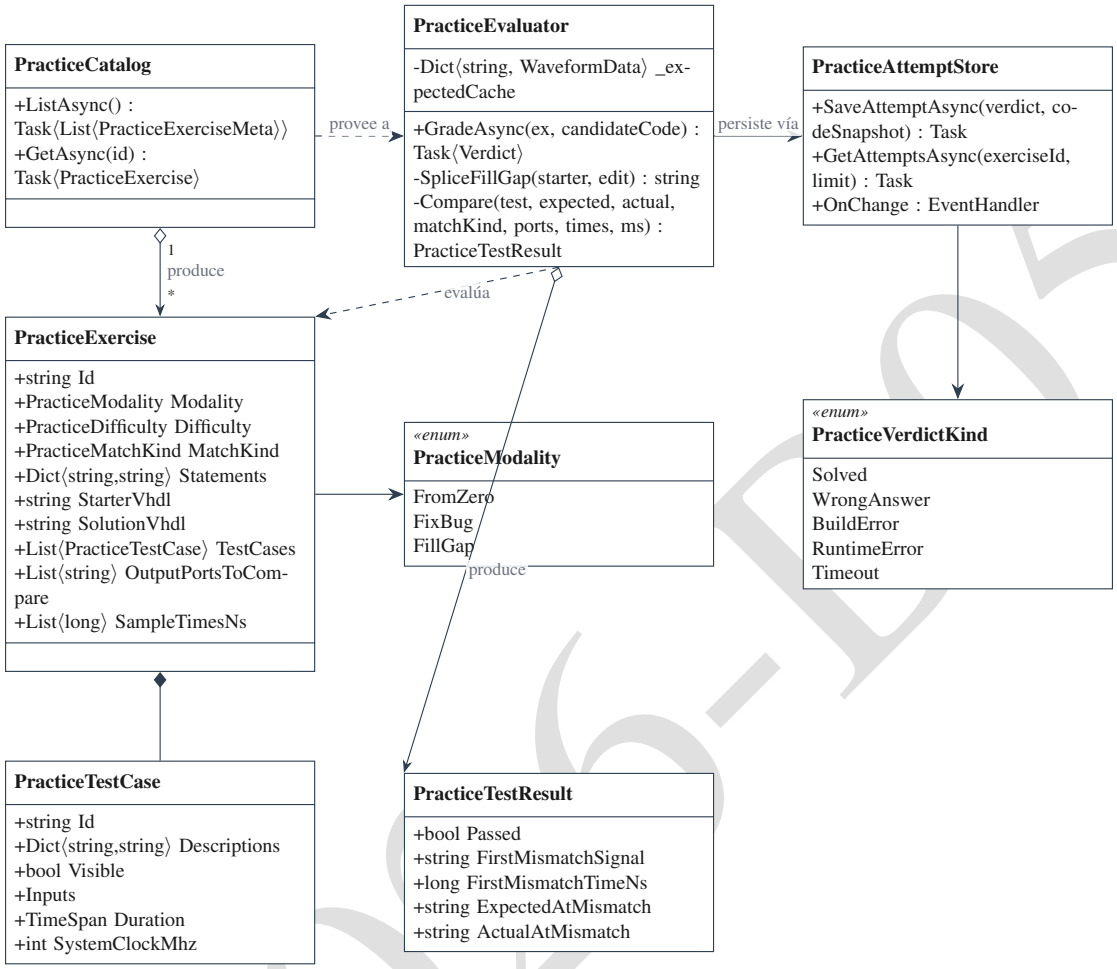


Figura 5.23: *Diagrama de clases del subsistema* Practice. `PracticeCatalog` *carga los ejercicios desde el manifiesto JSON;* `PracticeExercise` *describe la modalidad (*`FromZero`*,* `FixBug`*,* `FillGap`*), la dificultad y la lista de* `TestCase` *con puertos y tiempos de muestreo;* `PracticeEvaluator` *fusiona el código del estudiante con el* starter *según la modalidad, simula contra cada* `TestCase` *y produce un* `Verdict`*; los intentos se persisten en* `PracticeAttemptStore`*.*

5.2.8. Subsistema Visual Runtime

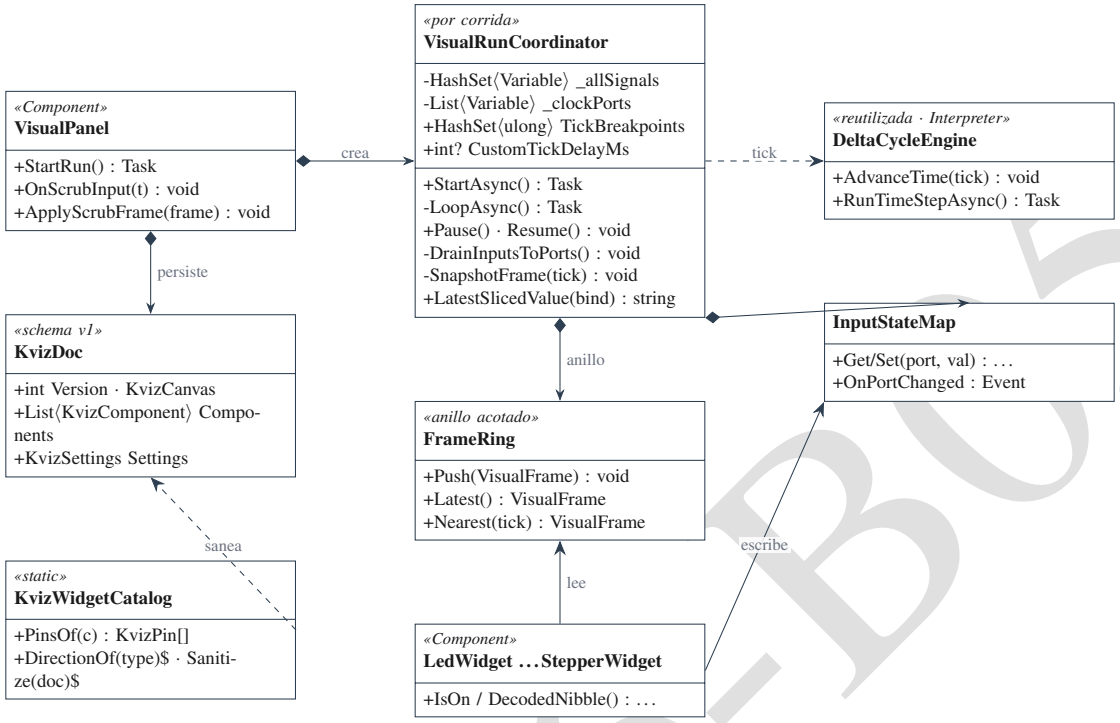


Figura 5.24: *Diagrama de clases del subsistema* Visual Runtime. `VisualPanel` *crea un* `VisualRunCoordinator` *por corrida, que reutiliza el* `DeltaCycleEngine` *del intérprete (no existe un segundo motor de simulación): su método* `LoopAsync` *escribe los puertos de entrada mediante* `DrainInputsToPorts` *y captura cada tick en un* `FrameRing` *acotado.* `InputStateMap` *concentra los valores impuestos por los widgets interactivos;* `KvizDoc` *(esquema* `.kviz` *v1) persiste el estado del lienzo y* `KvizWidgetCatalog` *depura los enlaces obsoletos. Los widgets de salida (*`LedWidget`*,* `SevenSegmentWidget`*,* `StepperWidget`*, …) leen del anillo y los de entrada (*`SwitchWidget`*,* `ButtonWidget`*) escriben en el mapa.*

5.2.9. Subsistema de Depuración en Vivo

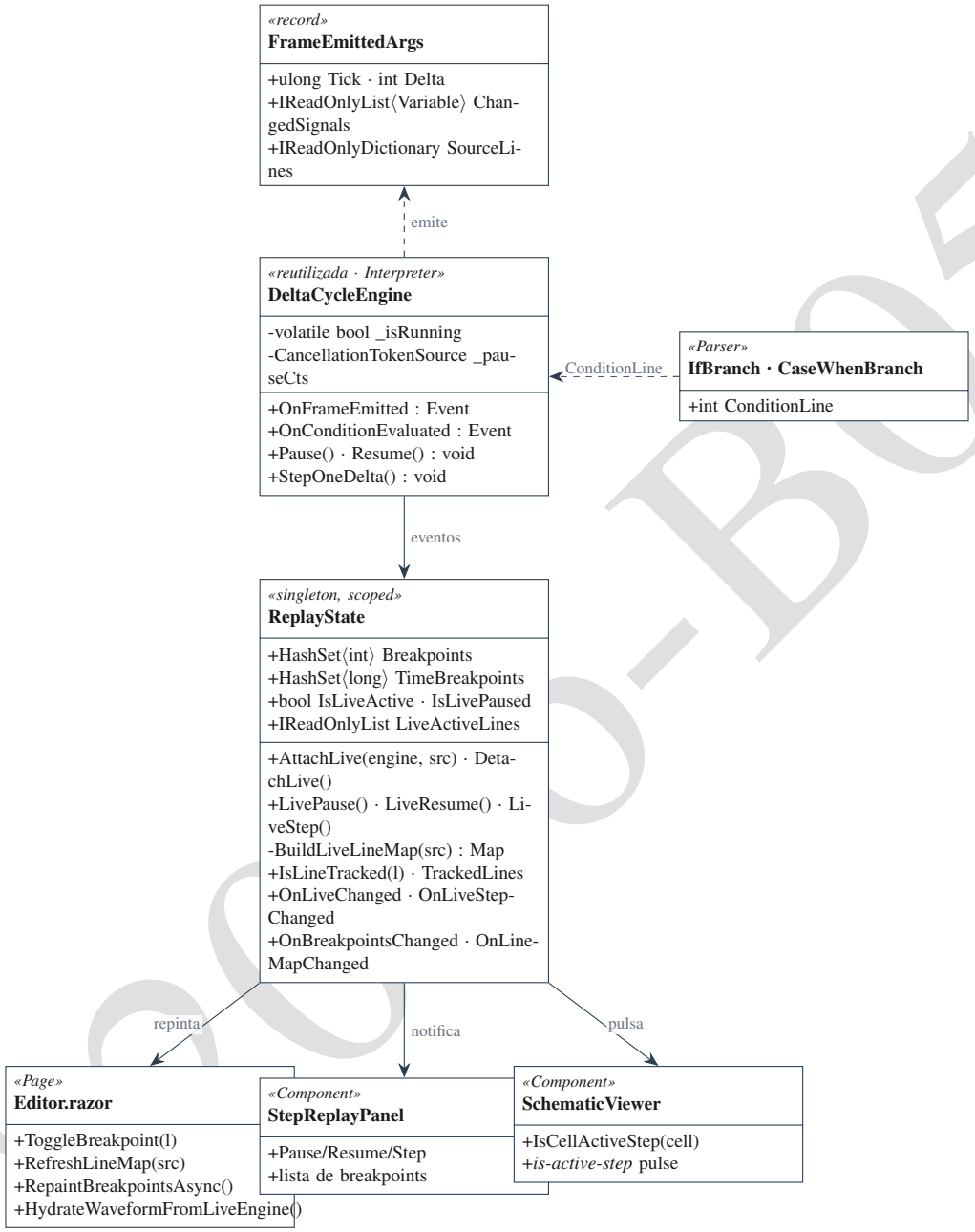


Figura 5.25: *Diagrama de clases del subsistema de* depuración en vivo. *El* `DeltaCycleEngine` *expone eventos* `OnFrameEmitted` *(con un* `FrameEmittedArgs` *que transporta las señales cambiadas y su línea de origen) y* `OnConditionEvaluated` *(línea de la rama* `if/elsif/case` *tomada), además de* `Pause/Resume/StepOneDelta`*.* `ReplayState` *(*scoped *por circuito) concentra los tres tipos de* breakpoint *(línea, tiempo y condición), construye el mapa señal→línea para clasificar los* breakpoints *no rastreables y notifica a* `Editor.razor` *(glifos en Monaco),* `StepReplayPanel` *y* `SchematicViewer` *(pulso* is-active-step*).*

5.2.10. Subsistema de Edición por Bloques

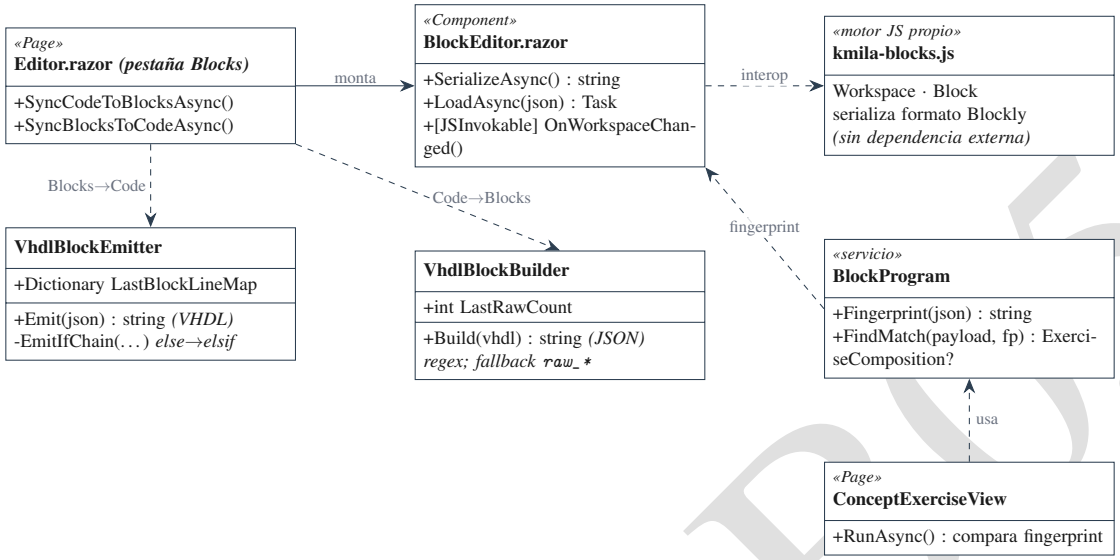


Figura 5.26: *Diagrama de clases del subsistema de* edición por bloques*. La pestaña* Blocks *del* `Editor.razor` *sincroniza en ambos sentidos con Monaco:* `VhdlBlockBuilder` *convierte VHDL a* workspace *JSON mediante un analizador por expresiones regulares (con* fallback *a bloques* `raw_*`*), y* `VhdlBlockEmitter` *regenera el VHDL manteniendo un* `LastBlockLineMap` *para ligar diagnósticos a bloques. El componente* `BlockEditor.razor` *se apoya en el motor propio* `kmila-blocks.js`*, que preserva el formato de serialización de Blockly;* `BlockProgram` *reduce el* workspace *a un* fingerprint *canónico que el módulo* Concepts *compara contra su catálogo pre-baked.*

5.3. Casos de Uso

Los casos de uso se organizan en dos grupos correspondientes a las fases del proyecto. El siguiente diagrama presenta los correspondientes a TT1 (interfaz y gestión).

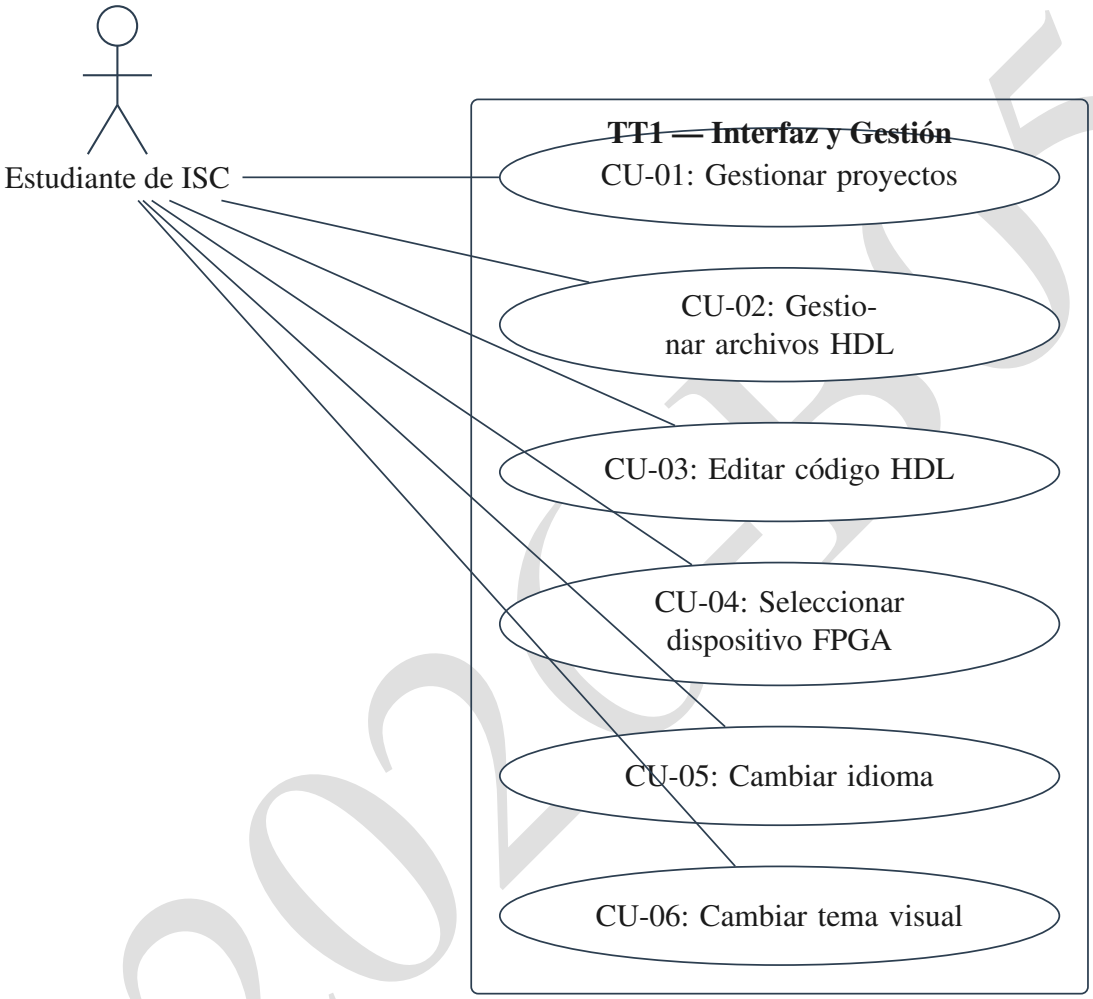


Figura 5.27: *Diagrama de casos de uso de TT1: interfaz y gestion.*

El siguiente diagrama presenta los casos de uso de TT2 (motor de simulación).

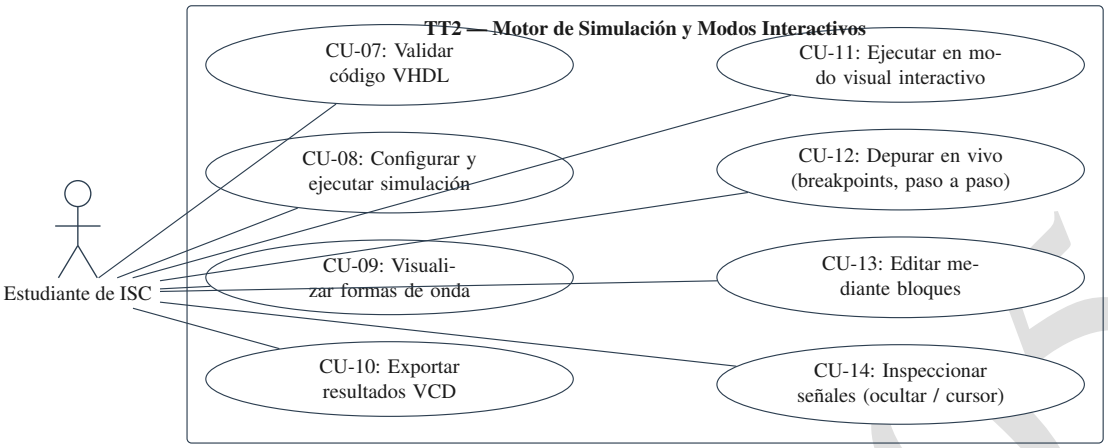


Figura 5.28: *Diagrama de casos de uso de TT2: motor de simulacion.*

5.3.1. CU-01 y CU-02: Gestión de Proyectos y Archivos HDL

CU-01: Gestionar Proyectos.

Campo	Descripción
ID	CU-01
Actor	Estudiante
RF relacionado	RF-01
Precondición	La aplicación se encuentra en ejecución
Flujo principal	1. El usuario navega a la vista de proyectos (`/projects`). 2. El usuario presiona "Crear proyecto". 3. El sistema muestra el modal `ProjectDetailsModal` solicitando nombre, descripción y ruta base. 4. El usuario ingresa los datos y confirma. 5. El repositorio `Projects` valida que no exista un proyecto con el mismo nombre y ruta. 6. El sistema crea el directorio en disco y registra el proyecto en SQLite. 7. El proyecto aparece en la lista de proyectos.
Flujo alternativo	5a. Ya existe un proyecto con el mismo nombre y ruta: el sistema notifica el error y no crea el proyecto.
Postcondición	El proyecto queda registrado en la base de datos con su directorio creado en el sistema de archivos.

CU-02: Gestionar Archivos HDL.

Campo	Descripción
ID	CU-02
Actor	Estudiante
RF relacionado	RF-02
Precondición	El usuario ha abierto un proyecto existente
Flujo principal	1. El usuario presiona "Nuevo archivo" en el panel lateral. 2. El sistema solicita el nombre del archivo. 3. El usuario ingresa un nombre con extensión `.vhd` o `.vhdl`. 4. El repositorio `ProjectFiles` crea el archivo en disco y registra una entrada en SQLite con el contenido inicial. 5. El archivo aparece en el listado de archivos del proyecto (`FilesCard`).
Flujo alternativo	3a. El usuario intenta crear un archivo con extensión no soportada: el sistema rechaza la operación. 6a. Al abrir un archivo, si el archivo en disco no existe pero sí en SQLite, el sistema recupera el contenido desde la base de datos.
Postcondición	El archivo persiste en disco y en SQLite (persistencia dual).

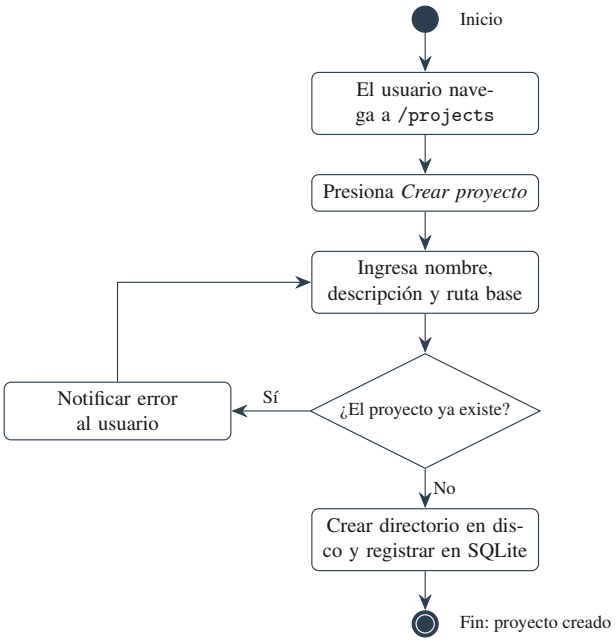


Figura 5.29: *Diagrama de flujo del CU-01: Gestionar Proyectos.*

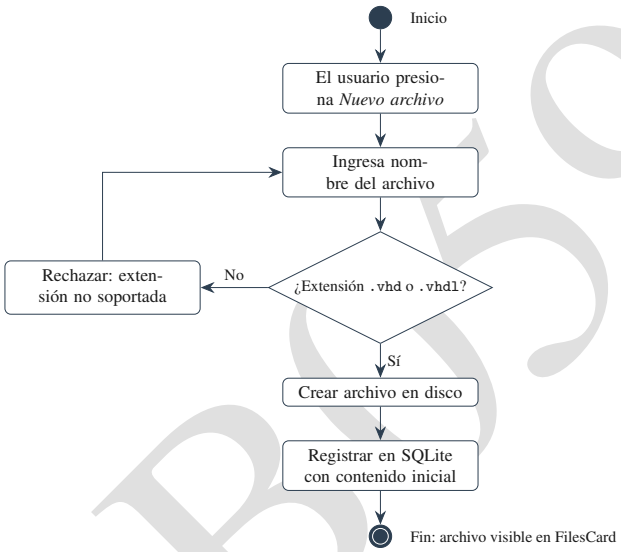


Figura 5.30: *Diagrama de flujo del CU-02: Gestionar Archivos HDL.*

5.3.2. CU-03 y CU-04: Edición de Código y Selección de Dispositivo FPGA

CU-03: Editar Código HDL.

Campo	Descripción
ID	CU-03
Actor	Estudiante
RF relacionado	RF-03
Precondición	Existe al menos un archivo abierto en el proyecto activo
Flujo principal	1. El usuario selecciona un archivo `.vhd` desde `FilesCard`. 2. El sistema carga el contenido en el editor Monaco (`StandaloneCodeEditor`) con resaltado de sintaxis VHDL. 3. El usuario edita el código. 4. El sistema actualiza el indicador de cambios sin guardar y la posición del cursor (línea y columna). 5. El usuario presiona Ctrl+S o el botón "Guardar". 6. El repositorio `ProjectFiles` escribe el contenido en disco y actualiza el respaldo en SQLite.
Flujo alternativo	2a. Si el archivo tiene extensión `.md`, el editor muestra vista dividida con previsualización Markdown.
Postcondición	El contenido editado persiste en disco y base de datos; el indicador de cambios sin guardar se desactiva.

CU-04: Seleccionar Dispositivo FPGA.

Campo	Descripción
ID	CU-04
Actor	Estudiante
RF relacionado	RF-05
Precondición	La aplicación se encuentra en ejecución
Flujo principal	1. El usuario selecciona un dispositivo FPGA desde el menú desplegable en la barra superior. 2. El servicio `JSONFileReader` carga el archivo JSON correspondiente al dispositivo. 3. El servicio `SelectedFPGA` actualiza el dispositivo activo. 4. La pestaña FPGA (`FPGATabContent`) muestra las especificaciones del dispositivo: LUTs, bloques de memoria, flip-flops, DSPs, PLLs y frecuencia máxima.
Flujo alternativo	2a. El archivo JSON del dispositivo no se encuentra: el sistema notifica el error.
Postcondición	El dispositivo seleccionado se utiliza como referencia para el cálculo de utilización de recursos.

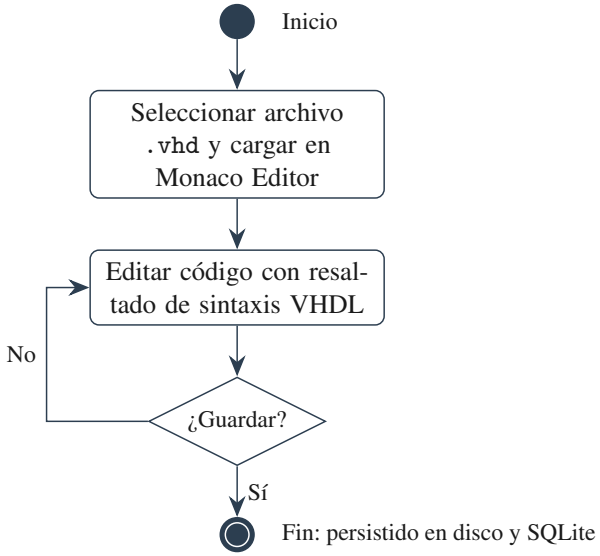


Figura 5.31: *Diagrama de flujo del CU-03: Editar Codigo HDL.*

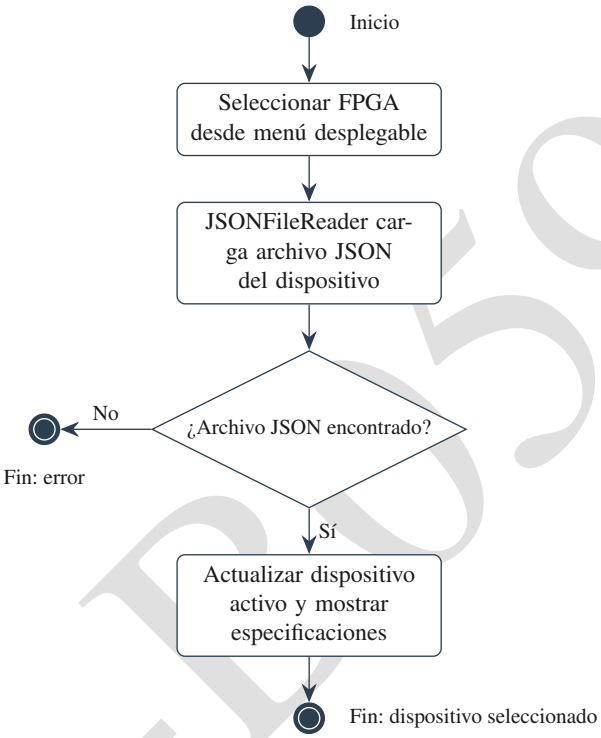


Figura 5.32: *Diagrama de flujo del CU-04: Seleccionar Dispositivo FPGA.*

5.3.3. CU-05 y CU-06: Preferencias de Idioma y Tema Visual

CU-05: Cambiar Idioma.

Campo	Descripción
ID	CU-05
Actor	Estudiante
RF relacionado	RF-09
Precondición	La aplicación se encuentra en ejecución
Flujo principal	1. El usuario selecciona un idioma desde el menú de configuración. 2. El servicio `Traductor` ejecuta `SetLanguageAsync(code)`. 3. El diccionario JSON del idioma seleccionado se carga desde `wwwroot/Dictionaries/`. 4. La preferencia se persiste en SQLite mediante `ApplicationDbContext.SetSettingAsync`. 5. El evento `StaticOnLangChanged` notifica a todos los componentes suscritos. 6. La interfaz se actualiza de forma reactiva sin recarga de página.
Flujo alternativo	1a. Si el idioma seleccionado es árabe (AR), el sistema activa el atributo `dir=rtl"` para escritura de derecha a izquierda.
Postcondición	La interfaz muestra todos los textos en el idioma seleccionado; la preferencia persiste entre sesiones.

CU-06: Cambiar Tema Visual.

Campo	Descripción
ID	CU-06
Actor	Estudiante
RF relacionado	RF-10
Precondición	La aplicación se encuentra en ejecución
Flujo principal	1. El usuario alterna entre tema claro y oscuro desde la configuración. 2. El sistema actualiza las propiedades CSS personalizadas (`-kmila-primary`, `-kmila-primary-light`, `-kmila-primary-dark`, `-kmila-accent`). 3. La preferencia se persiste en SQLite mediante la clave `KEY_THEME` de `AppSettings`. 4. El evento `StaticOnValueChanged` notifica a los componentes suscritos. 5. La interfaz refleja el nuevo tema sin recarga.
Flujo alternativo	—
Postcondición	La interfaz muestra la paleta de colores del tema seleccionado; la preferencia persiste entre sesiones.

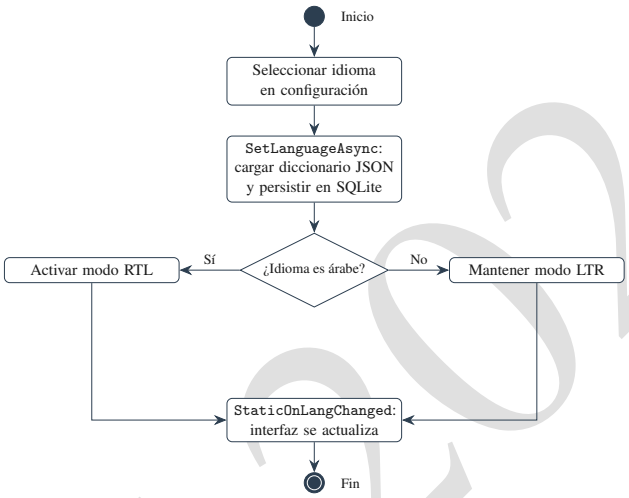


Figura 5.33: *Diagrama de flujo del CU-05: Cambiar Idioma.*

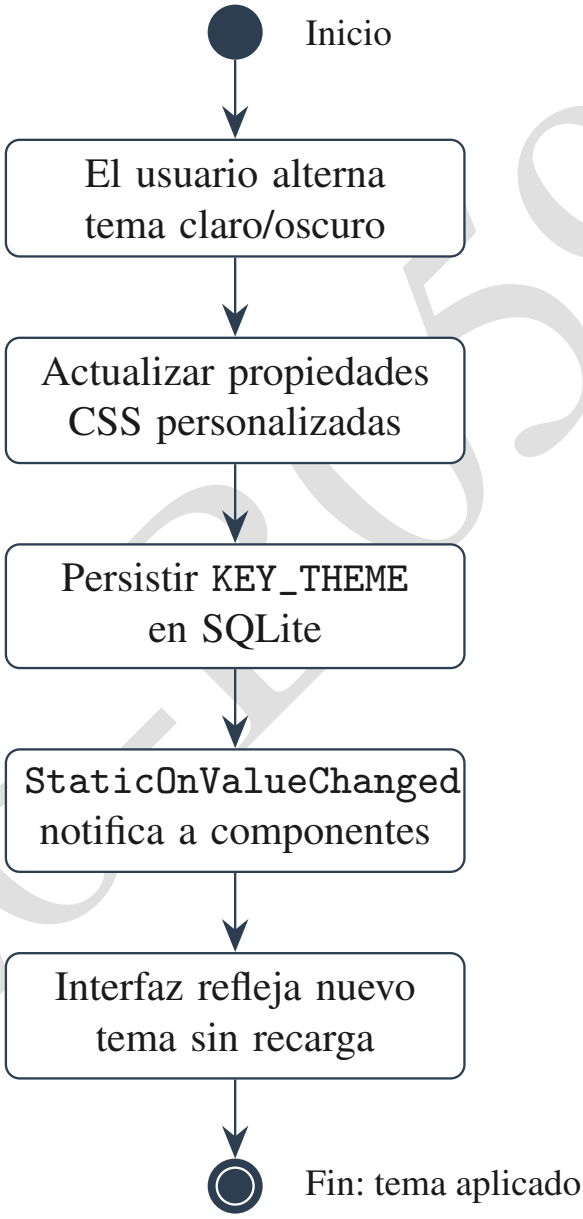


Figura 5.34: *Diagrama de flujo del CU-06: Cambiar Tema Visual.*

5.3.4. CU-07 y CU-08: Validación y Ejecución de Simulación (TT2)

CU-07: Validar Código VHDL.

Campo	Descripción
ID	CU-07
Actor	Estudiante
RF relacionado	RF-12, RF-13
Precondición	Existe un archivo `.vhd` o `.vhdl` abierto en el editor
Flujo principal	1. El usuario presiona "Validar programa" en `ButtonCollapse`. 2. `ProgramBuilder.Build(ProjectFile)` valida la extensión del archivo. 3. Se crea una instancia de `Parser` con la ruta del archivo. 4. `Parser.ParseFile()` ejecuta el análisis léxico y sintáctico mediante una máquina de estados (default, in-entity, in-architecture). 5. El sistema extrae entidades, puertos, señales y arquitecturas. 6. El componente `BuildProgressDisplay` muestra el progreso en tiempo real mediante el evento `OnBuildProgressChanged`. 7. Al completarse, el sistema muestra las entidades y recursos analizados en la pestaña de diagnósticos.
Flujo alternativo	4a. Error de sintaxis: `OnBuildError` notifica el error con el mensaje y la ubicación en el código.
Postcondición	`ProgramBuilder.Parser` contiene las entidades extraídas, habilitando la ejecución de simulación.

CU-08: Configurar y Ejecutar Simulación.

Campo	Descripción
ID	CU-08
Actor	Estudiante
RF relacionado	RF-06, RF-14, RF-15
Precondición	El código ha sido validado exitosamente (CU-07)
Flujo principal	1. El usuario presiona "Iniciar simulación". 2. El sistema muestra `SimulationParametersModal` con los puertos detectados. 3. El usuario configura la duración, escala temporal, frecuencia de reloj y estímulos por puerto (pares tiempo-valor). 4. `StimulusBuilder.BuildConfig()` convierte los parámetros a `SimulationConfig` (detecta relojes, calcula ticks, genera eventos de estímulo). 5. `SimulationRunner.RunAsync()` ejecuta la simulación tick por tick. 6. En cada tick, `DeltaCycleEngine` aplica estímulos, alterna relojes y ejecuta ciclos delta hasta alcanzar un estado estable. 7. `SignalHistory` registra los cambios de cada señal. 8. `OnSimulationCompleted` entrega el historial al componente `WaveformViewer`.
Flujo alternativo	6a. El usuario cancela la simulación: el sistema detiene la ejecución y entrega los resultados parciales.
Postcondición	El historial de señales está disponible para visualización y exportación.

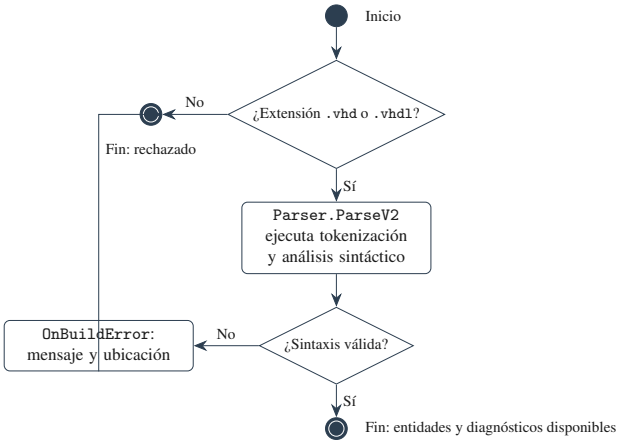


Figura 5.35: *Diagrama de flujo del CU-07: Validar Codigo VHDL.*

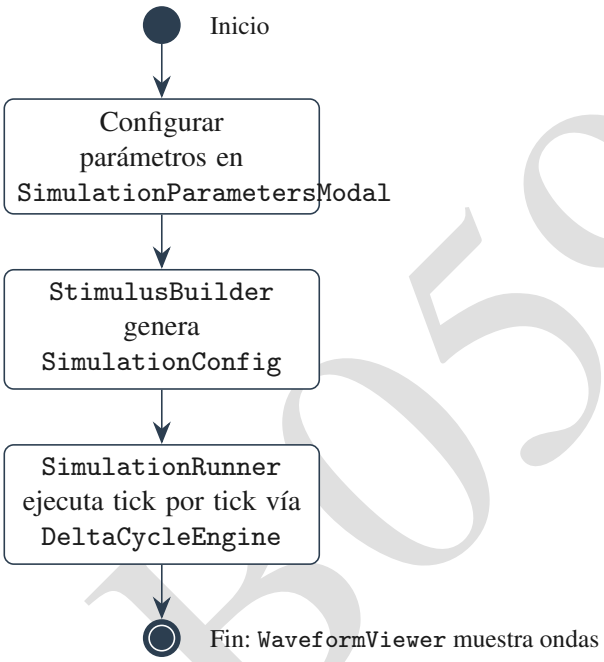


Figura 5.36: *Diagrama de flujo del CU-08: Configurar y Ejecutar Simulacion.*

5.3.5. CU-09 y CU-10: Visualización y Exportación de Resultados (TT2)

CU-09: Visualizar Formas de Onda.

Campo	Descripción
ID	CU-09
Actor	Estudiante
RF relacionado	RF-08
Precondición	Se ha ejecutado una simulación que produjo datos de señales
Flujo principal	1. `WaveformData.FromSignalHistory()` transforma el historial de señales en un modelo de trazas. 2. El componente `WaveformViewer` renderiza una traza SVG de línea escalonada independiente por cada señal. 3. Las señales se codifican por color según su tipo: verde para señales de reloj, azul para entradas, naranja para salidas y morado para vectores. 4. El usuario puede acercar/alejar el eje temporal y desplazarse horizontalmente.
Flujo alternativo	—
Postcondición	Las formas de onda se presentan al usuario de forma interactiva.

CU-10: Exportar Resultados VCD.

Campo	Descripción
ID	CU-10
Actor	Estudiante
RF relacionado	RF-18
Precondición	Se ha ejecutado una simulación que produjo datos de señales
Flujo principal	1. El usuario presiona “Exportar VCD” en `WaveformViewer`. 2. `SignalHistory.ExportToVcd()` genera el contenido en formato Value Change Dump (VCD). 3. El sistema escribe el archivo `.vcd` en el directorio del proyecto.
Flujo alternativo	—
Postcondición	El archivo VCD es compatible con visores externos como GTKWave.

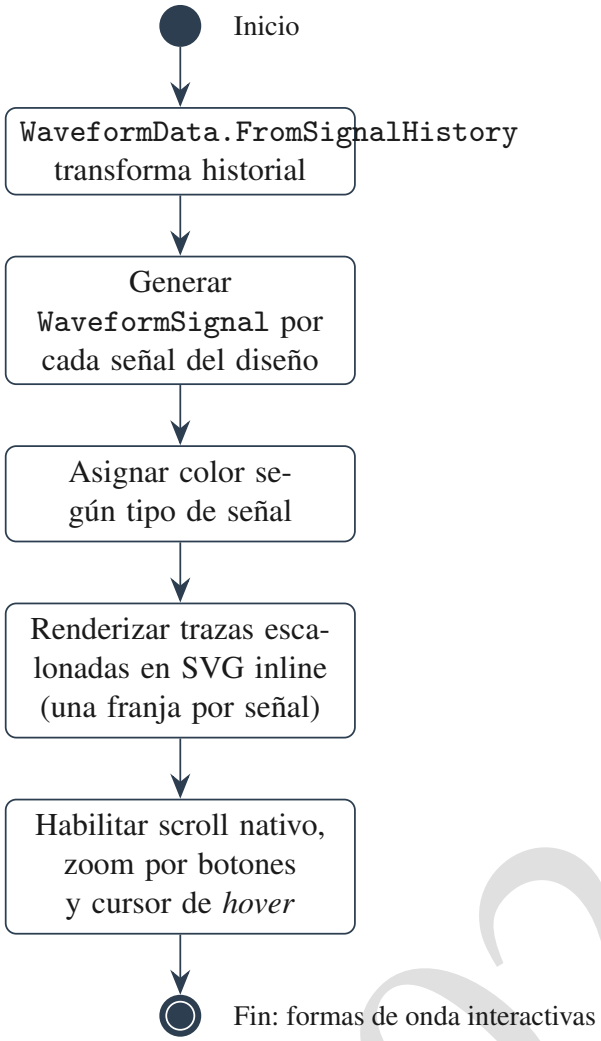


Figura 5.37: *Diagrama de flujo del CU-09: Visualizar Formas de Onda.*

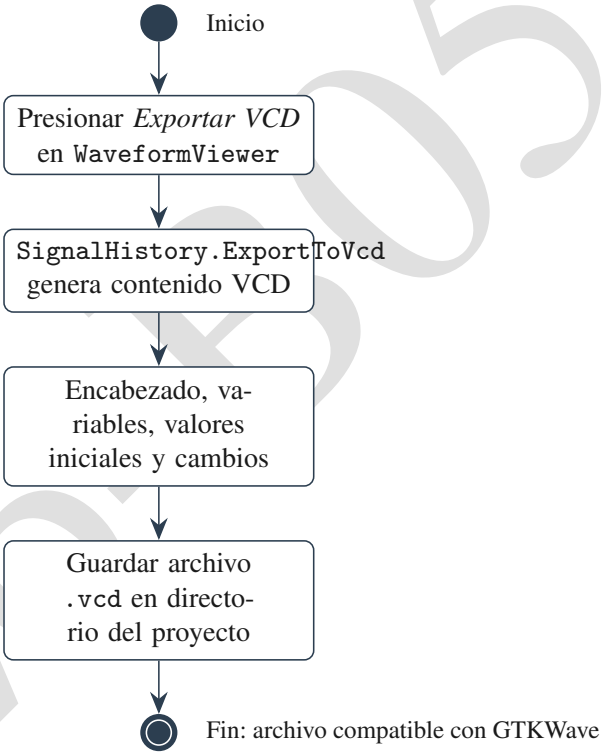


Figura 5.38: *Diagrama de flujo del CU-10: Exportar Resultados VCD.*

5.3.6. CU-11 a CU-14: Modos Interactivos (TT2)

Los modos interactivos incorporados durante TT2 añaden cuatro casos de uso al diagrama de la Figura 5.28: **CU-11** ejecutar en modo visual interactivo (*Visual Runtime*), **CU-12** depurar en vivo con *breakpoints* y paso a paso, **CU-13** editar mediante bloques, y **CU-14** inspeccionar señales (ocultar filas y cursor de *hover*, cuyo flujo se integra en el de CU-09). Las Figura 5.39, Figura 5.40 y Figura 5.41 detallan los tres primeros.

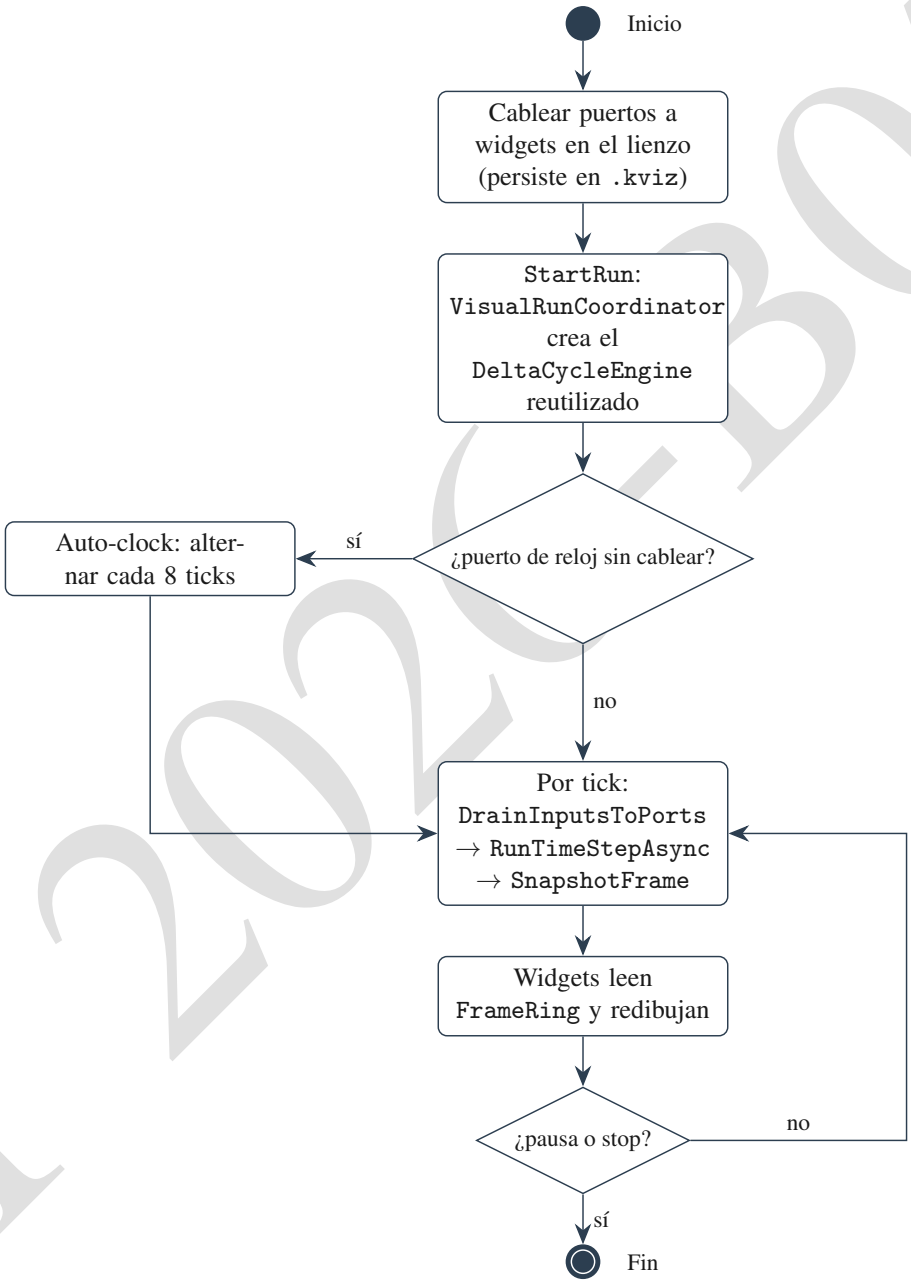


Figura 5.39: *Flujo del CU-11 (*Visual Runtime*): cablear puertos a widgets, arrancar el coordinador sobre el motor reutilizado, autogenerar el reloj cuando aplica, y ciclar volcado de entradas → paso del motor → captura de fotograma hasta pausar o detener.*

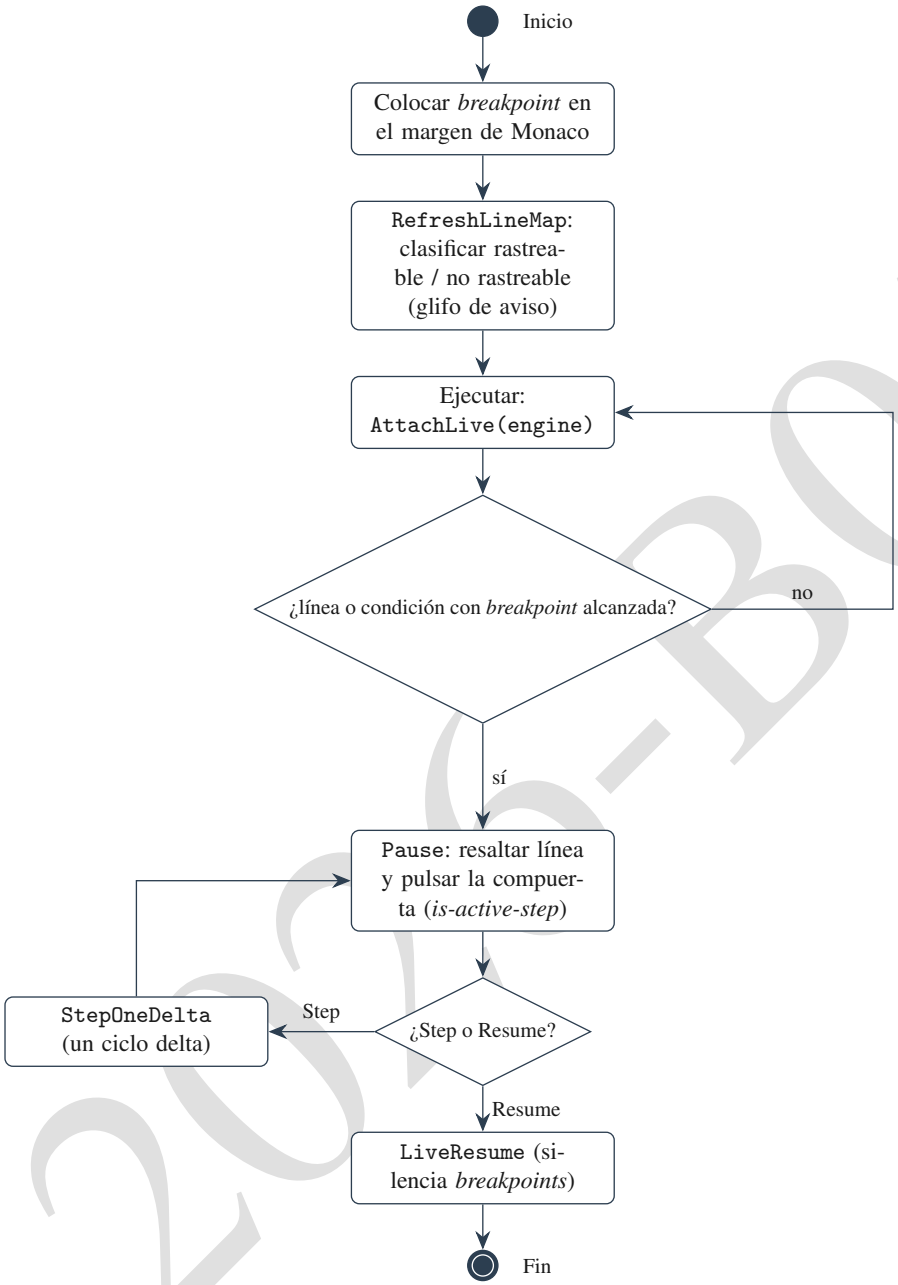


Figura 5.40: *Flujo del CU-12 (depuración en vivo): colocar y clasificar el* breakpoint*, adjuntar el motor en vivo, y ante una línea o condición con* breakpoint *pausar, resaltar y ofrecer paso a paso o reanudar.*

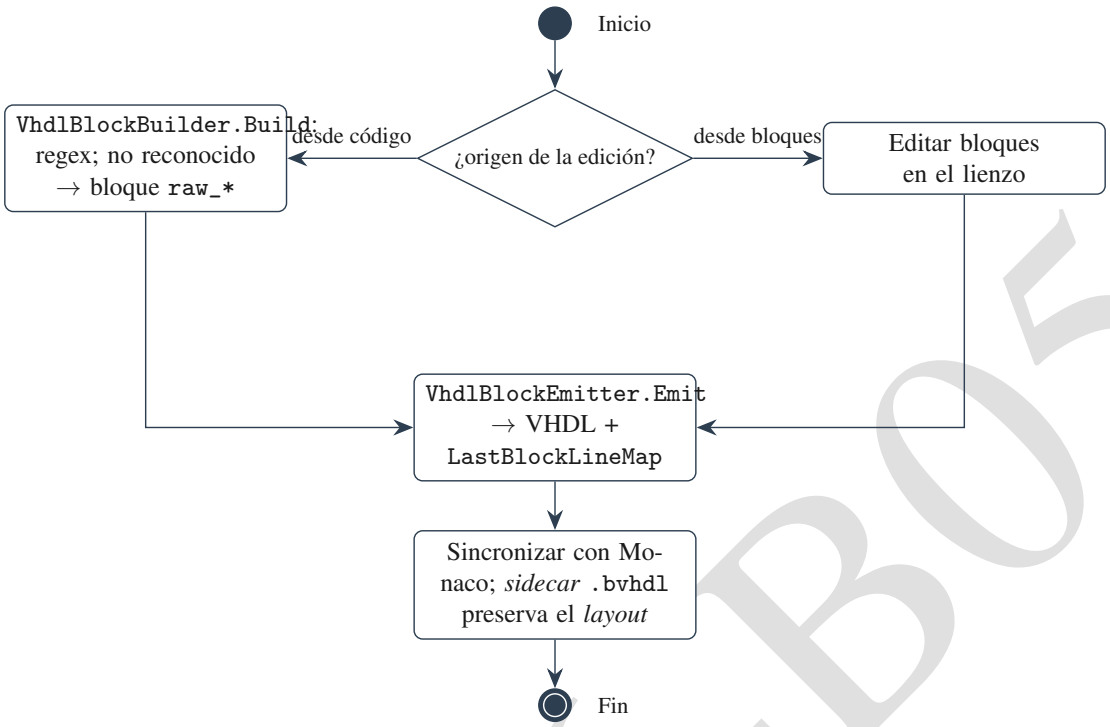


Figura 5.41: *Flujo del CU-13 (edición por bloques): según el origen (código o bloques), traducir con `VhdlBlockBuilder` o regenerar con `VhdlBlockEmitter`, sincronizar con Monaco y preservar el* layout *en el* sidecar `.bvhdl`.

5.4. Diagramas de Secuencia

5.4.1. Secuencia: Carga de Archivo HDL

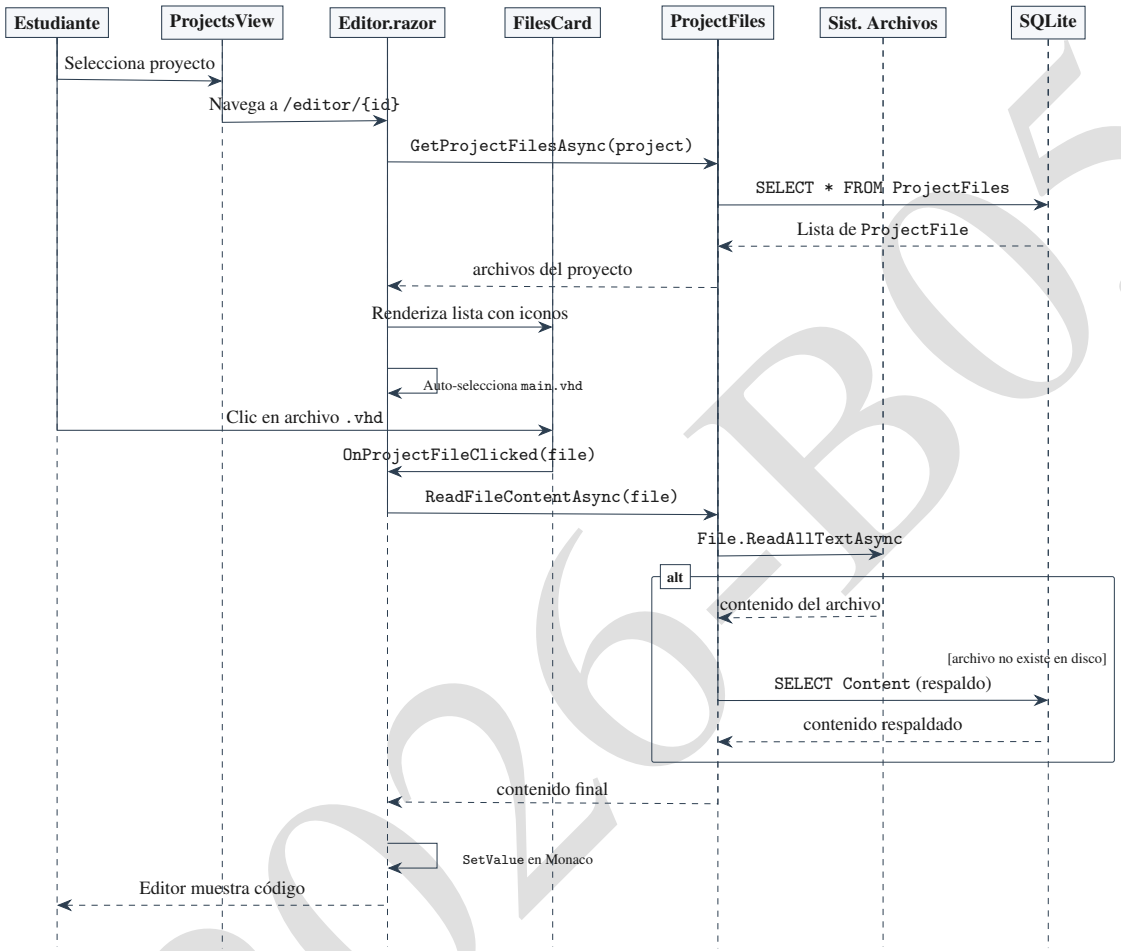


Figura 5.42: *Diagrama de secuencia: carga de archivo HDL. El repositorio ProjectFiles implementa persistencia dual: lee de disco y recurre a SQLite si el archivo fue eliminado.*

5.4.2. Secuencia: Validación de Código VHDL

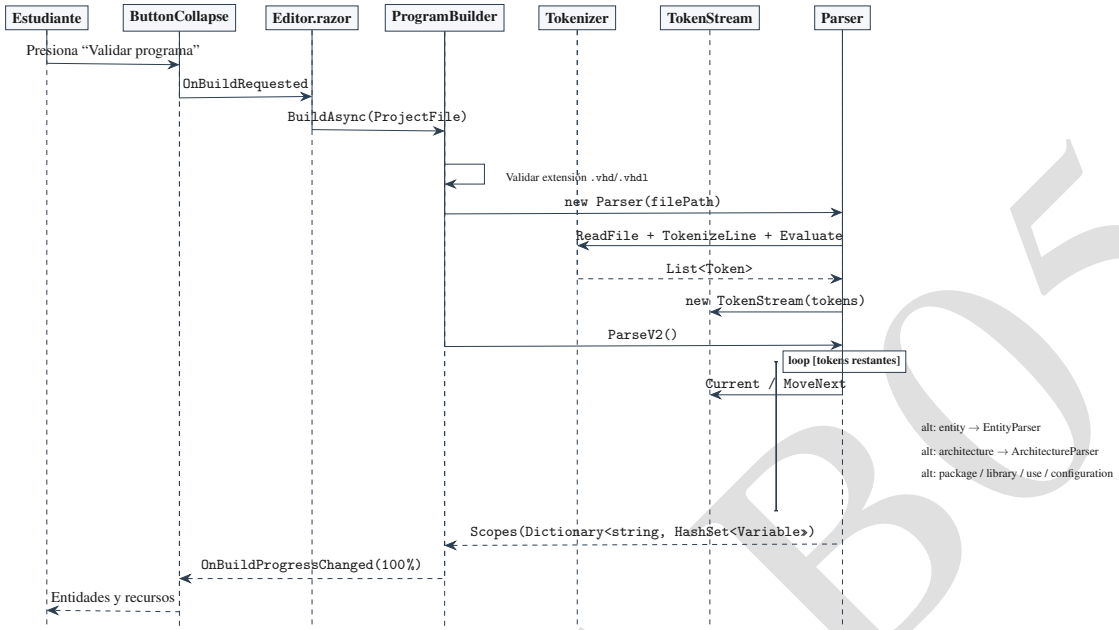


Figura 5.43: *Diagrama de secuencia: validacion de codigo VHDL. El Parser delega el analisis a EntityParser, ArchitectureParser y PackageParser segun el tipo del token actual del TokenStream.*

5.4.3. Secuencia: Configuración de Simulación

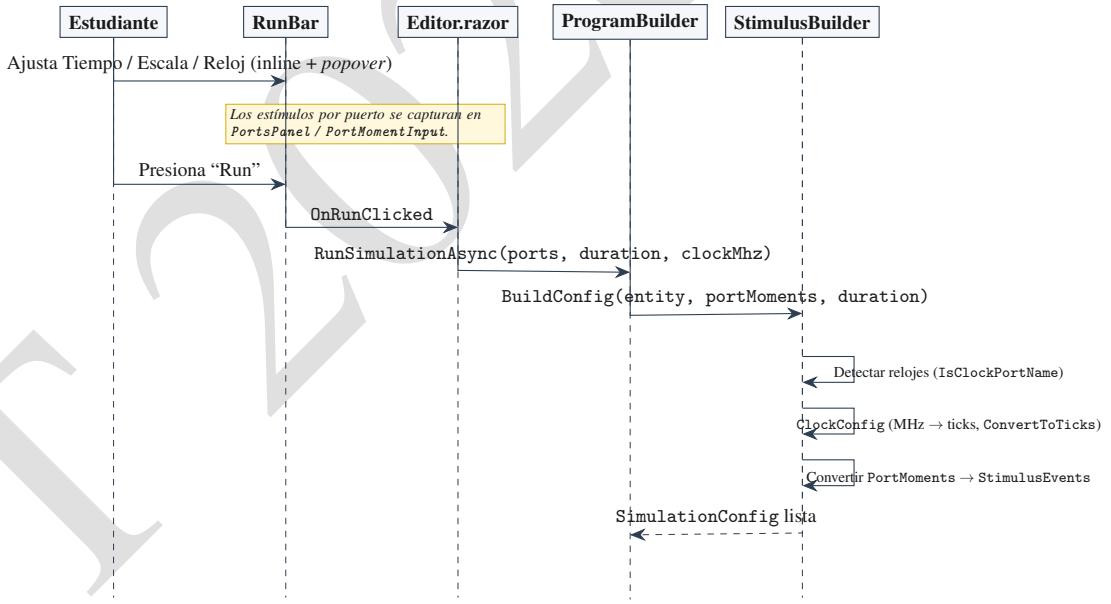


Figura 5.44: *Diagrama de secuencia: configuracion de simulacion. StimulusBuilder convierte los parametros de la interfaz al formato SimulationConfig del motor.*

5.4.4. Secuencia: Ejecución del Motor de Simulación

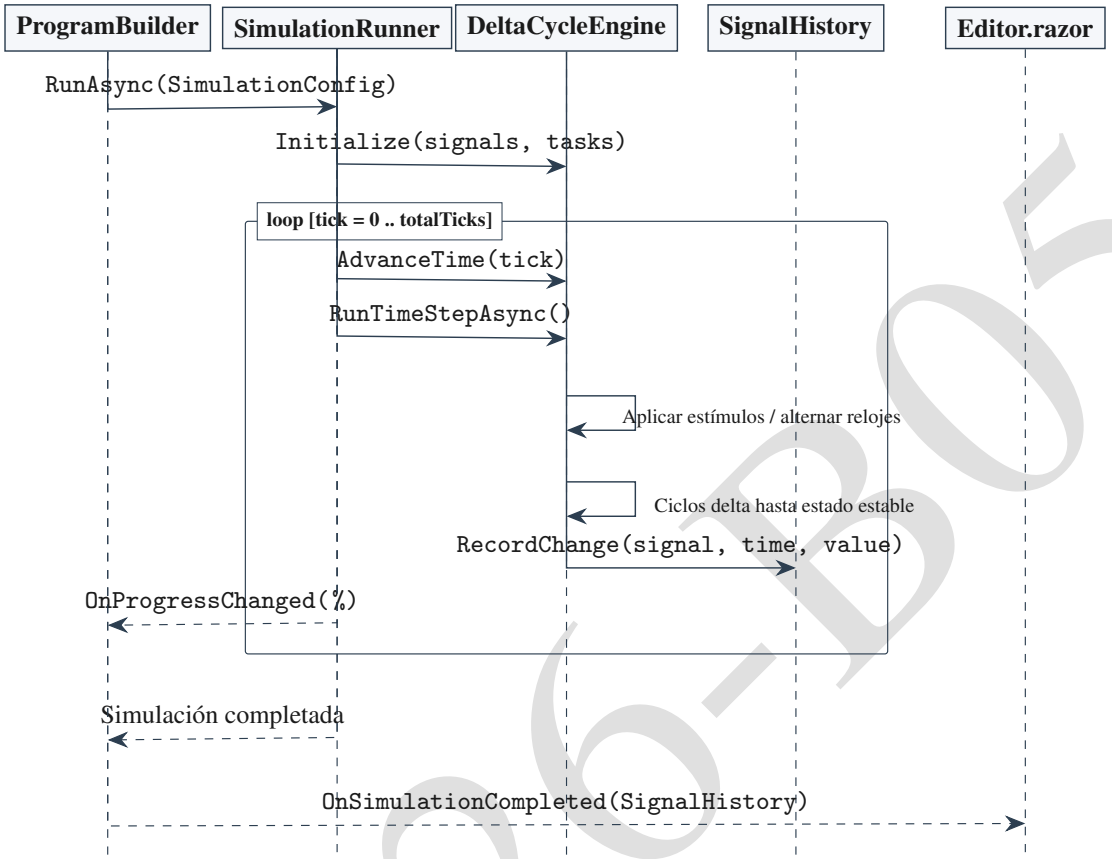


Figura 5.45: *Diagrama de secuencia: ejecucion del motor de simulacion. SimulationRunner itera tick por tick; DeltaCycleEngine resuelve ciclos delta y registra cambios en SignalHistory.*

5.4.5. Secuencia: Resolución de Sensitivity (TT2)

El motor de simulación calcula dinámicamente la lista de sensitivity efectiva de cada `process` VHDL para determinar cuándo debe re-evaluarse. Ante un cambio de señal, `SignalScheduler` notifica a `ProcessScheduler` los procesos sensibles, los cuales se reencolan para evaluación en el siguiente *delta cycle*.

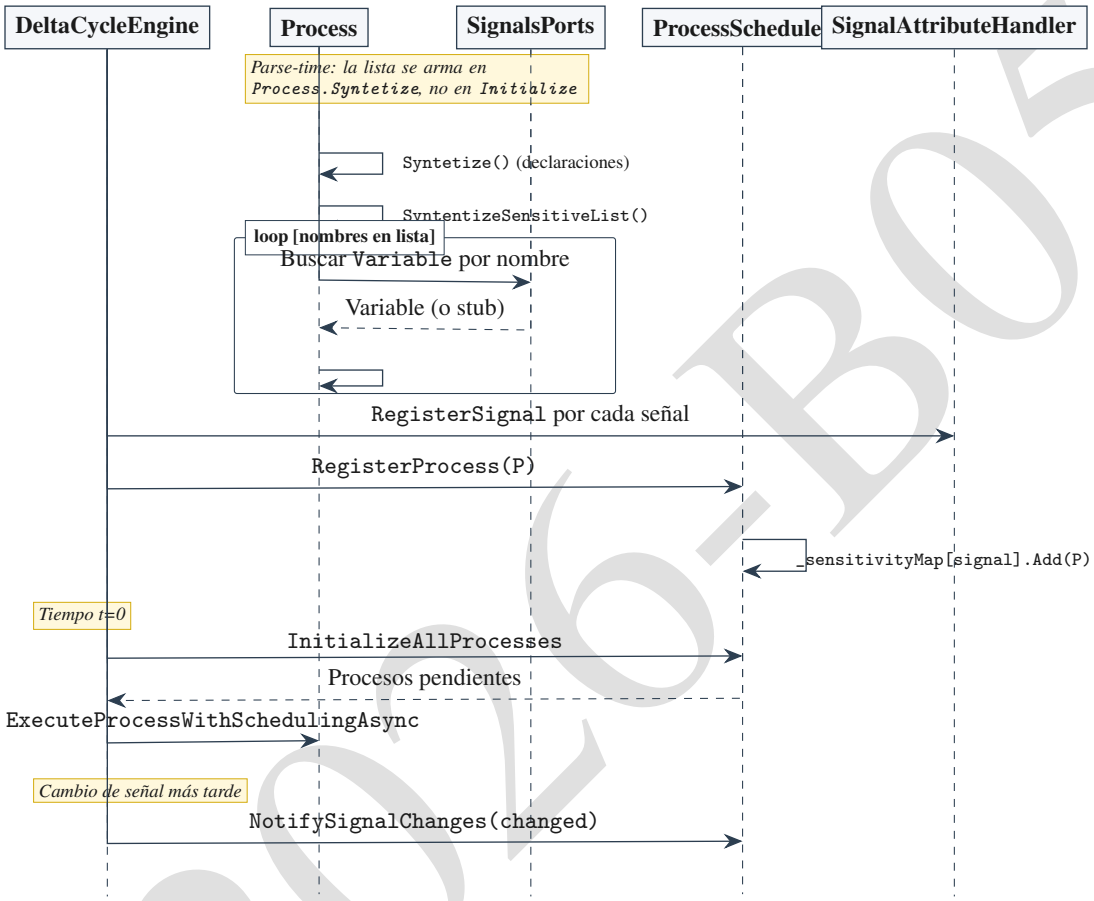


Figura 5.46: *Diagrama de secuencia: resolución de la lista de sensitivity. Cooperación entre `SignalScheduler` y `ProcessScheduler` para determinar qué procesos deben re-evaluarse tras un cambio de señal.*

5.4.6. Secuencia: Evaluación de Práctica (TT2)

El subsistema *Practice* ejecuta una simulación contra la solución de referencia para cada `TestCase` y compara los `WaveformData` resultantes con tolerancia numérica configurable.

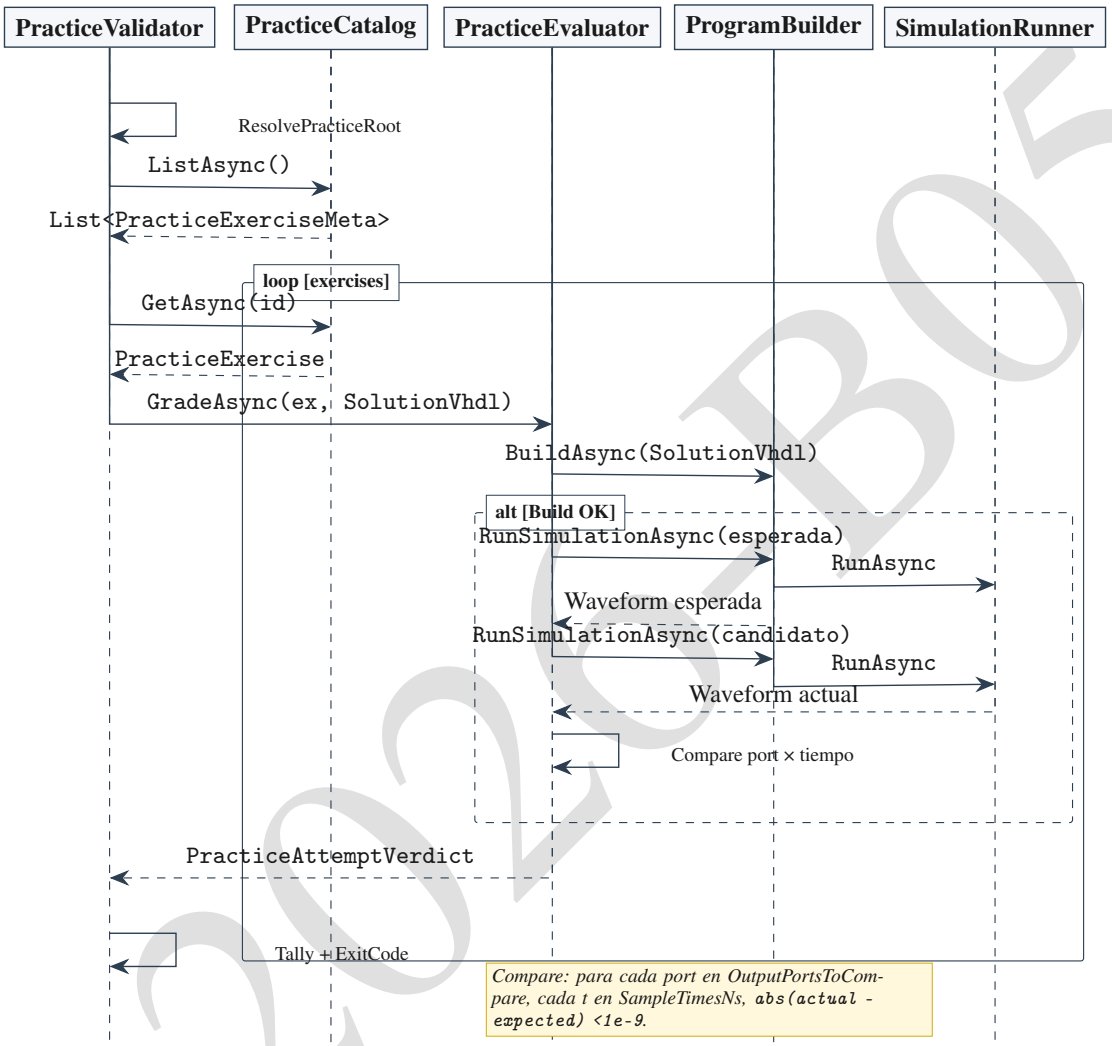


Figura 5.47: *Diagrama de secuencia: evaluación automática de práctica.* `PracticeEvaluator` *compila el código candidato y el código solución (con caché de soluciones), ejecuta ambas simulaciones, compara las formas de onda con tolerancia* 10^{-9} *y persiste el veredicto.*

5.4.7. Secuencia: Guardado de Archivo

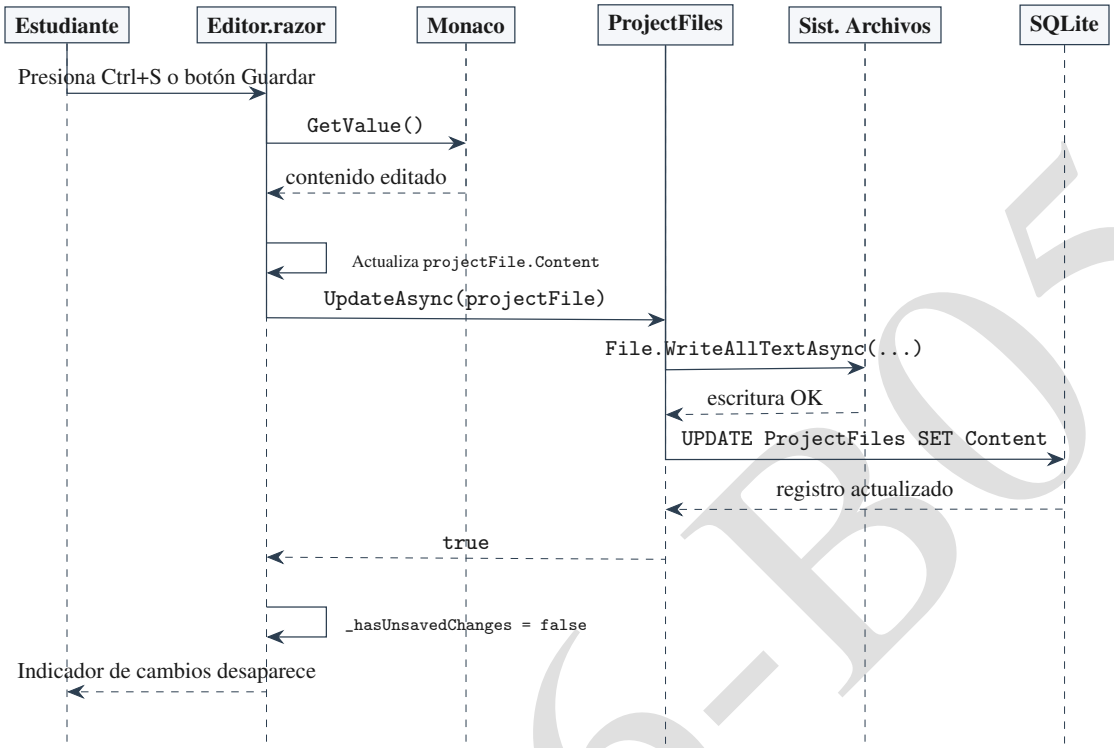


Figura 5.48: *Diagrama de secuencia: guardado de archivo. La persistencia dual escribe simultaneamente en disco y SQLite para garantizar la recuperacion ante perdida del archivo.*

5.4.8. Secuencia: Exportación de Resultados VCD

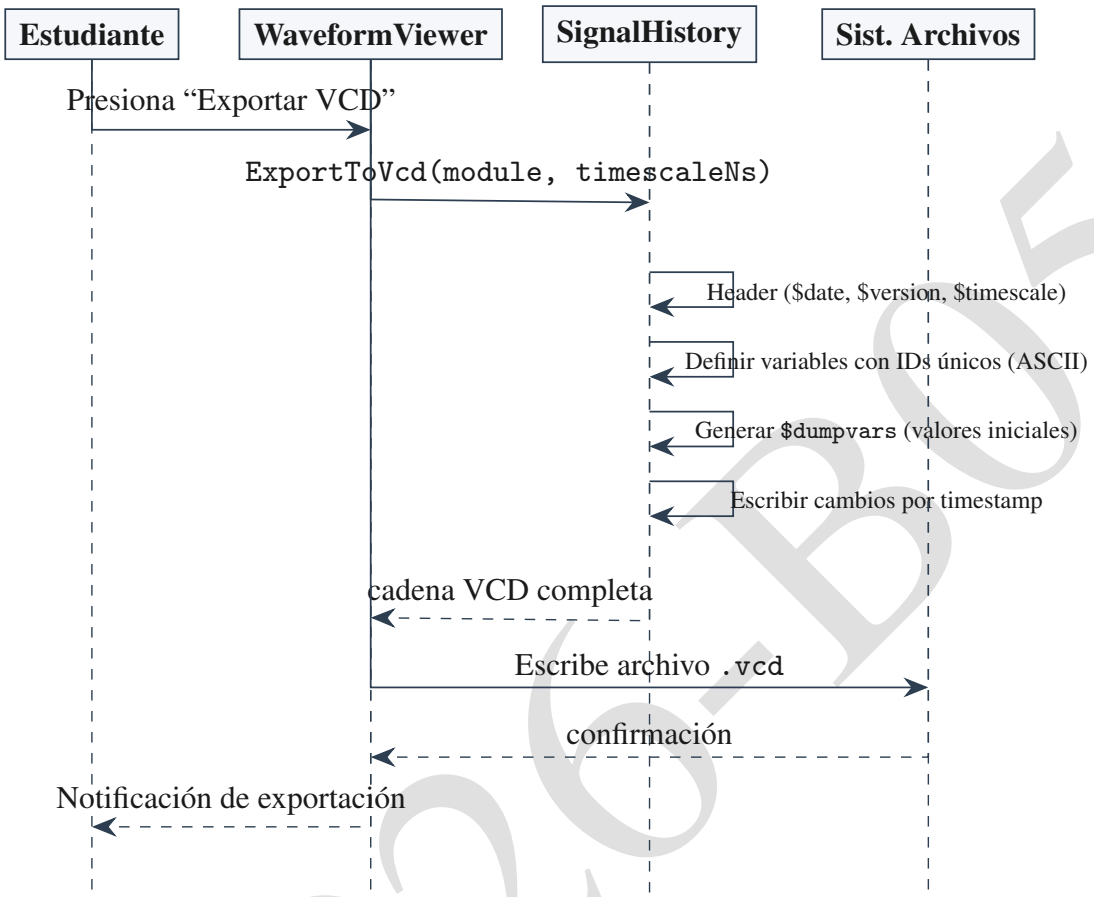


Figura 5.49: *Diagrama de secuencia: exportacion de resultados en formato VCD. El archivo generado es compatible con visores externos como GTKWave.*

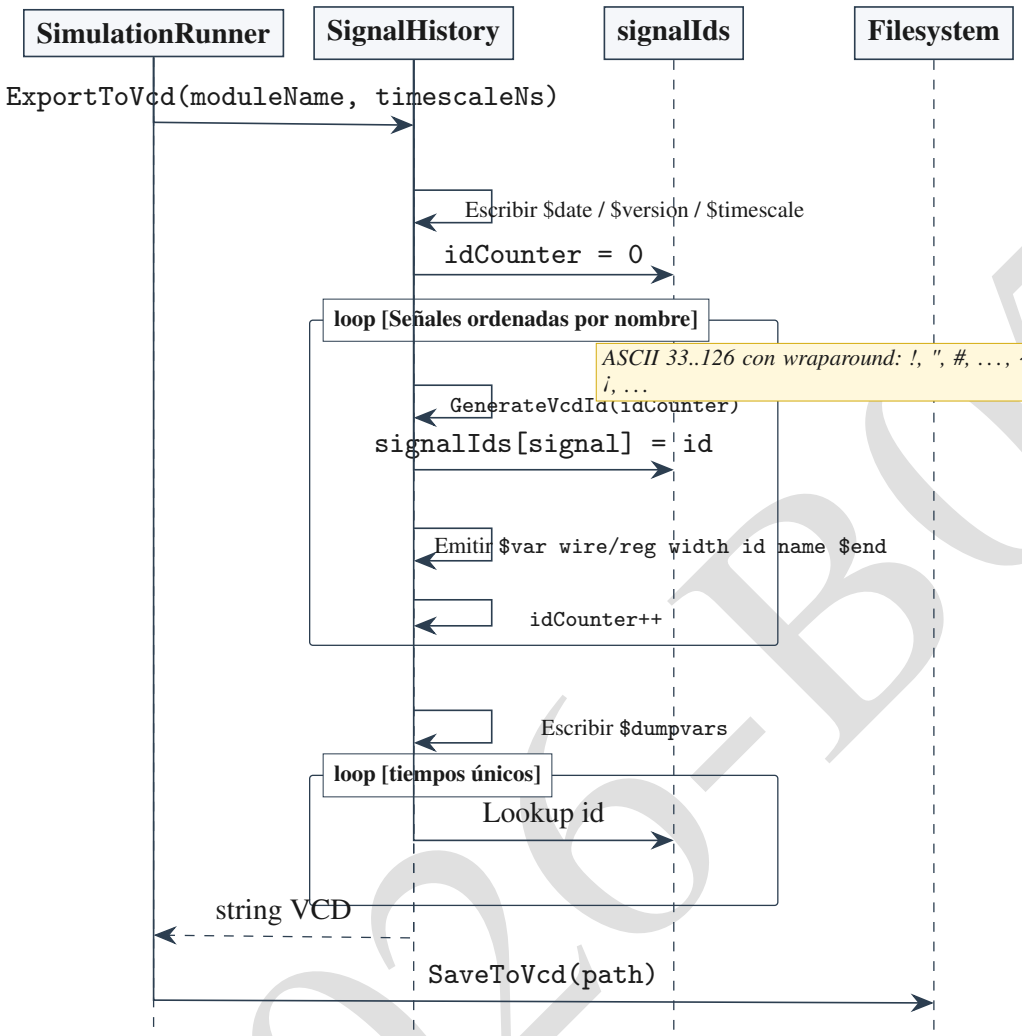


Figura 5.50: *Detalle del exportador VCD: asignación de identificadores cortos a cada señal del diseño. El exportador mapea cada* `Signal` *interna a un identificador codificado en* base-94 printable ASCII *para minimizar el tamaño del archivo VCD resultante.*

5.4.9. Secuencia: Pipeline de Replay post-simulación

Cada vez que una simulación termina con éxito, el editor dispara un proceso asíncrono que convierte el resultado en un documento reproducible paso a paso. La Figura 5.51 muestra la coordinación entre los servicios involucrados.

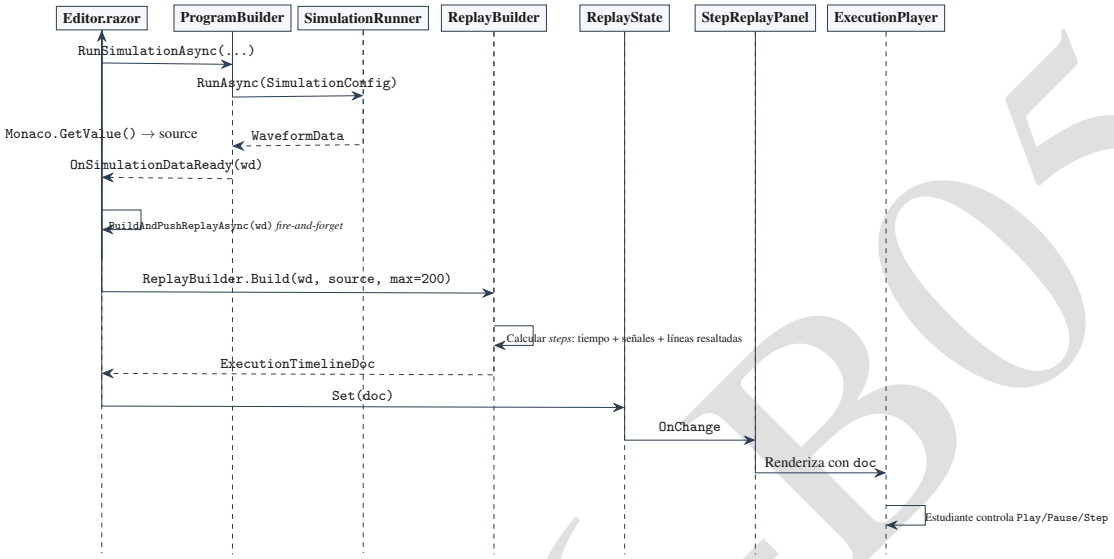


Figura 5.51: *Diagrama de secuencia del pipeline* Replay*: tras* `OnSimulationDataReady`*,* `Editor.razor` *invoca* `ReplayBuilder.Build` *con la* `WaveformData` *y el fuente actual; el documento resultante se inyecta en* `ReplayState`*, que notifica al* `StepReplayPanel` *para que renderice el* `ExecutionPlayer`*. El estudiante recorre la corrida con la barra de transporte sin necesidad de re-simular.*

5.5. Diagramas de Actividades

5.5.1. Actividad: Gestión de Proyecto y Edición

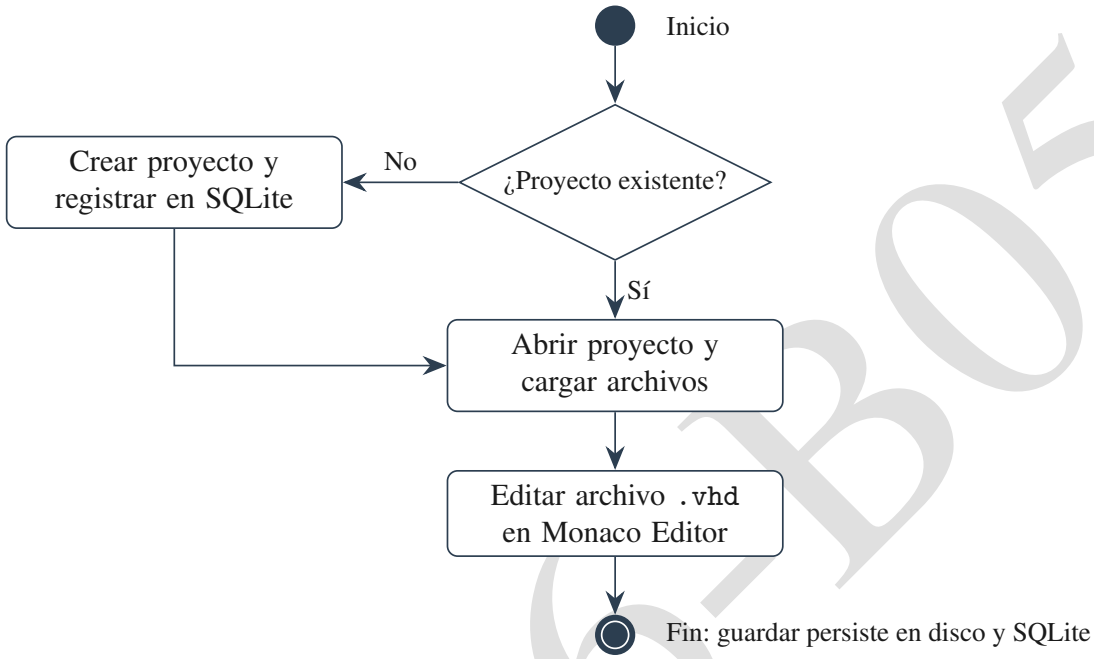


Figura 5.52: *Diagrama de actividades: gestion de proyecto y edicion. Cubre el flujo desde la creacion o apertura de un proyecto hasta el guardado del archivo editado.*

5.5.2. Actividad: Validación de Código VHDL

El proceso de validación se describe en dos partes: la preparación del Parser y el recorrido del archivo fuente.

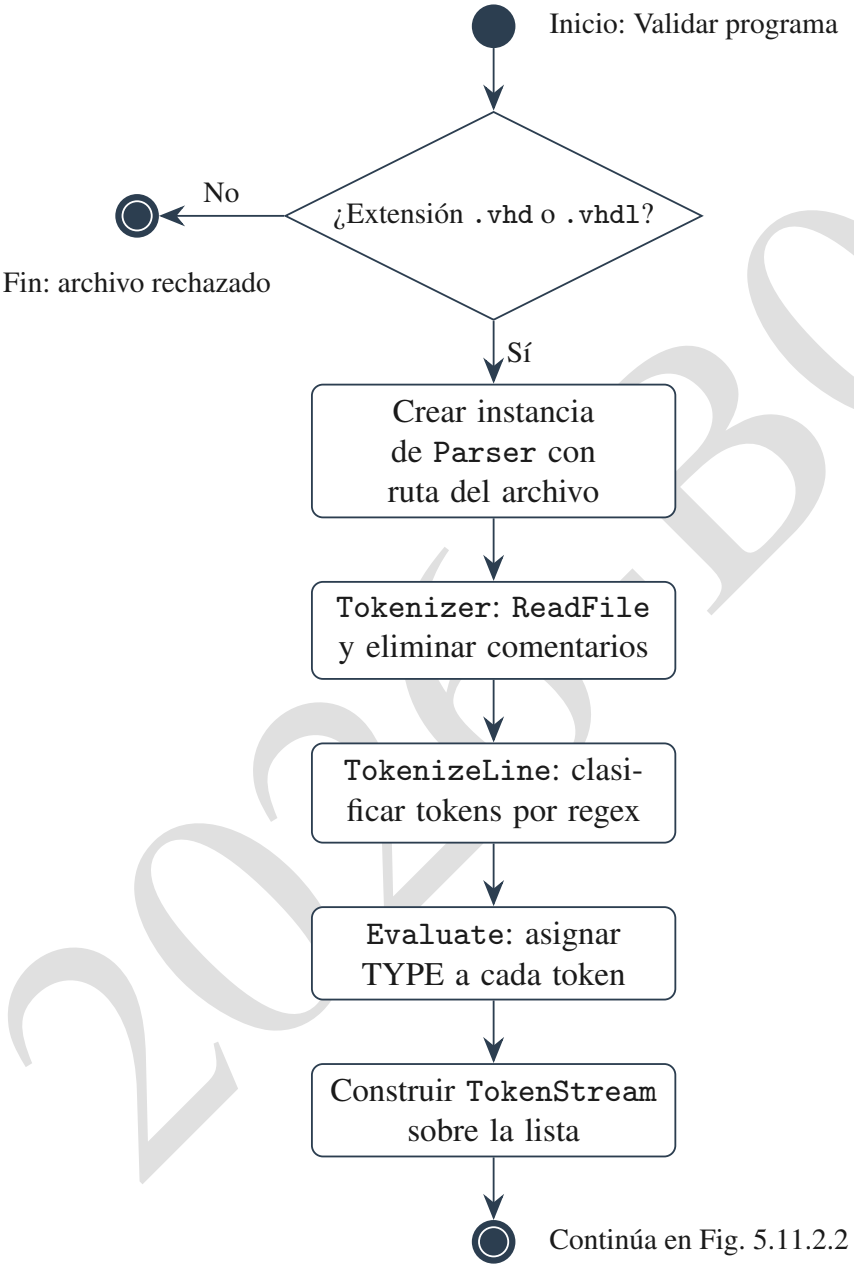


Figura 5.53: *Actividad de validacion: preparacion del Tokenizer (lectura del archivo, eliminacion de comentarios y clasificacion de tokens por regex) y construccion del TokenStream que alimenta a ParseV2.*

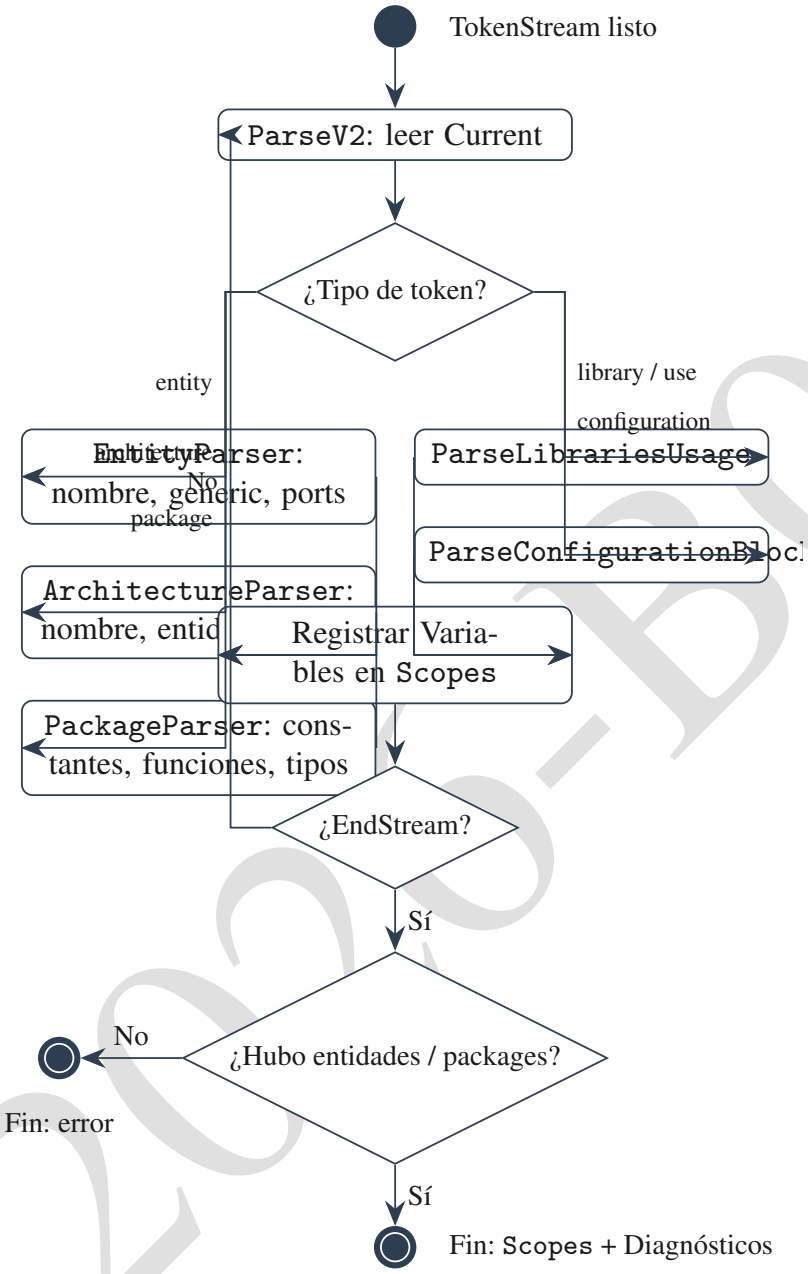


Figura 5.54: *Actividad de validacion: recorrido token por token del TokenStream. ParseV2 inspecciona el token actual y delega a EntityParser, ArchitectureParser, PackageParser, ParseLibrariesUsage o ParseConfigurationBlock; cada delegado registra sus Variables en el diccionario Scopes.*

5.5.3. Actividad: Configuración de Simulación

El proceso de configuración se describe en dos partes: la captura de parámetros y la generación de la configuración del motor.

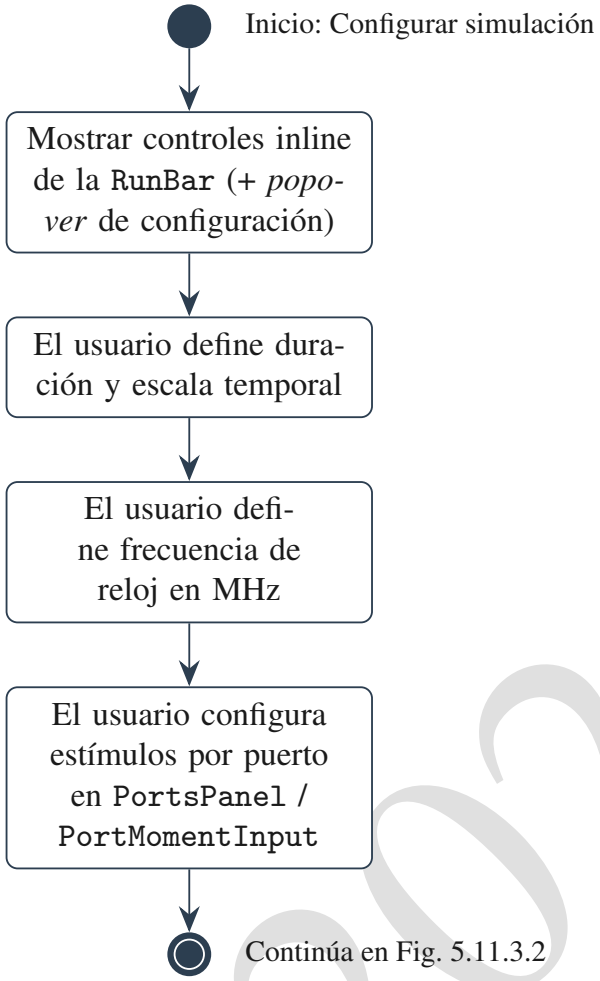


Figura 5.55: *Actividad de configuracion: captura de parametros del usuario.*

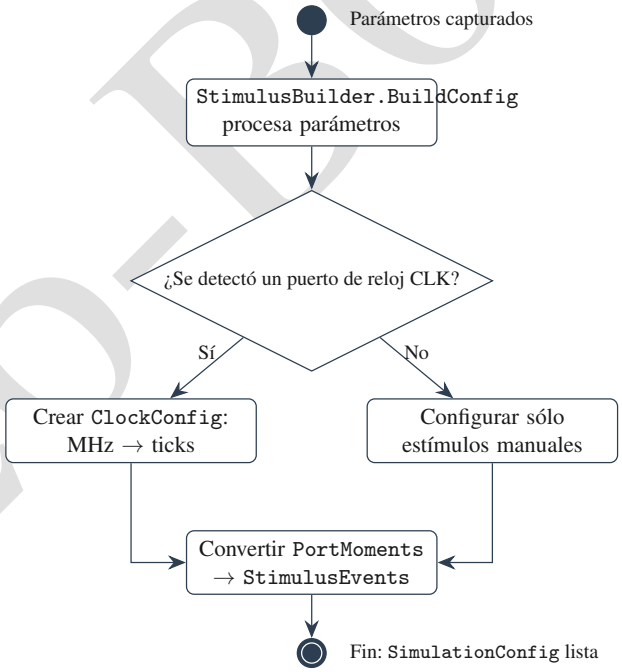


Figura 5.56: *Actividad de configuracion: StimulusBuilder genera SimulationConfig a partir de los parametros capturados.*

5.5.4. Actividad: Ejecución de Simulación

El proceso de ejecución se describe en dos partes: la inicialización del motor y el ciclo principal de simulación.

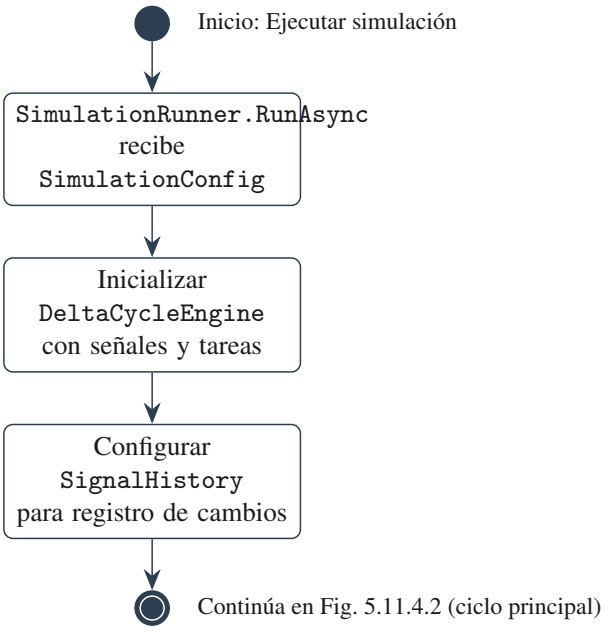


Figura 5.57: *Actividad de ejecucion: inicializacion del motor de simulacion.*

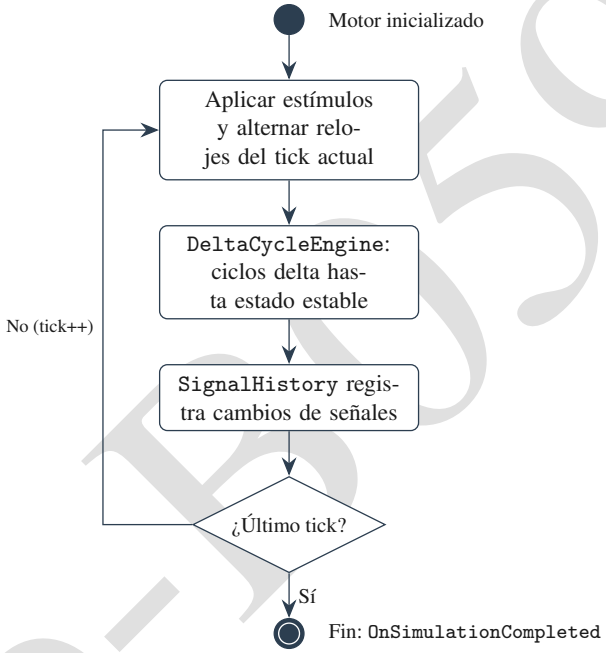


Figura 5.58: *Actividad de ejecucion: ciclo principal tick por tick. DeltaCycleEngine resuelve ciclos delta en cada instante de tiempo.*

5.5.5. Actividad: Visualización y Exportación

El proceso se describe en dos partes: la transformación y visualización de datos, y la exportación opcional.

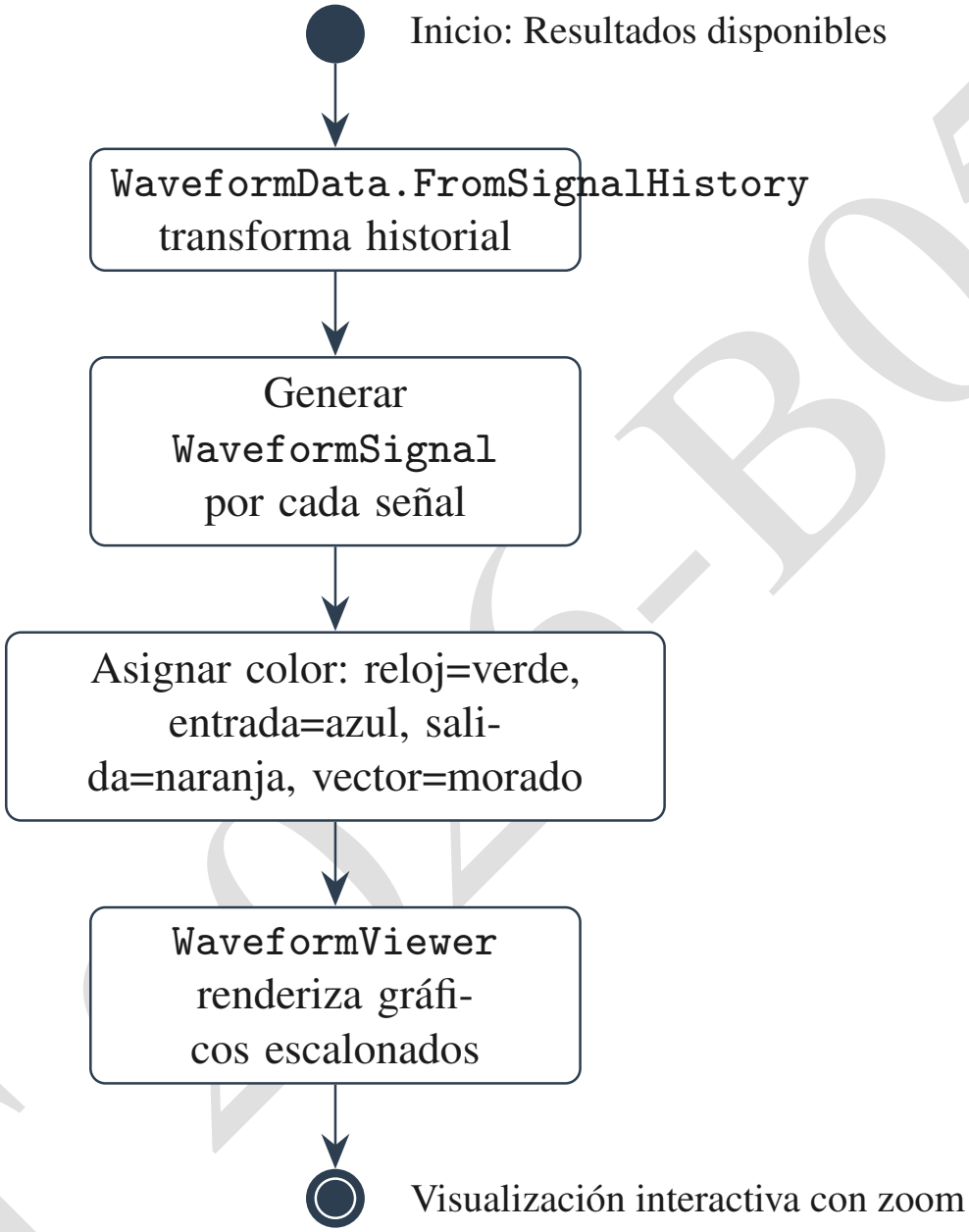


Figura 5.59: *Actividad de visualizacion: transformacion del historial de senales en graficos interactivos.*

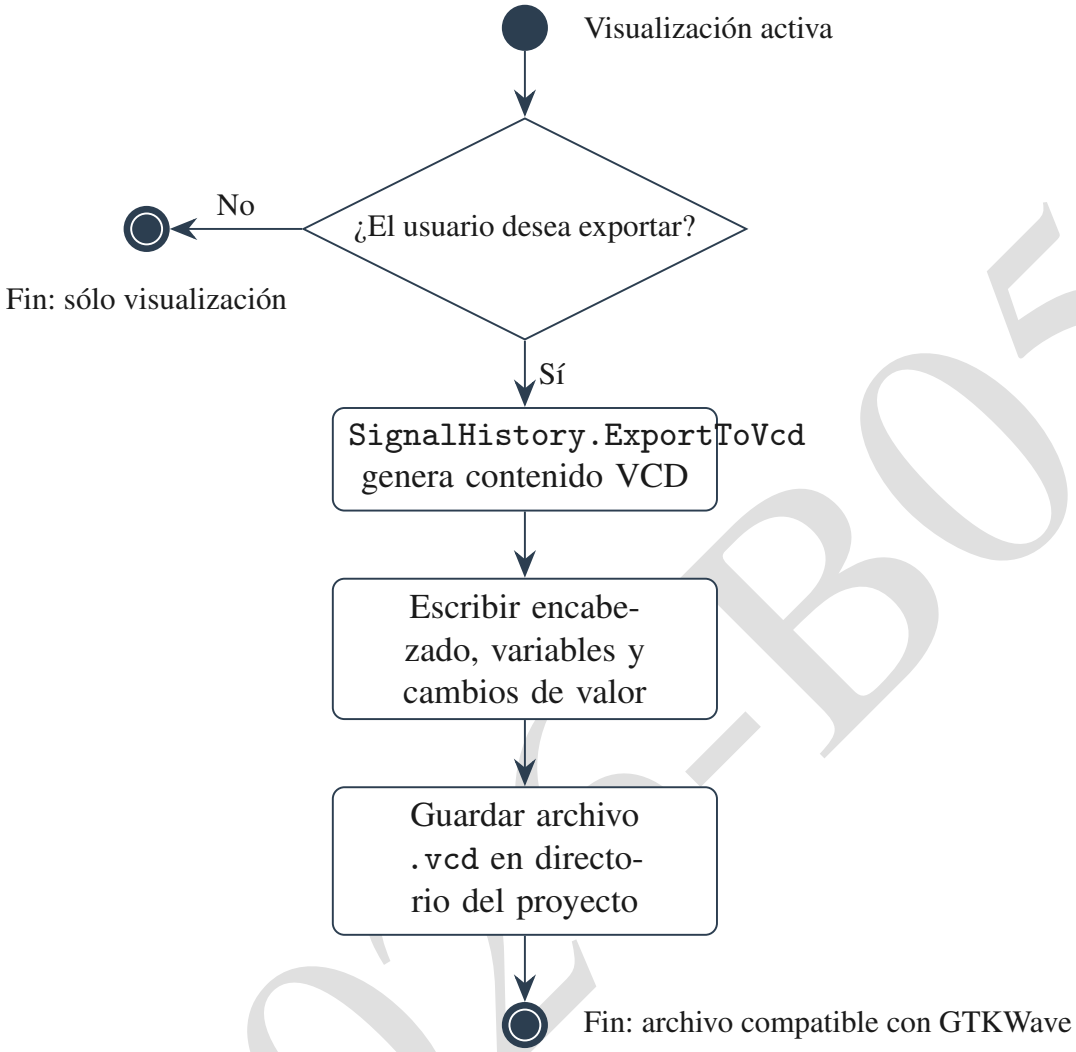


Figura 5.60: *Actividad de exportacion: generacion de archivo VCD a partir del historial de senales.*

5.5.6. Actividades del Parser

Las cuatro actividades siguientes documentan etapas internas del Parser que no están contempladas en los flujos de validación generales: el pipeline del Tokenizer, la construcción de bloques de configuración, la expansión de bloques `generate` y la resolución de tipos `record` con sus accesos por campo.

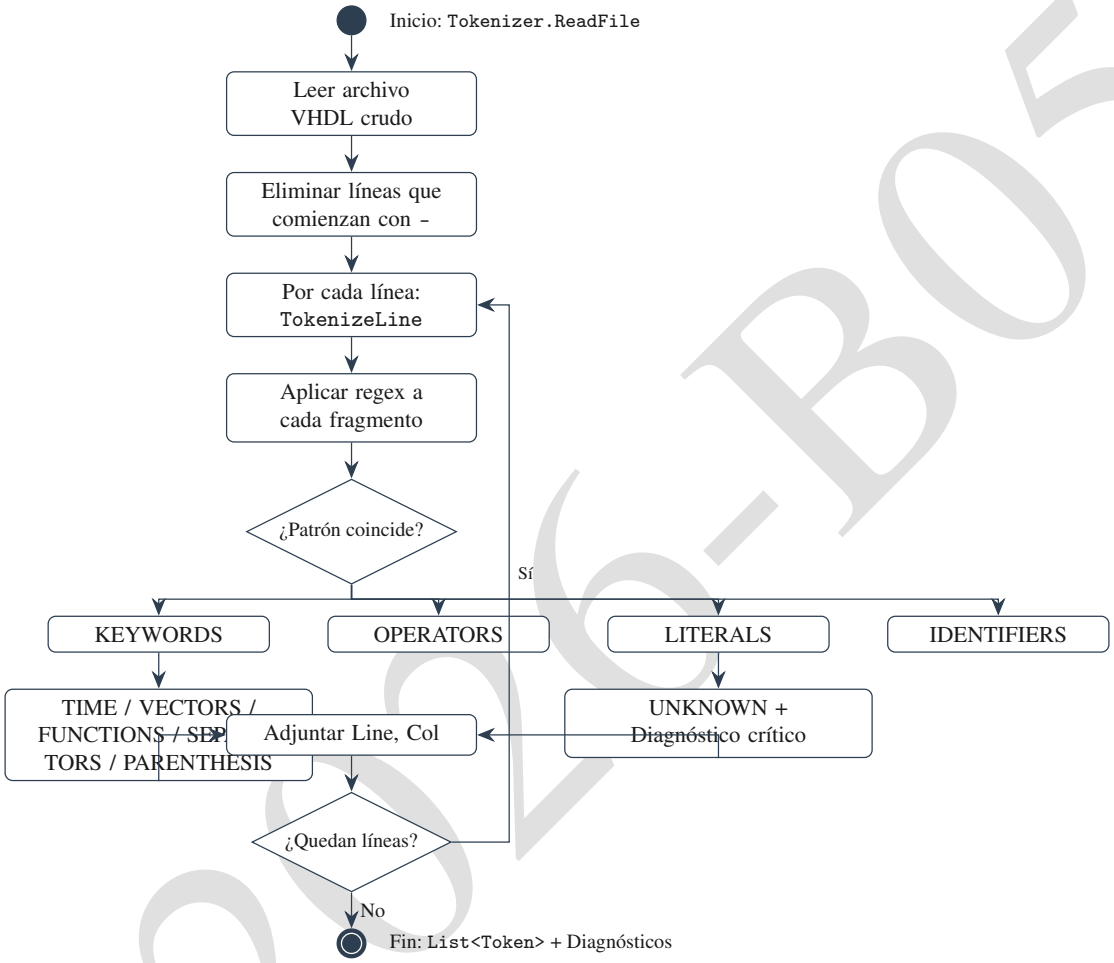


Figura 5.61: *Actividad del Tokenizer: pipeline de lectura, eliminación de comentarios, clasificación de tokens por expresión regular y construcción del TokenStream que alimenta a ParseV2.*

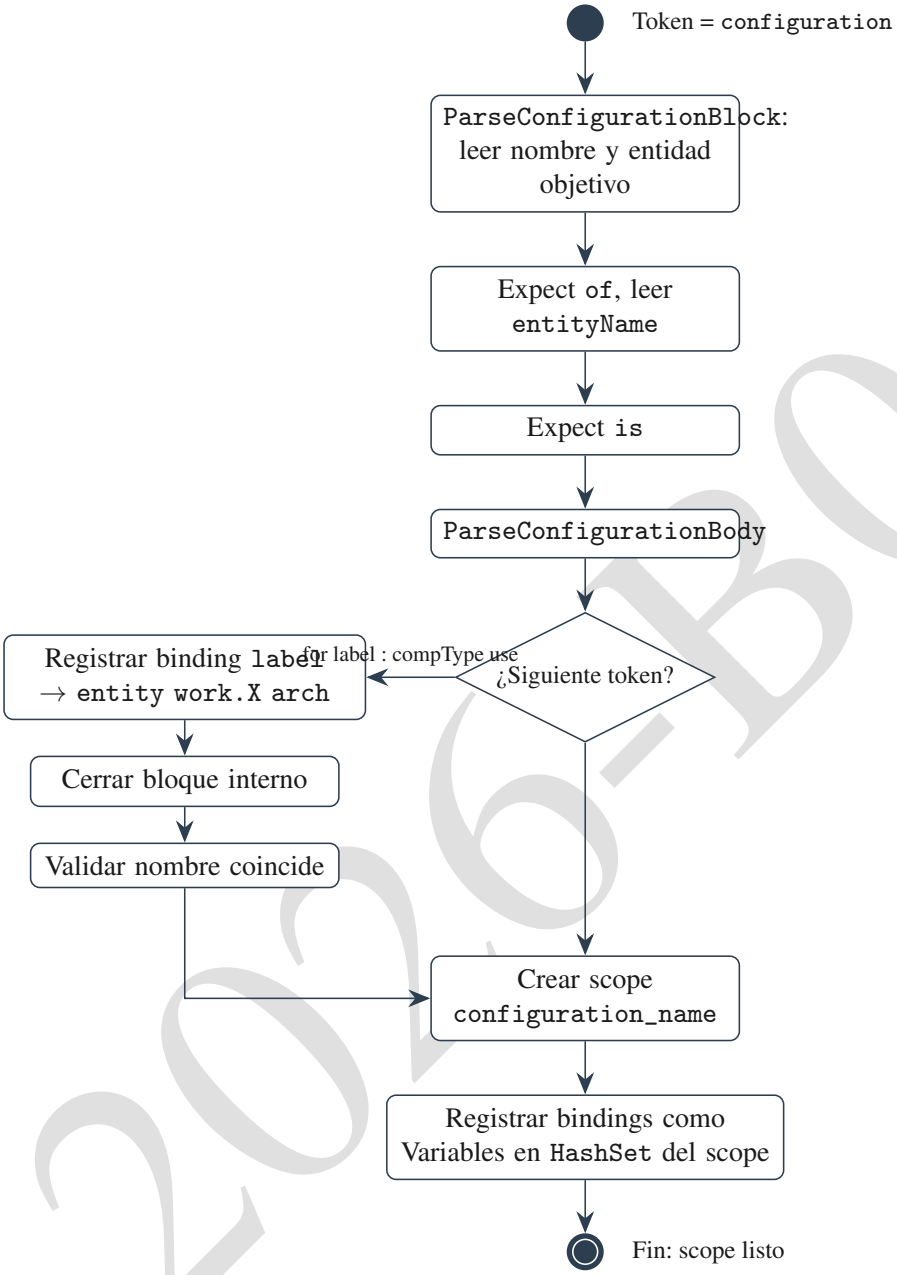


Figura 5.62: *Actividad: análisis de bloques `configuration`. ParseConfigurationBlock asocia las entidades con las arquitecturas correspondientes según la declaración `for use entity`, registrando las correspondencias en el scope actual.*

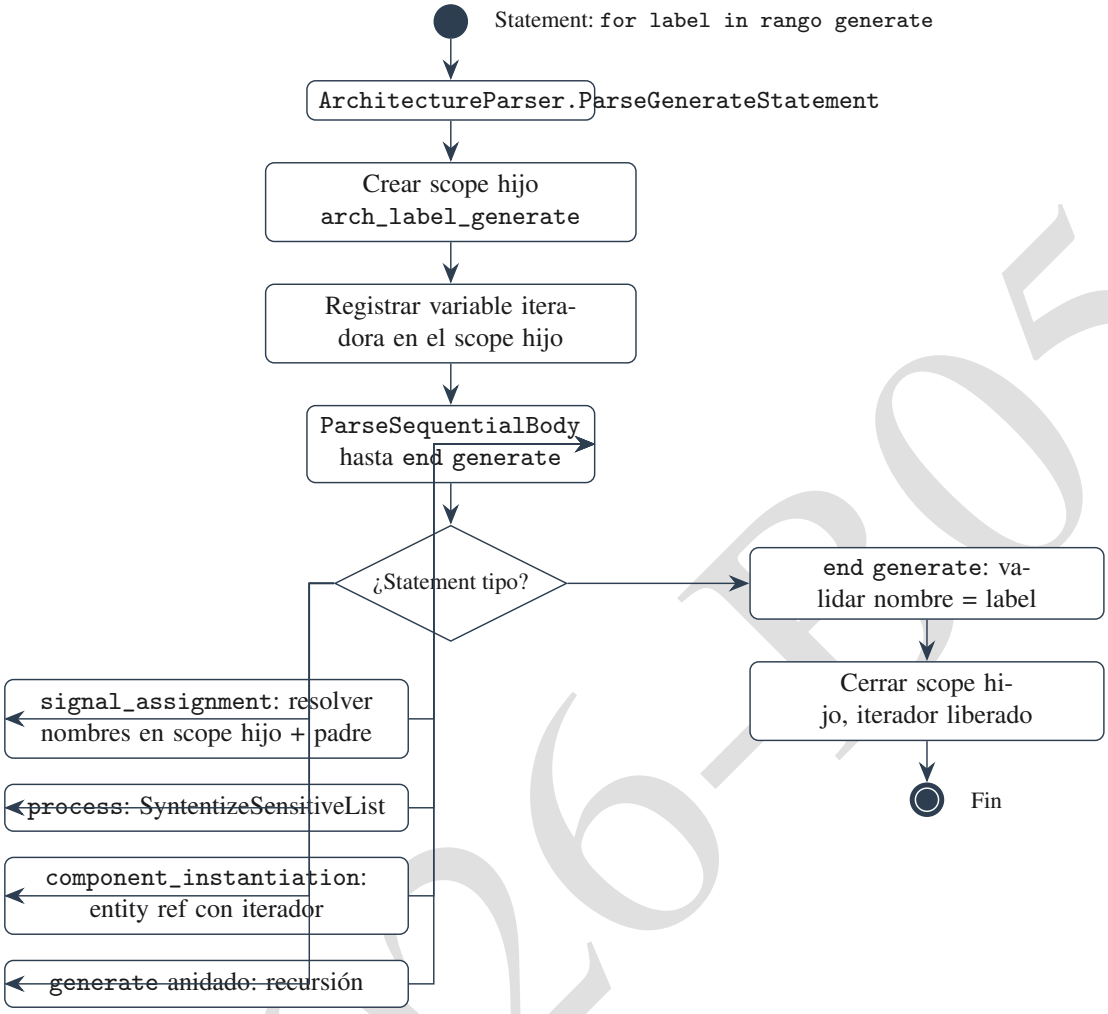


Figura 5.63: *Actividad: expansión de bloques `generate`. Cada iteración del `for-generate` o cada rama del `if-generate` crea un sub-scope independiente con su propio diccionario de Variables, permitiendo la instanciación replicada de hardware con índices distintos.*

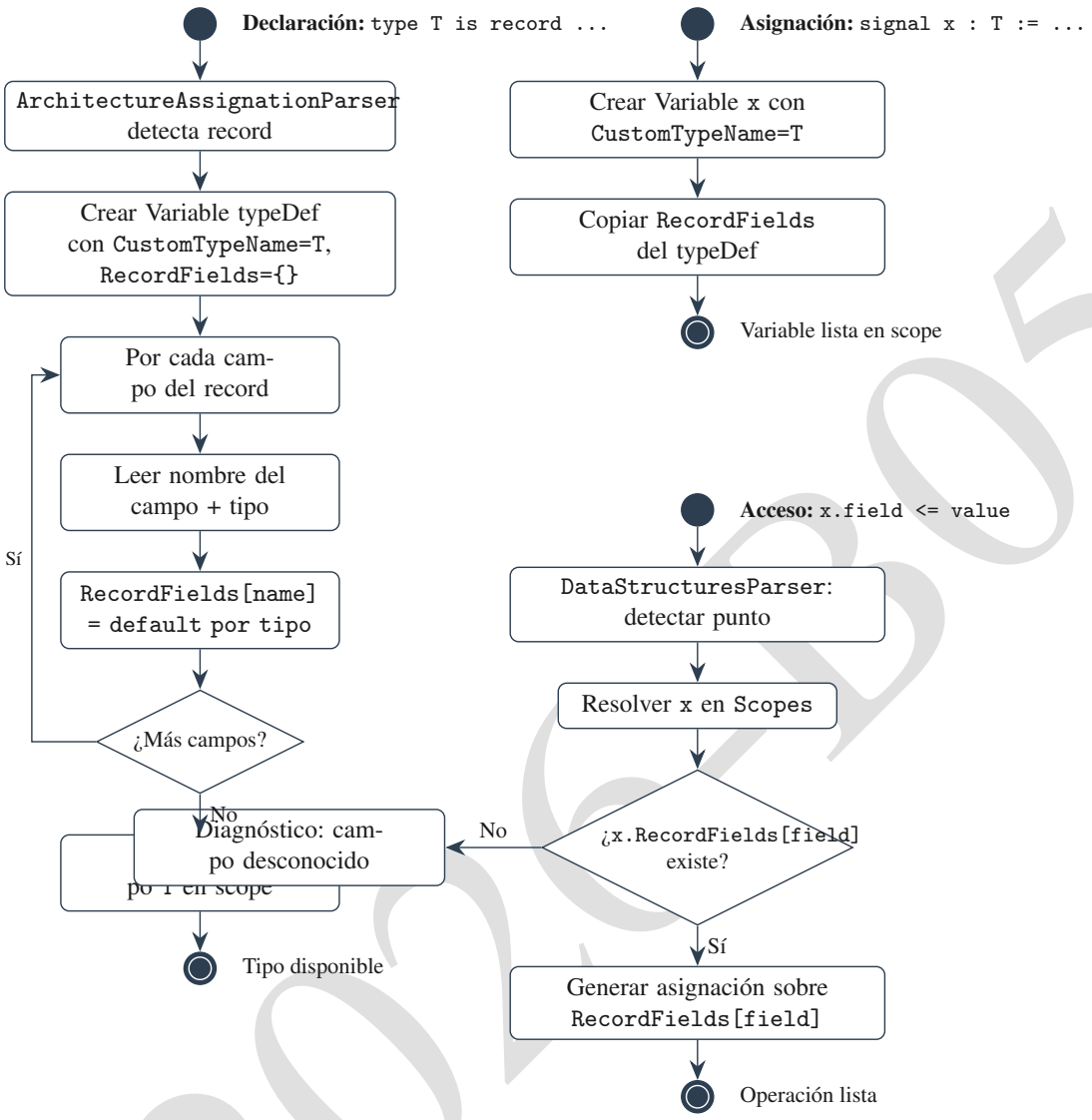


Figura 5.64: *Actividad: resolución de tipos `record`. La declaración `type T is record ...` registra una Variable typeDef con los campos del registro; cada acceso `x.field` se resuelve sobre el diccionario `RecordFields` de la instancia.*

5.5.7. Resiliencia del Debugger Lint

El Debugger implementa una etapa de *lint* previa a la simulación que detecta inconsistencias estructurales (puertos no conectados, señales no declaradas, asignaciones inválidas) sin abortar el análisis completo del archivo. La Figura 5.65 documenta la estrategia de recuperación tras errores no fatales.

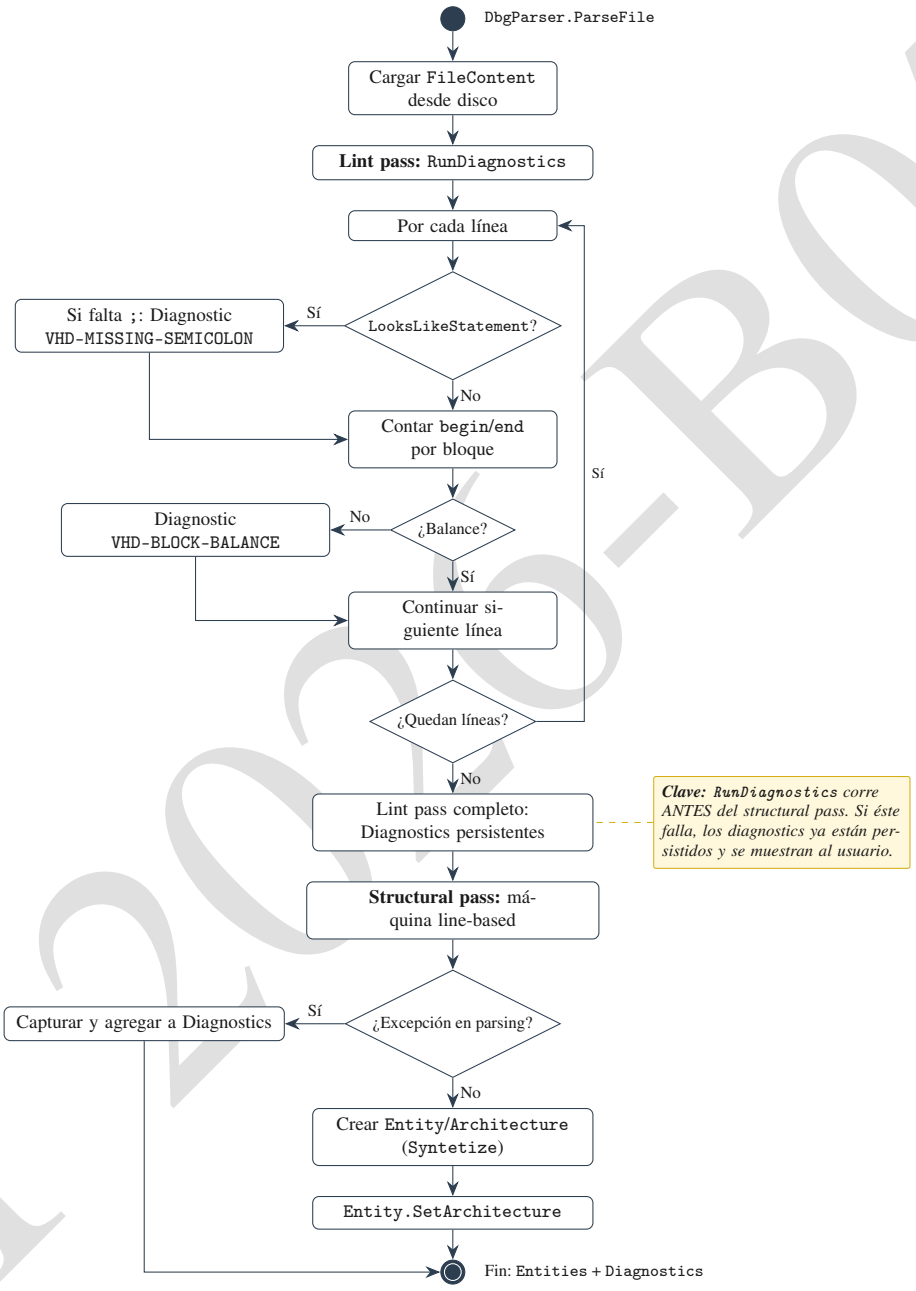


Figura 5.65: *Actividad: resiliencia del Debugger lint. Tras detectar un error no fatal, el Debugger registra el diagnóstico en la lista de errores y continúa con el siguiente nodo del AST, evitando la pérdida de información para errores subsecuentes.*

5.5.8. Despacho por modalidad en Practice

El subsistema *Practice* discrimina la lógica de evaluación según la modalidad del ejercicio: `FromZero` usa el código del estudiante íntegro, `FixBug` también lo usa íntegro (porque el *starter* ya traía el *bug*), y `FillGap` fusiona el *starter* con la edición preservando lo que el estudiante haya escrito *fuera* de los marcadores `BEGIN_EDIT/END_EDIT`. La Figura 5.66 documenta el flujo unificado de evaluación tras el despacho.

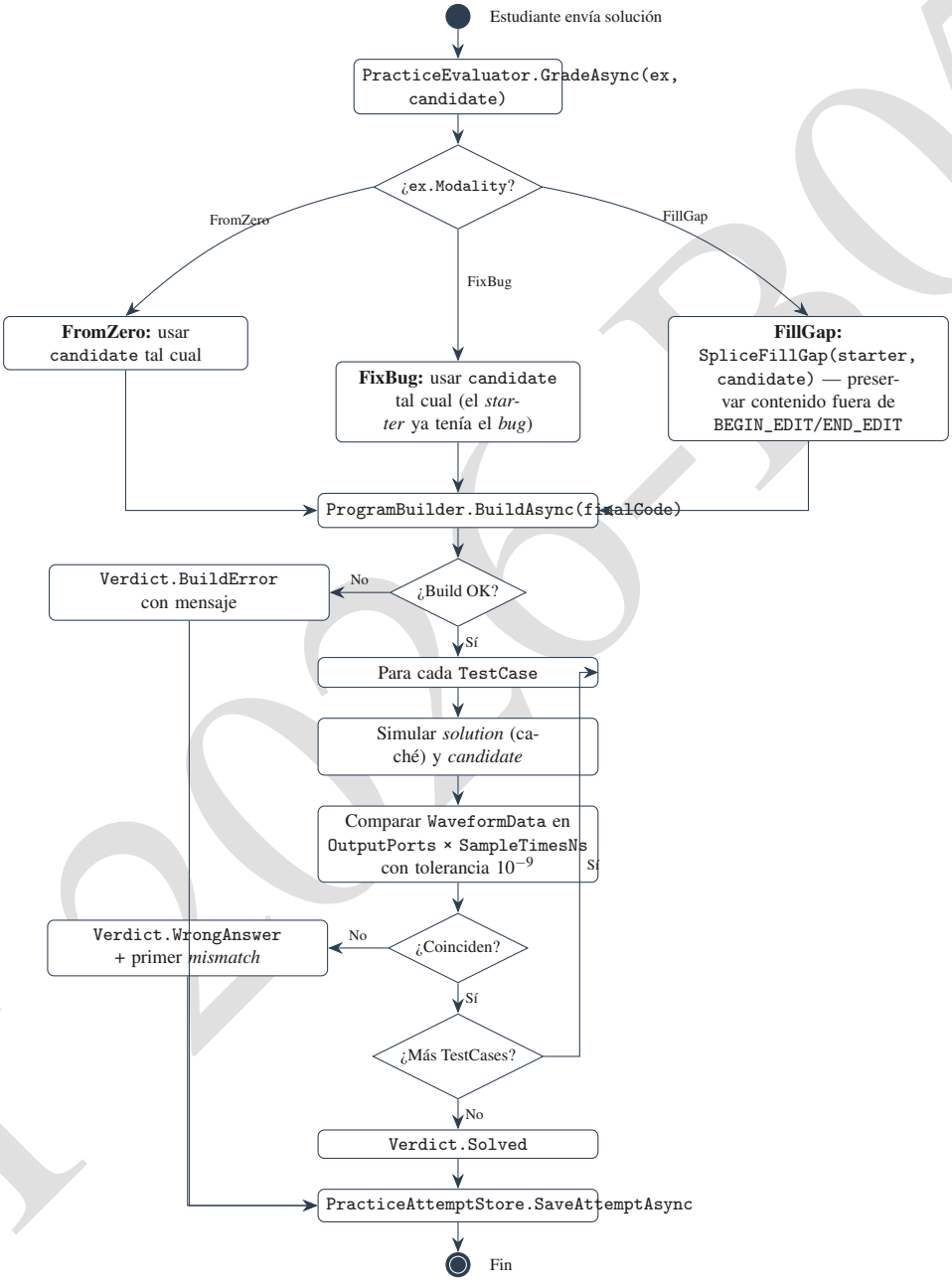


Figura 5.66: *Actividad: despacho por modalidad en `PracticeEvaluator.GradeAsync`. Tras determinar la modalidad, el flujo continúa de forma unificada: compilación, ejecución por* test case*, comparación tolerante de formas de onda y persistencia del* verdict *(`Solved`/`WrongAnswer`/`BuildError`/`RuntimeError`).*

5.5.9. Estados del Motor

Los siguientes dos diagramas de estados describen la dinámica temporal interna del motor de simulación: el ciclo delta del `DeltaCycleEngine` y la planificación temporal del `Timer` opcional para ejecución paso a paso.

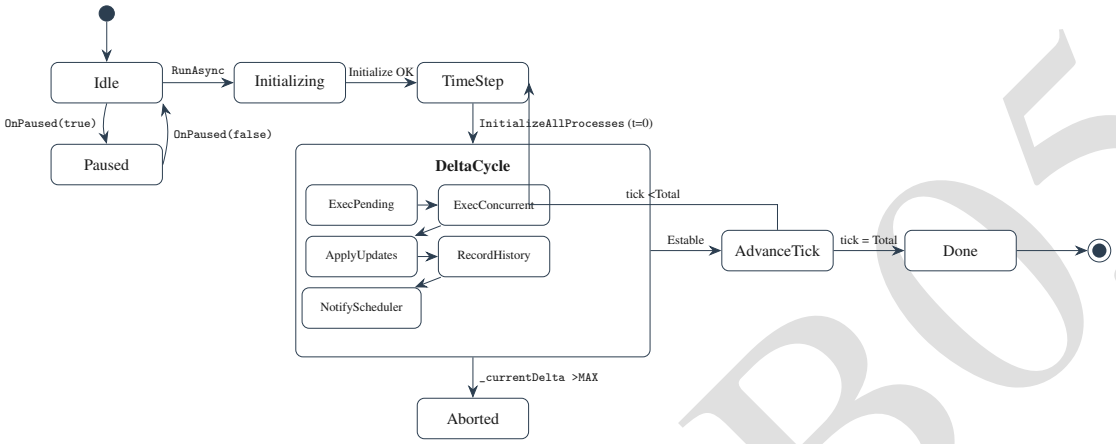
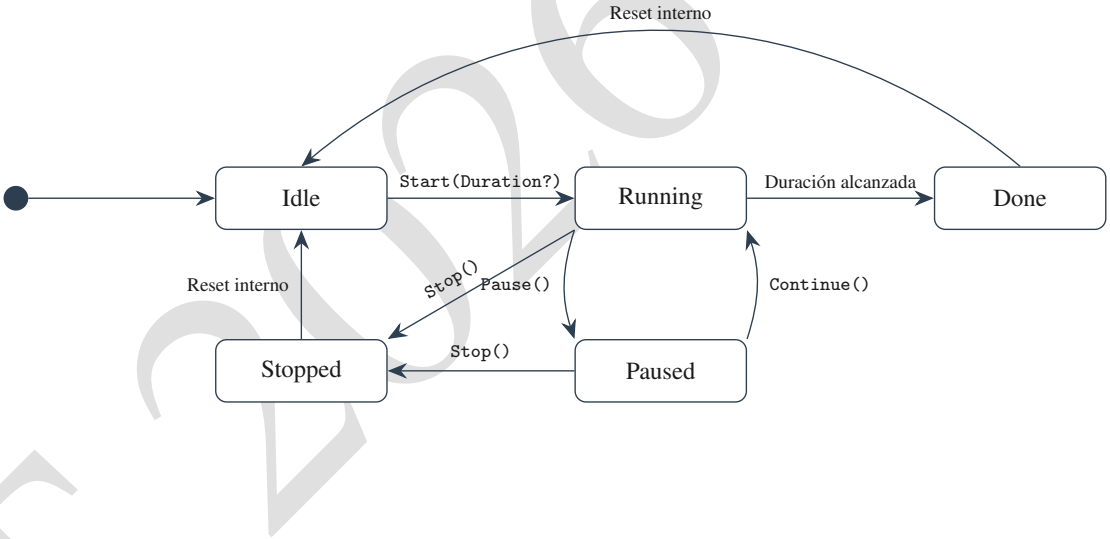


Figura 5.67: *Diagrama de estados: ciclo delta del motor. El `DeltaCycleEngine` itera entre los estados Idle, Evaluando, Propagando y Estable; aborta y reporta error si el contador de ciclos delta supera `MAX_DELTA_CYCLES`, indicador de un bucle combinacional.*



> ***Path dual:*** *(1)* Fast*: `SimulationRunner` sin `TimeMachine`. (2)* Interactivo*: `TimeMachine` + `DeltaCycleEngine`; `ConnectToTimeMachine()` suscribe a `OnPaused` para sincronizar pausa/resume.*

Figura 5.68: *Diagrama de estados: `Timer` de ejecución paso a paso. Permite pausar la simulación tras cada delta cycle (modo TimeMachine) para inspección interactiva, transicionando entre Idle, Running, Paused y Stopped.*

5.6. Subsistemas Internos

Esta sección documenta los subsistemas técnicos que extienden la funcionalidad central de Kmila: el `LibraryCompiler`, encargado de compilar y cachear las bibliotecas estándar de VHDL e IEEE, y los subsistemas de la aplicación (*Learn*, *Practice*, *Concepts*, *EditorDock*, *Runs Snapshot* y *Replay*) que materializan los componentes pedagógicos y de persistencia de corridas. La justificación pedagógica de los tres subsistemas didácticos (*Learn*, *Practice* y *Concepts*) se desarrolla por separado en la Sección 5.8. Se documentan además tres subsistemas interactivos incorporados durante TT2 que amplían el banco de trabajo del editor: el *Visual Runtime* (interacción en tiempo de ejecución con componentes virtuales), la *depuración en vivo* (*breakpoints*, paso a paso y recorrido sobre el esquemático) y la *edición por bloques* (traducción bidireccional entre VHDL y bloques visuales).

5.6.1. LibraryCompiler

`LibraryCompiler` resuelve y compila las cláusulas `use` de los archivos VHDL, manteniendo una caché basada en hash del contenido fuente para evitar recompilaciones innecesarias. Su flujo se descompone en tres vistas: la resolución por grafo dirigido acíclico (DAG) de dependencias, el *roundtrip* de la caché y la cascada de fuentes consultadas para compilar cada biblioteca.

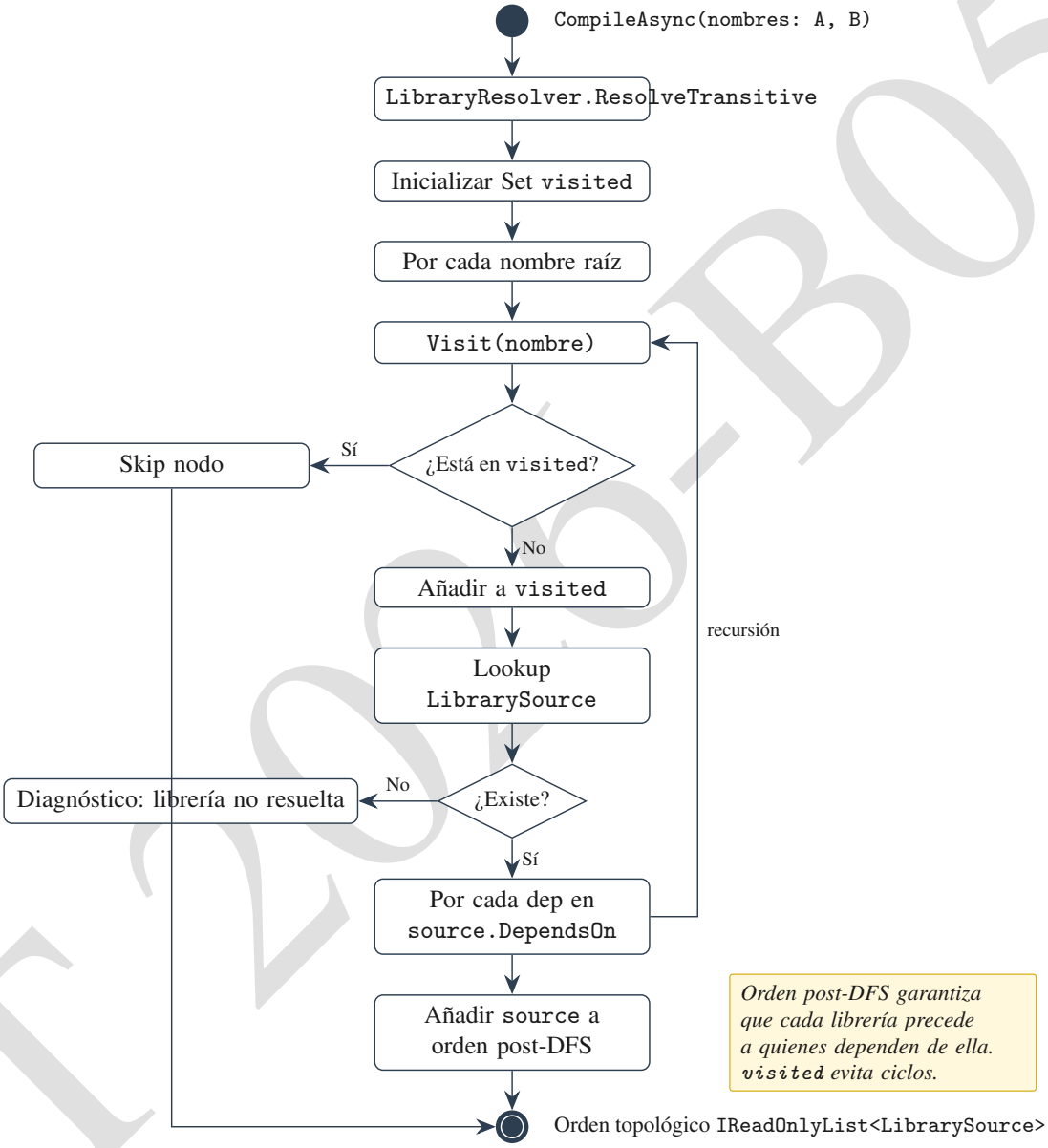


Figura 5.69: *LibraryCompiler: resolución de dependencias mediante DAG. Cada nodo es una biblioteca; las aristas representan dependencias `use` entre ellas. El resolver garantiza un orden topológico que evita ciclos.*

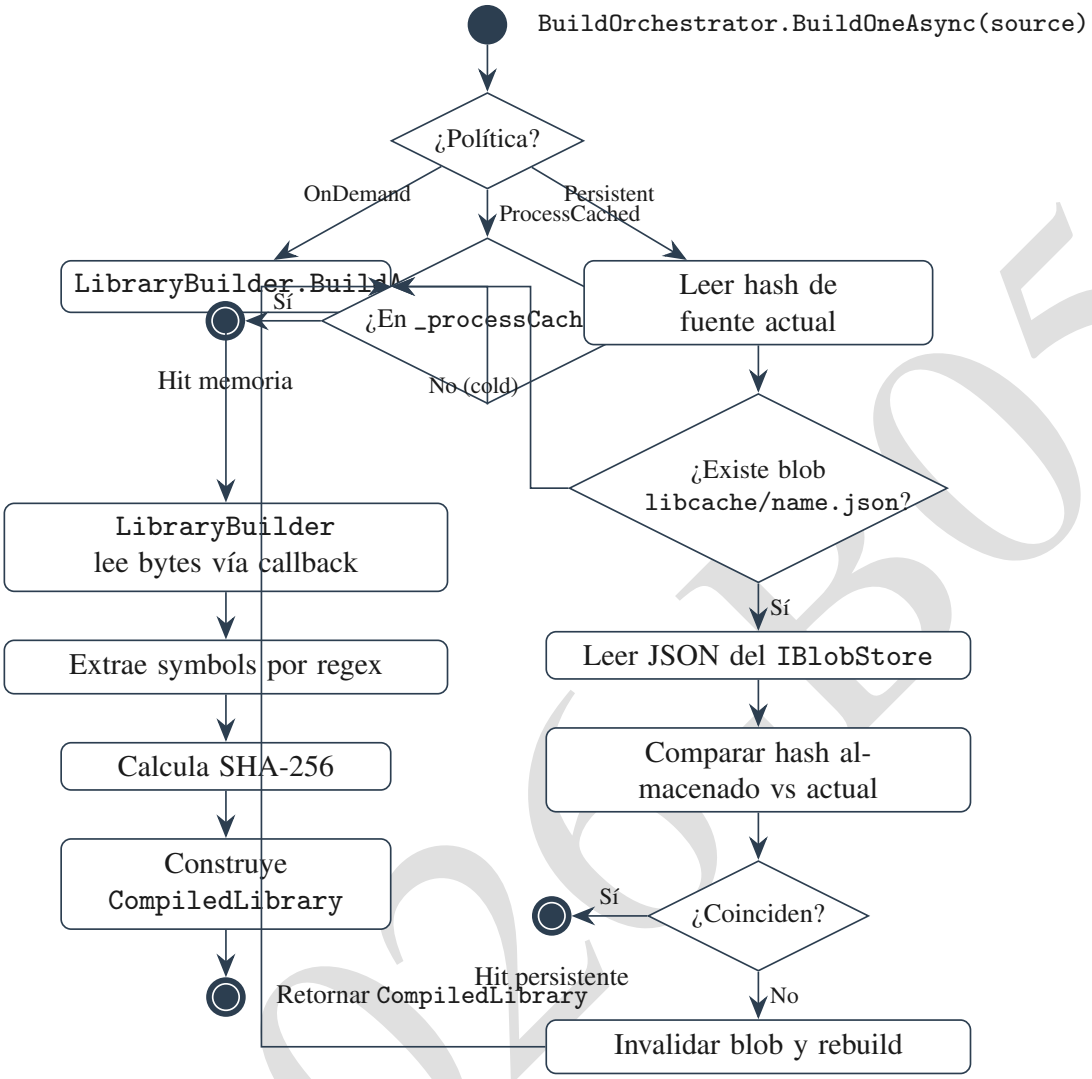


Figura 5.70: *LibraryCompiler:* roundtrip *de la caché de bibliotecas. El* `LibraryCache` *consulta primero la huella SHA-256 del fuente; ante* cache hit*, devuelve el* `CompiledLibrary` *preexistente; ante* cache miss*, dispara el pipeline de compilación y persiste el resultado.*

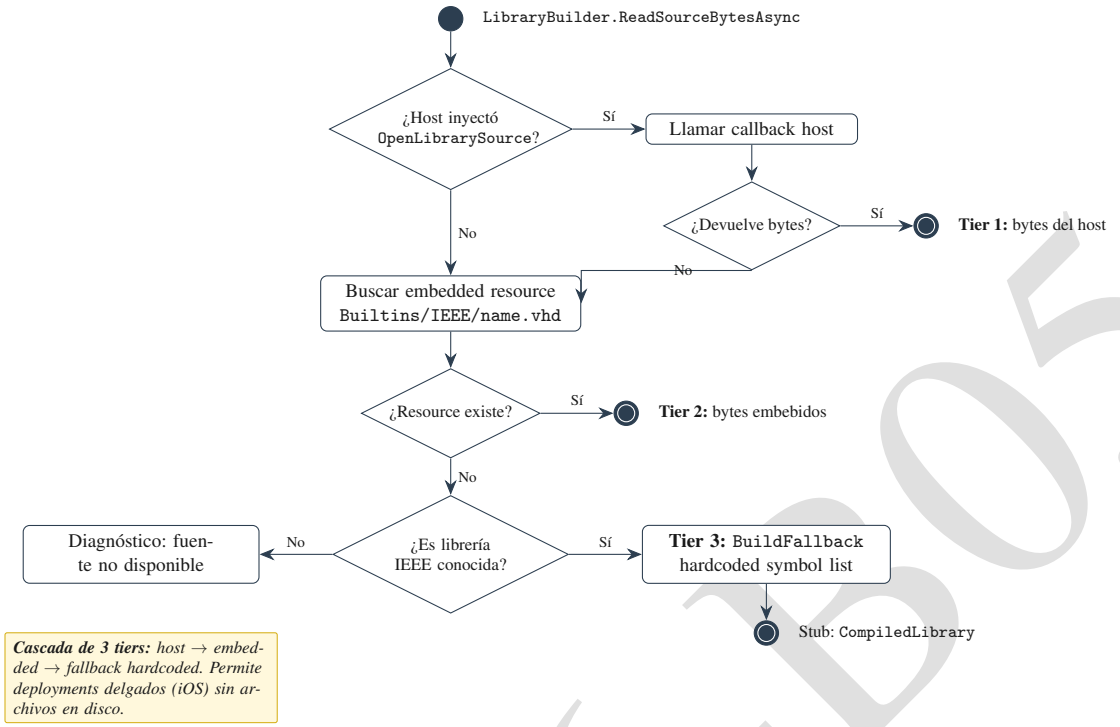


Figura 5.71: *LibraryCompiler: cascada de fuentes para resolver una biblioteca. El compilador busca primero en las bibliotecas IEEE empaquetadas, después en las bibliotecas de usuario distribuidas con el proyecto y finalmente en las bibliotecas locales del* workspace.

La Figura 5.72 muestra la vista estructural del módulo, complementaria a los tres flujos anteriores.

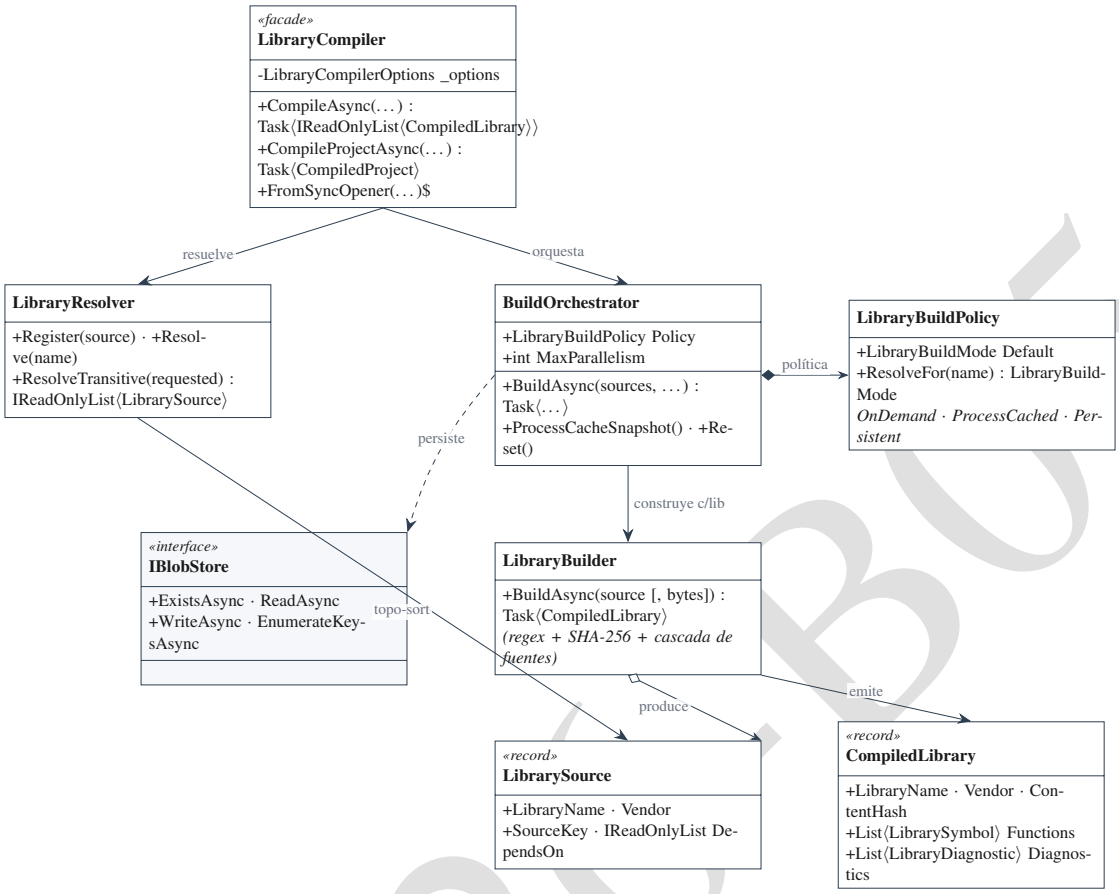


Figura 5.72: *Diagrama de clases del `LibraryCompiler`. La fachada expone `CompileAsync` y `CompileProjectAsync`; `LibraryResolver` ordena topológicamente las dependencias (`ResolveTransitive`); `BuildOrchestrator` aplica la `LibraryBuildPolicy` (*OnDemand / ProcessCached / Persistent*) y persiste el resultado vía la interfaz `IBlobStore`; `LibraryBuilder` produce cada `CompiledLibrary` a partir de una `LibrarySource` mediante extracción de símbolos por expresiones regulares y huella SHA-256.*

5.6.2. Subsistemas de la Aplicación

Seis subsistemas implementan funcionalidades transversales sobre Kmila.Shared: *Learn* (módulo pedagógico de lecciones), *Practice grading* (evaluación automática de ejercicios en tres modalidades), *Concepts* (módulo de modelos mentales de ejecución), *EditorDock* (composición del *workbench* multi-pestaña de 8 tabs), *Runs snapshot* (persistencia de corridas) y *Replay* (reproducción paso a paso de corridas terminadas).

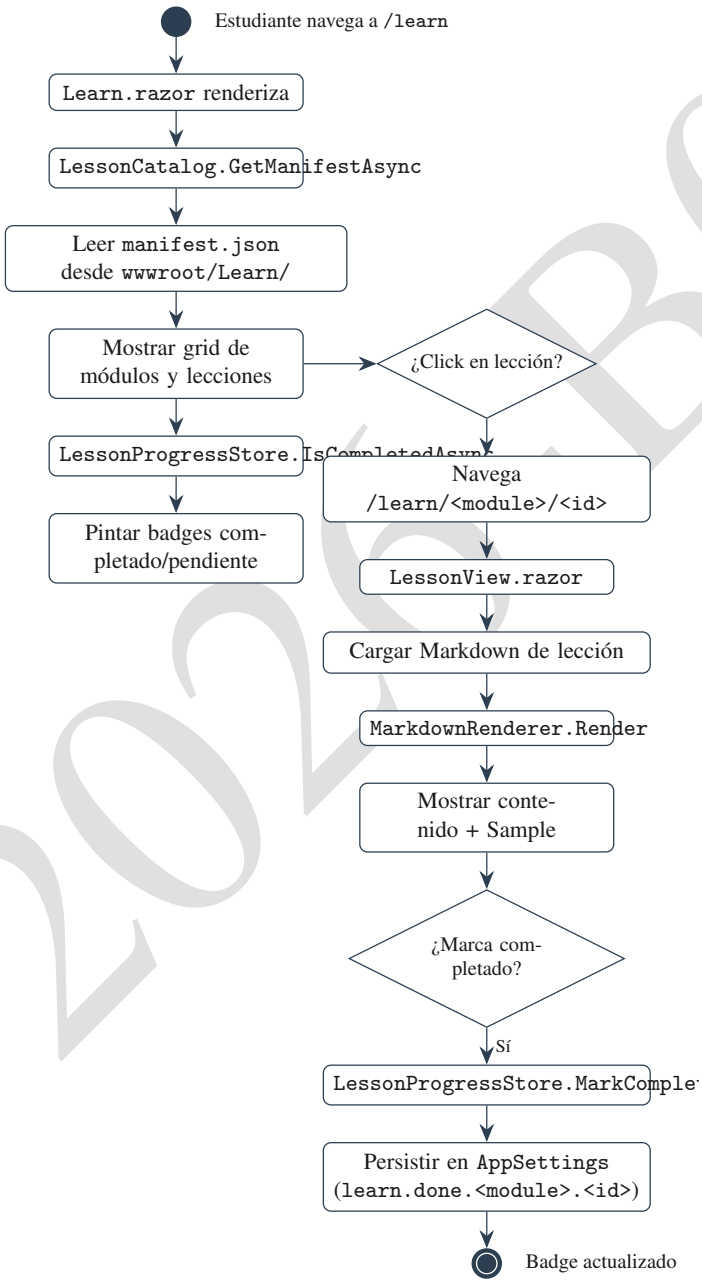


Figura 5.73: *Subsistema* Learn*: catálogo de lecciones organizadas por unidades didácticas.* `LessonCatalog` *carga el manifiesto desde* `learn-manifest.json`*;* `LessonProgress` *registra el avance del estudiante en* `AppSettings` *con claves* `lesson_*`*.*

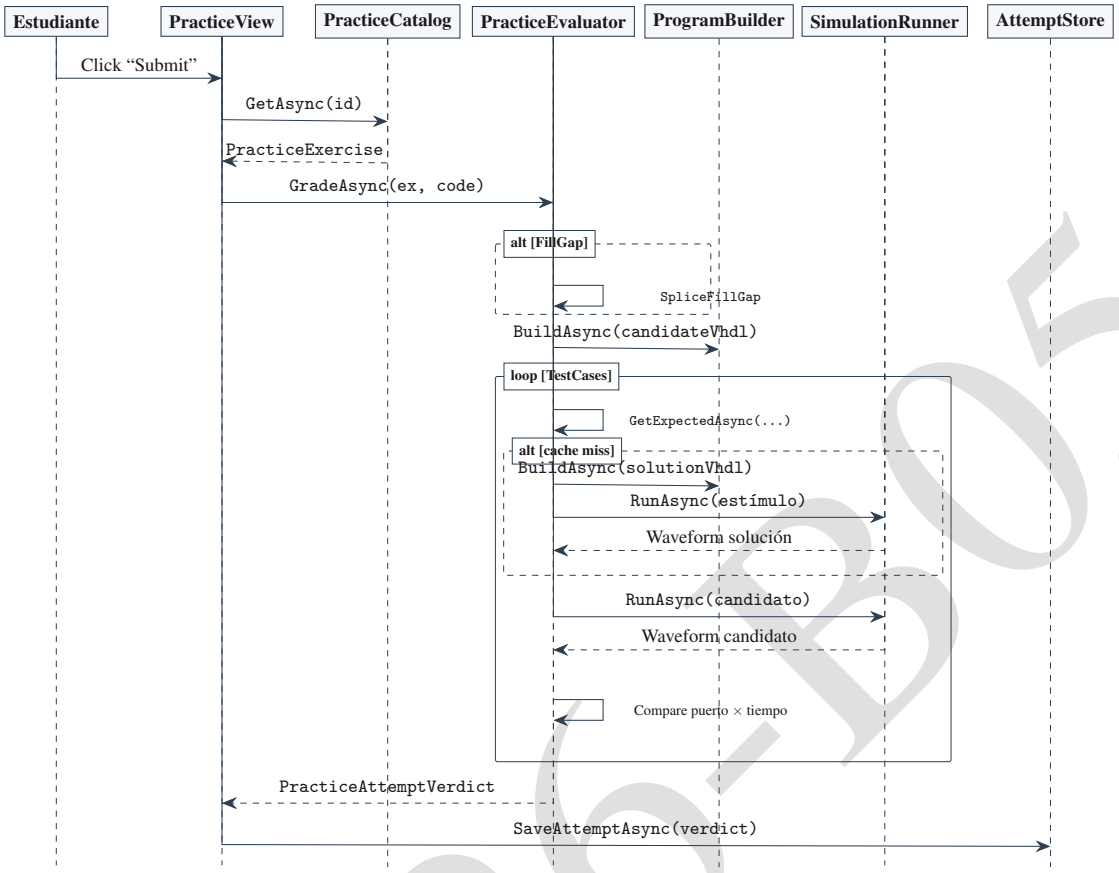
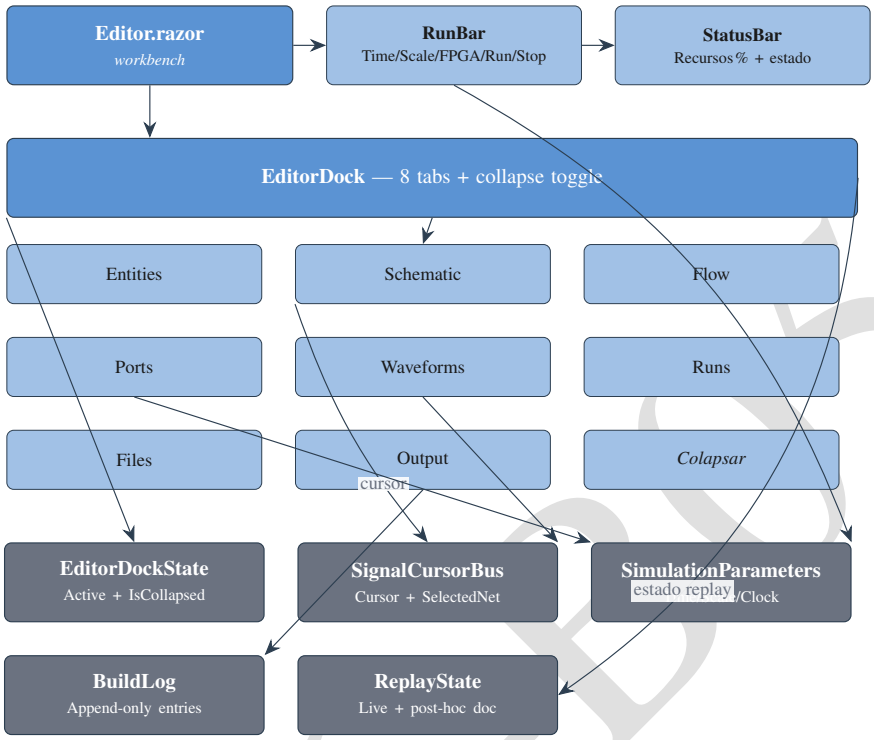


Figura 5.74: *Subsistema* Practice*: evaluación automática de ejercicios.* `PracticeEvaluator` *compara la simulación del código candidato contra la del código solución para cada* `TestCase`*, aplicando tolerancia numérica configurable; el veredicto se persiste en* `PracticeAttemptStore`*.*



Sincronización clave: *`SignalCursorBus` permite que el cursor temporal del visor de ondas (*Waveforms*) ilumine la conexión activa en el esquemático (*Schematic *en modo* Debug*). `ReplayState` captura la última corrida exitosa — `LiveDoc` para la corrida en curso, `Current` como instantánea post-hoc — y notifica al panel* Replay *para reproducirla paso a paso sobre el mismo componente `ExecutionPlayer` que usa el subsistema* Concepts.

Figura 5.75: *Subsistema* EditorDock*: composición del* workbench *multi-pestaña. `EditorDockState` sincroniza ocho pestañas (Entities, Schematic, Flow, Ports, Waveforms, Runs, Files, Output) y comparte el estado del cursor temporal vía `SignalCursorBus` entre Schematic (modo* Debug*) y Waveforms.*

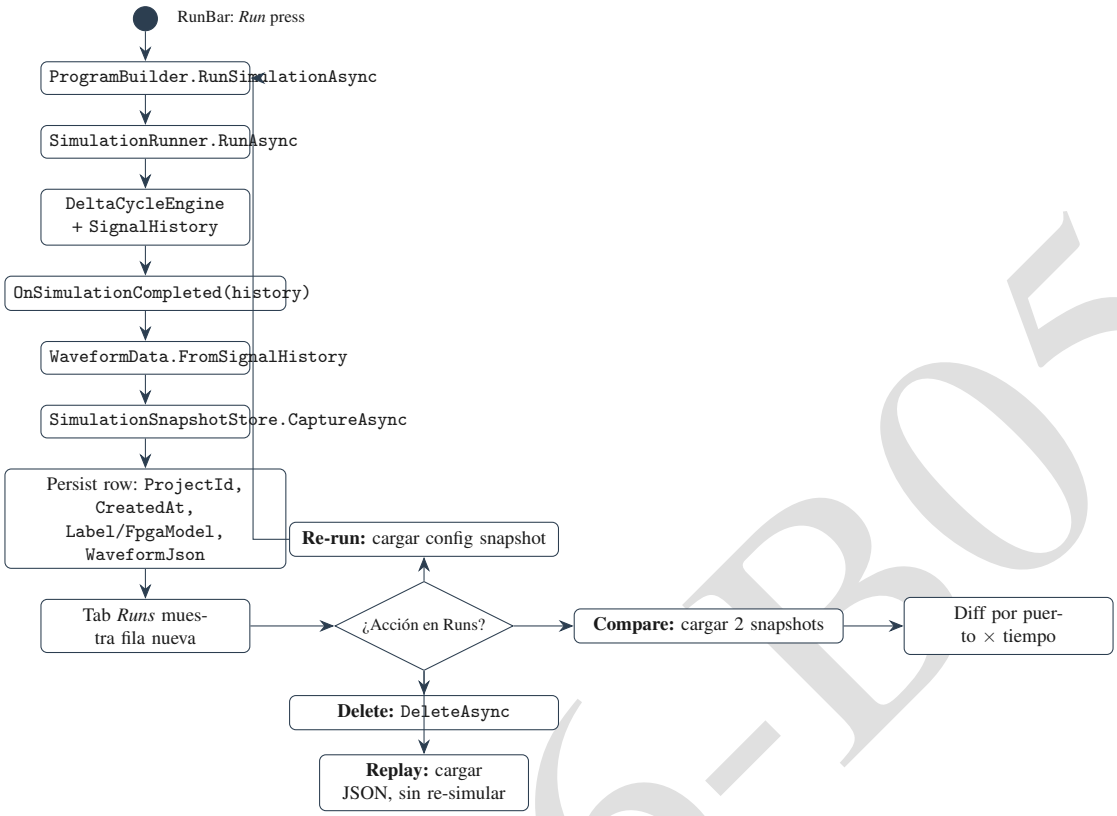


Figura 5.76: *Subsistema* Runs Snapshot*: persistencia de corridas completas de simulación.* `SimulationSnapshotStore` *serializa la configuración, los estímulos y el* `WaveformData` *resultante para permitir* replay *sin recompilación, comparación entre corridas y exportación.*

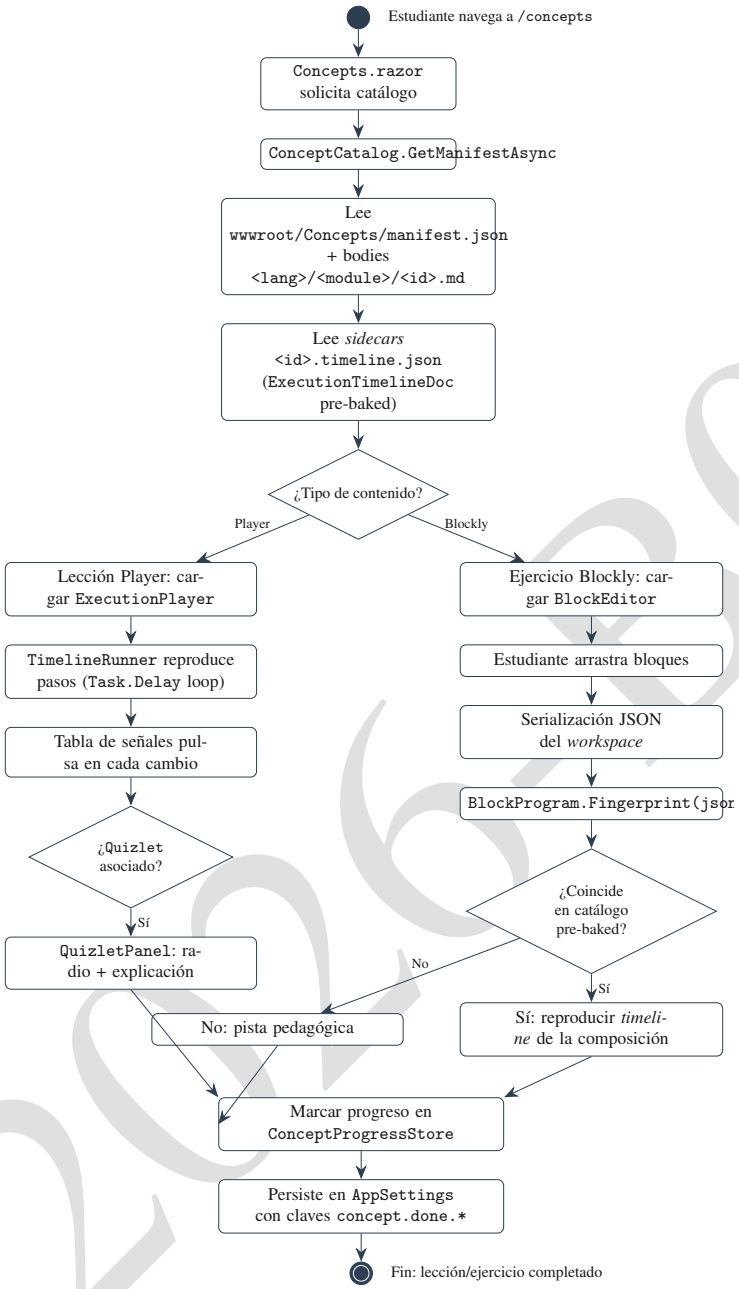


Figura 5.77: *Subsistema* Concepts*: módulo pedagógico para enseñar los tres modelos de ejecución (secuencial, concurrente, paralelo real en hardware).* `ConceptCatalog` *carga lecciones desde* `wwwroot/Concepts/manifest.json`*; cada lección puede llevar un* `ExecutionTimelineDoc` *pre-grabado que se reproduce con* `TimelineRunner` *sobre el componente* `ExecutionPlayer`*; los ejercicios usan el editor* Blockly *cuyo programa se reduce a un* fingerprint *canónico por el servicio* `BlockProgram` *y se compara contra un catálogo de composiciones pre-baked.*

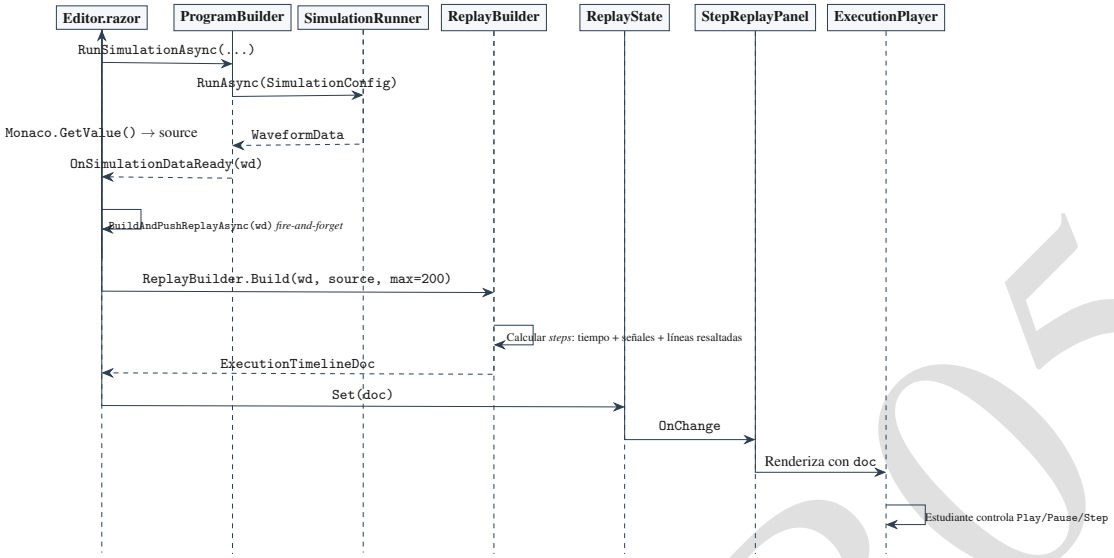


Figura 5.78: *Subsistema* Replay*: tras cada corrida exitosa,* `ReplayBuilder` *convierte la* `WaveformData` *resultante en un* `ExecutionTimelineDoc` *con* highlights *de línea por transición de señal;* `ReplayState` *mantiene el documento vigente y notifica al panel* `StepReplayPanel`*, que lo presenta sobre el mismo* `ExecutionPlayer` *usado por el subsistema* Concepts*.*

5.6.3. Subsistema Visual Runtime: interacción en tiempo de ejecución

El *Visual Runtime* (ruta `/visual`) es una superficie de simulación de *corrida infinita* donde los puertos del diseño se enlazan a componentes virtuales sobre un lienzo —LEDs, arreglos de LEDs, interruptores, bancos DIP, botones, *displays* de siete segmentos y motores paso a paso— con los que el estudiante *interactúa*: pulsa un botón y observa encenderse un LED, en lugar de leer una forma de onda *a posteriori*. Es la tercera capa de visualización pedagógica, junto al animador de compuertas (nivel lógico) y al animador de máquinas de estados (nivel de estado).

La restricción de diseño rectora es **reutilizar el motor de simulación existente**: el modo no introduce un segundo `DeltaCycleEngine` ni toca los servicios de simulación por lotes. El *seam* vive en `VisualRunCoordinator.LoopAsync`: tras un `InitializeSignals` que siembra valores por defecto, cada tick del bucle (i) verifica pausa y *breakpoints*, (ii) alterna los relojes automáticos, (iii) vuelca los valores impuestos por los widgets con `DrainInputsToPorts`, (iv) invoca `AdvanceTime` y `RunTimeStepAsync` sobre el motor reutilizado, y (v) captura el estado de los puertos con `SnapshotFrame`. La escritura de entradas no requiere un *hook* nuevo en el motor: al asignar `port.Value` se dispara `Variable.OnValueChange`, que marca el indicador de actividad (`_activeFlags`) que el planificador ya consulta al inicio de cada paso; así, la entrada en vivo viaja por exactamente la misma vía que el estímulo pre-construido de una corrida acotada. La Figura 5.79 detalla el bucle completo.

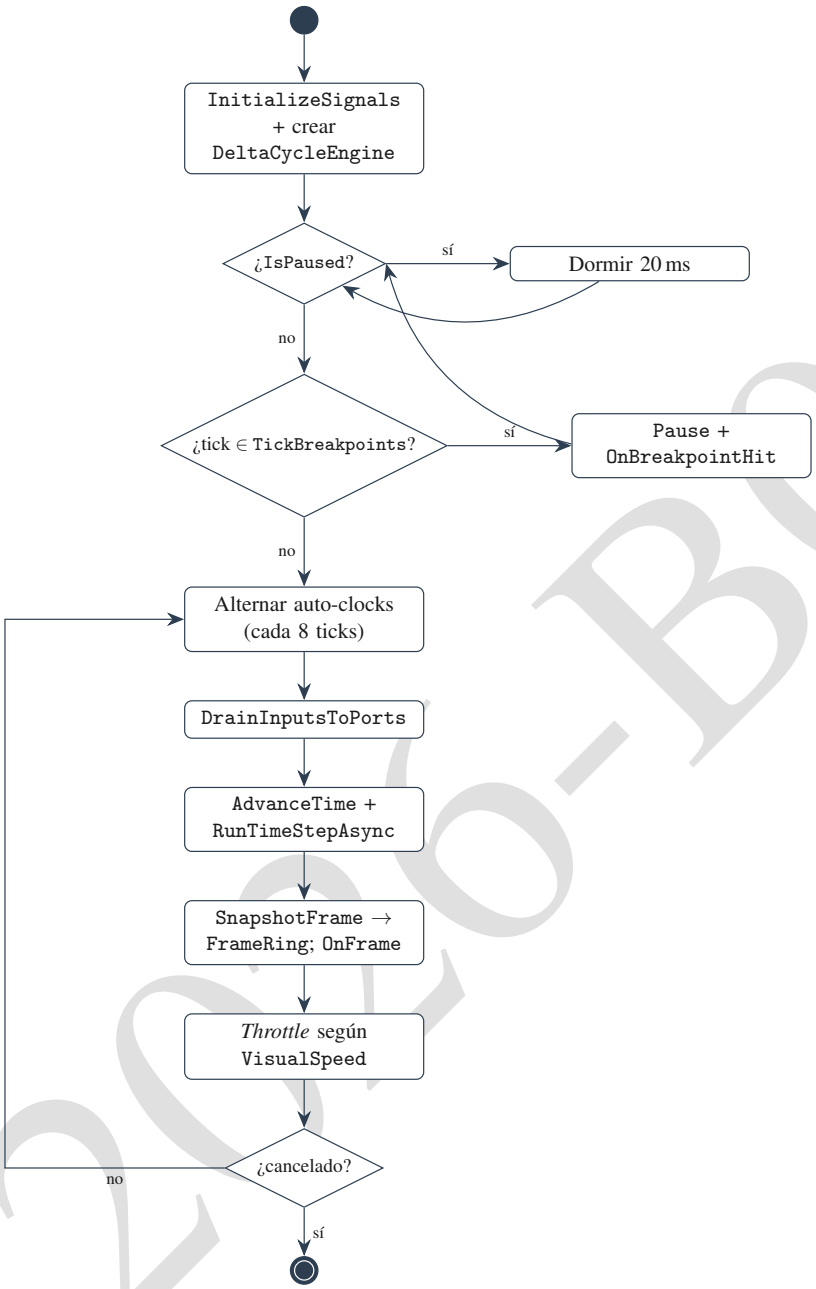


Figura 5.79: *Diagrama de actividad del bucle de corrida del* Visual Runtime*. Cada tick verifica pausa y* breakpoints *de tick, alterna los relojes automáticos (cada `CLOCK_HALF_PERIOD_TICKS = 8` ticks), vuelca las entradas de los widgets, ejecuta un paso del motor reutilizado, captura el fotograma en el `FrameRing` y aplica el* throttle *de velocidad antes de reiniciar.*

Como una corrida infinita no puede retener toda la historia, los fotogramas se guardan en un `FrameRing`: un búfer circular acotado (2000 ticks por omisión) que descarta los más antiguos. El `InputStateMap` concentra los valores impuestos por los widgets con semántica *último en escribir gana*, y una cola de rebote opcional (*button bounce*) superpone una breve alternancia de flancos por botón. Los relojes se detectan automáticamente: cualquier puerto de entrada cuyo nombre se lea como reloj (`clk`, `clock`, ...) se conmuta cada ocho ticks, salvo que el usuario haya cableado un interruptor o botón a ese puerto. La Figura 5.80 resume el flujo de datos y la Figura 5.24 el modelo de clases.

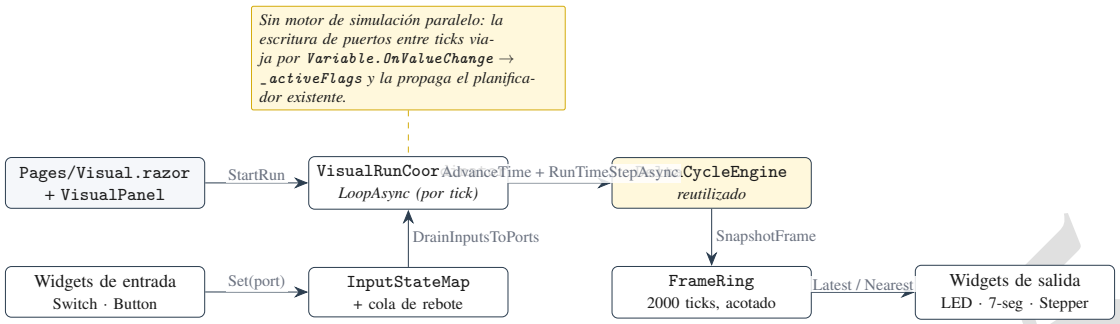


Figura 5.80: *Subsistema* Visual Runtime*: flujo de datos del* seam*.* `VisualPanel` *arranca el* `VisualRunCoordinator`*, que dirige el* `DeltaCycleEngine` *reutilizado tick a tick; los fotogramas se acumulan en el* `FrameRing` *del cual leen los widgets de salida, mientras los widgets de entrada escriben en el* `InputStateMap` *que el coordinador drena antes de cada paso.*

El estado del lienzo se persiste en un archivo *sidecar* `.kviz` (esquema v1, JSON) por proyecto: registra el tamaño del lienzo, la lista de componentes (identificador, tipo, posición, configuración y *bindings* pin→puerto) y los ajustes de la corrida. El motor de simulación nunca lee este archivo; es puro estado de interfaz. Los *bindings* admiten un sufijo de rebanada (`count[3]` para un bit, `count[7:0]` para un rango) resuelto en modo *solo lectura*: los widgets de salida leen la rebanada indicada, pero los de entrada siguen impulsando el puerto completo (la fan-in por bit se difirió). La Tabla 5.3 enumera el catálogo de componentes.

Tabla 5.3: *Catálogo de componentes virtuales del* Visual Runtime*.*

Componente	Dirección	Contrato
LED	salida	Pin `anode`; encendido si el bit leído es `'1'`; color configurable.
Arreglo de LEDs	salida	Pin `anodes` multibit; posición 0 = bit más significativo; longitud 1–64.
Interruptor	entrada	Pin `out`; estado retenido por la página (sobrevive Run/Stop).
Banco DIP	entrada	Modo por bit (`out0..outN`) o vector (`value`); 2–8 palancas numeradas desde el bit 0.
Botón	entrada	Pin `out`; presionado mientras se sostiene; rebote opcional.
Siete segmentos	salida	Modo crudo (8 pines `a..g`, `dp`) o decodificado (pin `value` de 4 bits, hex/dec).
Motor paso a paso	salida	4 pines de fase; decodifica los 8 patrones *full-step* y acumula el giro con *wrap-around*.

La Tabla 5.4 sintetiza las decisiones de diseño y sus limitaciones reconocidas: la más relevante es la latencia de ∼100–150 ms entre pulsación y respuesta visual en el objetivo web (impuesta por *SignalR* más el *throttle* de repintado de 30 Hz), frente al presupuesto de $<$50 ms del motor.

Tabla 5.4: *Decisiones de diseño del subsistema* Visual Runtime*.*

Decisión	Justificación
Reutilizar el `DeltaCycleEngine`	Se evita mantener un segundo motor y se garantiza que el comportamiento interactivo coincide con el de la simulación por lotes.
Búfer circular acotado	La corrida infinita no puede retener toda la historia; el *scrub* se limita a la ventana del anillo.
Truncamiento natural de ancho	Un desajuste de ancho puerto/widget se resuelve como en un *protoboard* (recorte/relleno), sin rechazar la conexión con un modal.
Entradas sostenidas + rebote opcional	Flancos limpios por omisión; el rebote se activa sólo cuando el ejercicio lo requiere.
Sidecar `.kviz` por lienzo	Separa el estado de interfaz del diseño VHDL; el simulador jamás lo lee.

5.6.4. Subsistema de Depuración en Vivo

La *depuración en vivo* convierte el mismo `DeltaCycleEngine` reutilizado en un depurador interactivo sobre la simulación del editor: permite pausar, reanudar y avanzar la corrida paso a paso, fijar *breakpoints* de línea, de condición y de tiempo, ver pulsar la compuerta en ejecución sobre el esquemático y la línea activa resaltada en Monaco, y reincorporar la vista a una corrida que sigue en curso. Con ello se cierra el hueco de la fase inicial, en la que el resaltado de líneas quedaba vacío por no instrumentar el árbol sintáctico.

El motor expone dos eventos y un control cooperativo de pausa. `OnFrameEmitted` se dispara una vez por ciclo delta con un `FrameEmittedArgs` que transporta las señales cambiadas y su línea de origen; `OnConditionEvaluated` se dispara con la línea de la rama `if`/`elsif` tomada o del *arm* `case` que casó. La pausa es cooperativa: un indicador volátil `_isRunning` y un `CancellationTokenSource` permiten que `Pause`, `Resume` y `StepOneDelta` detengan o avancen el bucle sin bloqueo activo. El servicio `ReplayState` (*scoped* por circuito) concentra la lógica y traduce las acciones de la interfaz en llamadas al motor, de modo que ningún componente lo manipula directamente. La Figura 5.81 muestra la secuencia completa desde la colocación de un *breakpoint* hasta la pausa.

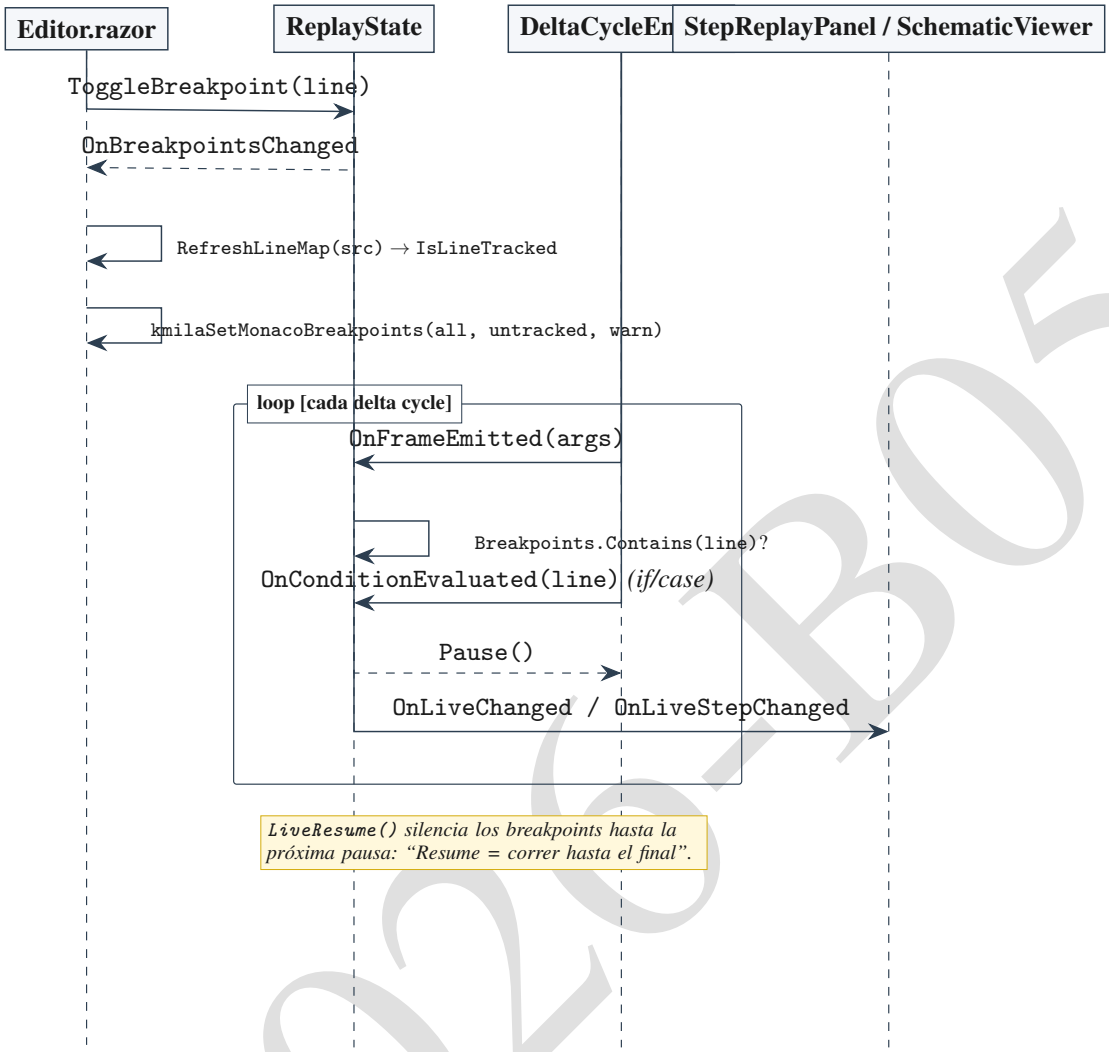


Figura 5.81: *Secuencia de la depuración en vivo. Al colocar un* breakpoint *el* `Editor` *reconstruye el mapa señal→línea y repinta los glifos de Monaco marcando los no rastreables; durante la corrida,* `OnFrameEmitted` *y* `OnConditionEvaluated` *permiten a* `ReplayState` *decidir la pausa y notificar a los paneles.*

Los *breakpoints* de línea se clasifican como *rastreables* o *no rastreables*. `ReplayState` construye por expresiones regulares un mapa que asocia cada señal con las líneas donde se le asigna un valor, y le agrega las líneas de condición (`if`/`elsif`/`when`); un *breakpoint* colocado fuera de ese conjunto —en una declaración, un comentario o una línea en blanco— nunca podría detener la ejecución. El editor lo advierte de antemano: `RepaintBreakpointsAsync` calcula el subconjunto no rastreable y lo envía a Monaco, que lo dibuja con un glifo atenuado y punteado y un *tooltip* de advertencia. Los *breakpoints* de condición requieren un tratamiento aparte porque la condición no cambia ninguna señal y por tanto la vía de transición no los atraparía: el analizador captura la línea del `keyword` en `IfBranch.ConditionLine` y `CaseWhenBranch.ConditionLine`, y el motor la emite al tomarse la rama. El recorrido sobre el esquemático se apoya en las líneas activas por delta (`LiveActiveLines`) y el evento `OnLiveStepChanged` —limitado a ∼10 Hz en corrida libre pero inmediato al pausar—, que el `SchematicViewer` traduce en el pulso *is-active-step* sobre la compuerta correspondiente. Finalmente, `HydrateWaveformFromLiveEngine` reconstruye la forma de onda desde la historia del motor al reincorporarse a una corrida en vuelo. La Figura 5.82 resume el cableado de eventos y la Figura 5.25 el modelo de clases.

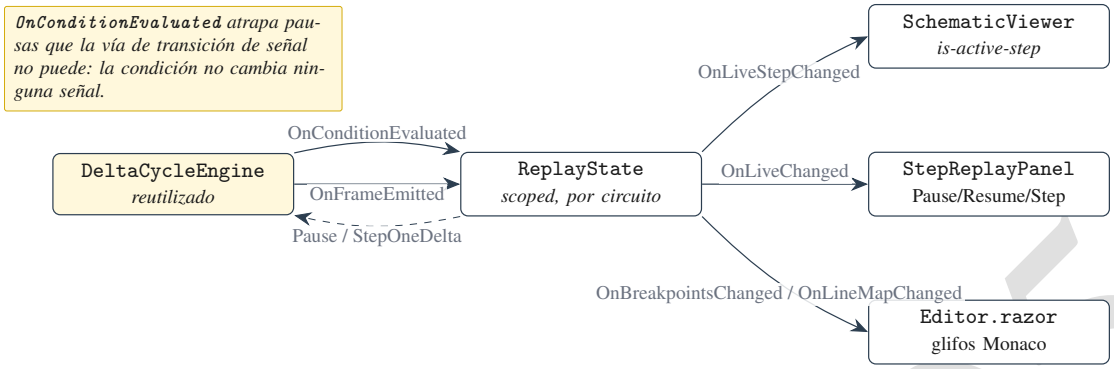


Figura 5.82: *Subsistema de* depuración en vivo*: cableado de eventos. El* `DeltaCycleEngine` *emite* `OnFrameEmitted` *y* `OnConditionEvaluated` *hacia* `ReplayState`*, que decide la pausa y propaga* `OnLiveChanged`*,* `OnLiveStepChanged`*,* `OnBreakpointsChanged` *y* `OnLineMapChanged` *al* `StepReplayPanel`*, al* `SchematicViewer` *y al* `Editor`*.*

Tabla 5.5: *Decisiones de diseño del subsistema de depuración en vivo.*

Decisión	Justificación
Reutilizar `ReplayState` y el motor	La depuración es aditiva sobre la vía de emisión de fotogramas ya existente; no se crea un subsistema de depuración separado.
`Resume` silencia *breakpoints*	El modelo mental es "reanudar = correr hasta el final"; los *breakpoints* se rearman en la siguiente pausa o paso.
Mapa de líneas por regex	Evita instrumentar el árbol sintáctico completo; el motor aporta la línea exacta (`SourceLine`) cuando la conoce, y las condiciones usan la línea capturada por el analizador.
Acumulación de *LiveDoc* opcional	La captura por fotograma es *opt-in* y se acota a 5000 cuadros para limitar la memoria en corridas prolongadas.
Estado *scoped* por circuito	Dos proyectos abiertos en dos pestañas no interfieren; el estado se pierde al recargar (por diseño).

5.6.5. Subsistema de Edición por Bloques

La pestaña *Blocks* del editor es una superficie de programación visual que traduce en **ambos sentidos** entre el código VHDL de Monaco y una representación por bloques arrastrables. A diferencia de la variante del módulo *Concepts* —que sólo compara el programa del estudiante contra un catálogo pre-baked por *fingerprint*—, esta pestaña es un editor de propósito general, aunque ambas comparten el componente `BlockEditor.razor` y el mismo motor de bloques.

El motor merece una aclaración de honestidad técnica. Originalmente los bloques se implementaron con *Blockly* v11 empaquetado sin conexión, lo que provocó un conflicto documentado con el cargador AMD (`define()`) de Monaco; la versión vigente sustituye Blockly por un motor propio basado en el DOM (`kmila-blocks.js`) que conserva el formato de serialización de Blockly y los nombres de *interop* (`kmilaBlockly*`), de modo que ninguna pieza posterior tuvo que cambiar. La evolución de esta decisión se documenta en la Sección 6.1.4.

El *pipeline* de traducción se compone de dos servicios tolerantes a fallos (ante entrada malformada devuelven cadena vacía en lugar de lanzar excepción). En sentido directo, `VhdlBlockEmitter.Emit` recorre el *workspace* JSON y emite VHDL; su rutina `EmitIfChain` aplana el patrón anidado *else*→*if* de los bloques en `elsif` propios, y mantiene un `LastBlockLineMap` que liga cada línea VHDL emitida al bloque de origen, de suerte que un diagnóstico

del simulador "en la línea *N*" se resalta sobre el bloque culpable. En sentido inverso, `VhdlBlockBuilder.Build` es un analizador por expresiones regulares —no un *parser* completo— que reconoce los constructos frecuentes y convierte cualquier construcción no reconocida en un bloque `raw_*` que preserva el texto original íntegro; `LastRawCount` alimenta un aviso cuando algo no pudo representarse. Un *sidecar* `.bvhdl` preserva el *layout* autoral que el analizador por regex no puede recuperar. El servicio `BlockProgram` reduce el *workspace* a un *fingerprint* canónico independiente de la posición pero sensible al orden, razón por la cual el módulo *Concepts* pre-hornea varias variantes conmutativas por ejercicio. La Figura 5.83 resume el *roundtrip* y la Figura 5.26 el modelo de clases.

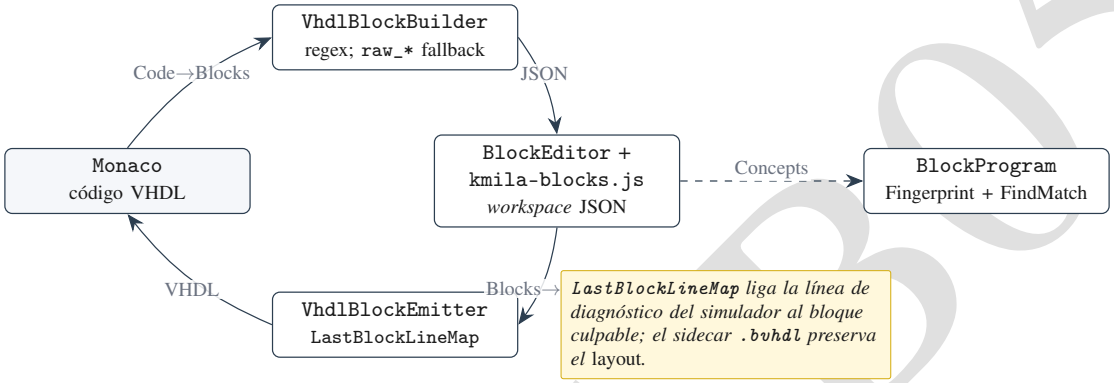


Figura 5.83: *Subsistema de* edición por bloques*:* roundtrip *VHDL↔bloques.* `VhdlBlockBuilder` *convierte el código de Monaco en* workspace *JSON (con* fallback *a bloques* `raw_*`*);* `VhdlBlockEmitter` *regenera el VHDL preservando el mapa línea→bloque;* `BlockProgram` *deriva el* fingerprint *que consume el módulo* Concepts*.*

Una auditoría del traductor produjo diecinueve hallazgos priorizados (F-1 a F-19). Se implementaron los tipos de puerto/señal vectoriales (F-1), los operadores booleanos binarios (F-2), la recuperación ante construcciones no reconocidas mediante bloques `raw` (F-3), la familia `case/when/others` (F-4), los literales de cadena de bits (F-5) y el clon de `falling_edge` (F-7). Quedan en cartera, entre otros, la conservación de comentarios en el *roundtrip* (F-6), el *slot* nativo de `elsif` (F-8), el analizador de expresiones con precedencia completa (F-9) y —marcado como de alto apalancamiento— las pruebas unitarias del constructor y el emisor (F-19). La Tabla 5.6 recoge las decisiones de diseño.

Tabla 5.6: *Decisiones de diseño del subsistema de edición por bloques.*

Decisión	**Justificación**
Motor propio en lugar de Blockly	Elimina el conflicto AMD con Monaco y ∼1 MB de dependencia empaquetada, y habilita el temado por *tokens*; el costo fue reimplementar arrastre, encaje y deshacer.
Conservar el formato de serialización	Al preservar el JSON de Blockly y los nombres de *interop*, el *fingerprint* y los catálogos de *Concepts* siguieron funcionando sin cambios.
Reconocedor por regex, no *parser*	Elección pragmática y tratable; degrada con elegancia a bloques `raw_*` ante entradas no reconocidas.
Sidecar `.bvhdl`	Preserva el *layout* autoral de los bloques que el reconocedor por texto no puede reconstruir.

5.6.6. Ruta de síntesis: del comportamiento al esquemático

Además de simular, el intérprete puede *sintetizar* el diseño a una red de compuertas (*netlist*) para alimentar la vista de esquemático. Esta ruta es independiente de la simulación: `Synthesizer.Synthesize(Entity)` es una función pura que recorre las *tasks* de la arquitectura y delega cada constructo al sub-sintetizador correspondiente (`ConcurrentAssignSynthesizer`, `ConditionalAssignSynthesizer`, `SelectedAssignSynthesizer`, `ProcessSynthesizer`), acumulando celdas y redes en un `SynthesisContext`. Las llamadas a funciones IEEE se bajan a celdas mediante `IEEEPrimitives.TryLower`. El `Netlist` resultante lo posiciona `LayeredLayout.Layout` antes de renderizarlo. La Figura 5.84 muestra la vista de clases y la Figura 5.85 el flujo.

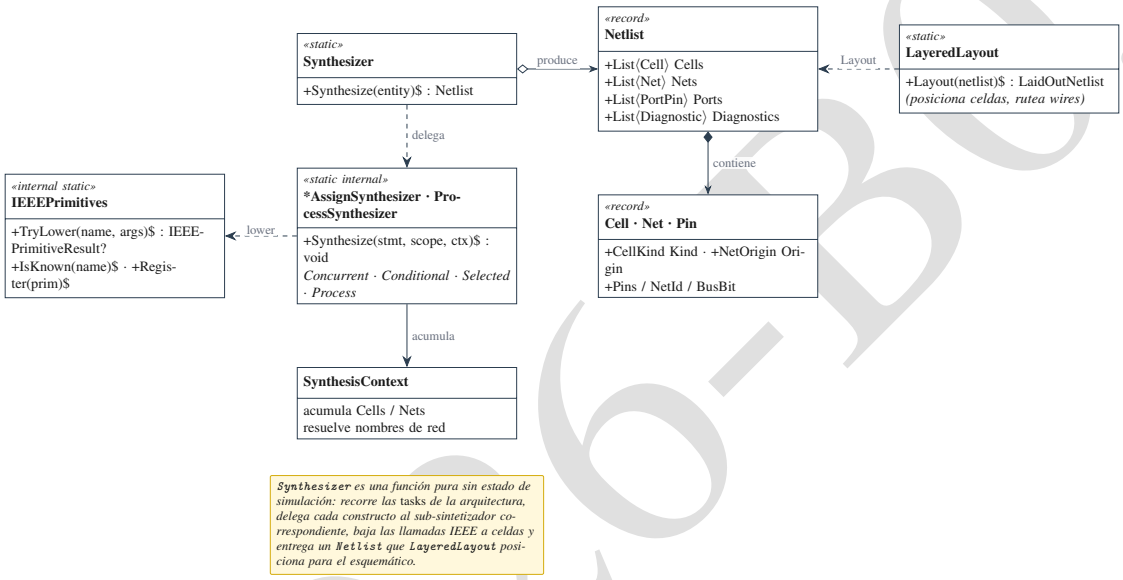


Figura 5.84: *Diagrama de clases de la ruta de síntesis. `Synthesizer` (función pura estática) delega en los sub-sintetizadores por constructo, que acumulan `Cell`/`Net` en el `SynthesisContext`; `IEEEPrimitives` baja las llamadas IEEE a celdas y `LayeredLayout` posiciona el `Netlist` para el esquemático.*

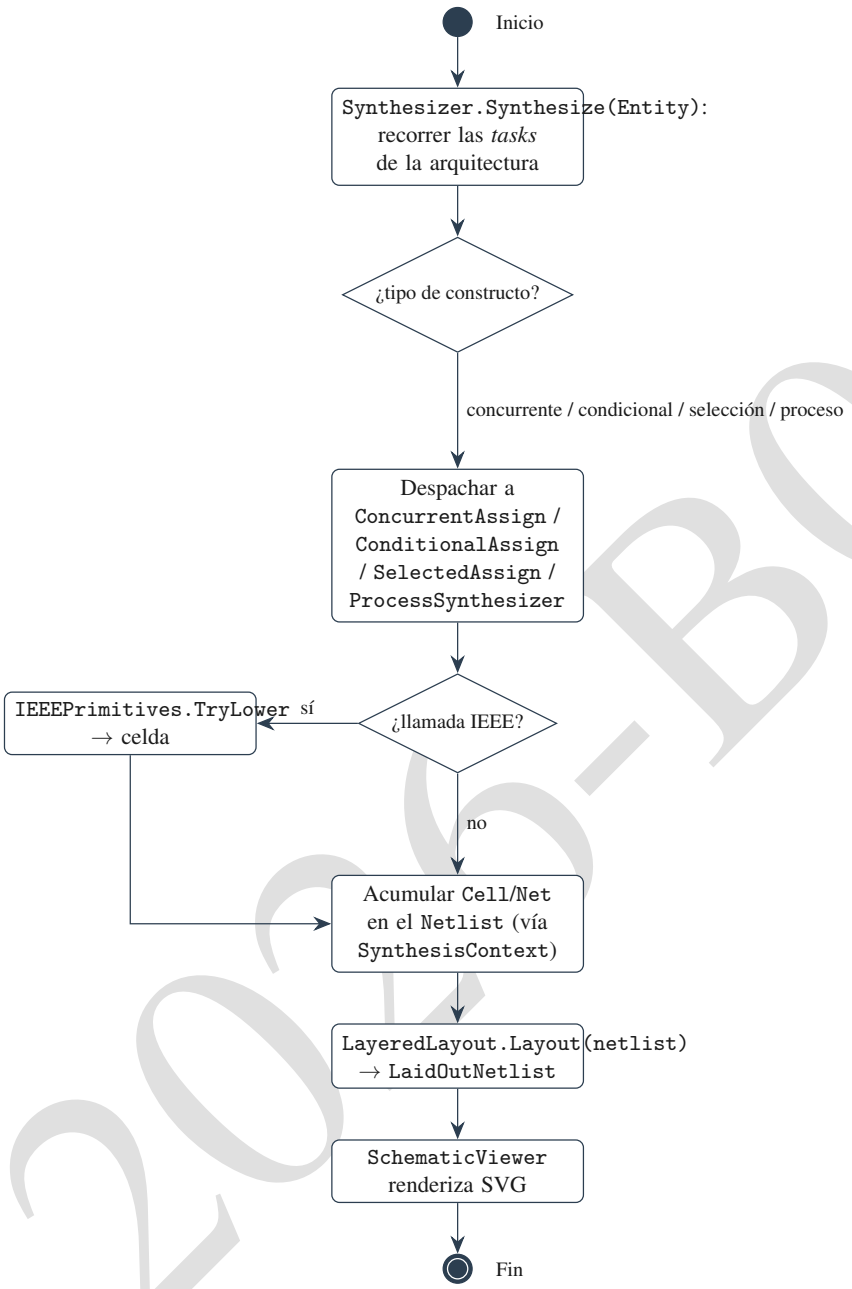


Figura 5.85: *Diagrama de actividad de la síntesis: recorrido de las* tasks *de la arquitectura, despacho por tipo de constructo, bajado de primitivas IEEE, acumulación del* `Netlist` *y posicionamiento con* `LayeredLayout` *para el* `SchematicViewer`.

5.6.7. Pipeline de exportación

Las tres vistas exportables (formas de onda, esquemático y diagrama de flujo) comparten un mismo patrón: un *exporter* estático construye la representación textual y un servicio de plataforma la persiste. `WaveformExporter.BuildSvg`, `SchematicExporter.BuildSvg` y `PlantUmlFlowEmitter.Emit` producen SVG, PNG (vía la interoperación JavaScript `kmilaSvgToPngBase64`), VCD o UML, que la interfaz `IFileExportService` guarda con `SaveTextAsync/SaveBytesAsync` en `Documents/Kmila/`. La Figura 5.86 y la Figura 5.87 detallan la estructura y la secuencia.

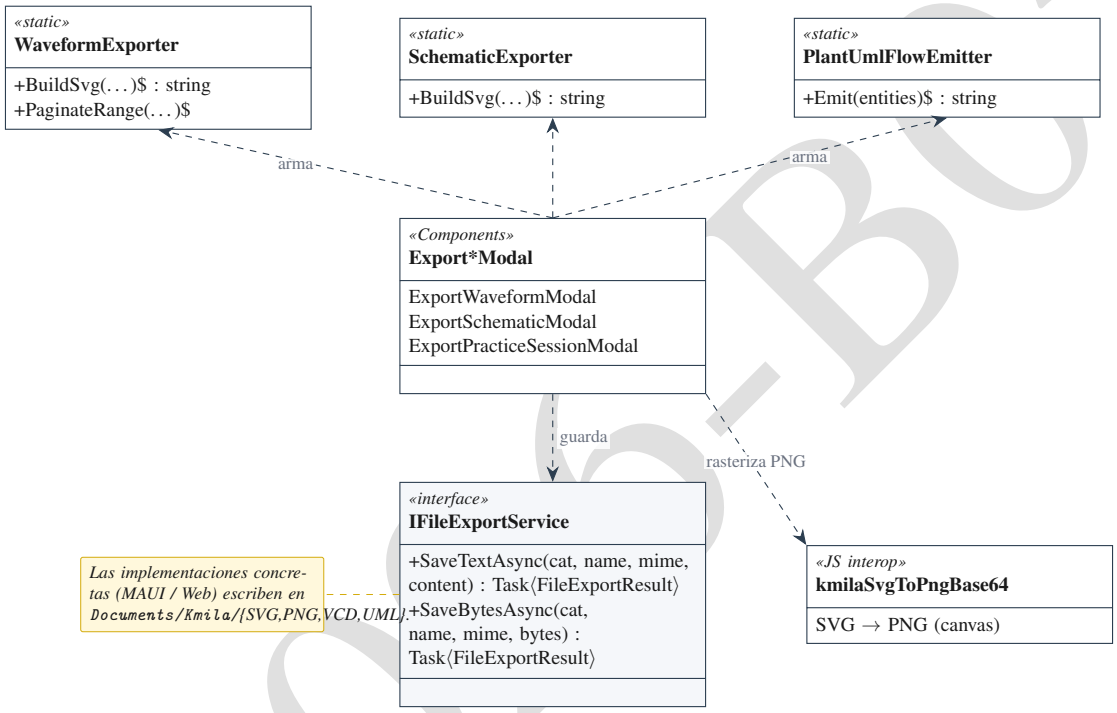


Figura 5.86: *Diagrama de clases del pipeline de exportación: los* exporters *estáticos (`WaveformExporter`, `SchematicExporter`, `PlantUmlFlowEmitter`) alimentan los modales de exportación, que persisten a través de `IFileExportService`; el paso a PNG se rasteriza en JavaScript.*

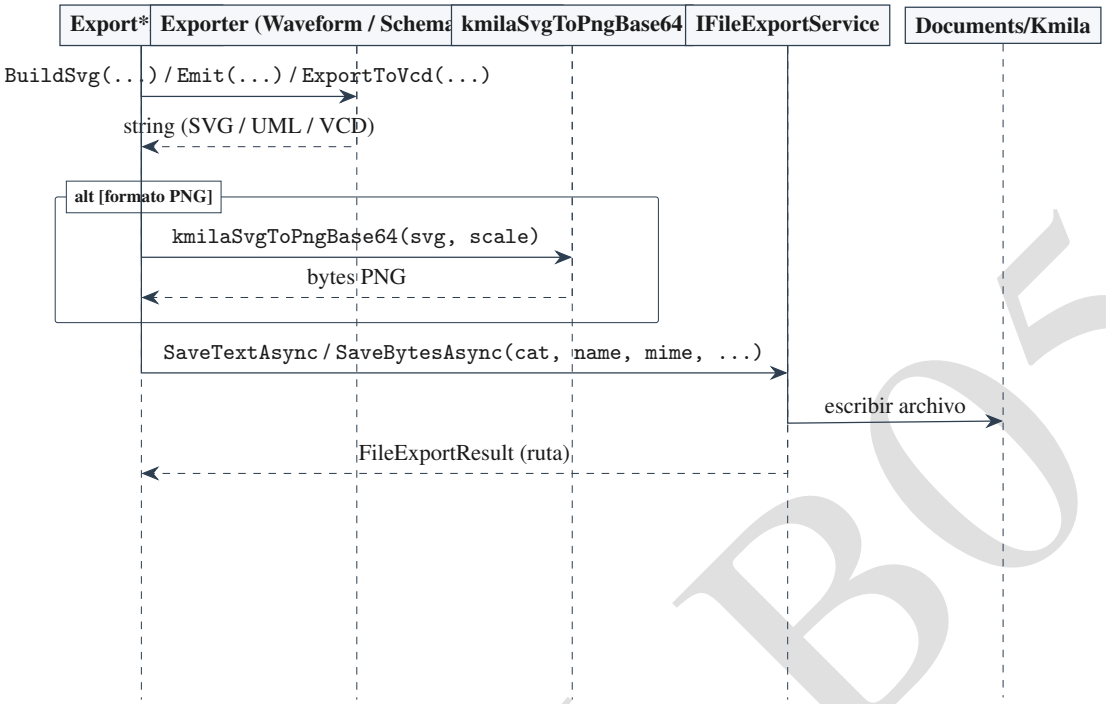


Figura 5.87: *Secuencia de exportación desde un modal: construcción del SVG/UML/VCD, rasterizado opcional a PNG y guardado mediante* `IFileExportService` *en el directorio de la aplicación.*

5.6.8. Coordinación de simulación

El `SimulationCoordinator` desacopla el ciclo de vida de una corrida del componente que la lanzó: al navegar fuera del `Editor` (que se desmonta), la simulación continúa y su instantánea se captura de todos modos. Es *singleton* en MAUI y *scoped* por circuito en la versión web, y se suscribe a los eventos `OnSimulationEnded`/`OnSimulationCompleted` del `ProgramBuilder` para registrar el proyecto activo y persistir la corrida vía `SimulationSnapshotStore.CaptureAsync`, notificando además con *toast* y notificación del sistema (Figura 5.88).

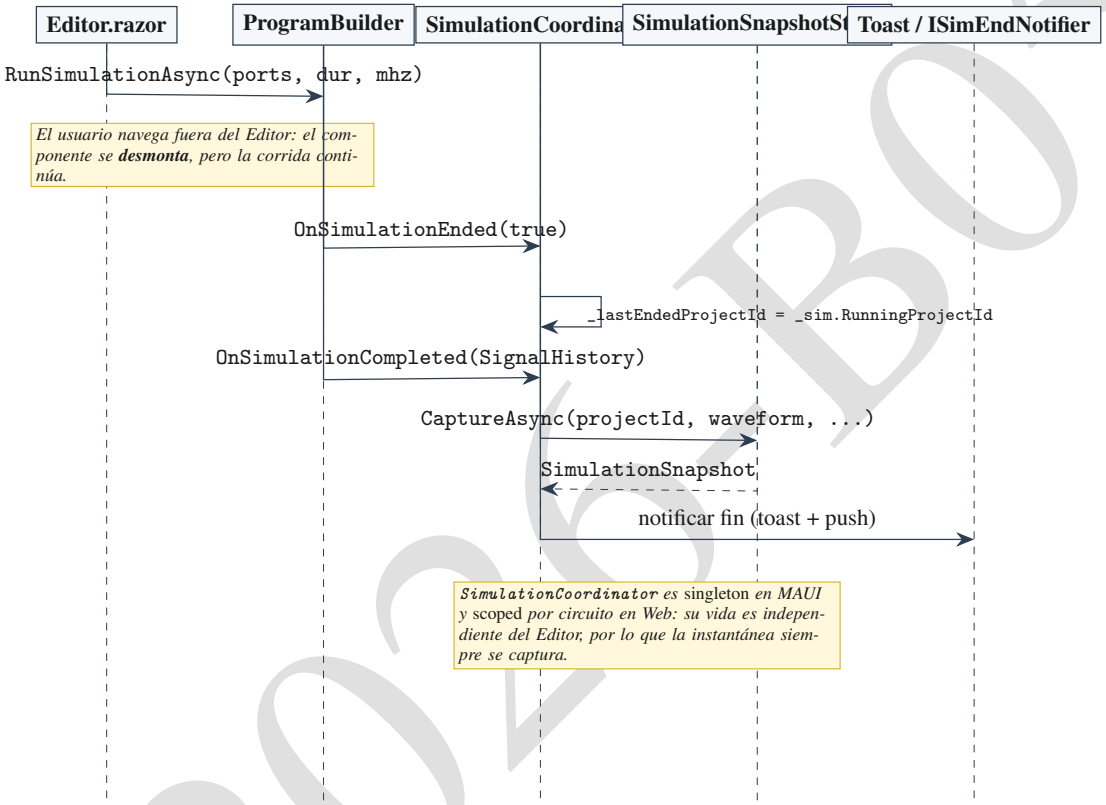


Figura 5.88: *Secuencia del `SimulationCoordinator`: la corrida sobrevive al desmontaje del `Editor`; al completarse, el coordinador captura la instantánea (`CaptureAsync`) y notifica al usuario, con independencia del componente que inició la simulación.*

5.7. Funcionamiento Interno del Simulador

Esta sección documenta de forma técnica y precisa cómo opera el simulador VHDL de Kmila: el pipeline completo desde la pulsación de *Run* hasta la presentación de las formas de onda, los algoritmos del motor de eventos discretos, la representación del tiempo, la diferenciación entre código concurrente y secuencial, el subsistema temporal interactivo (`TimeMachine`), la generación del archivo VCD y las garantías de compatibilidad con *GTKWave*. Las citas de código son del repositorio del proyecto al estado del 22 de mayo de 2026.

5.7.1. Pipeline de Simulación de Extremo a Extremo

Cuando el estudiante presiona *Run* en la barra del editor, el sistema atraviesa una secuencia coordinada de cinco etapas que involucran ocho componentes principales. La Tabla 5.7 resume el flujo y los archivos responsables.

Tabla 5.7: *Pipeline de simulación: etapas, componentes y artefactos producidos.*

Etapa	Componente	Acción / artefacto producido
1. Orquestación	`ProgramBuilder` (Kmila.Shared)	Recibe `RunSimulationAsync(portMoments, duration)`, dispara la compilación y delega al motor.
2. Análisis sintáctico	`Parser.ParseV2`	Tokeniza el archivo, popula el diccionario de *scopes* con `Entity`, `Architecture`, `Process`, `Variable`.
3. Análisis estructural	`DbgParser` (Debugger)	Pass de *lint* resiliente + reconstrucción line-based de la jerarquía.
4. Construcción de estímulos	`StimulusBuilder.BuildConfig`	Convierte `PortMomentInput` de la UI a `SimulationConfig` con `ClockConfig` si se detecta un puerto de reloj.
5. Simulación	`SimulationRunner` → `DeltaCycleEngine`	Ejecuta ciclo principal: por cada *tick*, evalúa todos los *delta cycles* hasta estabilidad.
6. Captura	`SignalHistory`	Cada cambio de señal se persiste como tupla `(timeMoment, value, sourceLine)`.
7. Notificación	`OnSimulationCompleted`	El motor entrega un `WaveformData` serializable al `Editor.razor`.
8. Visualización	`WaveformViewer.razor`	Renderiza las señales como funciones escalonadas sobre SVG/Canvas.

El motor admite dos *paths* de ejecución que el orquestador elige según las necesidades del usuario: el **path rápido** ejecuta `DeltaCycleEngine` a velocidad CPU completa sin instrumentación interactiva (típicamente <1 s para diseños <5000 transiciones de señal); el **path interactivo** entrelaza la ejecución con el módulo `TimeMachine`, permitiendo pausa/reanudación responsivas y ejecución paso a paso (Sección 5.7.6).

5.7.2. El motor: `DeltaCycleEngine`

`DeltaCycleEngine` implementa la semántica de eventos discretos del estándar IEEE 1076 mediante un bucle de *ciclos delta* [10]. La función `RunTimeStepAsync()` avanza la simulación un *tick* (resolución temporal mínima); dentro de un tick, el motor itera ciclos delta hasta que el sistema alcanza estabilidad (ninguna asignación pendiente, ningún proceso pendiente). El listado siguiente reproduce el bucle principal (`Interpreter/Services/DeltaCycleEngine.cs:21`–267):

```
1  public async Task RunTimeStepAsync()
2  {
3      _currentDelta = 0;
4
5      // En t=0, inicializa todos los procesos (despierta todos)
6      if (_currentTick == 0 && _currentDelta == 0)
7          _processScheduler.InitializeAllProcesses();
8      else
9      {
10         // Detecta señales con eventos externos (p. ej. reloj) y notifica
11         var changedSignals = new HashSet<Variable>();
12         foreach (var signal in _allSignals)
13             if (SignalAttributeHandler.Instance.GetEvent(signal))
14                 changedSignals.Add(signal);
15         if (changedSignals.Count > 0)
16             _processScheduler.NotifySignalChanges(changedSignals);
17     }
18
19     // Itera ciclos delta hasta estabilidad
20     bool hasMoreWork;
21     do {
22         hasMoreWork = await RunDeltaCycleAsync();
23         _currentDelta++;
24         if (_currentDelta > MAX_DELTA_CYCLES)
25             throw new InvalidOperationException(
26                 $"Delta cycle limit ({MAX_DELTA_CYCLES}) exceeded at tick {_currentTick
                       }. "+
27                 "Possible combinational loop in design.");
28     } while (hasMoreWork && _isRunning && !_cts.IsCancellationRequested);
29
30     OnSimulationStable?.Invoke();
31     SignalAttributeHandler.Instance.UpdateAllPreviousValues(_allSignals);
32 }
```

Listing 5.1: *Bucle principal de `RunTimeStepAsync`: ejecuta ciclos delta hasta estabilidad.*

La constante `MAX_DELTA_CYCLES = 1000` (línea 67) actúa como *guardia contra bucles combinacionales*: si la simulación lleva 1000 ciclos delta consecutivos sin estabilizarse dentro del mismo tick, el motor aborta con excepción explícita. Esta cota empírica es estándar en simuladores comerciales y previene cuelgues del programa al simular diseños con realimentación combinacional sin registro de por medio.

El bucle interno `RunDeltaCycleAsync` ejecuta una sola iteración de delta cycle en seis pasos verificables (`DeltaCycleEngine.cs:352`): (1) obtener procesos pendientes del `ProcessScheduler`; (2) ejecutar cada proceso registrando sus asignaciones en el `SignalScheduler`; (3) ejecutar las asignaciones concurrentes que no están dentro de procesos; (4) aplicar las actualizaciones pendientes con el método `ApplyCurrentUpdatesWithLines()` que además registra la línea fuente de cada cambio; (5) persistir los cambios en `SignalHistory` con el *time moment* compuesto `timeMoment` = `tick` × 1000 + `delta`; (6) notificar al `ProcessScheduler` para que marque como pendientes los procesos sensibles a las señales que cambiaron en este delta.

5.7.3. Programación de asignaciones: `SignalScheduler`

El servicio `SignalScheduler` mantiene **dos estructuras de datos paralelas** que materializan la distinción canónica del estándar IEEE 1076 entre transacciones que deben aplicarse en el delta actual y transacciones programadas para el futuro (`Interpreter/Services/SignalScheduler.cs:27-33`):

- `Queue<SignalUpdate> _currentDeltaUpdates` — cola FIFO de asignaciones a aplicar al final del delta cycle en curso. Cada asignación `signal <= value` dentro de un proceso o de una asignación concurrente se encola aquí, no se aplica inmediatamente. Esto materializa la regla VHDL de que *el valor de una señal no cambia hasta el final del delta cycle*.
- `SortedDictionary<ulong, Queue<SignalUpdate>> _futureUpdates` — diccionario ordenado por `futureTick`, asocia cada *tick futuro* con la lista de actualizaciones que deben dispararse en ese instante. Se popula con `ScheduleAfter()` cuando el código VHDL contiene cláusulas `after` (p. ej. `signal <= '1' after 10 ns`).

El método `ApplyCurrentUpdates()` drena la cola actual y devuelve el conjunto de señales cuyo valor efectivamente cambió, lo que sirve al `ProcessScheduler` para reactivar procesos sensibles. El método `AdvanceTime(newTick)` promueve las actualizaciones futuras al alcanzar su tick programado:

```
public void AdvanceTime(ulong newTick)
{
    while (_futureUpdates.Count > 0)
    {
        var firstEntry = _futureUpdates.First();
        if (firstEntry.Key > newTick) break;      // todavía no toca
        var updates = firstEntry.Value;
        while (updates.Count > 0)
            _currentDeltaUpdates.Enqueue(updates.Dequeue());
        _futureUpdates.Remove(firstEntry.Key);
    }
}
```

Listing 5.2: *Promoción de `_futureUpdates` a `_currentDeltaUpdates` al avanzar el tiempo.*

Esta estructura permite que el motor evalúe constructos VHDL como `signal <= value after 5 ns;` de manera correcta: `ScheduleAfter` calcula `futureTick = currentTick + delay/tickResolutionNs` y encola la actualización ahí; cuando el motor avanza el tiempo a ese tick, `AdvanceTime` la promueve al delta actual y el

cambio se aplica en el siguiente `ApplyCurrentUpdates`.

5.7.4. Activación de procesos: `ProcessScheduler`

El servicio `ProcessScheduler` (`Interpreter/Services/ProcessScheduler.cs`) mantiene un **mapa N:M de sensibilidad-a-procesos** de tipo `Dictionary<Variable, List<Process»`. Durante la elaboración (`Initialize`), por cada `Process` declarado en la arquitectura, el método `RegisterProcess(p)` recorre su `SensitiveList` y registra al proceso como subscriptor de cada variable de su lista de sensibilidad:

```
public void RegisterProcess(Process process)
{
    _allProcesses.Add(process);
    foreach (var signal in process.SensitiveList)
    {
        if (!_sensitivityMap.TryGetValue(signal, out var processes))
        {
            processes = new List<Process>();
            _sensitivityMap[signal] = processes;
        }
        processes.Add(process);
    }
}
```

Listing 5.3: *Registro de procesos en el mapa de sensibilidad.*

Cuando el `SignalScheduler` reporta señales modificadas al final del delta cycle, el motor invoca `NotifySignalChanges(changed)`: por cada señal cambiada, busca los procesos sensibles en `_sensitivityMap` y los *marca como pendientes* sin re-ejecutarlos en el delta actual — se ejecutarán en el siguiente delta cycle. Esto preserva la semántica VHDL de que un proceso despertado por una señal observa el valor de esa señal en el delta posterior al cambio, no en el delta del cambio mismo.

5.7.5. Atributos de señal: `'event`, `'last_value`, `'stable`, `'active`

`SignalAttributeHandler` (singleton) mantiene un *snapshot* del valor anterior de cada señal y expone los atributos VHDL más usados: `GetEvent(signal)` retorna `true` si el valor actual difiere del anterior (utilizado en `rising_edge(clk)`, `falling_edge(clk)`); `GetLastValue(signal)` retorna el valor del delta anterior; `GetStable(signal)` y `GetActive(signal)` se derivan de los mismos buffers. En cada *commit* de delta cycle, el método `OnDeltaCommit` congela los valores actuales como "previous" y sella el instante del cambio; al cierre de cada tick, `ClearActiveFlagsAtTickBoundary` reinicia los indicadores de actividad para que la detección de eventos del siguiente tick parta de un estado limpio.

5.7.6. Tiempo discreto y subsistema `TimeMachine`

La resolución temporal mínima del simulador es el *tick*. En el módulo `TimeMachine`, la constante `NS_PER_TICK = 10` (`TimeMachine/Models/TimeClock.cs:24`) establece que cada tick representa 10 ns de tiempo simulado. Esta resolución se eligió por dos razones: (i) es suficientemente fina para representar relojes en el rango típico de FPGA pedagógicas (1–100 MHz, períodos de 10–1000 ns); (ii) permite que un contador `ulong` de 64 bits cubra del orden de $1{,}8 \times 10^{19}$ ticks ≈ 18 *exasegundos*, eliminando cualquier preocupación por desbordamiento incluso en simulaciones extremadamente largas.

Para evitar incluso ese desbordamiento teórico, `TimeClock` expone una propiedad `Multiplier` que se incrementa cada vez que el contador `Tick` interno se reinicia, permitiendo simulaciones efectivamente ilimitadas.

El *time moment* interno del `DeltaCycleEngine` no es directamente el tick global, sino una codificación compuesta que captura también el ciclo delta dentro del tick: $\texttt{timeMoment} = \texttt{tick} \times 1000 + \texttt{delta}$ (`DeltaCycleEngine.cs:325`). Esta codificación permite que `SignalHistory` ordene cambios que ocurren en el mismo tick pero en delta cycles distintos, lo que es esencial para reproducir fielmente la secuencia "transacción posteada → delta cycle cerrado → valor visible".

5.7.6.1. Path dual: rápido vs. interactivo.

El `TimeMachine` es **opcional**. Cuando el usuario presiona *Run* sin habilitar pausa/step, el motor se ejecuta a velocidad CPU completa, encadenando `RunTimeStepAsync` hasta `TotalTicks`. Cuando el usuario habilita el modo interactivo, el `Timer` de `TimeMachine` (`TimeMachine/Repositories/Timer.cs`) impone un `Task.Delay(10)` entre ticks y entrelaza la simulación con eventos de pausa/reanudación gestionados por dos `CancellationTokenSource` independientes:


```
private CancellationTokenSource _cts = new(), _pauseCts = new();

private async Task StartSimulation(TimeSpan? Duration) {
    ulong maxTicks = Duration is null ? 0 : GetSimulationTicks(Duration.Value, _clock.Ns
        );
    while (!_cts.IsCancellationRequested) {
        if (!Paused) {
            CurrentTick = (ulong)_clock.Ns * _clock.Tick * (_clock.Multiplier > 0 ?
                _clock.Multiplier : 1);
            TimeElapsed = GetTimeFromTicks((ulong)CurrentTick, _clock.Ns);
            if (maxTicks != 0 && CurrentTick >= maxTicks) {
                OnSimulationEnded?.Invoke(true);
                break;
            }
            _clock.Next();
            OnSimulationTick?.Invoke((ulong)CurrentTick); // dispara delta cycle
            await Task.Delay(10);                          // 10 ms reales por tick
        } else {
            try { await Task.Delay(Timeout.Infinite, _pauseCts.Token); }
            catch (TaskCanceledException) { Paused = false; }
```

```
19          }
20      }
21  }
```

Listing 5.4: *Bucle principal del `Timer` en `TimeMachine`: doble token de cancelación para parar y para pausar.*

El `OnSimulationTick` se enlaza al `DeltaCycleEngine` mediante `ConnectToTimeMachine`, de modo que cada disparo del `Timer` provoca un `RunTimeStepAsync`. La pausa se implementa con un `Task.Delay(Timeout.Infinite)` sobre el token `_pauseCts`; el botón *Continue* cancela ese token y el bucle reanuda automáticamente.

5.7.7. Concurrencia vs. secuencialidad: dos mundos en el mismo delta

El simulador implementa la dicotomía VHDL canónica con dos colecciones diferenciadas dentro del motor:

- **Procesos** (`Process`, en `_processScheduler`) — contienen *código secuencial* en su cuerpo. Al ejecutarse, sus sentencias se evalúan una por una de arriba abajo (incluyendo bucles y condicionales), pero **las asignaciones a señales no surten efecto inmediatamente**: cada `signal <= value` se traduce a una llamada `_signalScheduler.Schedule(target, value)` que encola la actualización en `_currentDeltaUpdates`. Las variables (`variable :=`) sí se asignan inmediatamente.
- **Asignaciones concurrentes** (`Operation`, en `_concurrentTasks`) — viven fuera de cualquier proceso, en el cuerpo de la arquitectura. El motor las evalúa todas en cada delta cycle, en orden indeterminado, encolando sus actualizaciones igualmente en el `SignalScheduler`.

La diferencia operativa clave es que **ambos tipos de constructos encolan sus efectos en la misma cola** `_currentDeltaUpdates` dentro del mismo delta cycle. Esto garantiza que, al final del delta cycle, todas las señales modificadas adoptan sus nuevos valores *simultáneamente*, sin importar si la modificación provino de un proceso o de una asignación concurrente. Esta propiedad reproduce fielmente la semántica del estándar IEEE 1076: el orden textual de las asignaciones concurrentes en el archivo VHDL es irrelevante para el resultado de la simulación.

5.7.8. Visualización de señales: del `SignalHistory` al SVG

Tras cada delta cycle, `SignalHistory.RecordChangesWithLines` persiste las señales que cambiaron como tuplas `(timeMoment, value)` en un diccionario `Dictionary<Variable, List<Tuple<int, object>>>`, y mantiene la línea fuente de cada transición en una estructura paralela `_sourceLines` de tipo `Dictionary<Variable, List<int>>`. Esta estructura es *append-only*, ordenada implícitamente por orden de inserción (que coincide con el orden temporal).

Cuando el motor termina, el orquestador serializa el `SignalHistory` en un objeto `WaveformData` que viaja al `Editor.razor` mediante el evento `OnSimulationCompleted`. El componente `WaveformViewer.razor` convierte el `WaveformData` en SVG inline aplicando una función escalonada (*step function*) por cada señal: el valor entre dos puntos de muestreo consecutivos se mantiene constante (línea horizontal), y al llegar al siguiente cambio se aplica un salto vertical instantáneo seguido de la nueva línea horizontal. Para señales de varios bits, los valores se renderizan como bloques hexagonales que muestran el valor entre transiciones. El visor soporta *zoom* y *pan* mediante manipulación del atributo `viewBox` del SVG. La elección de SVG en línea sobre una biblioteca de gráficas de propósito general (como `Chart.js`, empleada en las primeras versiones de la interfaz) responde a la naturaleza del dato: las formas de onda digitales son funciones escalonadas con buses multi-bit rotulados por su valor, no series continuas interpoladas. El SVG permite transiciones exactas al píxel, etiquetas de valor por segmento de bus, ocultamiento de señales a nivel de elemento del DOM, *zoom*/*pan* por `viewBox` y exportación vectorial nativa (el mismo SVG alimenta las rutas de exportación a SVG y PNG), capacidades que un lienzo de gráficas no ofrece de forma natural.

El `WaveformViewer` incorpora dos mecanismos de inspección que operan sin recargar la traza. El primero es el **ocultado por señal**: un clic sobre la etiqueta de cualquier fila la retira del lienzo y la deposita en una *chip-bar* superior desde la cual puede restaurarse individualmente o en bloque. El conjunto de señales ocultas (`_hiddenSignals`) se respeta también en las tres rutas de exportación (VCD, SVG y PNG), de modo que el archivo generado refleja exactamente lo que el estudiante ve en pantalla. El segundo es el **cursor temporal de *hover***: al desplazar el puntero sobre el cuerpo de la traza, una línea vertical y una etiqueta muestran el instante exacto bajo el cursor. Este cursor se implementa en JavaScript (`kmilaAttachWaveformHover`), que lee los atributos `data-view-start-ns`, `data-view-end-ns` y `data-label-width` del cuerpo en cada cuadro; se descartó deliberadamente el manejador Razor `@onmousemove` porque en Blazor Server enrutaría cada movimiento del puntero por *SignalR*, y un cursor que se actualiza a 60 fps saturaría el circuito de red.

5.7.9. Generación VCD y compatibilidad con GTKWave

La exportación a formato VCD (*Value Change Dump*) está implementada en `SignalHistory.ExportToVcd` (`Interpreter/Services/SignalHistory.cs:463-570`) y se ajusta estrictamente al estándar IEEE 1364 sección 18.2 [12], que es el dialecto que *GTKWave* consume desde su primera versión. El listado siguiente reproduce el algoritmo de cabecera y declaración de variables:

```
public string ExportToVcd(string moduleName = "top", int timescaleNs = 1) {
    var sb = new StringBuilder();
    sb.AppendLine("$date");
    sb.AppendLine($"   {DateTime.Now:yyyy-MM-dd HH:mm:ss}");
    sb.AppendLine("$end");
    sb.AppendLine("$version");
    sb.AppendLine("   VHDL Interpreter v0.17.0");
    sb.AppendLine("$end");
    sb.AppendLine("$timescale");
    sb.AppendLine($"   {timescaleNs}ns");
    sb.AppendLine("$end");

    var signals = _history.Keys.OrderBy(k => k.Name).ToList();
    var signalIds = new Dictionary<Variable, string>();
    sb.AppendLine($"$scope module {moduleName} $end");
    int idCounter = 0;
    foreach (var signal in signals) {
        string id = GenerateVcdId(idCounter++);   // base-94 ASCII
        signalIds[signal] = id;
        int width = GetSignalWidth(signal);
        string vcdType = width == 1 ? "wire" : "reg";
        sb.AppendLine($"$var {vcdType} {width} {id} {signal.Name} $end");
    }
    sb.AppendLine("$upscope $end");
    sb.AppendLine("$enddefinitions $end");
    // ... $dumpvars + cambios por timestamp (ver §siguiente) ...
}
```

Listing 5.5: *Cabecera y declaración de variables del exportador VCD.*

La función `GenerateVcdId` asigna identificadores cortos a cada señal usando los caracteres ASCII imprimibles del rango 33 (`!`) al 126 (`~`), lo que da un alfabeto de 94 caracteres. Para diseños con más de 94 señales, el identificador pasa a dos caracteres en base 94 (índice $94 \rightarrow$ `i`, $95 \rightarrow$, ...), permitiendo hasta $94^2 = 8836$ señales únicas con identificadores de longitud 2, y así sucesivamente. Esta codificación es la canónica del estándar y es exactamente la que utilizan los exportadores de GHDL y ModelSim.

El cuerpo del archivo emite el directorio inicial `$dumpvars` con todos los valores iniciales, seguido de bloques agrupados por timestamp absoluto (precedido por `#`) que contienen solo las señales que cambiaron en ese instante. El siguiente fragmento muestra el archivo `counter_sim.vcd` generado por Kmila al simular un contador de 4 bits:

```
$date
   2026-01-24 09:57:43
```

```
$end
$version
   VHDL Interpreter v0.17.0
$end
$timescale
   10ns
$end
$scope module gtkwave_counter $end
$var wire 1 ! clk $end
$var reg 4 " count $end
$var reg 4 # counter_reg $end
$var wire 1 $ enable $end
$var wire 1 % reset $end
$upscope $end
$enddefinitions $end
$dumpvars
0!
b0000 "
b0000 #
1$
0%
$end
```

Listing 5.6: *Fragmento de `counter_sim.vcd` generado por Kmila — abierto sin modificaciones en GTKWave.*

5.7.9.1. Garantías de compatibilidad con GTKWave.

La paridad con GTKWave se sostiene en tres pilares verificables:

1. **Conformidad con el estándar IEEE 1364-2005 §18.2** [12] — el formato VCD es un estándar IEEE de 30 años de estabilidad. Cualquier exportador que respete las directivas `$date`, `$version`, `$timescale`, `$scope`, `$var`, `$dumpvars`, los timestamps con `#`, y la codificación de valores escalares (`0`, `1`, `x`, `z`) y vectoriales (`b<bits>`) es consumible por GTKWave.
2. **Validación cruzada con GHDL+GTKWave** — el directorio `Interpreter/` contiene archivos VCD de referencia (`reference_counter.vcd`, `reference_combinational.vcd`, `reference_shift_reg.vcd`) generados con GHDL para los mismos diseños VHDL. La comparación señal por señal en los puntos de muestreo coincide con tolerancia de ciclo delta entre las salidas de Kmila y GHDL.
3. **Validación interactiva con GTKWave 3.3.x** — los archivos producidos por `SignalHistory.ExportToVcd` se abren sin modificaciones en GTKWave 3.3.x bajo Linux y macOS, exhibiendo las mismas transiciones que el visor nativo de Kmila.

Las únicas limitaciones conocidas son: (a) el simulador no emite los metavalores `H` y `L` (*weak* 1/0) ni `W` (*weak unknown*) del sistema IEEE 1164 de 9 valores — se reducen a `1`, `0` y `x` respectivamente; (b) la cláusula VHDL `wait until` con condiciones complejas sólo se soporta de forma parcial en el alcance de TT2 (`wait for`, `wait on` y `wait until <signal> = <value>` sí están implementadas).

5.8. Subsistemas Pedagógicos

Kmila se distingue de las herramientas comerciales de simulación de HDL (*AMD Vivado*, *Intel Quartus*, *Siemens ModelSim*) y de los simuladores libres tradicionales (*GHDL*, *Icarus Verilog*, *Verilator*) por incorporar, junto al editor y al motor de simulación, tres **subsistemas pedagógicos** diseñados específicamente para acompañar al estudiante de ingeniería en su recorrido por las asignaturas de diseño digital de la ESCOM. Esta sección describe la motivación, la estructura y el contenido de los tres subsistemas y justifica su importancia pedagógica.

5.8.1. Contexto pedagógico: el currículo ESCOM como marco de referencia

Los subsistemas pedagógicos de Kmila se diseñaron tomando como marco de referencia los programas oficiales de cinco asignaturas del Plan de Estudios de la carrera de Ingeniería en Sistemas Computacionales (ISC) de la ESCOM, vigentes en el ciclo 2020 y consultables en los documentos anexos del directorio `Documentacion/Complementos/`:

1. **Matemáticas Discretas** (`matematicasDiscretas_ISC2020.pdf`) — aporta el sustrato formal de álgebra booleana, sistemas numéricos, funciones lógicas y simplificación de expresiones que precede a cualquier curso de diseño digital.
2. **Fundamentos de Diseño Digital** (`fundamentosDisenoDigital_ISC2020.pdf`) — introduce las compuertas lógicas, las tablas de verdad, los mapas de Karnaugh, los circuitos combinacionales (multiplexores, decodificadores, sumadores) y los primeros elementos secuenciales (*latches*, *flip-flops*).
3. **Diseño de Sistemas Digitales** (`disenoSistemasDigitales_ISC2020.pdf`) — profundiza en lógica secuencial, registros, contadores, máquinas de estados finitos (Mealy y Moore) y describe formalmente el modelado mediante VHDL.
4. **Arquitectura de Computadoras** (`arquitectutaComputadoras_ISC2020.pdf`) — contextualiza el diseño digital dentro de la arquitectura de sistemas (datapath, control, ALU, jerarquía de memoria) y motiva el uso del HDL para describir bloques de un procesador.
5. **Sistemas en Chip** (`sistemasChip_ISC2020.pdf`) — cierra el currículo con el diseño a nivel de sistema, integración IP, síntesis en FPGA y consideraciones de *co-design hardware-software*.

Kmila **no pretende reemplazar** a la docencia presencial, los laboratorios físicos ni el material didáctico de estas asignaturas: su rol es **complementar** y **apoyar** el proceso de aprendizaje cuando el estudiante no dispone de tarjeta FPGA, cuando requiere experimentar fuera del horario de laboratorio, o cuando necesita visualizar conceptos abstractos (ciclos delta, propagación de señales, modelos de ejecución) que son difíciles de comunicar con pizarrón únicamente. La filosofía es la del *laboratorio virtual de pre-práctica*: un entorno donde el alumno puede preparar y validar su diseño antes de acudir al laboratorio físico, o como alternativa cuando éste no esté disponible.

Bajo este enfoque se diseñaron tres subsistemas didácticos integrados que se describen en las subsecciones siguientes: *Learn* (lecciones estructuradas), *Practice* (ejercicios autoevaluables) y *Concepts* (modelos mentales de ejecución).

5.8.2. Subsistema Learn: lecciones estructuradas alineadas con el currículo ESCOM

El subsistema *Learn* es un catálogo de lecciones de VHDL organizadas en **módulos didácticos** progresivos, cada uno mapeado a una o varias asignaturas de la ESCOM. La Tabla 5.8 muestra la correspondencia entre los módulos de Learn y las asignaturas oficiales.

Tabla 5.8: *Módulos del subsistema* Learn *y su correspondencia con asignaturas de la ESCOM.*

Módulo	**Contenido (lecciones)**	**Asignatura**
Digital Fundamentals	Sistemas numéricos · Álgebra booleana · Las siete compuertas · Laboratorio de tabla de verdad	Matemáticas Discretas · Fund. Diseño Digital
VHDL Basics	Anatomía de un archivo · Tipos de datos · `signal` vs. `variable` · Primera entidad (MUX 2:1)	Fund. Diseño Digital
Combinational Logic	Asignaciones concurrentes · Decodificador 3:8 · Sumador 4 bits con *ripple carry* · Ejercicio *fix tristate*	Fund. Diseño Digital · Diseño de Sistemas Digitales
Sequential Logic	Procesos y listas de sensibilidad · Flip-flop D con reset asíncrono · Registro de desplazamiento 8 bits · Ejercicio *accidental latch*	Diseño de Sistemas Digitales
Finite-State Machines	Mealy vs. Moore · Semáforo · Máquina vendedora	Diseño de Sistemas Digitales · Arq. de Computadoras
Datapath & ALU	Registros · Banco de registros · ALU de 8 bits	Arq. de Computadoras
System-on-Chip	Buses · Protocolos AXI/AHB simplificados · IP *cores*	Sistemas en Chip

Cada lección sigue una estructura homogénea: encabezado en los siete idiomas soportados, cuerpo en *Markdown* con ejemplos de código VHDL, animaciones opcionales de compuertas (`GateAnimator`) o de máquinas de estados (`FsmAnimator`), un `ExecutionPlayer` *inline* cuando aplica, y — si la lección incluye una muestra (`Sample`) — un botón *"Try in Editor"* que crea un proyecto temporal con el código de la lección, abierto directamente en el editor con los estímulos pre-configurados. El servicio `LessonCatalog` hidrata el catálogo desde el manifiesto `wwwroot/Learn/manifest.json` con *fallback* multiidioma al inglés; el servicio `LessonProgressStore` registra el avance del estudiante en la tabla `AppSettings` de SQLite con claves prefijadas `lesson_*`.

5.8.3. Subsistema Practice: ejercicios autoevaluables al estilo competitivo

El subsistema *Practice* se inspira en las plataformas competitivas de práctica de programación más establecidas del mundo: **CodeForces** [66], **HackerRank** [67] y **LeetCode** [68]. Estas plataformas demostraron empíricamente que la combinación de (i) enunciados claros con casos de prueba ocultos, (ii) evaluación automática inmediata y granular, (iii) clasificación por dificultad y (iv) la posibilidad de iterar sobre un mismo problema hasta resolverlo, produce un avance pedagógico significativo en el dominio de los algoritmos y las estructuras de datos. Kmila traslada este modelo al dominio del diseño digital con HDL.

A diferencia de las plataformas mencionadas, que se restringen a una única modalidad (escribir el algoritmo desde cero), Kmila reconoce que el aprendizaje de un lenguaje de descripción de hardware atraviesa fases cognitivas distintas y, por tanto, ofrece **tres modalidades de ejercicio** con propósitos pedagógicos diferenciados (véase Tabla 5.9).

Tabla 5.9: *Las tres modalidades de ejercicio del subsistema* Practice *y su intención pedagógica.*

Modalidad	Mecánica	Habilidad ejercitada
FromZero	Se presenta el enunciado del problema y la lista de puertos de la entidad. El estudiante escribe la *entity* y la *architecture* completas desde un archivo vacío. El evaluador simula el código contra todos los casos de prueba.	Síntesis del problema en estructuras VHDL; fluidez en la sintaxis del lenguaje; comprensión de la especificación funcional.
FixBug	Se entrega código casi correcto que falla en uno o varios casos de prueba debido a un *bug* explícito (operador equivocado, índice mal escrito, condición invertida, asignación faltante). El estudiante debe diagnosticar el error y corregirlo.	Lectura crítica de código ajeno; depuración mediante el visor de formas de onda; correlación entre síntoma observable y causa en el código.
FillGap	Se entrega código parcialmente escrito con marcadores `-- BEGIN_EDIT` / `-- END_EDIT` que delimitan la región editable. El estudiante completa una función o un bloque puntual; `PracticeEvaluator` fusiona el código original con la edición del estudiante antes de simular, descartando cualquier modificación fuera de los marcadores.	Comprensión de un diseño existente; integración de un fragmento en un contexto dado; práctica de patrones de implementación específicos.

El catálogo está organizado en directorios físicos (`wwwroot/Practice/m1-from-zero/`, `m2-fix-bug/`, `m3-fill-gap/`), cada uno subdividido por nivel de dificultad (*easy*, *medium*, *hard*). Cada ejercicio contiene un manifiesto, el código de partida (cuando aplica), el código solución (oculto al estudiante), una lista de puertos de salida a comparar y una lista de *tiempos de muestreo* en nanosegundos donde la comparación se realiza. El catálogo actual abarca, entre las tres modalidades, más de cien ejercicios que van desde compuertas básicas y *multiplexores* (*easy*) hasta sumadores, FSMs, UARTs y arbitros *round-robin* (*hard*).

La evaluación se realiza mediante el servicio `PracticeEvaluator` (Figura 5.74): para cada *TestCase*, ejecuta dos simulaciones — una sobre la solución canónica y otra sobre el código del estudiante — y compara las `WaveformData` resultantes con tolerancia numérica de 10^{-9}, reportando el primer punto de discrepancia (señal, tiempo, valor esperado, valor obtenido) o un veredicto *Solved*. El veredicto y todos los intentos quedan persistidos en `PracticeAttemptStore` para permitir al estudiante revisar su historial.

5.8.4. Subsistema Concepts: el modelo mental de ejecución VHDL

El subsistema *Concepts* es la contribución pedagógica más original del proyecto. Su existencia responde a una observación empírica: **los estudiantes que dominan la sintaxis de VHDL frecuentemente fracasan al razonar sobre el comportamiento de su código** porque trasladan inconscientemente el modelo de ejecución secuencial de los lenguajes imperativos (C, Python, Java) al dominio del hardware. Esta confusión se manifiesta de tres formas características:

1. Interpretar `signal <=` como una asignación inmediata análoga a = en C, ignorando que las transacciones se posponen al cierre del ciclo delta.
2. Asumir que el orden textual de las asignaciones concurrentes dentro de una arquitectura determina el orden de evaluación, cuando en realidad las asignaciones concurrentes son *order-independent*.
3. Confundir *simulación* (tiempo discreto que avanza por eventos) con *paralelismo real* (electrones moviéndose simultáneamente en el silicio), creyendo que ambos modelos son equivalentes.

El módulo *Concepts* ataca estas confusiones con un currículo mini de seis lecciones (Tabla 5.10) seguidas de cuatro ejercicios scratch-style en editor visual *Blockly* [69].

Tabla 5.10: *Lecciones del subsistema* Concepts *y modelo de ejecución que cada una internaliza.*

Lección	Idea central
01. ¿Qué significa “ejecutar”?	Comparación lado a lado de los tres modelos: secuencial, concurrente, paralelo real. Concluye con un *quizlet* de tres opciones.
02. Secuencial: un paso a la vez	El cuerpo de un `process` VHDL se ejecuta de arriba abajo, una sentencia a la vez — como en C.
03. Concurrente: todo se dispara junto	Las asignaciones concurrentes en una arquitectura se evalúan todas en el mismo ciclo delta; el orden textual es irrelevante.
04. Por qué VHDL no puede hacer paralelismo real	Distinción simulación/hardware. Delta cycles como aproximación discreta del paralelismo continuo del FPGA.
05. Listas de sensibilidad y ciclos delta	Por qué una transacción registrada ahora se hace visible apenas en el siguiente ciclo delta. *Quizlet* sobre la regla.
06. Juntándolo todo	Un medio-sumador combinacional y un contador secuencial conviviendo en la misma arquitectura.

Cada lección que carga el reproductor (`ExecutionPlayer`) muestra simultáneamente: (i) el código VHDL en un editor Monaco de sólo lectura con resaltado de la línea actualmente en ejecución, (ii) una tabla en vivo del estado de las señales que se actualiza con cada paso, y (iii) una nota pedagógica localizada que explica qué hace el simulador en ese paso. El estudiante controla la reproducción con una barra de transporte (Play, Pause, Step-forward, Step-back, Restart, selector de velocidad 0.5×/1×/2×), permitiendo detenerse a observar exactamente el momento donde la intuición secuencial falla. El motor `TimelineRunner`, implementado en C# puro mediante un bucle `Task.Delay` con tokens de cancelación, evita la dependencia de *timers* de JavaScript.

Los ejercicios complementarios usan el editor visual *Blockly* para que la sintaxis no sea una barrera: el estudiante arrastra bloques (un *XOR*, un *AND*, una asignación, un envoltorio `on rising_edge(clk)`) y compone, por ejemplo, un medio-sumador o un contador. El servicio `BlockProgram` calcula un *fingerprint canónico* del programa

visual (formato `type:field=val|next` ordenado para neutralizar permutaciones equivalentes) y lo compara contra un catálogo pre-baked de composiciones válidas; si coincide, se reproduce el *timeline* correspondiente; si no, se ofrece una pista pedagógica.

5.8.5. Integración de los tres subsistemas con el editor

Los tres subsistemas no son piezas aisladas: están **integrados con el editor principal** para que el aprendizaje fluya sin interrupción hacia la práctica real. Cada lección con muestra de Learn ofrece un botón "Try in Editor" que crea un proyecto temporal con el código de la lección. Cada ejercicio de Practice abre el editor con el código de partida ya cargado y los estímulos configurados. La tab *Replay* del editor (Figura 5.78) emplea el mismo componente `ExecutionPlayer` que usa el subsistema *Concepts*, de forma que el estudiante encuentra una experiencia consistente: la habilidad de leer una traza paso a paso adquirida en las lecciones se transfiere directamente al análisis de sus propias simulaciones. Esta continuidad "aprender → practicar → construir → revisar" es el principal valor pedagógico añadido del proyecto.

5.9. Capturas de la Interfaz

Las capturas de las pantallas principales de la aplicación se presentan a continuación. Corresponden al prototipo funcional de Kmila ejecutándose en modo web (ASP.NET Core + Blazor Server, accedido vía navegador en `localhost:5297`), con el idioma español y tema oscuro activos. Las capturas complementarias (pantalla de inicio, selector de idioma, tema claro, internacionalización con idioma árabe/japonés, módulo de aprendizaje y configuración avanzada) se encuentran en el Anexo A.3.

5.9.1. Pantalla Principal — Editor de Código

El editor de código es el componente central de la aplicación. Está construido sobre Monaco Editor (el mismo motor de Visual Studio Code), configurado con resaltado de sintaxis VHDL, numeración de líneas, minimapa de navegación, indicador de posición del cursor y atajo de guardado rápido (`Ctrl+S`). La Figura 5.89 muestra un archivo `gates.vhd` abierto en el editor, correspondiente a la lección de exploración de tablas de verdad del módulo de aprendizaje.

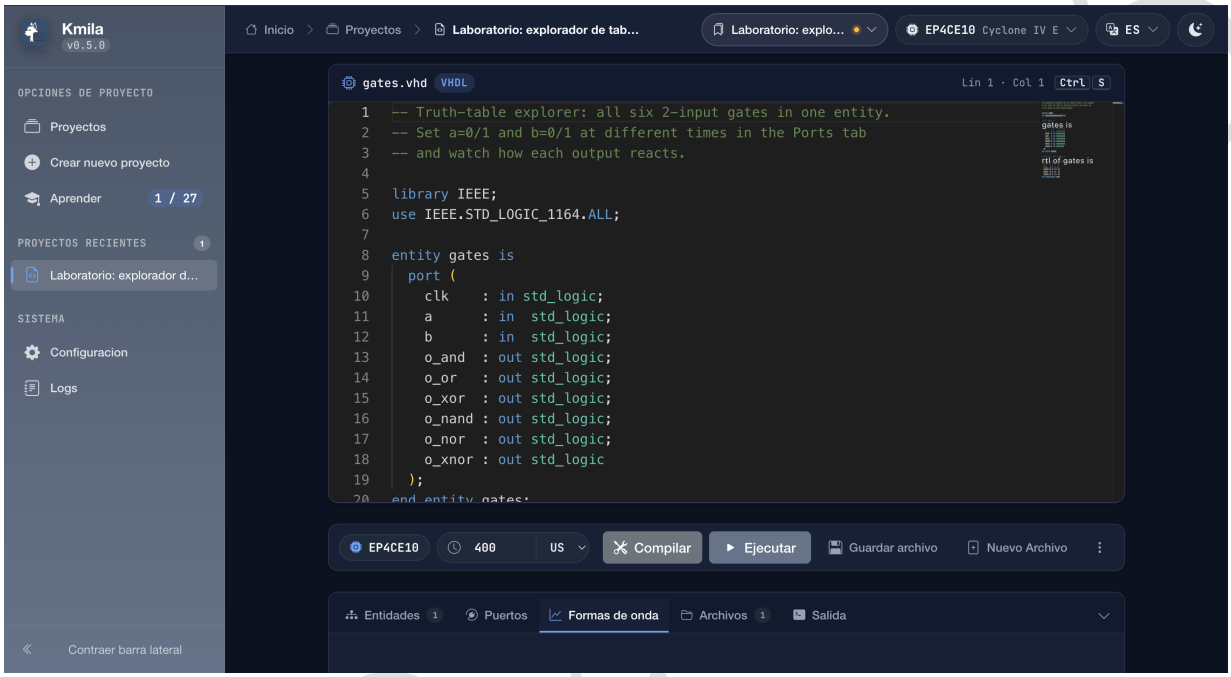


Figura 5.89: *Editor de código VHDL con Monaco Editor, mostrando el archivo `gates.vhd` del laboratorio de exploración de tablas de verdad. Se aprecia el resaltado de sintaxis, minimapa, numeración de líneas y la barra de acciones inferior (Compilar, Ejecutar, Guardar archivo, Nuevo Archivo) junto con el selector de FPGA (EP4CE10) y los parámetros temporales de simulación.*

El editor incorpora dos capacidades didácticas adicionales que distinguen a Kmila de un editor genérico: descripciones contextuales al posicionar el cursor sobre palabras clave VHDL (Figura 5.90) y plantillas de autocompletado inteligente con variantes pedagógicamente relevantes (Figura 5.91).

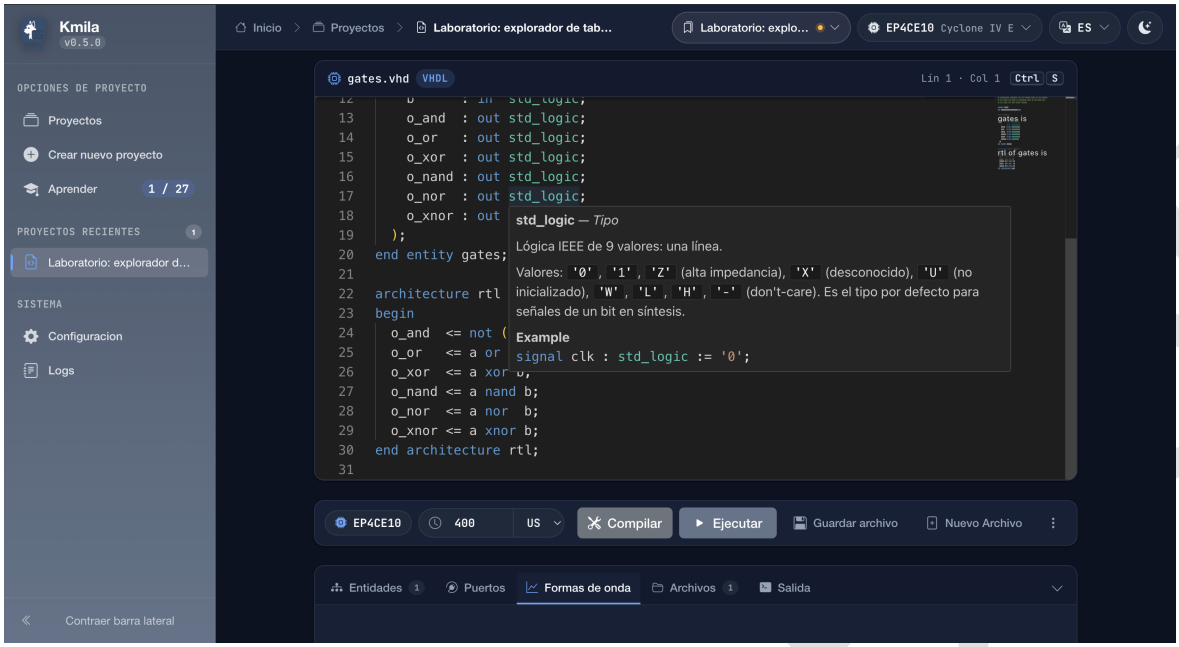


Figura 5.90: *Tooltip contextual sobre la palabra clave* `std_logic`*, mostrando la descripción del tipo junto con los nueve valores definidos por IEEE 1164 y un ejemplo de declaración. Este mecanismo refuerza el aprendizaje en el punto exacto donde el estudiante encuentra un concepto nuevo.*

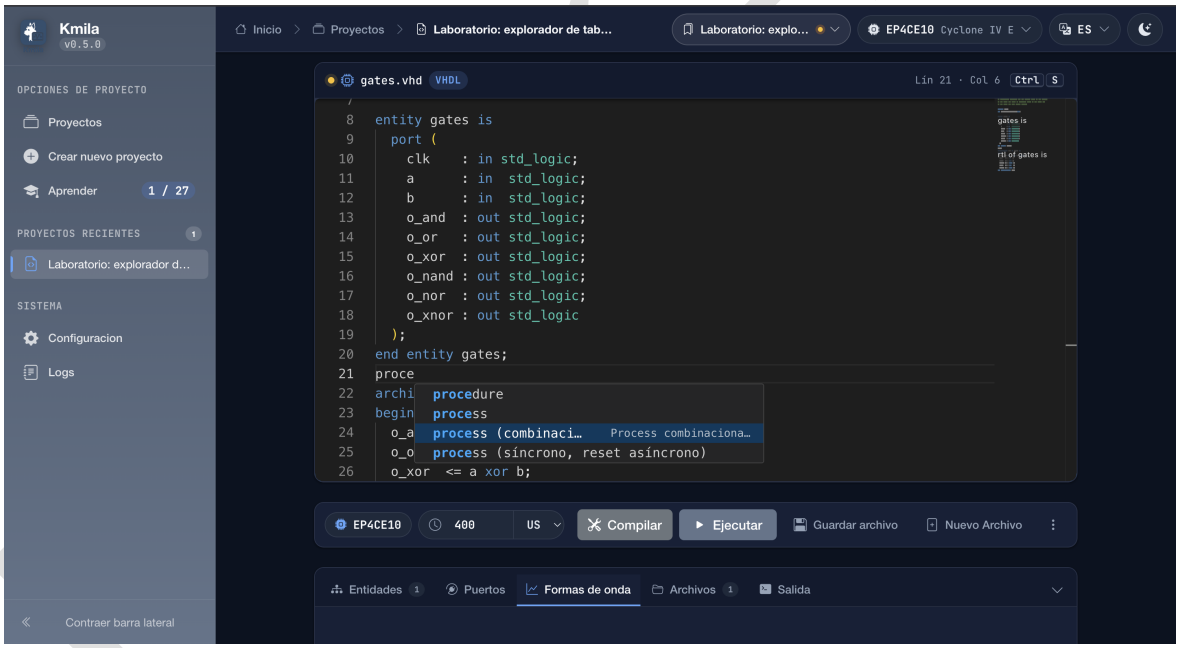


Figura 5.91: *Autocompletado con plantillas de snippets para* `process`*: se ofrecen tres variantes pedagógicamente diferenciadas (*procedure*,* process *genérico,* process combinacional*,* process síncrono con reset asíncrono*), guiando al estudiante hacia el estilo correcto según el caso de uso.*

5.9.2. Panel de Simulación

El panel inferior del editor agrupa las herramientas de análisis y configuración de simulación en pestañas. La pestaña *Puertos* (Figura 5.92) permite definir los estímulos de cada señal de entrada especificando momentos discretos en el eje temporal; cada fila representa un cambio de valor en un instante con su escala asociada (ns, us, ms). La pestaña *Entidades* (documentada en el Anexo A.3) permite inspeccionar la estructura de las entidades VHDL detectadas por el analizador. La pestaña *Uso de recursos* (Figura 5.93) presenta la estimación de utilización del dispositivo FPGA seleccionado.

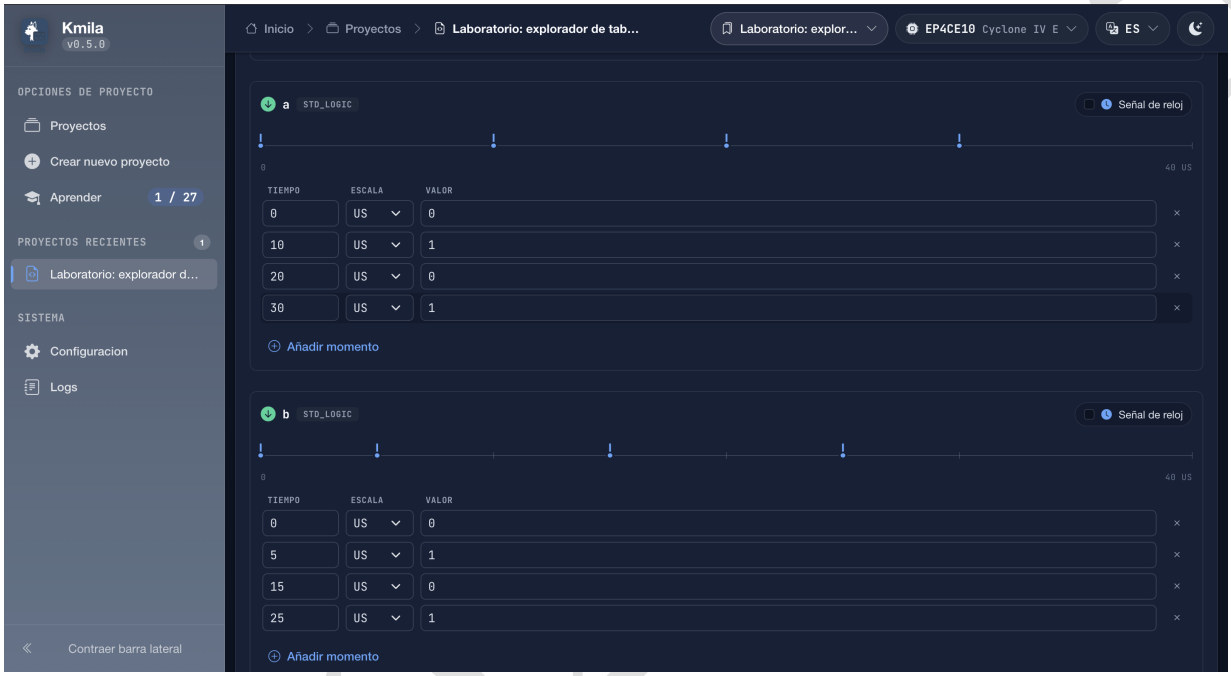


Figura 5.92: *Pestaña* Puertos*: configuración de estímulos para las señales de entrada* `a` *y* `b`*. Cada fila (Tiempo, Escala, Valor) define un instante en el que la señal cambia de valor. La línea temporal superior marca visualmente las transiciones programadas, y un interruptor permite designar cualquier señal como reloj (*clk*).*

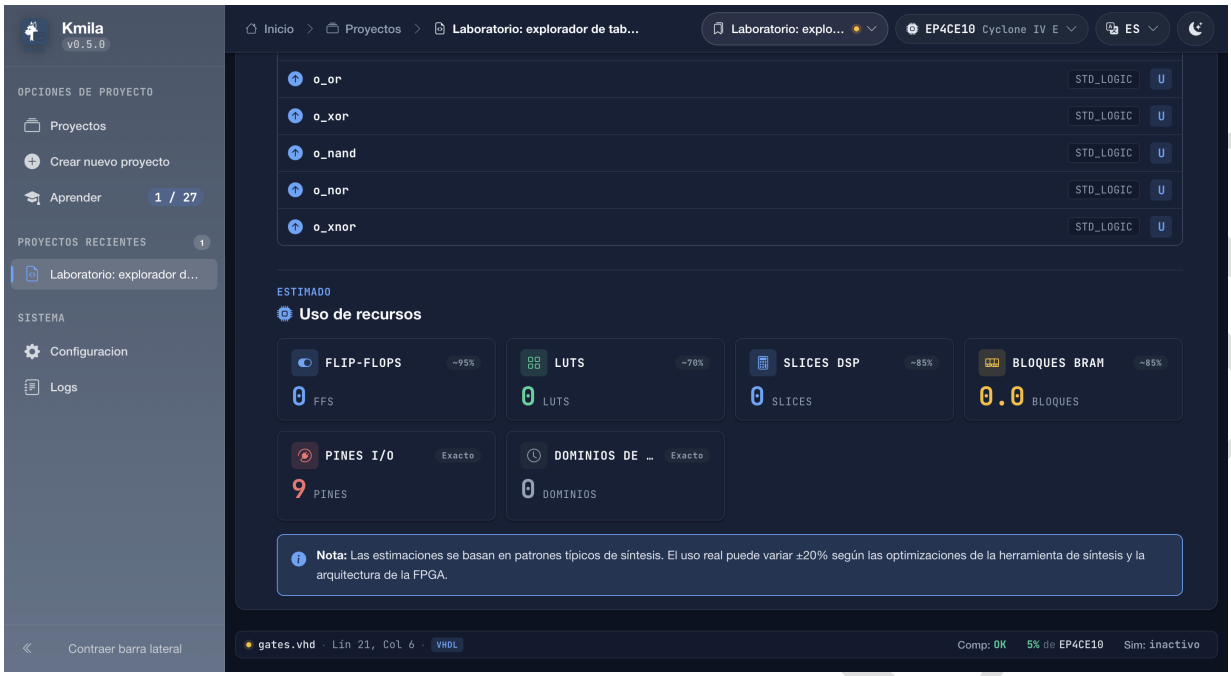


Figura 5.93: *Pestaña* Uso de recursos*: estimación de utilización del dispositivo FPGA seleccionado, desglosada en Flip-Flops, LUTs, Slices DSP, Bloques BRAM, Pines I/O y Dominios de reloj. La barra inferior indica la precisión de cada métrica (porcentaje de error esperado o resultado exacto) y una nota aclara la variabilidad inherente a las herramientas de síntesis reales.*

5.9.3. Visualizador de Señales (Ondas)

El visualizador de formas de onda, construido sobre SVG en línea, presenta la evolución temporal de las señales del diseño tras ejecutar una simulación. Las señales de entrada se muestran en azul, el reloj en verde (con resaltado sólido) y las salidas en naranja, permitiendo al estudiante identificar visualmente el tipo de señal sin necesidad de leer la leyenda. La Figura 5.94 ilustra la salida del laboratorio de exploración de tablas de verdad con nueve señales simultáneas.

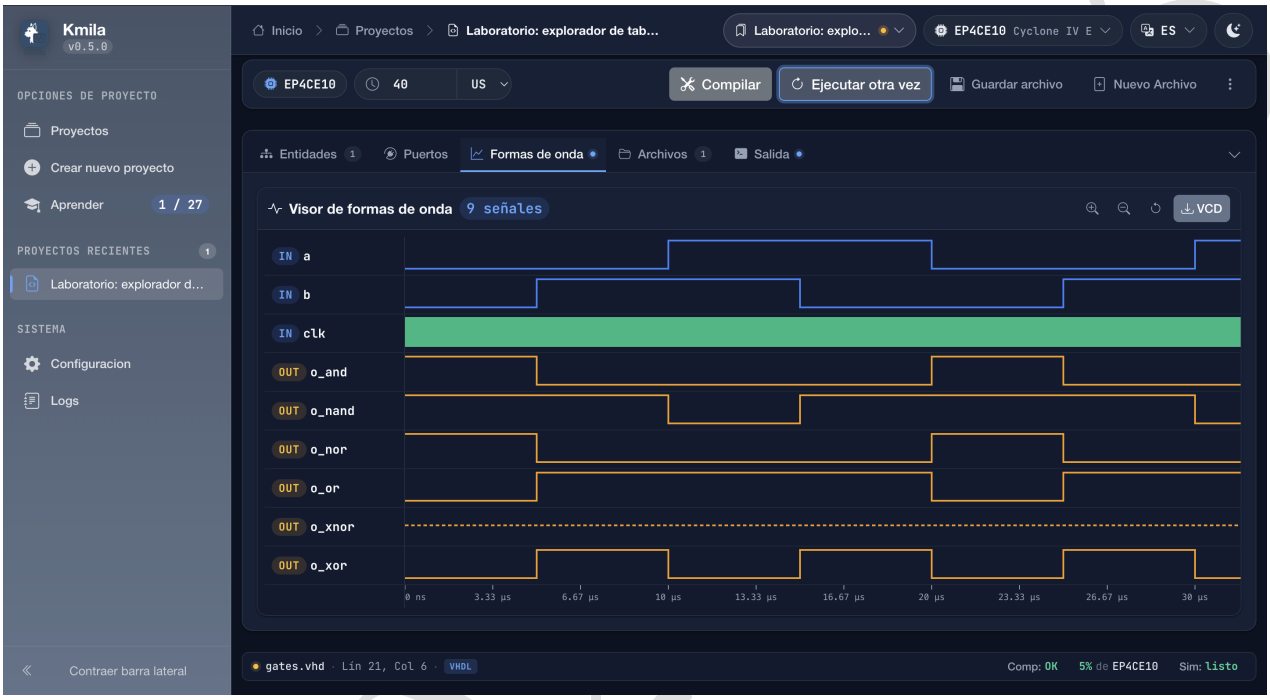


Figura 5.94: *Visualizador de formas de onda con nueve señales del laboratorio `gates.vhd`: tres entradas (`a`, `b`, `clk`), seis salidas combinacionales (`o_and`, `o_or`, `o_xor`, `o_nand`, `o_nor`, `o_xnor`). La línea temporal cubre el rango completo de simulación, con controles de acercamiento/alejamiento y botón de exportación al formato estándar VCD.*

> *Las formas de onda de este apartado son **salida real** del motor de simulación ya construido (`DeltaCycleEngine` y sus componentes, Sección 5.7.2), no ejemplos ilustrativos: la equivalencia funcional de esa salida frente al simulador de referencia GHDL se valida sobre 148 casos en la Sección 6.2.3.*

5.9.4. Panel de Navegación y Proyecto

La vista de *Proyectos* (Figura 5.95) presenta los proyectos existentes en tarjetas con resumen, fecha de última modificación y acceso directo al editor. El usuario puede crear un proyecto nuevo mediante un modal dedicado (Figura 5.96) o abrir rápidamente un ejemplo precargado. Una vez dentro del editor, el selector superior de proyecto ofrece el flujo de guardado: el proyecto activo se marca como **modificado** automáticamente al detectar cambios no guardados, y el menú desplegable permite guardar los cambios en la versión vigente o crear una nueva versión guardada (Figura 5.97).

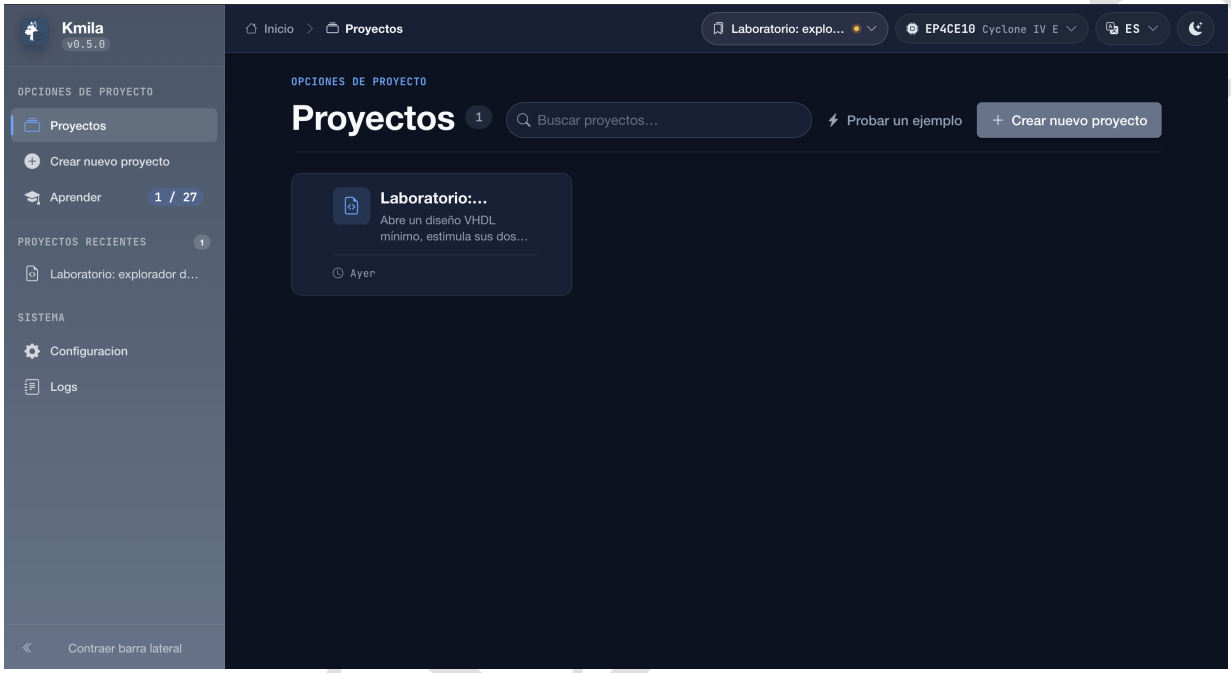


Figura 5.95: *Vista de* Proyectos*: lista de proyectos existentes con tarjetas de resumen. La barra superior ofrece búsqueda por nombre y los accesos directos* Probar un ejemplo *(que abre un proyecto de demostración) y* Crear nuevo proyecto *(que despliega el modal de la Figura 5.96).*

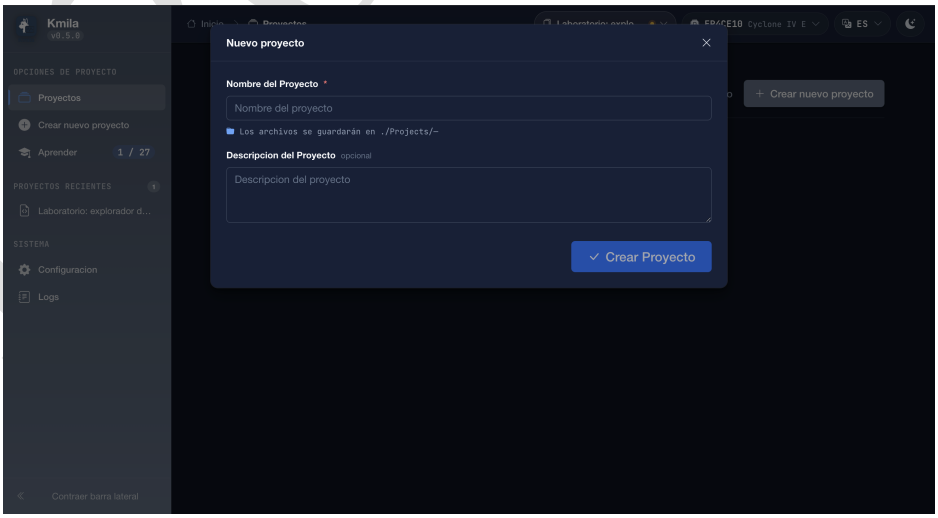


Figura 5.96: *Modal de creación de proyecto: el usuario indica nombre (obligatorio) y descripción (opcional). La ubicación de los archivos se preconfigura en* `./Projects/` *bajo el directorio de la aplicación, reforzando el principio de operación local sin servidores.*

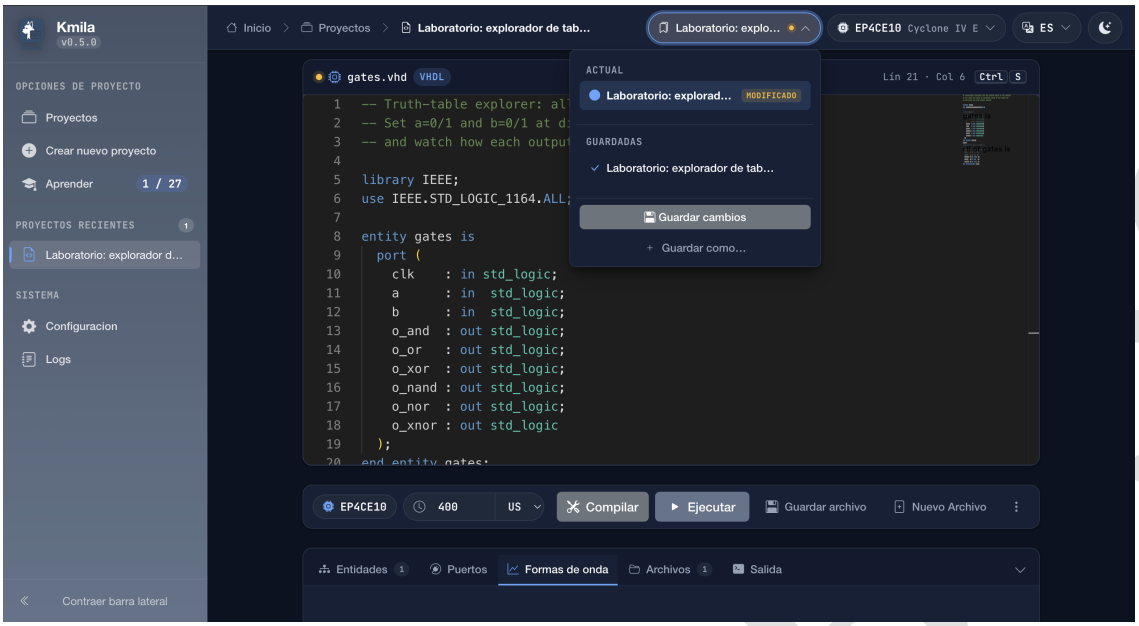


Figura 5.97: *Selector de proyecto activo con submenú de guardado. La sección* ACTUAL *marca el proyecto en edición con la etiqueta* MODIFICADO *cuando contiene cambios sin guardar; la sección* GUARDADAS *lista las versiones persistidas. Los botones* Guardar cambios *y* Guardar como... *cierran el flujo de versionado manual.*

5.9.5. Panel de Diagnóstico y Errores

El diagnóstico se presenta en dos niveles de detalle. La pestaña *Salida* del panel inferior del editor (Figura 5.98) muestra el registro cronológico del ciclo actual de compilación y simulación, con códigos de color por severidad. Para un análisis más profundo, la aplicación expone un visor de logs dedicado (Figura 5.99) accesible desde el menú *Sistema* de la barra lateral, que agrega todos los eventos del ciclo de vida de la aplicación.

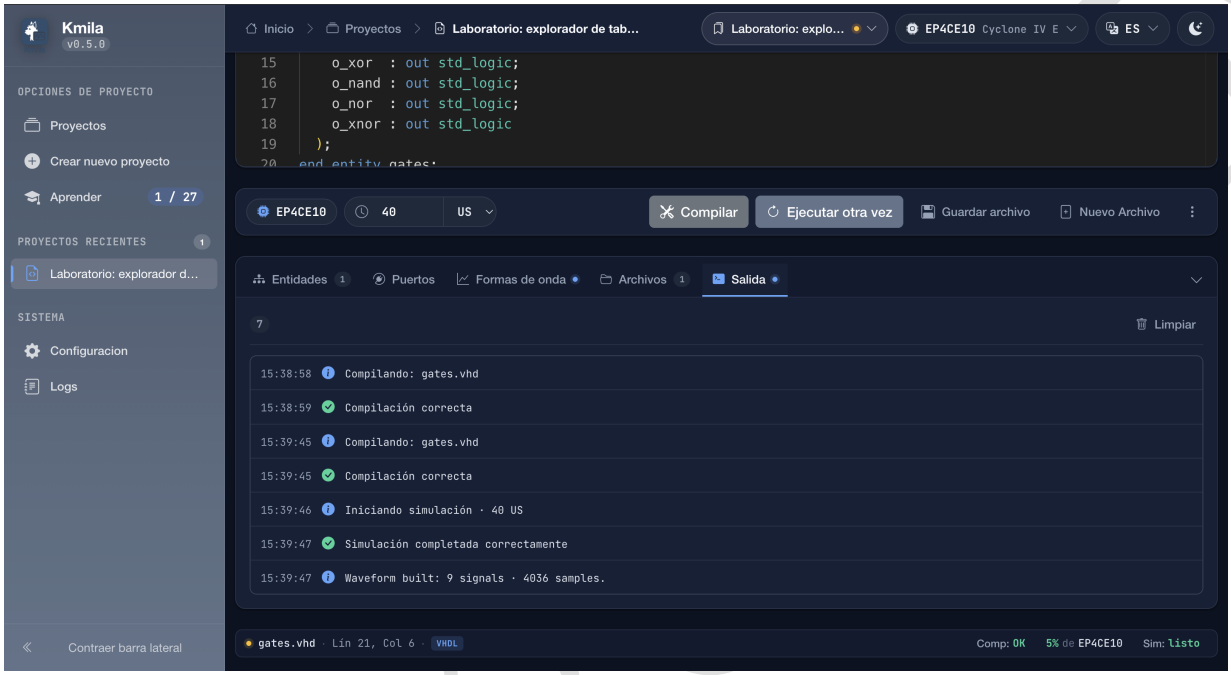


Figura 5.98: *Pestaña* Salida *del editor: log cronológico de la compilación y ejecución del proyecto activo, con indicadores de éxito (verde), información (azul) y advertencias. El botón* Limpiar *reinicia el panel.*

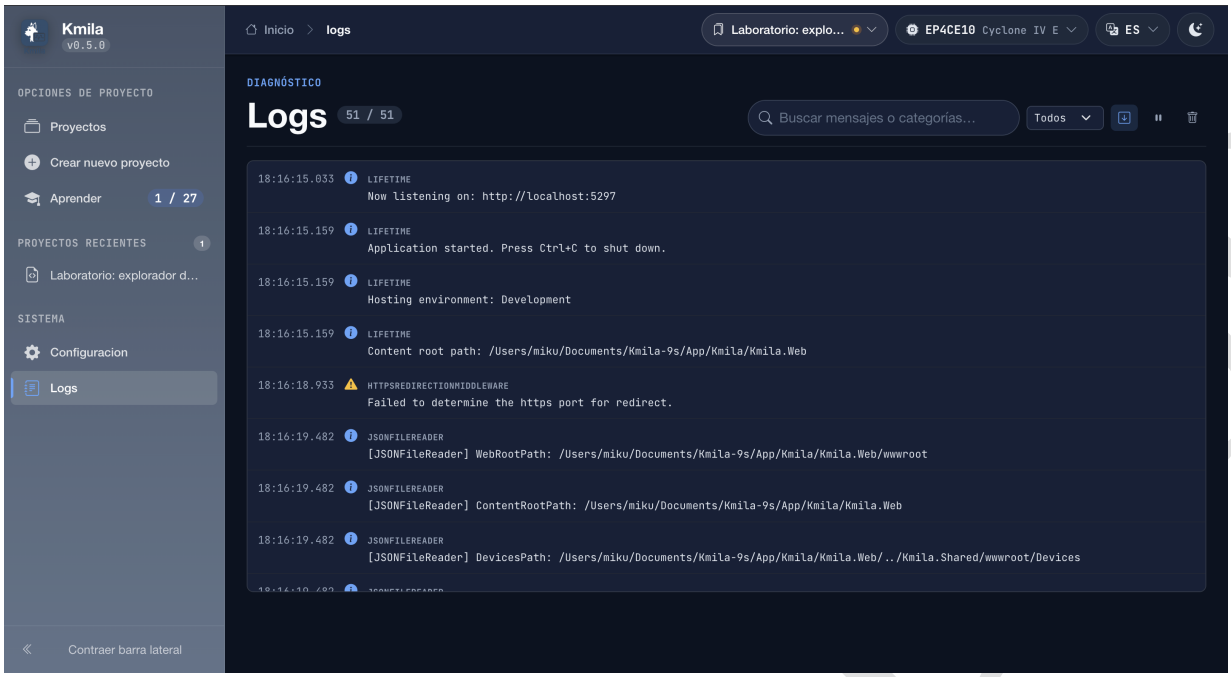


Figura 5.99: *Visor de logs del sistema: concentra 51 eventos del ciclo de vida de la aplicación. Cada entrada incluye hora, categoría (LIFETIME, JSONFILEREADER, HTTPSREDIRECTIONMIDDLEWARE, etc.), mensaje y nivel de severidad. Incluye búsqueda por texto, filtro por categoría y controles de exportación.*

5.9.6. Selector de Dispositivos FPGA

El selector de dispositivos FPGA, localizado en la barra superior del editor, despliega el catálogo completo agrupado por familia del fabricante seleccionado. La Figura 5.100 muestra el menú desplegado con los nueve modelos Intel (Altera) Cyclone IV E disponibles. El cambio de dispositivo recalcula automáticamente la estimación de utilización de recursos (Figura 5.93).

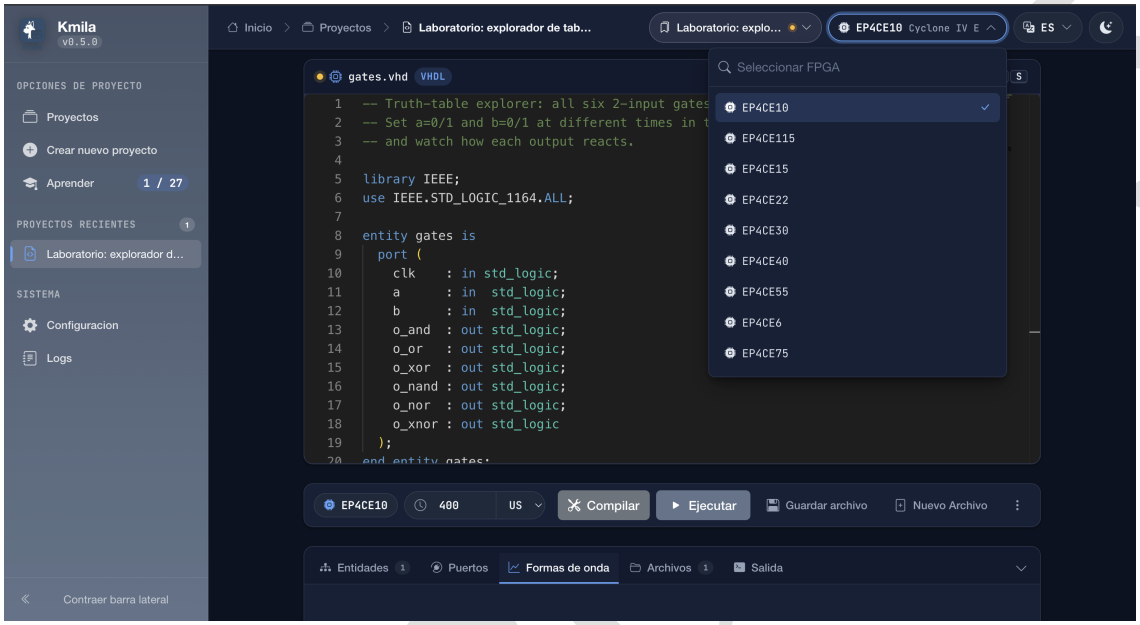


Figura 5.100: *Selector de dispositivos FPGA en la barra superior del editor, mostrando los nueve modelos Intel (Altera) Cyclone IV E del catálogo: EP4CE6, EP4CE10 (seleccionado), EP4CE15, EP4CE22, EP4CE30, EP4CE40, EP4CE55, EP4CE75 y EP4CE115. Los 34 dispositivos del catálogo completo se distribuyen entre tres fabricantes según la Sección 2.3.2.*

Capítulo 6

Implementación, Resultados y Conclusiones

Este capítulo cierra el documento de Trabajo Terminal con la documentación de lo efectivamente construido durante TT2, las pruebas y validación realizadas, los resultados obtenidos respecto a los objetivos planteados en la Sección 1.2, las conclusiones del trabajo y las líneas de investigación derivadas. A diferencia del Capítulo 5, que describe el diseño *proyectado* del sistema, este capítulo documenta el sistema *realizado*, incluyendo las desviaciones respecto al diseño original y su justificación técnica.

6.1. Implementación del Sistema

Esta sección documenta el sistema efectivamente construido al cierre de TT2, contrastándolo con el diseño proyectado en el Capítulo 5 y justificando las desviaciones técnicas.

6.1.1. Vista General de Módulos Entregados

El sistema se entrega organizado en siete proyectos `.NET 9` que constituyen el producto en operación, tres herramientas de línea de comandos de apoyo al desarrollo y la validación, y dos andamios Unity reservados para trabajo futuro. La Tabla 6.1 resume cada componente, su tecnología, su estado de madurez y su responsabilidad.

Tabla 6.1: *Módulos entregados al cierre de TT2.*

Módulo	Tecnología	Estado	Responsabilidad
`App/Kmila`	.NET 9, MAUI + Blazor	Productivo	Aplicación: `Kmila.Shared` (páginas, componentes y servicios), `Kmila` (host MAUI nativo) y `Kmila.Web` (host Blazor Server + API local `/api/v1`).
`Parser`	C# / .NET 9	Productivo	Análisis léxico y sintáctico de VHDL; produce el diccionario de *scopes*.
`Interpreter`	C# / .NET 9	Productivo	Motor de ejecución (`DeltaCycleEngine`, planificadores de señal y proceso, `SignalHistory`) y ruta de síntesis a *netlist*.
`Debugger`	C# / .NET 9	Productivo	Capa de pre-procesamiento: *lint* y pase estructural que hidrata los tipos del `Interpreter`.
`TimeMachine`	C# / .NET 9	Productivo	Fuente de tiempo discreto (10 ns por *tick*) y control de ejecución.
`LibraryCompiler`	C# / .NET 9	Productivo	Resolución y compilación de librerías `IEEE` (`std_logic_1164`, `numeric_std`) con caché por *hash*.
`PracticeValidator`	C# / .NET 9 (CLI)	Herramienta	Califica sin conexión los ejercicios de práctica contra sus casos de prueba.
`RoundTripAudit`	C# / .NET 9 (CLI)	Herramienta	Valida masivamente la ida y vuelta VHDL $\leftrightarrow$ bloques.
`SimulatorBench`	C# / .NET 9 (CLI)	Herramienta	Arnés de exactitud que contrasta Kmila contra GHDL (Sección 6.2.3).
`Simulator` / `KmilaFactorySim`	Unity 6, URP	Andamio	Interfaces 3D proyectadas; plantillas sin lógica propia (trabajo futuro).
`Modelos3D`	Recursos	Recurso	Modelos 3D de componentes electrónicos para las interfaces Unity.

Los seis módulos de *backend* (`Parser`, `Interpreter`, `Debugger`, `TimeMachine`, `LibraryCompiler` y la biblioteca compartida `Kmila.Shared`) constituyen el pipeline descrito en el Capítulo 5; los andamios Unity permanecen como plantillas sin implementación y se declaran explícitamente como trabajo futuro para no sobreestimar el alcance efectivamente cubierto.

6.1.2. Estadísticas del Código

La Tabla 6.2 presenta el tamaño del código fuente escrito a mano, medido como número de líneas de los archivos `.cs`, `.razor`, `.ts` y `.js` propios, excluyendo artefactos de compilación (`obj/`, `bin/`), dependencias de terceros (`node_modules/`, `wwwroot/lib/`) y archivos minificados. El total asciende a aproximadamente **129 500 líneas**, concentradas en la aplicación (`App/Kmila`) y el motor de ejecución (`Interpreter`).

Tabla 6.2: *Líneas de código por módulo (código propio, sin artefactos ni dependencias).*

Módulo	**Líneas de código**
`App/Kmila` (Kmila.Shared + Kmila + Kmila.Web)	69 612
`Interpreter`	35 076
`Tests` (Playwright + xUnit + rendimiento)	12 434
`Parser`	6 957
`SimulatorBench`	2 312
`LibraryCompiler`	1 152
`TimeMachine`	811
`Debugger`	438
`PracticeValidator`	398
`RoundTripAudit`	319
Total	≈ **129 509**

La cobertura de pruebas creció de forma marcada respecto a la línea base de TT1, que reportaba 230 pruebas concentradas en el *backend* (`Parser`, `Interpreter`, `Debugger`, `TimeMachine`). Al cierre de TT2 el sistema cuenta con varias superficies de prueba complementarias, detalladas en la Sección 6.2: **170** pruebas unitarias `xUnit` (`[Fact]`/`[Theory]`), un arnés propio de pruebas del motor, un corpus de **54** archivos VHDL de exploración, **77** especificaciones de extremo a extremo en Playwright y un banco de exactitud de **148** casos contrastados contra GHDL.

6.1.3. Desviaciones Respecto al Diseño Original

Durante TT2 el sistema construido creció por encima del diseño proyectado en el Capítulo 5 en tres frentes principales, todos ellos aditivos sobre el motor de simulación y justificados por su valor pedagógico:

1. **Visual Runtime (Sección 5.6.3).** No previsto en el diseño original, este modo de simulación interactiva enlaza los puertos del diseño a componentes virtuales sobre un lienzo (LEDs, botones, interruptores, siete segmentos, motor paso a paso) y los ejecuta en corrida infinita. La desviación se justifica porque materializa la experiencia "sin hardware físico" que motiva todo el proyecto, y se implementó reutilizando el `DeltaCycleEngine` existente en lugar de un segundo motor.
2. **Depuración en vivo (Sección 5.6.4).** El diseño contemplaba únicamente la *reproducción* paso a paso de una corrida ya finalizada (RF-21). La implementación amplió esa capacidad a una depuración *en vivo* sobre la simulación en curso —*breakpoints* de línea, de condición y de tiempo, avance por ciclo delta y recorrido sobre el esquemático—, cerrando el hueco del resaltado de líneas vacío de la fase inicial.
3. **Edición por bloques de propósito general (Sección 5.6.5).** El diseño sólo preveía bloques como mecanismo de ejercicio dentro de *Concepts*. La implementación añadió una pestaña *Blocks* en el editor real que traduce en ambos sentidos entre VHDL y bloques, convirtiendo un recurso didáctico acotado en una herramienta de autoría.

A estas se suman las desviaciones ya anticipadas al cierre de TT1: el propio módulo *Concepts* con `ExecutionPlayer` y editor de bloques; la cascada de tres niveles de fuentes del `LibraryCompiler`; y el *dock* multi-pestaña del editor, más rico que el *mockup* original.

6.1.4. Decisiones Técnicas no Previstas

Varias decisiones técnicas surgieron durante la implementación sin estar contempladas en el diseño:

- **Sustitución de Blockly por un motor propio.** El editor de bloques usó inicialmente *Blockly* v11 empaquetado sin conexión, pero su envoltura UMD entraba en conflicto con el cargador AMD (`define()`) de Monaco, lo que ocasionaba que el editor de código quedara vacío de forma intermitente. La solución definitiva fue reemplazar Blockly por un motor propio basado en el DOM (`kmila-blocks.js`) que *conserva el formato de serialización de Blockly* y los nombres de *interop*; así se eliminó el conflicto y $\sim$1 MB de dependencia sin que ninguna pieza posterior (el *fingerprint* de `BlockProgram`, los catálogos de *Concepts*) tuviera que cambiar.
- **Reutilización del motor para los modos interactivos.** Tanto el *Visual Runtime* como la depuración en vivo se construyeron sobre el `DeltaCycleEngine` y el `ReplayState` existentes en lugar de motores paralelos. El *seam* del *Visual Runtime* descubrió que la escritura de puertos entre ticks podía viajar por la vía de `Variable.OnValueChange` ya presente, evitando por completo un *hook* nuevo en el motor.
- Decisiones menores previas: caché del `LibraryCompiler` por *hash* SHA-256 del fuente; uso de `TimelineRunner` en C# puro (bucle `Task.Delay` cancelable) en lugar de temporizadores JavaScript para el `ExecutionPlayer`; y codificación *base-94* de identificadores en el exportador VCD.

6.2. Pruebas y Validación

Esta sección documenta el conjunto de pruebas aplicadas al sistema y los resultados de la validación frente a los requerimientos funcionales y no funcionales establecidos en el Capítulo 4.

6.2.1. Pruebas Unitarias Automatizadas

La verificación unitaria se apoya en tres superficies complementarias:

- **Suite** `xUnit` (`dotnet test`). Reúne **170** pruebas anotadas con `[Fact]`/`[Theory]`, alojadas principalmente en `Tests/Unit/Kmila.Shared.Tests` (158) y en `Parser` (7). Cubren el constructor y el emisor del editor de bloques (`VhdlBlockBuilder`, `VhdlBlockEmitter`), la fidelidad de la ida y vuelta (`RoundTripFidelity`), los atributos de señal, el manejo de rebanadas (*slices*), los *frames* de proceso y la equivalencia entre los modos de ejecución rápido y visual (`FastRunnerEquivalence`). Es la superficie de regresión autoritativa.
- **Arnés de pruebas del motor.** El proyecto `Interpreter` incluye un conjunto de clases de prueba (`DeltaCycleEngineTest` `SynthesisTests`, `IEEEPrimitivesTests`, `IEEELoweringEndToEndTests`, `NestedIEEECallTests`, `PortStrictnessTests`, `LibraryCompilerTests`, `PracticeExerciseTests`) que se ejecutan mediante el comando `dotnet run - --test-delta`. Verifican la semántica de ciclos delta, el descenso de funciones IEEE a celdas primitivas y la topología de las llamadas anidadas.
- **Corpus de exploración. 54** archivos `test*.vhdl` que el ejecutable del `Interpreter` procesa por lotes como prueba de humo sobre construcciones VHDL de dificultad creciente.

Estas pruebas se ejecutan de forma rutinaria durante el desarrollo antes de integrar cada cambio, y constituyen la red de seguridad que —conforme a la metodología XP adoptada (Sección 3.5.1)— habilita las refactorizaciones agresivas del motor realizadas durante TT2.

6.2.2. Pruebas de Integración

La interacción entre módulos se ejerce de extremo a extremo por dos mecanismos automatizados:

- **Pipeline completo vía el API local.** El banco de exactitud `SimulatorBench` envía cada diseño a la aplicación a través del *endpoint* `POST /api/v1/simulate`, que recorre la cadena real `Parser` → `Interpreter` (`DeltaCycleEngine`) → exportador VCD sin arneses artificiales. De este modo, los 148 casos del corpus (Sección 6.2.3) son, además de una prueba de exactitud, una prueba de integración del pipeline completo sobre compuertas, decodificadores, multiplexores, sumadores, contadores, máquinas de estado, registros de desplazamiento y funciones IEEE.
- **Ida y vuelta VHDL ↔ bloques.** La herramienta `RoundTripAudit` recorre el corpus de diseños convirtiendo cada archivo VHDL a bloques y de vuelta a VHDL, verificando la idempotencia a nivel de bytes y la ausencia de pérdidas silenciosas. Esta prueba valida la consistencia del editor de bloques con el editor de código, ambos alojados en `Kmila.Shared`.

La persistencia se apoya en el almacén local `SQLite` descrito en el Capítulo 5; el ciclo de guardar un proyecto, cerrar la aplicación y reabrirla conservando la estructura de archivos y la configuración se ejerce de forma continua durante el uso y en las pruebas de extremo a extremo de Playwright (Sección 6.2.4).

6.2.3. Validación Funcional contra GHDL

La prueba más exigente del sistema consiste en verificar que el motor de simulación propio produce los mismos resultados que un simulador VHDL de referencia. Para ello se contrasta Kmila contra **GHDL** [6], el simulador VHDL libre de referencia en la industria y la academia, mediante el banco automatizado `SimulatorBench`.

Metodología

Cada caso del corpus contiene, de forma *deliberadamente separada*, dos fuentes VHDL independientes: un archivo `kmila.vhdl`, implementación del problema escrita a partir de la especificación y ejecutada por **Kmila**; y una carpeta `reference/` con el diseño y su *testbench* de fuente pública, ejecutados por **GHDL**. El banco genera el VCD dorado con GHDL, obtiene el VCD de Kmila a través del API `/api/v1/simulate`, y compara ambas trazas con un *diferenciador semántico* (`VcdParser` + `SemanticDiffer`). La Tabla 6.3 resume el flujo.

Tabla 6.3: *Pipeline de validación funcional Kmila frente a GHDL.*

Etapa	Descripción
Referencia (GHDL)	`reference/*.vhdl` → `ghdl -a/-e/-r --vcd` → `golden.vcd`.
Bajo prueba (Kmila)	`kmila.vhdl` → `POST /api/v1/simulate` → `kmila.vcd`.
Normalización	`VcdParser` normaliza ambas trazas a femtosegundos y aplana la jerarquía de *scopes*.
Comparación	`SemanticDiffer` contrasta los puertos de salida y registra coincidencias y discrepancias en `diff.json`.

Alcance de la equivalencia

Es importante declarar con precisión qué afirma esta validación. El diferenciador compara *únicamente los puertos declarados de salida* (`out`/`inout`); las entradas y los relojes constituyen el estímulo y no se comparan. Ambas trazas se cuantizan a una resolución de **1 ns** antes de contrastarse, lo que absorbe tanto las diferencias de sub-nanosegundo por ciclos delta como la distinta escala temporal de los volcados —GHDL emite con *timescale* de 1 fs y Kmila de 1 ps—. Los metavalores `x`, `u` y `z` se tratan como comodines, y se omite la muestra en $t = 0$ porque GHDL vuelca tras la inicialización mientras que Kmila vuelca antes del primer delta. La afirmación validada es, por tanto, la **equivalencia de los puertos de salida a resolución de 1 ns, tolerante a metavalores y omitiendo el instante inicial**: el criterio adecuado para responder si el sistema se comporta funcionalmente igual que el simulador de referencia, que es lo relevante desde el punto de vista pedagógico.

Resultados

Sobre un corpus de **148 casos**, la salida de Kmila coincide con la de GHDL en el **100 % de los casos (148/148)**, sin una sola discrepancia en un total de **2 454 muestras** de puertos de salida comparadas. El corpus cubre lógica combinacional (compuertas, decodificadores, codificadores, multiplexores), aritmética (sumadores, restadores, comparadores, multiplicación `numeric_std`), elementos secuenciales (*flip-flops*, *latches*), contadores (binario, en anillo, Johnson, ascendente/descendente, LFSR), registros de desplazamiento, máquinas de estado Moore y Mealy, decodificadores de siete segmentos, memorias ROM/arreglos, elaboración jerárquica con `generic map`, funciones y procedimientos de usuario, y construcciones de VHDL-2008 (`case?`, atributos de señal). La Tabla 6.4 muestra una **selección representativa** con el número de muestras comparadas por caso; el corpus completo, junto con la procedencia y la licencia de cada uno de los 148 casos —de los cuales el 68 % proviene de repositorios públicos e independientes de terceros— se detalla en el Sección A.5, de modo que la evaluación es reproducible y no depende de casos ajustados al sistema.

Tabla 6.4: *Selección representativa de la validación Kmila frente a GHDL (muestras de puertos de salida comparadas / discrepancias).*

Caso	Categoría	Muestras / Discrep.
`001-and-gate`	Combinacional (AND)	2 / 0
`002-decoder-3to8`	Decodificador	8 / 0
`010-mux-8to1`	Multiplexor	510 / 0
`003-counter-8bit`	Contador (bus)	10 / 0
`029-lfsr`	LFSR	40 / 0
`086-sec-moore`	FSM Moore	5 / 0
`092-sec-mealy`	FSM Mealy	5 / 0
`100-traffic`	FSM (semáforo)	27 / 0
`114-seven-seg`	Display 7 segmentos	2 / 0
`027-add-sub`	Aritmética `numeric_std`	11 / 0
`147-case-matching`	VHDL-2008 `case?`	5 / 0
`148-signal-event-in-wait`	Atributo `'event`	10 / 0
Corpus completo	**148 casos**	**2 454 / 0**

La Figura 6.1 y la Figura 6.2 contrastan visualmente las trazas de Kmila y de GHDL para una compuerta combinacional y un contador de 8 bits, renderizadas directamente desde los VCD reales; en ambas la forma de onda de Kmila reproduce la de GHDL.

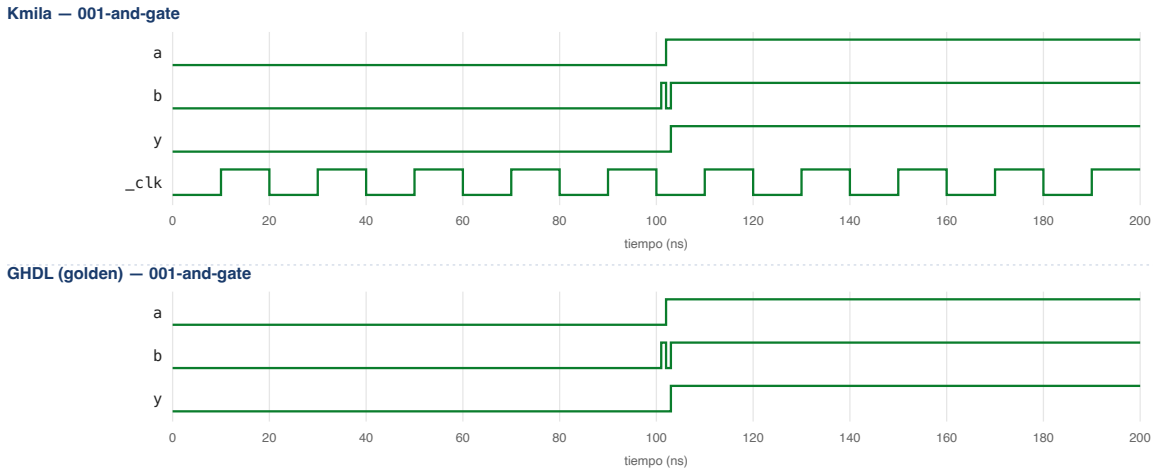


Figura 6.1: *Validación del caso `001-and-gate`: trazas de Kmila (arriba) y GHDL (abajo) para la compuerta AND, renderizadas desde los VCD. La salida `y` sube cuando `a` y `b` valen 1 en ambos motores.*

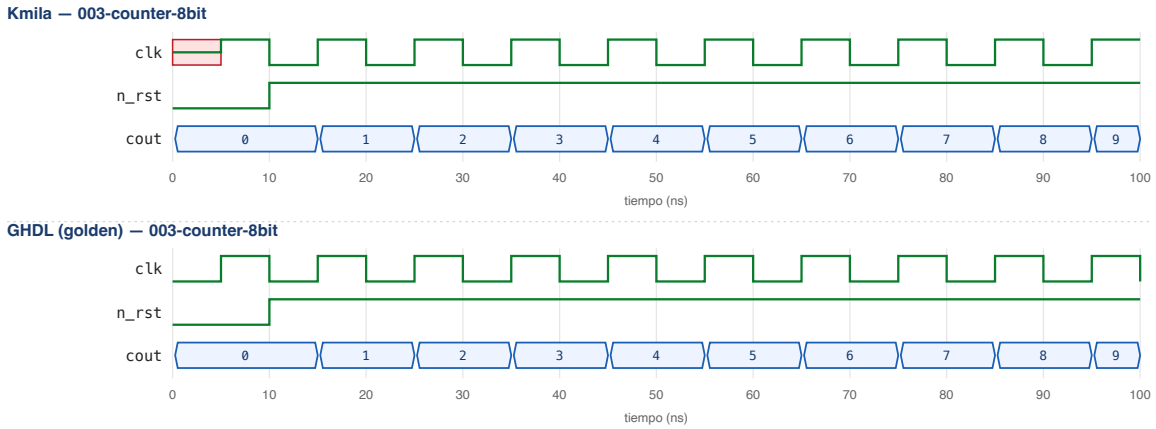


Figura 6.2: *Validación del caso `003-counter-8bit`: el bus `cout` incrementa idénticamente en Kmila y en GHDL.*

Sobre el visor de formas de onda

El VCD es un formato estándar (parte de IEEE 1076 [10]) y puede inspeccionarse en cualquier visor. La herramienta clásica del ecosistema libre es GTKWave [7]; sin embargo, su distribución vigente está descontinuada y resultó incompatible con el entorno de desarrollo empleado (macOS sobre Apple Silicon), donde falla al iniciar. Para la inspección visual se adoptó **Surfer**, un visor moderno, nativo para Apple Silicon y de mantenimiento activo, que lee VCD sin dificultad. La Figura 6.3 muestra el VCD del contador de Kmila abierto en Surfer. De los 148 VCD de Kmila, únicamente el del caso `100-traffic` desencadena un fallo del analizador interno de Surfer —un defecto del visor, no de Kmila, pues el mismo archivo es procesado sin problema por GHDL, por el diferenciador del banco (que lo marca como aprobado) y por el renderizador propio empleado en las figuras anteriores.

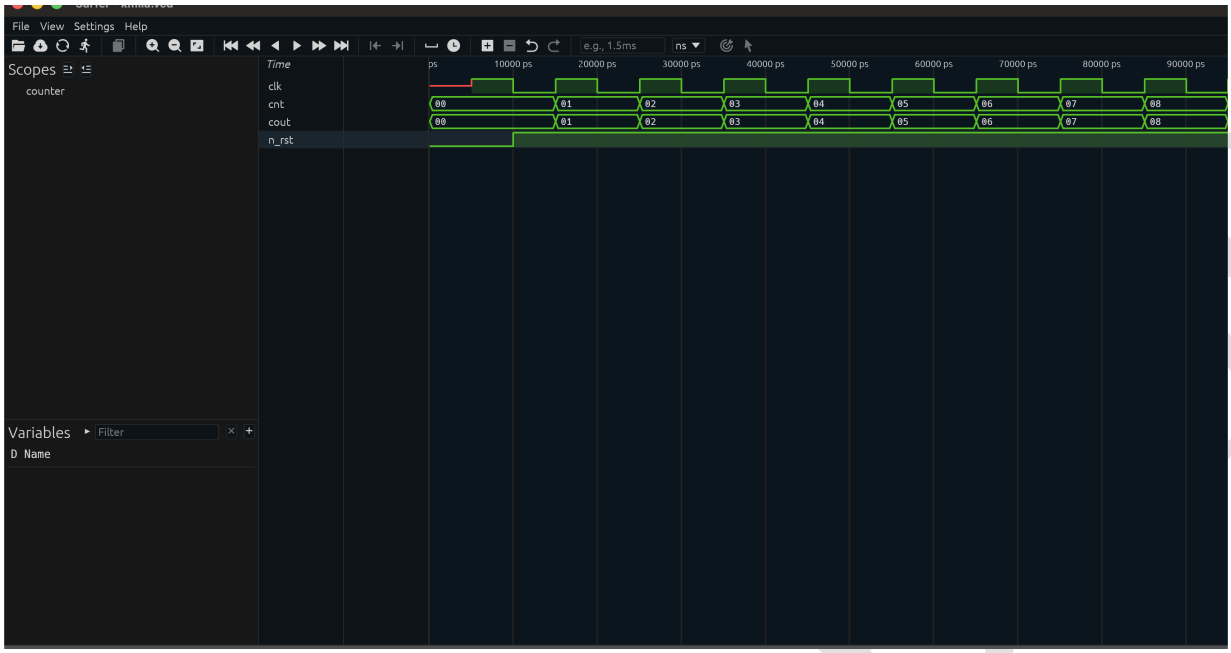


Figura 6.3: *VCD del contador de 8 bits generado por Kmila, abierto en el visor Surfer: los buses* `cnt` *y* `cout` *incrementan por cada flanco de reloj.*

6.2.4. Pruebas Multiplataforma

La cobertura multiplataforma se verificó con distinto grado de formalidad según la plataforma, complementando la evidencia visual recogida durante TT1 (Sección 3.2.5). La Tabla 6.5 resume el estado; se declaran explícitamente las plataformas cuya verificación no fue exhaustiva para no sobreestimar la cobertura.

Tabla 6.5: *Matriz de verificación funcional por plataforma.*

Plataforma	**Verificación**	**Alcance**
Web (Blazor, Linux/escritorio)	Automatizada	Suite de extremo a extremo en Playwright (77 especificaciones) sobre Chromium: navegación, gestión de proyectos, editor, pipeline de compilación y ejecución, formas de onda, esquemático, *runtime* visual y subsistemas pedagógicos.
Android	En dispositivo real	Auditoría de UI/UX sobre Xiaomi 13T Pro (Android 15) con verificación por CDP; corrección de defectos de interfaz en dispositivo (véase la auditoría móvil del proyecto).
macOS (Mac Catalyst)	Manual	Ejecución del *host* MAUI y del *host* web; base de la evidencia visual de TT1.
iOS	Parcial	*Build* y API `/api/v1` habilitados (opt-in); verificación funcional exhaustiva pendiente.
Windows	Pendiente	Objetivo *TFM* declarado; verificación funcional no realizada en este periodo.

La superficie de ejecución compartida (`Kmila.Shared`) es común a todas las plataformas, por lo que la validación funcional del motor (Sección 6.2.3) es independiente del *host* y aplica por igual a todas ellas; la matriz anterior verifica la capa de interfaz y de integración específica de cada plataforma.

continúa en la página siguiente

ID	Nombre	Estado	Evidencia

6.2.5. Cumplimiento de Requerimientos Funcionales

La Tabla 6.6 traza las **39** requerimientos funcionales declarados en el Capítulo 4 (RF-01 a RF-39) contra su estado de cumplimiento y la evidencia que lo respalda. De los 39, **37 se cumplen** y **2 se cumplen parcialmente** (con los pendientes documentados como trabajo futuro en la Sección 6.5.1); ninguno queda sin cumplir. Los requerimientos del motor de simulación se respaldan con la validación contra GHDL (Sección 6.2.3); los de interfaz y gestión, con la suite de extremo a extremo (Sección 6.2.4) y la evidencia visual del Anexo de capturas.

Tabla 6.6: *Trazabilidad de cumplimiento de los requerimientos funcionales.*

ID	Nombre	Estado	Evidencia
RF-01	Gestionar proyectos	Cumplido	Playwright; Sección 6.2.4
RF-02	Gestionar archivos HDL	Cumplido	Playwright; persistencia SQLite
RF-03	Editar código HDL	Cumplido	Editor Monaco
RF-04	Navegar estructura del proyecto	Cumplido	Panel de archivos del *dock*
RF-05	Seleccionar dispositivo FPGA	Cumplido	Catálogo de 34 dispositivos
RF-06	Configurar parámetros de simulación	Cumplido	`RunBar` (duración, reloj)
RF-07	Visualizar diagnósticos y recursos	Cumplido	`Debugger` (*lint*); `Resources` (estimación FPGA)
RF-08	Visualizar formas de onda	Cumplido	`WaveformViewer`; Sección 6.2.3
RF-09	Cambiar idioma de la interfaz	Cumplido	Siete idiomas (RNF-05)
RF-10	Cambiar tema visual	Cumplido	*Settings*
RF-11	Persistir datos localmente	Cumplido	Almacén SQLite
RF-12	Analizar sintaxis VHDL	Cumplido	`Parser`; 148/148 casos del banco
RF-13	Generar representación intermedia	Cumplido	Diccionario de *scopes*
RF-14	Ejecutar simulación	Cumplido	`DeltaCycleEngine`; 148/148 vs GHDL
RF-15	Controlar simulación	Cumplido	Pausar/Reanudar/Detener
RF-16	Ejecutar paso a paso	Cumplido	Depuración en vivo / *replay*

continúa en la página siguiente

ID	Nombre	Estado	Evidencia
RF-17	Insertar puntos de interrupción	Cumplido	*Breakpoints* de línea/condición/tiempo
RF-18	Exportar resultados a VCD	Cumplido	`SignalHistory.ExportToVcd`; usado por el banco
RF-19	Visualizar esquemático del *netlist*	Cumplido	Ruta de síntesis; Sección 6.3.1
RF-20	Visualizar diagrama de flujo	Cumplido	`PlantUmlFlowEmitter`
RF-21	Reproducir simulación paso a paso	Cumplido	`ReplayState`
RF-22	Comparar corridas de simulación	Cumplido	Modo *compare* del `WaveformViewer`
RF-23	Persistir *snapshots* de simulación	Parcial	Persistencia básica; entidades avanzadas diferidas (Sección 6.5.1)
RF-24	Aprender VHDL con lecciones estructuradas	Cumplido	Subsistema *Learn*
RF-25	Practicar con evaluación automática	Cumplido	Subsistema *Practice* + calificador
RF-26	Comprender modelos de ejecución	Cumplido	*Concepts* + `ExecutionPlayer`
RF-27	Construir programas con bloques visuales	Cumplido	Editor de bloques
RF-28	Persistir progreso pedagógico	Cumplido	Almacenamiento local
RF-29	Reportar errores automáticamente	Cumplido	Reporte con OAuth de GitHub
RF-30	Exportar y empaquetar proyectos	Cumplido	Exportación de *bundles*
RF-31	Gestionar *assets* de usuario	Cumplido	`UserAssetStore`
RF-32	Sincronizar vistas del *workbench*	Cumplido	*Dock* multi-pestaña
RF-33	Compilar librerías estándar VHDL	Cumplido	`LibraryCompiler` (`std_logic_1164`, `numeric_std`)
RF-34	Ejecutar en modo visual interactivo	Parcial	*Visual Runtime* operativo; *fan-in* por bit diferido (Sección 6.5.1)
RF-35	Persistir el lienzo visual	Cumplido	Archivo `.kviz`

continúa en la página siguiente

ID	Nombre	Estado	Evidencia
RF-36	Depurar la simulación en vivo	Cumplido	Depuración en vivo sobre el motor
RF-37	Recorrer la ejecución sobre el esquemático	Cumplido	Modo Debug del esquemático
RF-38	Editar código mediante bloques	Cumplido	Pestaña *Blocks*; `RoundTripAudit`
RF-39	Inspeccionar el visor de ondas	Cumplido	`WaveformViewer` (SVG en línea)

6.2.6. Cumplimiento de Requerimientos No Funcionales

La Tabla 6.7 verifica los requerimientos no funcionales definidos en el Capítulo 4 (tomando su numeración como canónica). Varios cuentan con evidencia cuantitativa recogida durante TT1 y reutilizada aquí.

Tabla 6.7: *Verificación de los requerimientos no funcionales.*

ID	Categoría	Verificación
RNF-01	Multiplataforma	Cobertura Web/Android/macOS verificada (Sección 6.2.4); iOS/Windows parciales.
RNF-02	Usabilidad	Interfaz unificada con *dock* y *runbar*; auditorías de UI en dispositivo real (sin estudio formal SUS).
RNF-03	Rendimiento	LCP de 45 ms frente al umbral de 200 ms (Figura 3.4); tiempos de simulación en Sección 6.3.1.
RNF-04	Portabilidad web	*Host* Blazor Server operativo en Linux; disponibilidad sin conexión (Figura 3.3).
RNF-05	Internacionalización	Siete idiomas (incluye árabe con RTL); cadenas nuevas de TT2 en ES/EN, resto con *fallback* a inglés (Sección 6.5.1).
RNF-06	Confiabilidad	148/148 casos reproducibles contra GHDL (Sección 6.2.3); 170 pruebas `xUnit`.
RNF-07	Mantenibilidad	Arquitectura modular (siete proyectos); red de pruebas de regresión.
RNF-08	Extensibilidad	Registro extensible de primitivas IEEE; catálogo de *widgets* del *Visual Runtime*.
RNF-09	Persistencia	Almacén SQLite + sidecar `.kviz`; entidades avanzadas diferidas.
RNF-10	Compatibilidad	Distribución de 43–147 MB frente a decenas de GB de las herramientas comerciales (Sección 3.2.4); huella de memoria 124–947 MB según plataforma (Sección 3.2.6).

Quedan como verificación pendiente los aspectos que requieren instrumentos formales fuera del alcance de este trabajo: una evaluación de usabilidad con usuarios (SUS) y una auditoría de accesibilidad (WCAG).

6.3. Resultados

6.3.1. Métricas de Desempeño

El desempeño del motor de simulación se caracterizó con el mismo corpus de 148 casos empleado en la validación. Ejecutando el pipeline completo (análisis, simulación y exportación VCD a través del API) sobre los 148 diseños, el tiempo de pared total fue de **12.1 s**; por caso, la mediana fue de **55 ms**, el percentil 95 de **271 ms** y el máximo de **716 ms**. Estos valores confirman que la simulación de diseños de tamaño didáctico es interactiva (por debajo del umbral de respuesta perceptible en la gran mayoría de los casos).

Para corridas de tiempo simulado largo, el motor ofrece dos modos de ejecución sobre el mismo `DeltaCycleEngine`: un **modo visual** que visita cada *tick* —útil para la depuración en vivo y el *Visual Runtime*— y un **modo rápido** que salta a los *ticks* donde puede ocurrir un evento (flanco de reloj, estímulo, actualización agendada o despertar de proceso), colapsando corridas de escala de segundos (10^8 *ticks* a 10 ns) al número de eventos reales. Ambos modos son intercambiables a través de la interfaz `ISimulationRunner` y se verifican byte a byte como equivalentes mediante comparación de VCD, de modo que la aceleración no altera el resultado.

En cuanto a la comparación con la combinación de referencia abierta (GHDL + un visor de ondas), Kmila no busca competir en velocidad bruta de simulación —GHDL compila a nativo— sino en integración: reúne en una sola aplicación multiplataforma el editor, el motor, la visualización de ondas y de esquemático, y los subsistemas pedagógicos, sin instalación de *toolchains* ni configuración. La equivalencia funcional de resultados frente a GHDL (Sección 6.2.3) es precisamente lo que habilita esa integración sin sacrificar corrección.

6.3.2. Comparativa Final con Herramientas Existentes

La Tabla 6.8 refina la comparación de la Sección 1.4.6 con las cifras finales de TT2, contrastando Kmila con la combinación de referencia abierta (GHDL más un visor de ondas) y con las suites comerciales (Vivado, Quartus). La ventaja documentada en TT1 en cobertura multiplataforma, peso de distribución e integración se confirma, y se añaden las capacidades incorporadas durante TT2.

Tabla 6.8: *Comparativa final de Kmila frente a alternativas.*

Dimensión	**Kmila**	**GHDL + visor**	**Vivado / Quartus**
Costo	Libre	Libre	Libre/comercial
Plataformas nativas	6 (objetivo)	Escritorio	Escritorio
Instalación / *toolchain*	Sin configuración	Manual, por herramienta	Instalador grande
Peso de distribución	43–147 MB	Cientos de MB	Decenas de GB
Entorno integrado (editor+motor+ondas)	Sí	No (piezas separadas)	Sí
Simulación VHDL	Sí (148/148 vs GHDL)	Sí (referencia)	Sí
Visor de ondas / VCD	Sí (integrado)	Sí (GTKWave)	Sí
Esquemático del *netlist*	Sí	No	Sí
Runtime visual interactivo	Sí	No	No
Depuración en vivo sobre el diseño	Sí	Limitada	Parcial
Subsistemas pedagógicos	Sí (Learn/Practice/Concepts)	No	No
Sin requerir hardware físico	Sí	Sí	Sí

La aportación diferencial de Kmila no reside en superar a GHDL en velocidad de simulación —imposible, dado que este compila a código nativo— sino en *empaquetar* una simulación funcionalmente equivalente dentro de un entorno integrado, ligero, multiplataforma y orientado a la enseñanza, eliminando la barrera de instalación y configuración que la combinación GHDL + GTKWave impone al estudiante.

6.3.3. Validación Pedagógica

El protocolo no comprometió una prueba piloto con estudiantes, por lo que —en lugar de resultados de usabilidad medidos con instrumentos formales— esta sección analiza la *cobertura del temario* de las dos asignaturas de diseño digital del plan ISC 2020 de ESCOM: *Fundamentos de Diseño Digital* [70] y *Diseño de Sistemas Digitales* [71]. El análisis se apoya en evidencia concreta: el corpus de 148 diseños validados contra GHDL (Sección 6.2.3) actúa como prueba de que el sistema soporta correctamente las construcciones que cada tema requiere.

La Tabla 6.9 mapea los temas nucleares de ambas asignaturas contra las capacidades demostradas de Kmila.

Tabla 6.9: *Cobertura del temario de diseño digital (ISC 2020) por las capacidades de Kmila.*

Tema	**Soporte en Kmila (evidencia)**
Álgebra booleana y compuertas	Casos combinacionales validados (`and`, `or`, `xor`, De Morgan).
Lógica combinacional	Decodificadores, codificadores, multiplexores y demultiplexores validados.
Aritmética digital	Semi-sumador, sumador completo, sumador/restador y multiplicación `numeric_std` validados.
Lógica secuencial	*Flip-flops* (D, JK) y *latches* validados.
Contadores y registros	Contadores binario, en anillo, Johnson, ascendente/descendente, LFSR y registros de desplazamiento validados.
Máquinas de estado finito	FSM Moore y Mealy, detectores de secuencia y controlador de semáforo validados.
Lenguajes de descripción de hardware (VHDL)	Editor con resaltado, análisis sintáctico, síntesis a *netlist* y edición por bloques bidireccional.
Simulación y verificación	Motor de eventos discretos con formas de onda, exportación VCD y equivalencia verificada contra GHDL.
Dispositivos programables (FPGA)	Catálogo de 34 dispositivos y estimación de recursos (LUT, *flip-flops*, BRAM, DSP).
Modelo de ejecución (concurrencia)	Subsistema *Concepts* con `ExecutionPlayer` dedicado a la ejecución secuencial/concurrente.

El análisis muestra que las capacidades de Kmila cubren los temas nucleares de ambas asignaturas, y que los subsistemas *Learn*, *Practice* y *Concepts* ofrecen, además del simulador, un recorrido didáctico alineado con dicho temario. La evaluación empírica de su efectividad pedagógica con estudiantes queda planteada como línea de trabajo futuro (Sección 6.5.3).

6.4. Conclusiones

6.4.1. Cumplimiento de Objetivos

El **objetivo general** —desarrollar una aplicación multiplataforma, gratuita y de código abierto para la simulación y depuración interactiva de HDL sin hardware especializado— **se cumple**: el sistema entregado ejecuta simulación VHDL funcionalmente equivalente a GHDL (Sección 6.2.3), incorpora depuración en vivo sobre el diseño, y opera sin FPGA en una base de código compartida orientada a seis plataformas.

Los seis **objetivos específicos** declarados en la Sección 1.3 correspondían al alcance de TT1 (análisis, arquitectura, interfaz, selector FPGA, visualizador de ondas y validación web). Todos se cumplieron en TT1 y se consolidaron en TT2; en particular, el quinto objetivo —que preveía un visualizador de ondas *preparado para su integración futura con el motor de simulación*— alcanzó en TT2 esa integración: el motor construido (`DeltaCycleEngine`) alimenta hoy el visualizador con datos de simulación reales. La Tabla 6.10 resume la evidencia.

Tabla 6.10: *Cumplimiento de los objetivos específicos.*

Objetivo	Evidencia	Sección
1. Análisis de requerimientos	39 RF y 10 RNF con trazabilidad de cumplimiento	Sección 6.2.5
2. Arquitectura modular	Siete proyectos .NET 9; pipeline verificado de extremo a extremo	Sección 6.1.1
3. Interfaz gráfica unificada	Editor Monaco, gestión de proyectos y *dock*; verificación E2E	Sección 6.2.4
4. Selector de dispositivos FPGA	Catálogo de 34 modelos con recursos de hardware	Sección 6.2.5 (RF-05)
5. Visualizador de formas de onda	Visualizador integrado con el motor; equivalente a GHDL	Sección 6.2.3
6. Validación web (Linux/Blazor)	Suite de 77 especificaciones E2E sobre el *host* Blazor	Sección 6.2.4

Además del cumplimiento de los objetivos, TT2 los superó con tres capacidades no comprometidas originalmente —el *Visual Runtime*, la depuración en vivo y la edición por bloques de propósito general— documentadas como desviaciones aditivas en la Sección 6.1.3.

6.4.2. Aportaciones del Trabajo

El trabajo aporta, en síntesis:

1. **Un simulador VHDL propio, integrado y validado.** Un motor de eventos discretos escrito desde cero cuya salida se verificó *funcionalmente equivalente* a GHDL en 148 de 148 casos, empaquetado dentro de una aplicación multiplataforma sin dependencia de *toolchains* externos. Constituye evidencia de que un entorno de simulación de calidad de referencia puede ofrecerse de forma integrada y accesible.
2. **Reducción de la barrera de instalación en dos a tres órdenes de magnitud.** Una distribución de 43–147 MB frente a las decenas de GB de las suites comerciales, ejecutable sin configuración en seis plataformas objetivo desde una única base de código.
3. **Herramientas pedagógicas originales.** El subsistema *Concepts* con su `ExecutionPlayer` para visualizar el modelo de ejecución concurrente de VHDL, el editor de bloques bidireccional VHDL $\leftrightarrow$ bloques, y el *Visual Runtime* interactivo, que materializan el aprendizaje "sin hardware físico" que motiva el proyecto.
4. **Código abierto para la comunidad.** El sistema queda disponible como software libre para estudiantes y docentes de ESCOM y de la comunidad hispanohablante, junto con un banco de exactitud reutilizable (`SimulatorBench`) que documenta y hace reproducible la validación contra GHDL.

6.4.3. Aprendizajes

El desarrollo del proyecto dejó aprendizajes técnicos y de proceso. En lo técnico, destacan la construcción de un analizador y un intérprete de un lenguaje real, la implementación de la semántica de eventos discretos con ciclos delta —cuya sutileza (por ejemplo, el intercambio correcto de dos señales en un mismo delta) sólo se domina al depurar casos concretos contra un simulador de referencia—, y el desarrollo de una arquitectura multiplataforma con .NET 9, MAUI y Blazor desde una base de código compartida. En lo de proceso, la adopción de Extreme Programming (Sección 3.5.1) mostró el valor de una red de pruebas de regresión como habilitador de refactorizaciones agresivas, y la validación sistemática contra una herramienta externa (GHDL) como disciplina para evitar la deriva de correctitud.

> *Esta subsección es de carácter personal y conviene ampliarla al cierre del periodo con la reflexión propia del autor.*

6.5. Trabajo Futuro

6.5.1. Funcionalidades Pendientes

Algunas funcionalidades quedaron identificadas pero fuera del alcance temporal de TT2, y se documentan aquí como trabajo futuro directo (su trazabilidad frente a las RF se recoge en la Sección 6.2.5):

- **Fan-in por bit en el *Visual Runtime*.** Los widgets de entrada impulsan el puerto completo; la escritura por rebanada (que varios widgets controlen distintos bits de un mismo puerto) requiere lógica de superposición en el coordinador y se difirió.
- **Cierre de los hallazgos del traductor de bloques.** Restan de la auditoría F-1–F-19 la conservación de comentarios en el *roundtrip* (F-6), el *slot* nativo de `elsif` (F-8), el analizador de expresiones con precedencia completa (F-9) y —de alto apalancamiento— las pruebas unitarias del constructor y el emisor (F-19).
- **Traducción de las cadenas nuevas de interfaz.** Los textos de los tres subsistemas incorporados en TT2 están autorados en español e inglés; los idiomas restantes (FR, DE, ZH, JA, AR) caen al inglés con el aviso estándar.
- **Capturas del prototipo para el anexo.** La evidencia visual de los subsistemas nuevos (*Visual Runtime*, depuración en vivo, pestaña *Blocks*) queda pendiente de incorporarse al Anexo de capturas.
- Ideas capturadas previamente: verificación formal con PSL e integración con GHDL como *backend* opcional de validación cruzada.

6.5.2. Optimizaciones Identificadas

Durante la implementación se identificaron optimizaciones no abordadas por restricciones de tiempo o alcance:

- **Compilación AOT en la modalidad web.** La huella de memoria de 947 MB medida corresponde a una configuración `Debug`; una compilación AOT y una medición en `Release` deberían reducirla de forma significativa.
- **Persistencia incremental de *snapshots*.** Guardar los estados de simulación de forma incremental reduciría el tiempo de reconstrucción durante la reproducción paso a paso.
- **Paralelización del motor para diseños grandes.** El planificador de procesos podría explotar paralelismo para diseños con muchas señales, más allá del alcance didáctico actual.

6.5.3. Líneas de Investigación Derivadas

El trabajo abre líneas de investigación abordables en proyectos derivados o de posgrado:

- **Estudio empírico de efectividad pedagógica.** Evaluar con estudiantes el impacto de la visualización paso a paso de la ejecución VHDL (subsistema *Concepts*) sobre la comprensión de la concurrencia, mediante un diseño pre/post con grupo de control.
- **Extensión a otros HDL.** Reutilizar la arquitectura modular del `Parser` para incorporar SystemVerilog, ampliando el alcance del entorno.
- **Puente a FPGA físicas de bajo costo.** Integrar cadenas de herramientas abiertas (Yosys, nextpnr) para llevar los diseños validados en Kmila a dispositivos Lattice iCE40, cerrando el ciclo de simulación a implementación.

Referencias

[1] Amazon México. “Tarjeta de desarrollo FPGA Altera Cyclone IV EP4CE6,” visitado feb. de 2026. dirección: https://www.amazon.com.mx/dp/B09MYSM25M

[2] Digilent Inc. “Basys 3 Artix-7 FPGA Trainer Board,” visitado feb. de 2026. dirección: https://digilent.com/shop/basys-3-artix-7-fpga-trainer-board/

[3] Coursera. “FPGA Design for Embedded Systems Specialization,” visitado feb. de 2026. dirección: https://www.coursera.org/specializations/fpga-design

[4] Intel Corporation. “Intel Quartus Prime Design Software,” visitado feb. de 2026. dirección: https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime.html

[5] Advanced Micro Devices. “AMD Vivado Design Suite,” visitado feb. de 2026. dirección: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html

[6] T. Gingold. “GHDL — Open-source analyzer, compiler, simulator and synthesizer for VHDL,” visitado feb. de 2026. dirección: https://ghdl.github.io/ghdl/

[7] T. Bybell. “GTKWave Electronic Waveform Viewer,” visitado feb. de 2026. dirección: https://gtkwave.sourceforge.net/

[8] Siemens EDA. “ModelSim — FPGA Simulation,” visitado feb. de 2026. dirección: https://eda.sw.siemens.com/en-US/ic/modelsim/

[9] Escuela Superior de Cómputo. “Catálogo de Trabajos Terminales,” visitado feb. de 2026. dirección: https://www.escom.ipn.mx/

[10] IEEE, *IEEE Standard VHDL Language Reference Manual*, IEEE, 2009.

[11] P. J. Ashenden, *The Designer’s Guide to VHDL*, 3.ª ed. Burlington, MA, USA: Morgan Kaufmann, 2008.

[12] IEEE, *IEEE Standard for Verilog Hardware Description Language*, IEEE, 2006.

[13] D. E. Thomas y P. R. Moorby, *The Verilog Hardware Description Language*, 5.ª ed. New York, NY, USA: Springer, 2002.

[14] GHDL contributors. “GHDL Documentation: Backends and Features,” visitado feb. de 2026. dirección: https://ghdl.github.io/ghdl/about.html

[15] S. Williams. “Icarus Verilog,” visitado feb. de 2026. dirección: https://github.com/steveicarus/iverilog

[16] W. Snyder. “Verilator: Open-source SystemVerilog simulator and lint system,” visitado feb. de 2026. dirección: https://www.veripool.org/verilator/

[17] C. Wolf. “Yosys Open SYnthesis Suite,” visitado feb. de 2026. dirección: https://github.com/YosysHQ/yosys

[18] Doulos Ltd. “EDA Playground: Online IDE for HDL simulation,” visitado feb. de 2026. dirección: https://www.edaplayground.com/

[19] H. Wong. “HDLBits: Practice digital hardware design,” visitado feb. de 2026. dirección: https://hdlbits.01xz.net/

[20] M. Materzok, “DigitalJS: a Visual Verilog Simulator for Teaching,” en *Proc. 8th Computer Science Education Research Conference (CSERC ’19)*, ACM, 2020. DOI: 10.1145/3375258.3375272

[21] C. Burch, “Logisim: a graphical system for logic circuit design and simulation,” *Journal on Educational Resources in Computing (JERIC)*, vol. 2, n.º 1, págs. 5-16, mar. de 2002. DOI: 10.1145/545197.545199

[22] Logisim Evolution contributors. “Logisim-evolution: Digital logic design tool and simulator,” visitado feb. de 2026. dirección: https://github.com/logisim-evolution/logisim-evolution

[23] Microsoft. “.NET Multi-platform App UI (.NET MAUI) documentation,” visitado feb. de 2026. dirección: https://learn.microsoft.com/en-us/dotnet/maui/

[24] Microsoft. “ASP.NET Core Blazor documentation,” visitado feb. de 2026. dirección: https://learn.microsoft.com/en-us/aspnet/core/blazor/

[25] S. Brown y Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*, 3.ª ed. New York, NY, USA: McGraw-Hill, 2009.

[26] S. A. Edwards, “Design and Verification Languages,” Columbia University, New York, NY, USA, inf. téc., 2004. visitado feb. de 2026. dirección: https://www.cs.columbia.edu/~sedwards/papers/edwards2004design.pdf

[27] IEEE, *IEEE Standard for SystemVerilog — Unified Hardware Design, Specification, and Verification Language*, IEEE, 2018.

[28] H. Foster, “Part 6: The 2022 Wilson Research Group Functional Verification Study,” *Verification Horizons*, nov. de 2022. visitado feb. de 2026. dirección: https://blogs.sw.siemens.com/verificationhorizons/

[29] P. P. Chu, *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version*. Hoboken, NJ, USA: John Wiley & Sons, 2008.

[30] IEEE, *IEEE Standard for Standard SystemC® Language Reference Manual*, IEEE, 2023.

[31] J. Decaluwe. “MyHDL: From Python to Silicon,” visitado abr. de 2026. dirección: https://www.myhdl.org/

[32] Amaranth HDL contributors. “Amaranth HDL: A modern hardware definition language,” visitado abr. de 2026. dirección: https://amaranth-lang.org/

[33] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek y K. Asanović, “Chisel: Constructing Hardware in a Scala Embedded Language,” en *Proc. 49th Annual Design Automation Conference (DAC ’12)*, ACM, 2012, págs. 1212-1221. DOI: 10.1145/2228360.2228584

[34] C. Papon. “SpinalHDL: A Scala based HDL,” visitado abr. de 2026. dirección: https://github.com/SpinalHDL/SpinalHDL

[35] Arvind, “Bluespec: A Language for Hardware Design, Simulation, Synthesis and Verification,” en *Proc. 1st ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2003)*, IEEE, 2003. DOI: 10.1109/MEMCOD.2003.1210092

[36] S. M. Trimberger, "Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology," *Proceedings of the IEEE*, vol. 103, n.º 3, págs. 318-331, 2015. DOI: 10.1109/JPROC.2015.2392104

[37] Xilinx, Inc. "7 Series FPGAs Data Sheet: Overview (DS180)," visitado abr. de 2026. dirección: https://docs.amd.com/v/u/en-US/ds180_7Series_Overview

[38] Advanced Micro Devices. "AMD Completes Acquisition of Xilinx," visitado abr. de 2026. dirección: https://www.amd.com/en/newsroom/press-releases/2022-2-14-amd-completes-acquisition-of-xilinx.html

[39] Intel Corporation. "Cyclone IV Device Handbook," visitado abr. de 2026. dirección: https://www.intel.com/content/www/us/en/docs/programmable/683375/current/device-handbook.html

[40] Intel Corporation. "Intel Completes Acquisition of Altera," visitado abr. de 2026. dirección: https://newsroom.intel.com/news-releases/intel-completes-acquisition-of-altera/

[41] Altera Corporation. "Altera Launches as a Standalone FPGA Company," visitado abr. de 2026. dirección: https://www.altera.com/about/company-overview.html

[42] Lattice Semiconductor. "Lattice Semiconductor Completes Acquisition of SiliconBlue Technologies," visitado abr. de 2026. dirección: https://www.latticesemi.com/About/Newsroom/PressReleases/2011/12/LatticeCompletesAcquisitionofSiliconBlue

[43] Lattice Semiconductor. "iCE40 LP/HX Family Data Sheet," visitado abr. de 2026. dirección: https://www.latticesemi.com/Products/FPGAandCPLD/iCE40

[44] Lattice Semiconductor. "ECP5 and ECP5-5G Family Data Sheet," visitado abr. de 2026. dirección: https://www.latticesemi.com/Products/FPGAandCPLD/ECP5

[45] Microsoft. "C# documentation," visitado mar. de 2026. dirección: https://learn.microsoft.com/en-us/dotnet/csharp/

[46] Microsoft. "What's new in .NET 9," visitado mar. de 2026. dirección: https://learn.microsoft.com/en-us/dotnet/core/whats-new/dotnet-9/overview

[47] D. R. Hipp. "SQLite: A self-contained, serverless, zero-configuration, transactional SQL database engine," visitado mar. de 2026. dirección: https://www.sqlite.org/

[48] F. Krueger. "sqlite-net: Simple, powerful, cross-platform SQLite client and ORM for .NET," visitado mar. de 2026. dirección: https://github.com/praeclarum/sqlite-net

[49] I. Sommerville, *Software Engineering*, 10.ª ed. Boston, MA, USA: Pearson, 2016.

[50] T. Ohno, *Toyota Production System: Beyond Large-Scale Production*. Portland, OR, USA: Productivity Press, 1988.

[51] D. J. Anderson, *Kanban: Successful Evolutionary Change for Your Technology Business*. Sequim, WA, USA: Blue Hole Press, 2010.

[52] H. Kniberg y M. Skarin, *Kanban and Scrum: Making the Most of Both*. Toronto, Canada: C4Media / InfoQ Enterprise Software Development Series, 2010.

[53] K. Beck, *Extreme Programming Explained: Embrace Change*, 1.ª ed. Reading, MA, USA: Addison-Wesley, 1999.

[54] K. Beck y C. Andres, *Extreme Programming Explained: Embrace Change*, 2.ª ed. Boston, MA, USA: Addison-Wesley Professional, 2004.

[55] K. Beck, *Test-Driven Development: By Example*. Boston, MA, USA: Addison-Wesley Professional, 2003.

[56] C. Larman, *Agile and Iterative Development: A Manager's Guide*. Boston, MA, USA: Addison-Wesley Professional, 2003.

[57] K. Schwaber y M. Beedle, *Agile Software Development with Scrum*. Upper Saddle River, NJ, USA: Prentice Hall, 2002.

[58] K. Schwaber y J. Sutherland. "The Scrum Guide — The Definitive Guide to Scrum: The Rules of the Game," visitado mayo de 2026. dirección: https://scrumguides.org/scrum-guide.html

[59] R. S. Pressman y B. R. Maxim, *Software Engineering: A Practitioner's Approach*, 9.ª ed. New York, NY, USA: McGraw-Hill Education, 2020.

[60] A. Cockburn, *Crystal Clear: A Human-Powered Methodology for Small Teams*. Boston, MA, USA: Addison-Wesley Professional, 2004.

[61] M. Poppendieck y T. Poppendieck, *Lean Software Development: An Agile Toolkit*. Boston, MA, USA: Addison-Wesley Professional, 2003.

[62] S. R. Palmer y J. M. Felsing, *A Practical Guide to Feature-Driven Development*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2002.

[63] W. W. Royce, "Managing the Development of Large Software Systems," en *Proc. IEEE WESCON*, 1970, págs. 1-9.

[64] P. Kruchten, *The Rational Unified Process: An Introduction*, 3.ª ed. Boston, MA, USA: Addison-Wesley Professional, 2004.

[65] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland y D. Thomas. "Manifesto for Agile Software Development," visitado mayo de 2026. dirección: https://agilemanifesto.org/

[66] M. Mirzayanov. "Codeforces: Programming Competitions and Contests," visitado mayo de 2026. dirección: https://codeforces.com/

[67] HackerRank Inc. "HackerRank: Practice Coding, Prepare for Interviews," visitado mayo de 2026. dirección: https://www.hackerrank.com/

[68] LeetCode. "LeetCode: The World's Leading Online Programming Learning Platform," visitado mayo de 2026. dirección: https://leetcode.com/

[69] N. Fraser, "Ten Things We've Learned from Blockly," en *Proc. 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*, IEEE, 2015, págs. 49-50. DOI: 10.1109/BLOCKS.2015.7369000

[70] Escuela Superior de Cómputo. "Programa académico: Fundamentos de Diseño Digital — Plan ISC 2020," Instituto Politécnico Nacional, visitado mayo de 2026.

[71] Escuela Superior de Cómputo. "Programa académico: Diseño de Sistemas Digitales — Plan ISC 2020," Instituto Politécnico Nacional, visitado mayo de 2026.

[72] Xilinx, Inc. “Spartan-6 Family Overview (DS160),” visitado abr. de 2026. dirección: https://docs.amd.com/v/u/en-US/ds160

[73] Lattice Semiconductor. “iCE40 UltraPlus Family Data Sheet,” visitado abr. de 2026. dirección: https://www.latticesemi.com/Products/FPGAandCPLD/iCE40UltraPlus

[74] Kreijstal. “AND-gate-cosim-madness: co-simulación de una compuerta AND.” Sin licencia declarada; 1 caso usado en SimulatorBench, visitado sep. de 2026. dirección: https://github.com/Kreijstal/AND-gate-cosim-madness

[75] JulyWitch. “vhdl_ghdl_examples: ejemplos VHDL para GHDL.” Sin licencia declarada; 14 casos usados en SimulatorBench, visitado sep. de 2026. dirección: https://github.com/JulyWitch/vhdl_ghdl_examples

[76] CodiieSB. “VHDL-DFlipFlop: implementación de un flip-flop D en VHDL.” Licencia GPL-3.0; 1 caso usado en SimulatorBench, visitado sep. de 2026. dirección: https://github.com/CodiieSB/VHDL-DFlipFlop

[77] MayElfkiy. “Sequence-Detector-VHDL: detector de secuencia en VHDL.” Sin licencia declarada; 1 caso usado en SimulatorBench, visitado sep. de 2026. dirección: https://github.com/MayElfkiy/Sequence-Detector-VHDL

[78] pmassolino. “vhdl-examples: ejemplos de circuitos en VHDL.” Sin licencia declarada; 12 casos usados en SimulatorBench, visitado sep. de 2026. dirección: https://github.com/pmassolino/vhdl-examples

[79] fcayci. “vhdl-digital-design: material de diseño digital en VHDL.” Sin licencia declarada; 8 casos usados en SimulatorBench, visitado sep. de 2026. dirección: https://github.com/fcayci/vhdl-digital-design

[80] ARC-Lab-UF. “vhdl-tutorial: tutorial de VHDL.” Licencia GPL-3.0; 1 caso usado en SimulatorBench, visitado sep. de 2026. dirección: https://github.com/ARC-Lab-UF/vhdl-tutorial

[81] LemurPwned. “classic-fpga: ejemplos de diseño digital en VHDL.” Licencia GPL-3.0; 16 casos usados en SimulatorBench, visitado sep. de 2026. dirección: https://github.com/LemurPwned/classic-fpga

[82] susanacanel. “proyectos-vhdl: colección de diseños VHDL.” Sin licencia declarada; 31 casos usados en SimulatorBench, visitado sep. de 2026. dirección: https://github.com/susanacanel/proyectos-vhdl

[83] Mohamedsalem80. “VHDL-tutorial: tutorial de VHDL.” Licencia Apache-2.0; 1 caso usado en SimulatorBench, visitado sep. de 2026. dirección: https://github.com/Mohamedsalem80/VHDL-tutorial

[84] peioazk. “counter: contador en VHDL.” Sin licencia declarada; 1 caso usado en SimulatorBench, visitado sep. de 2026. dirección: https://github.com/peioazk/counter

[85] tomas-fryza. “vhdl-examples: ejemplos didácticos en VHDL.” Licencia MIT; 3 casos usados en SimulatorBench, visitado sep. de 2026. dirección: https://github.com/tomas-fryza/vhdl-examples

[86] AlbertoMarquillas. “vhdl-dsp-building-blocks: bloques DSP en VHDL.” Licencia MIT; 7 casos usados en SimulatorBench, visitado sep. de 2026. dirección: https://github.com/AlbertoMarquillas/vhdl-dsp-building-blocks

[87] Martoni. “VHDL_Lib: biblioteca de componentes VHDL.” Sin licencia declarada; 1 caso usado en SimulatorBench, visitado sep. de 2026. dirección: https://github.com/Martoni/VHDL_Lib

[88] Aliflori. “VHDL-Digital-Logic-Labs: prácticas de lógica digital en VHDL.” Licencia MIT; 3 casos usados en SimulatorBench, visitado sep. de 2026. dirección: https://github.com/Aliflori/VHDL-Digital-Logic-Labs

Apéndice A

Anexos

A.1. Ejemplo de Código VHDL

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity AND_Gate is
    Port (
        A : in  STD_LOGIC;
        B : in  STD_LOGIC;
        Y : out STD_LOGIC
    );
end AND_Gate;

architecture Behavioral of AND_Gate is
begin
    Y <= A and B;
end Behavioral;
```

Listing A.1: *Ejemplo de una compuerta AND en VHDL.*

A.2. Ejemplo de Código Verilog

El alcance de este trabajo se limita a VHDL; Verilog se contempla únicamente como una posible extensión futura y no forma parte del sistema entregado. El siguiente ejemplo se incluye con carácter ilustrativo, para contrastar la sintaxis de ambos lenguajes.

```
module and_gate (
    input  wire a,
    input  wire b,
    output wire y
);
    assign y = a & b;
endmodule
```

Listing A.2: *Ejemplo de una compuerta AND en Verilog (ilustrativo; fuera del alcance).*

A.3. Capturas Complementarias del Prototipo

Este anexo agrupa las capturas del prototipo funcional de Kmila que complementan los mockups presentados en la Sección 5.9. Se incluyen vistas del resto de la aplicación: la pantalla de inicio, el selector de idioma y su internacionalización, los dos temas visuales, pestañas adicionales del panel de simulación, la vista de configuración avanzada y el módulo de aprendizaje. Todas las capturas provienen de la versión web del prototipo (`localhost:5297`) el 21 de abril de 2026.

A.3.1. Pantalla de Inicio

Figura A.1: *Pantalla de inicio de Kmila: presenta el nombre del sistema, el lema del proyecto, la mascota (*Kmila*) y los accesos directos a la gestión de proyectos y al módulo de aprendizaje.*

Figura A.2: *Sección* Características Principales *de la pantalla de inicio: visualización dinámica de variables, multiplataforma, autonomía sin conexión, función pre-laboratorio y accesibilidad.*

A.3.2. Internacionalización (7 idiomas)

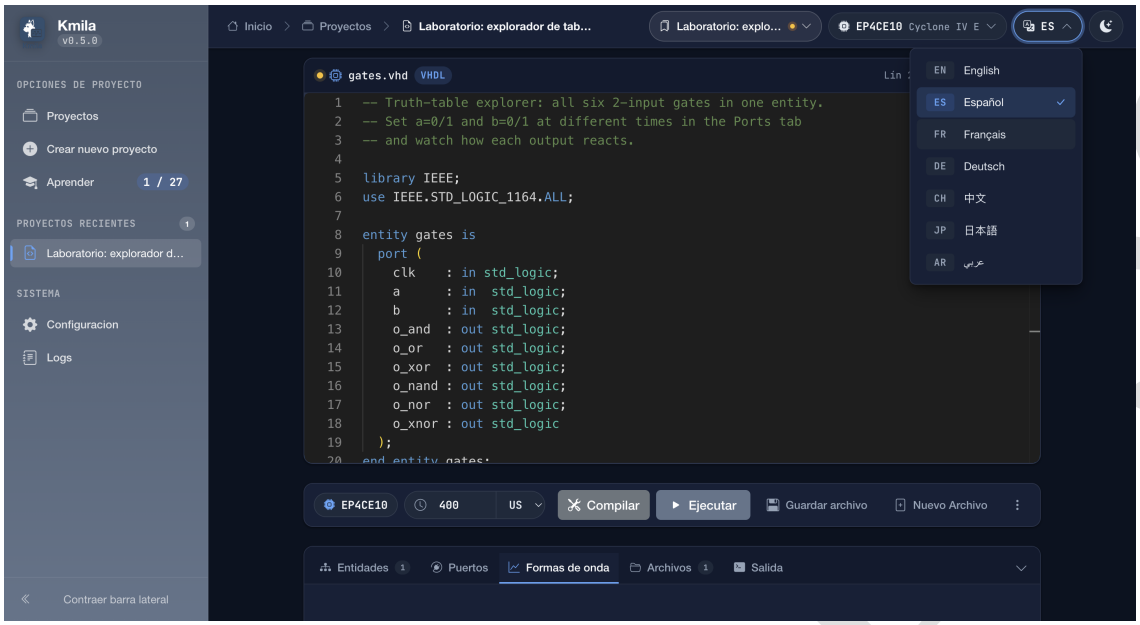


Figura A.3: *Selector de idioma en la barra superior: los siete idiomas soportados por Kmila (Inglés, Español, Francés, Alemán, Chino simplificado, Japonés y Árabe) con sus nombres nativos.*

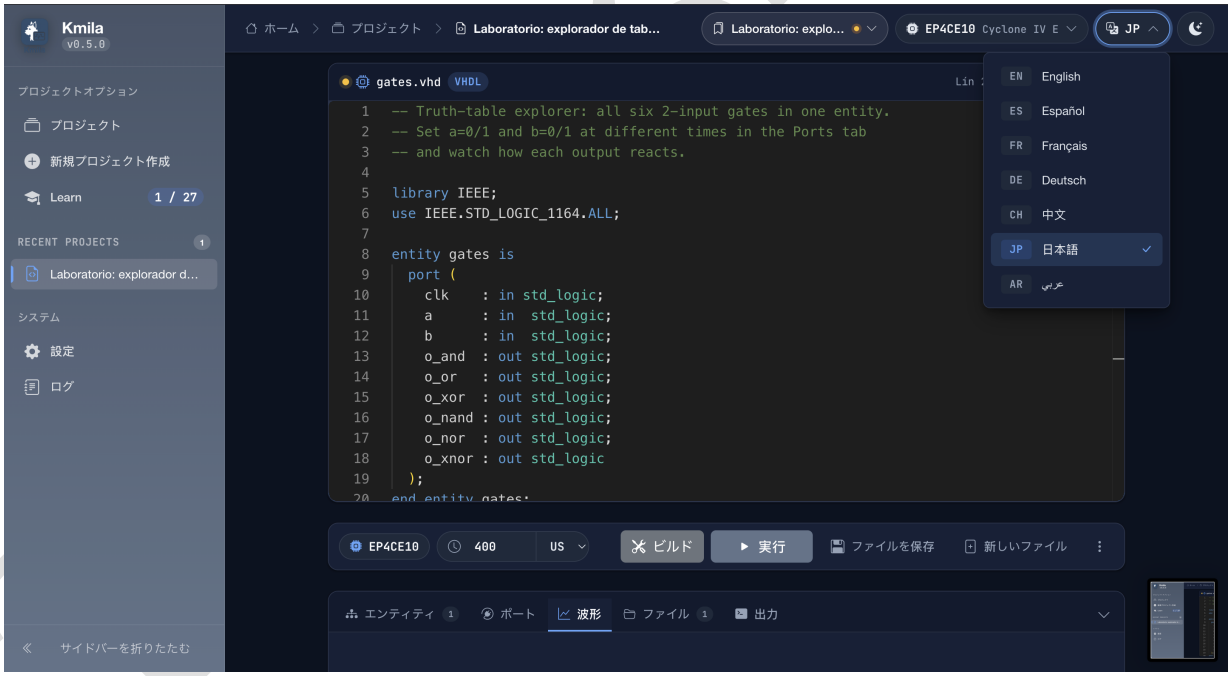


Figura A.4: *Interfaz completa en japonés (*Nihongo*): demuestra la integración reactiva del cambio de idioma sin recarga de página. Todos los rótulos de la barra lateral, barra superior, botones de acción y pestañas del panel inferior se traducen simultáneamente a partir del diccionario JSON correspondiente.*

A.3.3. Temas Visuales (claro y oscuro)

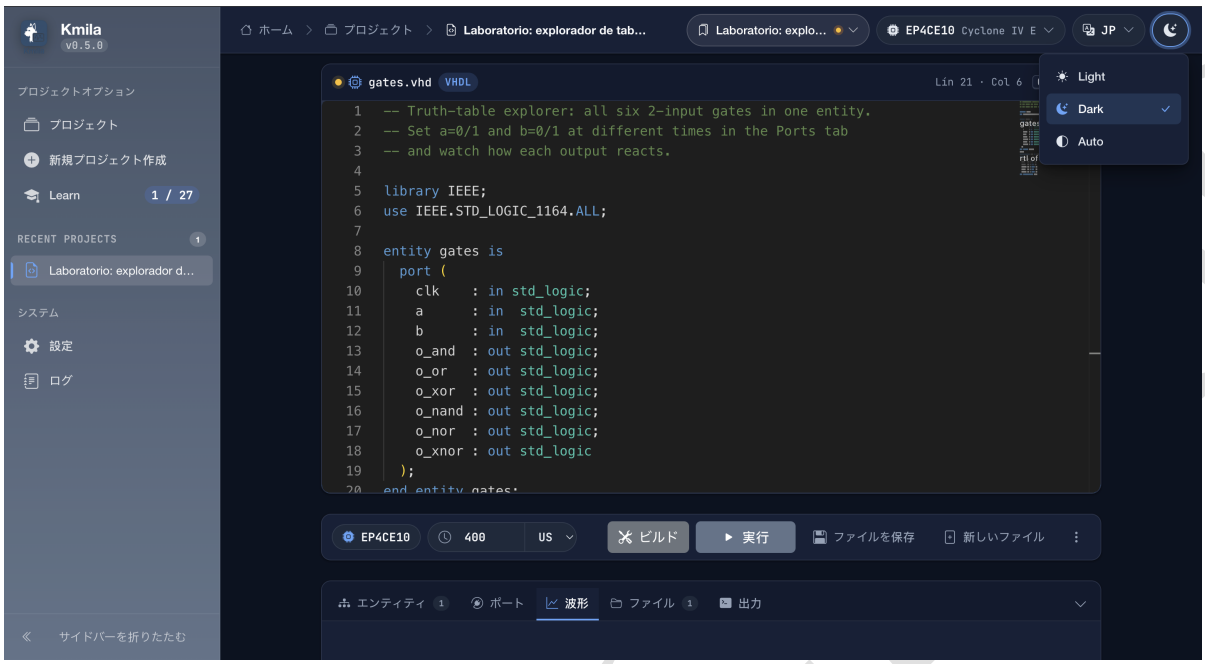


Figura A.5: *Selector de tema visual mostrando las tres opciones disponibles (*Light, Dark, Auto*) con el tema* Dark *activo, configuración por defecto.*

Figura A.6: *Aplicación con el tema* Light *aplicado: todos los componentes (Monaco, paneles, modales, tooltips) se adaptan a la paleta clara.*

A.3.4. Pestaña Entidades del Panel de Simulación

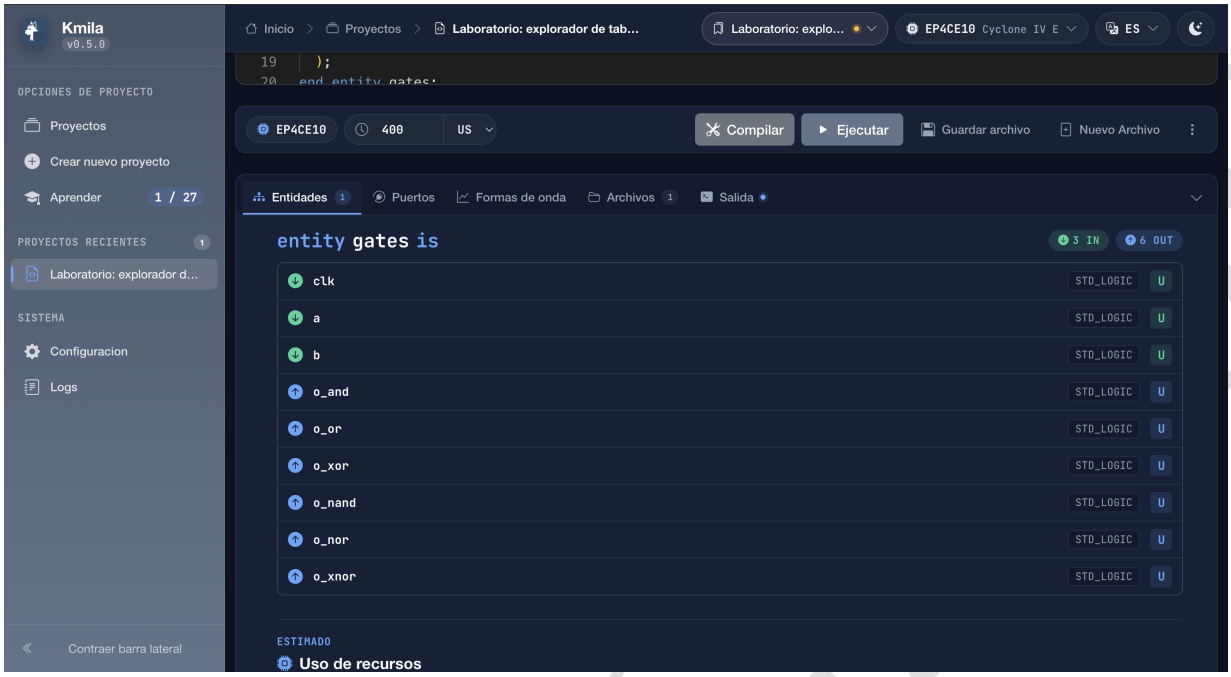


Figura A.7: *Pestaña* Entidades *del panel inferior del editor: muestra la firma de la entidad* `gates` *detectada por el analizador sintáctico, con tres señales de entrada (*`clk`*,* `a`*,* `b`*) y seis señales de salida, todas de tipo* `STD_LOGIC`*. El panel es consumido por el resto de pestañas (Puertos, Formas de onda) para conocer qué señales pueden estimularse o visualizarse.*

A.3.5. Configuración Avanzada

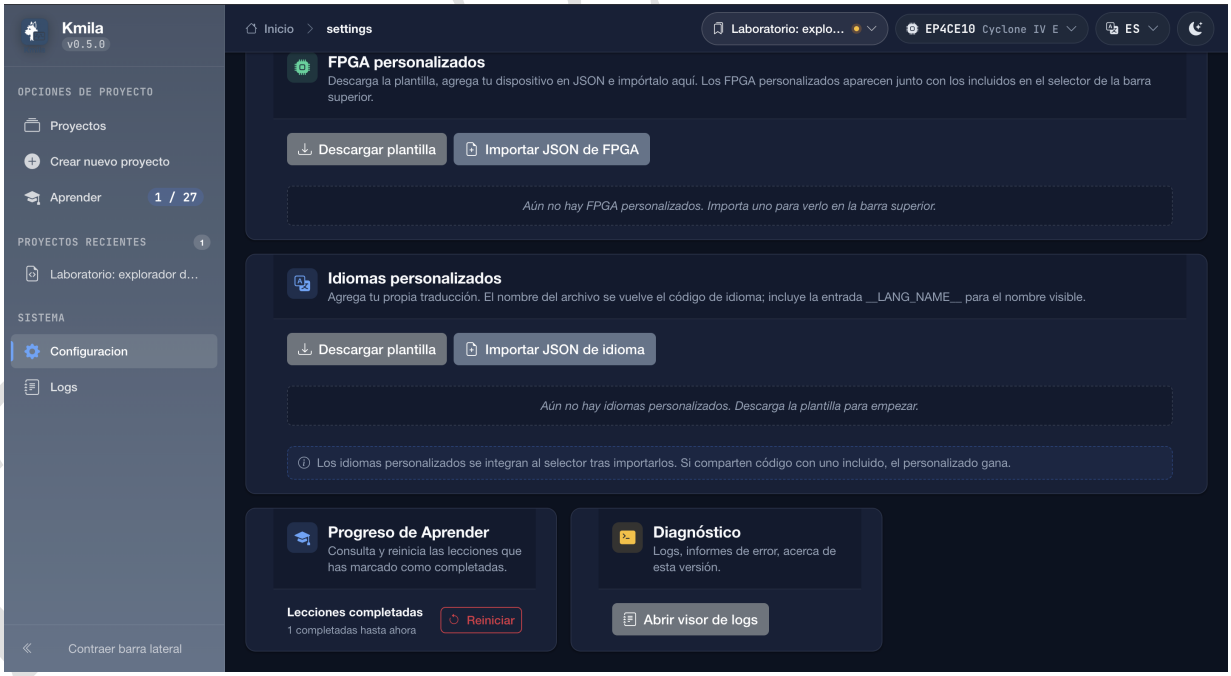


Figura A.8: *Vista de* Configuración*: permite al usuario extender el catálogo mediante la importación de archivos JSON de FPGAs personalizados (plantilla descargable incluida) y diccionarios JSON de idiomas personalizados (con la clave reservada* `__LANG_NAME__` *para el nombre visible). También incluye acceso rápido al progreso del módulo de aprendizaje y al visor de logs.*

A.3.6. Módulo de Aprendizaje

El módulo *Aprender* integra un plan de estudios progresivo con 27 lecciones distribuidas en cinco niveles de dificultad. Las lecciones incluyen archivos VHDL de ejemplo ejecutables directamente en el editor.

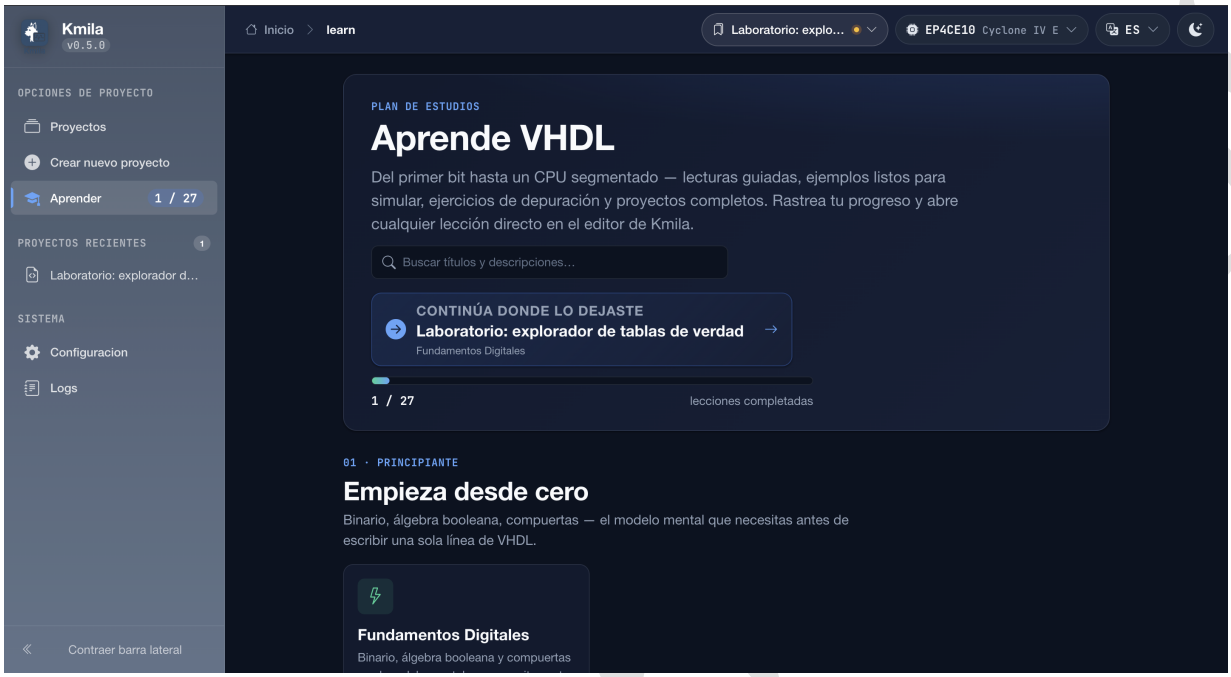


Figura A.9: *Pantalla principal del módulo* Aprender*: presenta el plan de estudios completo, una barra de búsqueda sobre títulos y descripciones, la tarjeta* Continúa donde lo dejaste *(recuperación del último progreso) y un indicador de progreso global (1 de 27 lecciones completadas en esta sesión).*

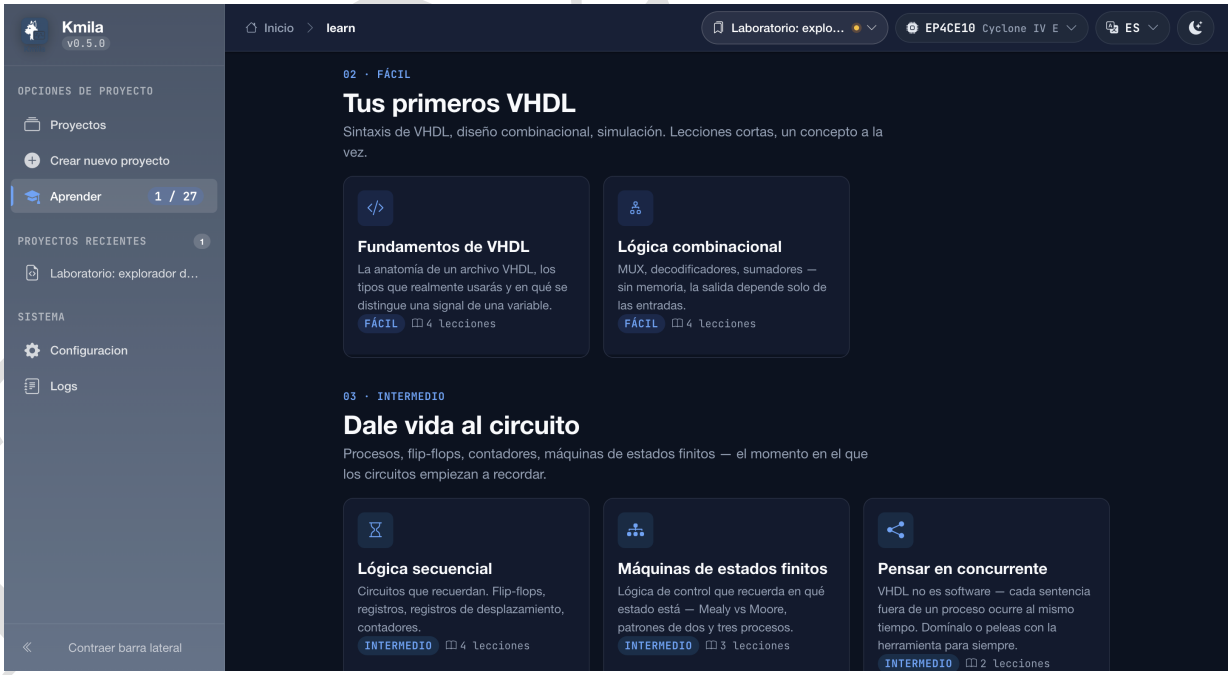


Figura A.10: *Niveles* Fácil *(Fundamentos de VHDL, Lógica combinacional) e* Intermedio *(Lógica secuencial, Máquinas de estados finitos, Pensar en concurrente): cada módulo presenta el número de lecciones y una descripción breve.*

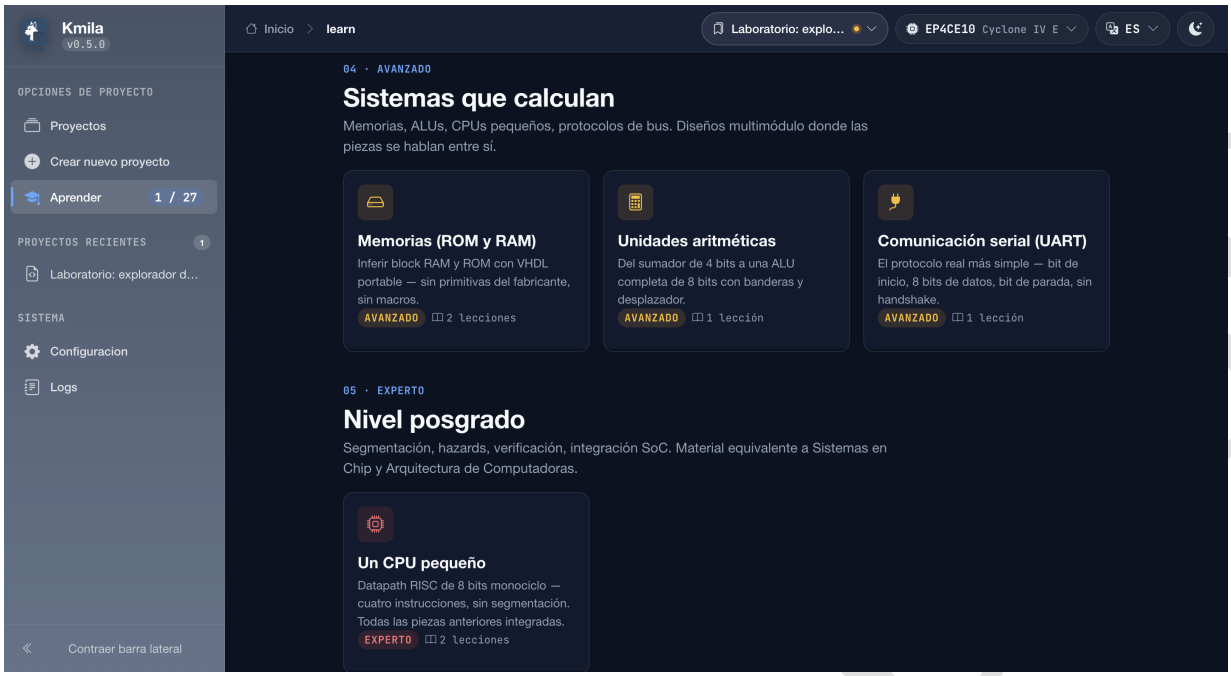


Figura A.11: *Niveles* Avanzado *(Memorias ROM/RAM, Unidades aritméticas, Comunicación serial UART) y* Experto *(Un CPU pequeño basado en datapath RISC de 8 bits). El módulo de aprendizaje complementa los capítulos de diseño digital del plan de estudios de ISC en la ESCOM.*

A.4. Especificación Detallada de Dispositivos FPGA Soportados

Este anexo presenta la especificación completa de los 34 dispositivos FPGA que conforman el catálogo de Kmila, complementando la síntesis presentada en la Sección 2.3.2. La información se distribuye en dos tablas por fabricante para preservar la orientación vertical del documento: la tabla *(A)* agrupa los recursos lógicos, de memoria y aritméticos (DSP); la tabla *(B)* agrupa los recursos de temporización (PLLs, DCM/MMCM), voltaje del núcleo y entrada/salida. Los valores fueron extraídos directamente de los archivos `*.json` del directorio `App/Kmila/Devices/` del repositorio del proyecto, que a su vez se fundamentan en las hojas de datos oficiales de cada fabricante [37], [39], [43], [44], [72], [73].

A.4.1. AMD (Xilinx)

Tabla A.1: *AMD (Xilinx)* (A)*: recursos lógicos, memoria y DSP.*

Modelo	Familia	Unid. lóg.	Tipo unid.	FFs	BRAM	Tipo BRAM	Kb	DSP	Tipo DSP
XC6SLX4	Spartan-6	3 840	Logic Cells	4 800	12	BRAM 18 Kb	216	8	DSP48A1
XC6SLX9	Spartan-6	9 152	Logic Cells	11 440	32	BRAM 18 Kb	576	16	DSP48A1
XC6SLX16	Spartan-6	14 579	Logic Cells	18 224	32	BRAM 18 Kb	576	32	DSP48A1
XC6SLX25	Spartan-6	24 051	Logic Cells	30 064	52	BRAM 18 Kb	936	38	DSP48A1
XC6SLX45	Spartan-6	43 661	Logic Cells	54 576	116	BRAM 18 Kb	2 088	116	DSP48A1
XC6SLX75	Spartan-6	74 637	Logic Cells	93 296	172	BRAM 18 Kb	3 096	180	DSP48A1
XC6SLX100	Spartan-6	101 261	Logic Cells	126 576	268	BRAM 18 Kb	4 824	180	DSP48A1
XC6SLX150	Spartan-6	147 443	Logic Cells	184 304	268	BRAM 18 Kb	4 824	180	DSP48A1
XC7A12T	Artix-7	12 800	Logic Cells	16 000	20	BRAM 36 Kb	720	40	DSP48E1
XC7A15T	Artix-7	16 640	Logic Cells	20 800	25	BRAM 36 Kb	900	45	DSP48E1
XC7A25T	Artix-7	23 360	Logic Cells	29 200	32	BRAM 36 Kb	1 170	80	DSP48E1
XC7A35T	Artix-7	33 280	Logic Cells	41 600	50	BRAM 36 Kb	1 800	90	DSP48E1
XC7A50T	Artix-7	52 160	Logic Cells	65 200	75	BRAM 36 Kb	2 700	120	DSP48E1
XC7A75T	Artix-7	75 520	Logic Cells	94 400	105	BRAM 36 Kb	3 780	180	DSP48E1
XC7A100T	Artix-7	101 440	Logic Cells	126 800	135	BRAM 36 Kb	4 860	240	DSP48E1
XC7A200T	Artix-7	215 360	Logic Cells	269 200	365	BRAM 36 Kb	13 140	740	DSP48E1

Tabla A.2: *AMD (Xilinx)* (B)*: temporización, voltaje y E/S.*

Modelo	PLLs	DCM/MMCM	F_{PLL} (MHz)	V_{core} (V)	I/O máx.	Bancos
XC6SLX4	2	4	1080	1.2	132	4
XC6SLX9	2	4	1080	1.2	200	4
XC6SLX16	2	4	1080	1.2	232	4
XC6SLX25	4	8	1080	1.2	266	4
XC6SLX45	4	8	1080	1.2	358	4
XC6SLX75	6	12	1080	1.2	498	6
XC6SLX100	6	12	1080	1.2	498	6
XC6SLX150	6	12	1080	1.2	576	6
XC7A12T	5	5	800	1.0	170	—
XC7A15T	5	5	800	1.0	250	—
XC7A25T	5	5	800	1.0	250	—
XC7A35T	5	5	800	1.0	250	5
XC7A50T	5	5	800	1.0	250	5
XC7A75T	6	6	800	1.0	300	—
XC7A100T	6	6	800	1.0	500	6
XC7A200T	10	10	800	1.0	500	10

El valor "—" indica que el descriptor JSON del catálogo no especifica el número de bancos de I/O para ese dispositivo.

A.4.2. Intel (Altera)

Tabla A.3: *Intel (Altera)* (A)*: recursos lógicos, memoria y DSP.*

Modelo	Familia	Unid. lóg.	Tipo unid.	FFs	BRAM	Tipo BRAM	Kb	DSP	Tipo DSP
EP4CE6	Cyclone IV E	6 272	LEs	6 272	30	M9K 9 Kb	270	15	Mult. 18x18
EP4CE10	Cyclone IV E	10 320	LEs	10 320	46	M9K 9 Kb	414	23	Mult. 18x18
EP4CE15	Cyclone IV E	15 408	LEs	15 408	56	M9K 9 Kb	504	56	Mult. 18x18
EP4CE22	Cyclone IV E	22 320	LEs	22 320	66	M9K 9 Kb	594	66	Mult. 18x18
EP4CE30	Cyclone IV E	28 848	LEs	28 848	66	M9K 9 Kb	594	66	Mult. 18x18
EP4CE40	Cyclone IV E	39 600	LEs	39 600	126	M9K 9 Kb	1 134	116	Mult. 18x18
EP4CE55	Cyclone IV E	55 856	LEs	55 856	260	M9K 9 Kb	2 340	154	Mult. 18x18
EP4CE75	Cyclone IV E	75 408	LEs	75 408	305	M9K 9 Kb	2 745	200	Mult. 18x18
EP4CE115	Cyclone IV E	114 480	LEs	114 480	432	M9K 9 Kb	3 888	266	Mult. 18x18

Tabla A.4: *Intel (Altera)* (B)*: temporización, voltaje y E/S.*

Modelo	PLLs	DCM/MMCM	F_{PLL} (MHz)	V_{core} (V)	I/O máx.	Bancos
EP4CE6	2	0	1300	1.2	179	8
EP4CE10	2	0	1300	1.2	179	8
EP4CE15	4	0	1300	1.2	343	8
EP4CE22	4	0	1300	1.2	153	8
EP4CE30	4	0	1300	1.2	532	8
EP4CE40	4	0	1300	1.2	532	8
EP4CE55	4	0	1300	1.2	374	8
EP4CE75	4	0	1300	1.2	426	8
EP4CE115	4	0	1300	1.2	528	8

Los dispositivos Cyclone IV E no incorporan DCMs ni MMCMs (gestión avanzada de reloj exclusiva de arquitecturas Xilinx); la generación de señales de reloj se realiza mediante los PLLs nativos. **Paquetes disponibles por modelo:** EP4CE6: E144, F256, U256, M164; EP4CE10: E144, F256, U256; EP4CE15: F256, U256, F484; EP4CE22: E144, F256, U256; EP4CE30: F484, F780; EP4CE40: F484, F780; EP4CE55: U484, F484, F780; EP4CE75: U484, F484, F780; EP4CE115: F484, F780.

A.4.3. Lattice Semiconductor

Tabla A.5: *Lattice* (A)*: recursos lógicos, memoria y DSP.*

Modelo	Familia	Unid. lóg.	Tipo unid.	FFs	BRAM	Tipo BRAM	Kb	DSP	Tipo DSP
iCE40HX1K	iCE40 HX	1 280	LUTs	1 280	16	EBR 4 Kb	64	0	N/A
iCE40HX4K[†]	iCE40 HX	—	LUTs	—	—	EBR 4 Kb	—	0	N/A
iCE40HX8K	iCE40 HX	7 680	LUTs	7 680	32	EBR 4 Kb	128	0	N/A
iCE40UP3K	iCE40 UltraPlus	2 800	LUTs	2 800	20	EBR 4 Kb	80	4	MAC 16x16
iCE40UP5K	iCE40 UltraPlus	5 280	LUTs	5 280	30	EBR 4 Kb	120	8	MAC 16x16
LFE5U-12	ECP5	12 000	LUTs	12 000	32	EBR 18 Kb	576	12	Mult. 18x18
LFE5U-25	ECP5	24 000	LUTs	24 000	56	EBR 18 Kb	1 008	28	Mult. 18x18
LFE5U-45	ECP5	44 000	LUTs	44 000	108	EBR 18 Kb	1 944	72	Mult. 18x18
LFE5U-85	ECP5	84 000	LUTs	84 000	208	EBR 18 Kb	3 744	156	Mult. 18x18

Tabla A.6: *Lattice* (B)*: temporización, voltaje y E/S.*

Modelo	PLLs	DCM/MMCM	F_{PLL} (MHz)	V_{core} (V)	I/O máx.	Bancos
iCE40HX1K	1	0	275	1.2	96	—
iCE40HX4K[†]	—	0	275	1.2	178	—
iCE40HX8K	1	0	275	1.2	206	—
iCE40UP3K	1	0	275	1.2	39	3
iCE40UP5K	1	0	275	1.2	39	3
LFE5U-12	2	2	—	1.1	197	7
LFE5U-25	2	2	—	1.1	245	7
LFE5U-45	4	4	—	1.1	259	7
LFE5U-85	4	4	—	1.1	365	8

[†] El iCE40HX4K comparte die con el iCE40HX8K; Lattice no publica cifras oficiales de LUTs, FFs ni BRAM efectiva para esta variante. Los dispositivos ECP5 no reportan frecuencia máxima de PLL en el catálogo del software. La sigla EBR corresponde a *Embedded Block RAM*.

A.5. Procedencia de los Casos de Validación

La validación funcional del motor contra GHDL (Sección 6.2.3) se apoya en un corpus de **148 casos**. Para garantizar la imparcialidad de la evaluación y evitar cualquier sesgo por casos ajustados a conveniencia, la mayoría del corpus proviene de **repositorios públicos e independientes de terceros**. La procedencia de cada caso se registra en su archivo `meta.json` dentro de `SimulatorBench/Cases/`, con los campos `sourceUrl`, `license` y `fetchedAt`; los casos se recolectaron entre junio y julio de 2026, y las licencias aquí reportadas se verificaron contra la API de GitHub el 15 de septiembre de 2026.

De los 148 casos, **101 (68 %) provienen de 15 repositorios públicos de GitHub** escritos por autores independientes; **46 son pruebas de regresión internas** redactadas para este trabajo, cada una diseñada para fijar un comportamiento específico del motor (por ejemplo, atributos de señal, esperas dentro de cuerpos anidados o síntesis de cuerpos de función); y **1 es una prueba de humo interna**. La Tabla A.7 resume la composición.

Tabla A.7: *Composición del corpus de validación por procedencia y licencia.*

Origen	Casos	Licencia
Repositorios públicos de GitHub (terceros)	101	GPL-3.0 (18), MIT (13), Apache-2.0 (1), sin licencia declarada (69)
Pruebas de regresión internas (propias)	46	Propia
Prueba de humo interna (propia)	1	Propia
Total	**148**	

Los 69 casos marcados como *sin licencia declarada* provienen de repositorios públicos que no incluyen un archivo `LICENSE`; en todos ellos se conserva la atribución mediante la referencia al repositorio de origen. La Tabla A.8 detalla la fuente y la licencia de cada uno de los 148 casos.

Tabla A.8: *Procedencia y licencia de cada caso de validación.*

Caso	Fuente	Licencia
`000-smoke-and`	Prueba de humo (interna)	Propia
`001-and-gate`	`AND-gate-cosim-madness` [74]	Sin licencia
`002-decoder-3to8`	`vhdl_ghdl_examples` [75]	Sin licencia
`003-counter-8bit`	`vhdl_ghdl_examples` [75]	Sin licencia
`004-dff`	`VHDL-DFlipFlop` [76]	GPL-3.0
`005-encoder-8to3`	`vhdl_ghdl_examples` [75]	Sin licencia
`006-demux-1to8`	`vhdl_ghdl_examples` [75]	Sin licencia
`007-boolfn-4bit`	`vhdl_ghdl_examples` [75]	Sin licencia
`008-seqdet-fsm`	`Sequence-Detector-VHDL` [77]	Sin licencia
`009-boolfn-minset`	`vhdl_ghdl_examples` [75]	Sin licencia
`010-mux-8to1`	`vhdl_ghdl_examples` [75]	Sin licencia
`011-decoder-cond`	`vhdl_ghdl_examples` [75]	Sin licencia

continúa en la página siguiente

Caso	Fuente	Licencia
012-decoder-sel	vhdl_ghdl_examples [75]	Sin licencia
013-mux-cond	vhdl_ghdl_examples [75]	Sin licencia
014-mux-sel	vhdl_ghdl_examples [75]	Sin licencia
015-encoder-16to4	vhdl_ghdl_examples [75]	Sin licencia
016-demux-1to32	vhdl_ghdl_examples [75]	Sin licencia
017-fsm-2state	vhdl_ghdl_examples [75]	Sin licencia
018-or-gate	vhdl-examples (pmassolino) [78]	Sin licencia
019-not-gate	vhdl-examples (pmassolino) [78]	Sin licencia
020-xor-gate	vhdl-examples (pmassolino) [78]	Sin licencia
021-adder	vhdl-examples (pmassolino) [78]	Sin licencia
022-mux4x1	vhdl-examples (pmassolino) [78]	Sin licencia
023-flipflop-d	vhdl-examples (pmassolino) [78]	Sin licencia
024-flipflop-jk	vhdl-examples (pmassolino) [78]	Sin licencia
025-counter-pmas	vhdl-examples (pmassolino) [78]	Sin licencia
026-shift-register	vhdl-examples (pmassolino) [78]	Sin licencia
027-add-sub	vhdl-examples (pmassolino) [78]	Sin licencia
028-state-machine	vhdl-examples (pmassolino) [78]	Sin licencia
029-lfsr	vhdl-examples (pmassolino) [78]	Sin licencia
030-half-adder	vhdl-digital-design [79]	Sin licencia
031-full-adder	vhdl-digital-design [79]	Sin licencia
032-parity	vhdl-digital-design [79]	Sin licencia
033-priority	vhdl-digital-design [79]	Sin licencia
034-hamming	vhdl-digital-design [79]	Sin licencia
035-fsm-fcayci	vhdl-digital-design [79]	Sin licencia
036-fsm-counter	vhdl-digital-design [79]	Sin licencia
037-adder4	vhdl-digital-design [79]	Sin licencia
042-prio4	vhdl-tutorial (ARC-Lab-UF) [80]	GPL-3.0
045-compar	classic-fpga [81]	GPL-3.0
046-dff-lemur	classic-fpga [81]	GPL-3.0
047-counter-lemur	classic-fpga [81]	GPL-3.0
048-ring-counter	classic-fpga [81]	GPL-3.0
049-grey	classic-fpga [81]	GPL-3.0
050-one-hot	classic-fpga [81]	GPL-3.0
051-mux1-lemur	classic-fpga [81]	GPL-3.0
052-piso	classic-fpga [81]	GPL-3.0
053-sipo	classic-fpga [81]	GPL-3.0
054-abs1	classic-fpga [81]	GPL-3.0
055-seqdet-lemur	classic-fpga [81]	GPL-3.0

continúa en la página siguiente

Caso	Fuente	Licencia
056-code-det	classic-fpga [81]	GPL-3.0
057-barrel	classic-fpga [81]	GPL-3.0
059-reg-adder	classic-fpga [81]	GPL-3.0
060-trans	classic-fpga [81]	GPL-3.0
061-compar-fsm	classic-fpga [81]	GPL-3.0
062-cod4a2	proyectos-vhdl [82]	Sin licencia
063-comp-n	proyectos-vhdl [82]	Sin licencia
065-mux4-suc	proyectos-vhdl [82]	Sin licencia
066-gray2bin	proyectos-vhdl [82]	Sin licencia
067-paridad	proyectos-vhdl [82]	Sin licencia
068-summag	proyectos-vhdl [82]	Sin licencia
071-nat2aik	proyectos-vhdl [82]	Sin licencia
072-ffd-susi	proyectos-vhdl [82]	Sin licencia
073-ffd-clen	proyectos-vhdl [82]	Sin licencia
074-contm	proyectos-vhdl [82]	Sin licencia
075-cont-anillo	proyectos-vhdl [82]	Sin licencia
076-johnson8	proyectos-vhdl [82]	Sin licencia
077-contsec	proyectos-vhdl [82]	Sin licencia
079-ffjk	proyectos-vhdl [82]	Sin licencia
080-contmpot2	proyectos-vhdl [82]	Sin licencia
081-latch-d	proyectos-vhdl [82]	Sin licencia
082-latch-sr	proyectos-vhdl [82]	Sin licencia
084-and2ecu	proyectos-vhdl [82]	Sin licencia
085-compnandn	proyectos-vhdl [82]	Sin licencia
086-sec-moore	proyectos-vhdl [82]	Sin licencia
087-deco3a8	proyectos-vhdl [82]	Sin licencia
088-decon	proyectos-vhdl [82]	Sin licencia
089-muxn	proyectos-vhdl [82]	Sin licencia
091-compnorn	proyectos-vhdl [82]	Sin licencia
092-sec-mealy	proyectos-vhdl [82]	Sin licencia
093-sec-mooress	proyectos-vhdl [82]	Sin licencia
095-regsinc	proyectos-vhdl [82]	Sin licencia
097-bidir	proyectos-vhdl [82]	Sin licencia
098-tcqrs	proyectos-vhdl [82]	Sin licencia
099-sec-mealyss	proyectos-vhdl [82]	Sin licencia
100-traffic	VHDL-tutorial (Mohamedsalem80) [83]	Apache-2.0
101-johnson7	proyectos-vhdl [82]	Sin licencia
102-counter-arst	counter (peioazk) [84]	Sin licencia

continúa en la página siguiente

Caso	Fuente	Licencia
103-dff-qbar	vhdl-examples (fryza) [85]	MIT
105-ha-marq	vhdl-dsp-building-blocks [86]	MIT
106-comb-marq	vhdl-dsp-building-blocks [86]	MIT
107-deco-marq	vhdl-dsp-building-blocks [86]	MIT
108-demux-marq	vhdl-dsp-building-blocks [86]	MIT
109-cnt9-marq	vhdl-dsp-building-blocks [86]	MIT
112-ser-fsm	vhdl-dsp-building-blocks [86]	MIT
113-edges-det	VHDL_Lib [87]	Sin licencia
114-seven-seg	VHDL-Digital-Logic-Labs [88]	MIT
115-ud-cnt8	VHDL-Digital-Logic-Labs [88]	MIT
116-7seg-mux	VHDL-Digital-Logic-Labs [88]	MIT
117-buffer	vhdl-dsp-building-blocks [86]	MIT
118-comparator2	vhdl-examples (fryza) [85]	MIT
119-demorgan	vhdl-examples (fryza) [85]	MIT
120-multi-hop-pure	Regresión interna (Kmila)	Propia
121-multi-hop-synced-rom	Regresión interna (Kmila)	Propia
122-rom-array-lookup	Regresión interna (Kmila)	Propia
123-rom-literal-lookup	Regresión interna (Kmila)	Propia
124-named-aggregate-funcs	Regresión interna (Kmila)	Propia
125-case-ident-match	Regresión interna (Kmila)	Propia
126-case-ident-hex	Regresión interna (Kmila)	Propia
127-case-slice-selector	Regresión interna (Kmila)	Propia
128-unsigned-slice-assign	Regresión interna (Kmila)	Propia
129-enum-eq-conditional	Regresión interna (Kmila)	Propia
130-arith-in-vector-range	Regresión interna (Kmila)	Propia
131-indexed-lhs-write	Regresión interna (Kmila)	Propia
132-entity-work-inst	Regresión interna (Kmila)	Propia
133-others-init-readback	Regresión interna (Kmila)	Propia
134-others-rhs-clocked-reset	Regresión interna (Kmila)	Propia
135-mul-width-tripwire	Regresión interna (Kmila)	Propia
136-fn-multi-caller-scope	Regresión interna (Kmila)	Propia
137-entity-work-generic-map	Regresión interna (Kmila)	Propia
138-multi-level-elab	Regresión interna (Kmila)	Propia
139-lfsr-through-fn	Regresión interna (Kmila)	Propia
140-multiarg-userfn	Regresión interna (Kmila)	Propia
141-wait-in-if-body	Regresión interna (Kmila)	Propia
142-wait-in-case-arm	Regresión interna (Kmila)	Propia
143-wait-in-nested-loop	Regresión interna (Kmila)	Propia

continúa en la página siguiente

Caso	Fuente	Licencia
144-procedure-wait	Regresión interna (Kmila)	Propia
145-procedure-if-branch	Regresión interna (Kmila)	Propia
146-procedure-time-formal	Regresión interna (Kmila)	Propia
147-case-matching	Regresión interna (Kmila)	Propia
148-signal-event-in-wait	Regresión interna (Kmila)	Propia
149-signal-last-value	Regresión interna (Kmila)	Propia
150-signal-stable-time-arg	Regresión interna (Kmila)	Propia
151-fn-inline-and2	Regresión interna (Kmila)	Propia
152-fn-inline-mux2	Regresión interna (Kmila)	Propia
153-proc-inline-signal-assign	Regresión interna (Kmila)	Propia
154-fn-local-var	Regresión interna (Kmila)	Propia
155-fn-case-mux	Regresión interna (Kmila)	Propia
156-proc-in-fn	Regresión interna (Kmila)	Propia
157-fn-two-callsites	Regresión interna (Kmila)	Propia
158-process-local-ram	Regresión interna (Kmila)	Propia
159-fn-early-return-simple	Regresión interna (Kmila)	Propia
160-fn-early-return-chain	Regresión interna (Kmila)	Propia
161-fn-branch-local-var	Regresión interna (Kmila)	Propia
162-fn-case-arm-local-var	Regresión interna (Kmila)	Propia
163-fn-early-return-with-tail-ops	Regresión interna (Kmila)	Propia
164-fn-xor-literal-vector	Regresión interna (Kmila)	Propia
165-fn-for-loop-count-ones	Regresión interna (Kmila)	Propia