elektron ámbar

Original Article

Revista elektron, Vol. 5, No. 1, pp. 56-63 (2021)
https://doi.org/10.37537/rev.elektron.5.1.113.2021

FlowHDL, lenguaje de programación visual para el diseño digital de lógica programable

FlowHDL, visual programming language for digital design of programmable logic

Anibal Fernando Antonelli[1] and Carlos Arturo Gayoso[2]

Laboratorio de Componentes Electrónicos, Facultad de Ingeniería, Universidad Nacional de Mar del Plata (FI-UNMdP)
Juan B. Justo 2002, Mar del Plata, Buenos Aires, Argentina
[1]anibal.antonelli@fi.mdp.edu.ar
[2]cgayoso@fi.mdp.edu.ar

Recibido: 16/10/20; Aceptado: 04/02/21

***Resumen*— Dentro del diseño digital de lógica programable se cuenta con una gran cantidad de herramientas de software, sin embargo se observa que existe la necesidad de disponer de mayores facilidades para los usuarios de estos sistemas. Los principios necesarios de usabilidad pueden ser provistos por los lenguajes de programación visual. Por otro lado, para generar código VHDL o Verilog sintentizable en una FPGA a través de un sistema visual coherente, es necesario superar múltiples inconvenientes. En este trabajo se presenta una nueva herramienta visual para el diseño de circuitos electrónicos digitales.**

Palabras clave: lenguaje de programación visual, diseño digital, VHDL, máquinas de estado, tablas de verdad, desarrollo de software.

Within the digital design of programmable logic there is a large number of software, however the need for greater facilities for the users of these systems is observed. The necessary principles of usability can be provided by visual programming languages. On the other hand, to generate synthesizable VHDL or Verilog code in an FPGA through a coherent visual system, it is necessary to overcome multiple drawbacks. This work presents a new visual tool for the design of digital electronic circuits.

Keywords: visual programming language, digital design, VHDL, state machines, truth tables, software development.

I. Introducción

Si bien se dispone de una gran cantidad de herramientas de diseño digital, se observa que existe la necesidad de contar con aquellas que simplifiquen la descripción, entendimiento y desarrollo de circuitos digitales. Lenguajes de programación visuales como LabView [1] se vienen desarrollando en éste sentido. También herramientas de código abierto como RKH [2] para máquinas de estado, o Icestudio [3] para interconexión de componentes. En éste contexto, la mejora en los tiempos de diseño y aprendizaje, aprovechando el recurso visual deben ser explotados aún más para el desarrollo del diseño digital.

Además, en la actualidad también se cuenta con herramientas que facilitan el trabajo de describir hardware a partir lenguajes de alto nivel como C, C++, SystemC [4], Matlab [5] e incluso en el último tiempo Rust [6]. Éstas herramientas aceleran el proceso de desarrollo de sistemas digitales, pero no siempre garantizan resultados óptimos en término de área utilizada y rendimiento del sistema [7]. De ésta forma, lenguajes que faciliten y enriquezcan el diseño digital a toda escala sigue siendo de interéz.

Según Marriott y Meyer [8] un lenguaje visual es un conjunto de diagramas que representan sentencias válidas en ese mismo lenguaje. Tales diagramas se pueden entender como una colección de símbolos en un espacio bidimensional o tridimensional. Usando este paradigma, se puede lograr el diseño de sistemas de forma más accesible para los usuarios [9].

En el presente trabajo se introduce un lenguaje de programación visual para el diseño digital denominado FlowHDL. Al igual que en VHDL y Verilog, donde existen especificaciones funcionales denominadas Entidades y Módulos respectivamente, en FlowHDL se cuenta también con nodos funcionales, los cuales pueden instanciar componentes particulares y ser interconectados a través de sus compuertas de entrada y salida. Una vez que se cuenta con el Lenguaje de Descripción de Hardware (HDL por sus siglas en ingles), los componentes se sintetizan en elementos electrónicos digitales y las interconexiones en conexiones físicas que transmiten señales eléctricas.

Con el presente Lenguaje Visual se busca agilizar y lograr un entendimiento estructural del sistema en su conjunto [9] en el proceso de diseño digital de la lógica programable. El mismo por ser un lenguaje de programación visual con manipulación directa desde la interfaz, puede ser utilizado en una etapa inicial para la enseñanza; pero en la medida que vaya madurando puede llegar a tener un general en el desarrollo de un proyectos.

II. Implementación

El sistema se ha implementado utilizado el lenguaje de programación C++ debido a la necesidad de utilizar la Programación Orientada a Objetos (OOP, por sus siglas en inglés). Se utiliza la herencia para realizar definiciones de código generales que funcionaran más allá de las particularidades del sistema. Así, las funcionalidades de las conexiones, los espacios (slots) donde se encastran las conexiones, y los nodos que contienen transformaciones de datos, se definen de manera abstracta en la lógica central del sistema. Con los aspectos generales cubiertos, las particularidades se modelan y solucionan. El sistema constituye

así un framework [10], el cual se detallará más adelante.

II-A. Estructura del sistema

Se cuenta con tres grandes bloques en la arquitectura del software (Fig. 1). Desde la interfaz de usuario se crean y conectan nodos funcionales a través de la interacción directa, los nodos pueden provenir de Entidades en archivos VHDL de proyectos pre-existentes, tablas de verdad creadas desde cero o importadas desde archivos con el formato utilizado por el software Espresso Logic Minimizer [11], máquinas de estado para ser creadas en la interfaz o nodos jerárquicos que contendrán otros nodos visuales de cualquiera de los tipos ya citados. La estructura funcional de mayor jerarquía (top Entity en VHDL), es un nodo jerárquico que comienza vacío en el proyecto. Del diseño se encuentra con el nodo funcional padre, en donde los sub-nodos se conectan.

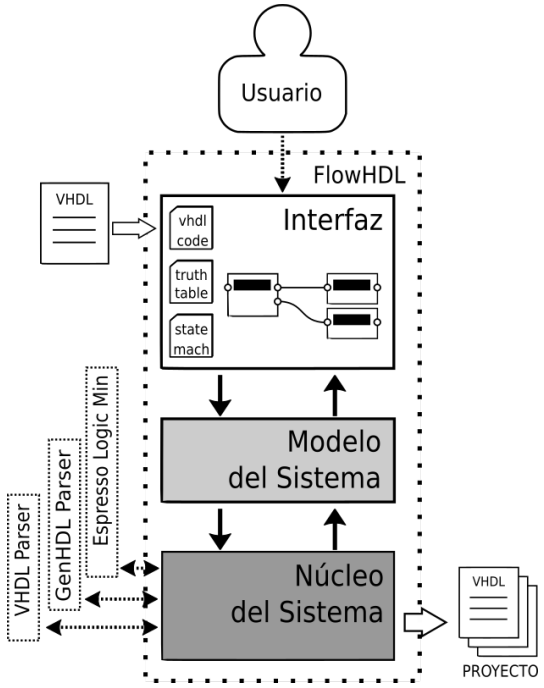

Figura 1: Diagrama del sistema FlowHDL.

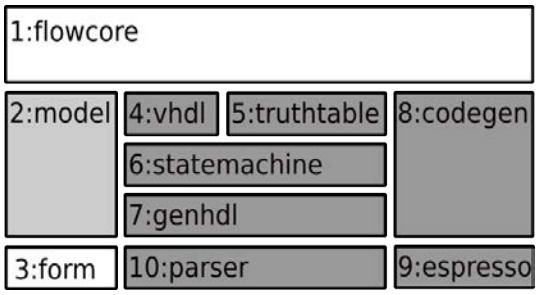

Figura 2: Bloques principales del sistema FlowHDL.

Como se observa en la Fig. 2 se encuentran los bloques del sistema correspondientes a cada una de las capas, así `1:flowcore` y `3:from` corresponden a la interfaz de usuario. El bloque `1:flowcore` se ha programado como un framework visual que interactúa con las clases internas abstractas del modelo. Es importante destacar que este framework visual se ha logrado a partir de realizar amplias modificaciones a la herramienta de código abierto Node Editor [12]. En `2:model` se agrupan las implementaciones concretas de los modelos para cada uno de los tipos de elementos: VHDL, tablas de verdad, maquinas de estado y una representación textual intermedia denominada genHDL. Por otro lado `3:from` contiene los formularios visuales de edicion de estos elementos.

Como se muestra en la Fig. 2 `4:vhdl`, `5:truthtable`, `6:statemachine` y `7:genhdl` son encapsulados por `2:model` para utilizarse desde el framework visual `1:flowcore` siendo trasparente para éste último. También todos estos bloques del núcleo mencionados, contienen estructuras y objetos que validan y almacenan los datos necesarios para generar VHDL válido a partir de la utilización de la librería de generación de código `8:condegen`.

Por ejemplo, el bloque `5:truthtable` mantiene las estructuras de datos que almacenan las tablas de verdad, a través de `8:condegen` se genera un archivo entendible por Espresso Logic Minimizer. El bloque `9:esperesso` se comunica con el sistema Espresso Logic Minimizer y éste devuelva las funciones lógicas que representan cada salida respecto a las entradas. Por último se transforman las funciones en formato Espresso a formato VHDL a través de un conjunto de expresiones regulares.

En el bloque `10:parser` se toma como entrada código VHDL y genHDL. Allí se mantiene una comunicación con subsistemas que analizan los lenguajes de entrada textual, generando así objetos JSON [13], que se devolverán a FlowHDL y servirán para crear los elementos necesarios en los bloques `4:vhdl` y `7:genhdl`. Estos subsistemas se han desarrollado en el lenguaje de programación Rust, dejando su estudio para futuros trabajos.

II-B. Clases del sistema

Teniendo tres elementos visuales, dos ranuras (slots) y una conexión, se puede tener funcionalidades subyacentes diferentes: cuando estas ranuras contienen puertos VHDL (algo transparente para `1:flowcore`), se llamará a una funcionalidad diferente a cuando se conecten dos estados en una máquina de estados; pero, las clases abstractas generales que rigen esta funcionalidad son las mismas en ambos casos. Con esto se logra tener menor repetición de código y mayor facilidad de extender el sistema, ya que las generalidades no deberían cambiar si hay un buen diseño de software. Como ya se ha dicho, de esta forma, se da la posibilidad de extender el sistema visual en forma de framework. Las clases abstractas que brindan esta posibilidad se encuentran en el modelo de datos.

A continuación se verán las especificaciones de las clases ranuras y conexiones para observar la especificación de los nodos ver el Apéndice A.

II-B1. Ranuras: En la Fig. 3 se puede observar la clase abstracta `SlotModel` que representa un elemento al que se pueden agregar conexiones; se procede a detallar algunos de sus métodos concretos y abstractos:

- `haveConnectionSpace`: método abstracto que devuelve un valor booleano verdadero en caso que tenga espacio para realizar una conexión.

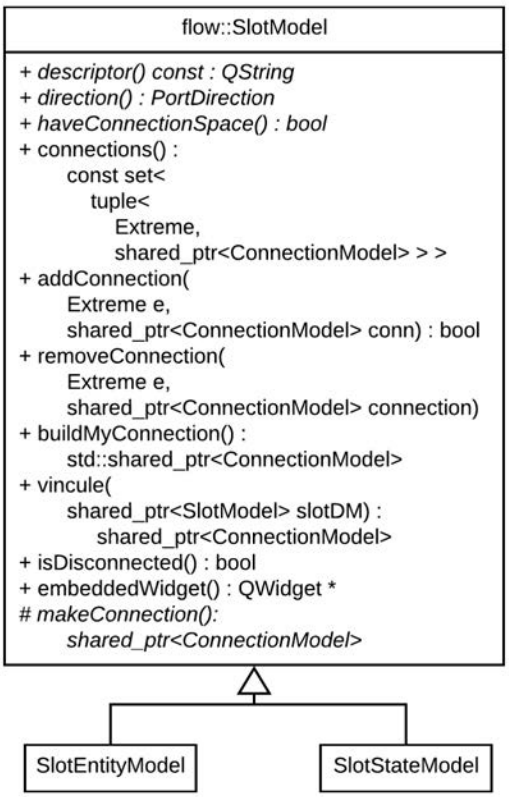

Figura 3: Diagrama de clases para el modelo de slots.

- `connections`: método concreto que devuelve un conjunto de todos los modelos de conexiones que contiene junto al dato del extremo de la conexión (extremo A o B).

- `addConnection`: método concreto que agrega una conexión, se devuelve un valor booleano dependiendo del éxito o fracaso de la operación.

- `removeConnection`: método concreto que elimina una conexión.

- `buildMyConnection`: método concreto que crea una conexión y se agrega a el mismo a ésta, para así retornarla al llamador; la conexión es creada a través del método abstracto `makeConnection` que se detalla a continuación.

- `vincule`: método concreto similar al `buildMyConnection` pero a este se le da otro slot al cual vincularse; también es necesario llamar al método `makeConnection`.

- `makeConnection`: método abstracto que crea una instancia de un modelo de conexión, actualmente hay dos tipos de clases concretas de slots, los que se corresponden con una entidad VHDL y los que se corresponden a una máquina de estados (`SlotEntityModel` y `SlotStateModelConcrete` respectivamente), las implementaciones de este método se corresponderá a la creación de un `ConnectionEntityModel` y un `ConnectionStateModel` respectivamente (clases que se detallarán más adelante).

Se observa también en la Fig. 3 las clases ya mencionadas `SlotEntityModel` y `SlotStateModel`. Se procede a detallar cada una de ellas:

- `SlotEntityModel`: clase que internamente contiene un slot VHDL, con lo cual solo puede ser conectado a otro slot VHDL.

- `SlotStateModel`: clase que internamente contiene un estado de una máquina de estados, con lo cual solo puede ser conectado a otro estado.

II-B2. Conexiones: En la Fig. 4 se puede observar la clase abstracta `ConnectionModel` que representa un elemento de conexión; se procede a detallar algunos de sus métodos:

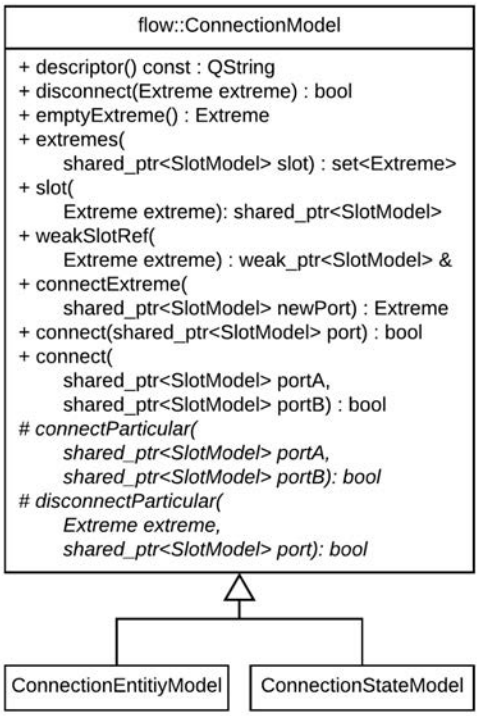

Figura 4: Diagrama de clases para el modelo de conexiones.

- `disconnect`: método concreto que desconecta un extremo de la conexión (ya sea el extremo A o B) devuelve un valor booleano verdadero en caso que sea correcta la desconexión y falso en caso contrario.

- `emptyExtreme`: método concreto que devuelve el extremo que este disponible, puede ser A (`Extreme::a`), B (`Extreme::b`) o ninguno (`Extreme::none`) en caso que no haya disponibilidad.

- `extremes`: método concreto que dado un modelo de slot devuelve a que extremos corresponde, puede corresponder a ambos extremos en el caso que esté conectado consigo mismo (por ejemplo en un estado de máquina de estado) o el conjunto vacío en caso que no se contenga ese slot.

- `slot`: método concreto que dado un extremo devuelve el slot correspondiente, vacío (`nullptr` para punteros en C++) en caso de estar desconectado ese extremo.

- `connectExtreme`: método concreto que dado un modelo de slot en caso de ser posible, al utilizar `connectParticular`, se conecta y devuelve el extremo al que fue conectado, en caso de no haber sido posible devuelve el extremo con valor ninguno (`Extreme::none`).

- `connect`: método concreto que dado uno o dos modelos de slot realiza la conexión utilizando `connectExtreme`, devuelve verdadero en caso de haber

realizado la conexión correctamente o falso en caso contrario.

- `connectParticular`: método abstracto que realiza la conexión de dos modelos de slot particulares, devuelve verdadero en caso de haber realizado la conexión correctamente o falso en caso contrario.

- `disconnectParticular`: método abstracto que realiza la desconexión de un modelo de slot particular, devuelve verdadero en caso de haber realizado la desconexión correctamente o falso en caso contrario.

Se observa también en la Fig. 4 las clases ya mencionadas `ConnectionEntityModel` y `ConnectionStateModel`. Se procede a detallar cada una de ellas:

- `ConnectionEntityModel`: clase que internamente contiene un vinculo VHDL, el mismo puede generar una señal VHDL o ser una conexión directa con un puerto de entrada o salida dependiendo de factores internos particulares de VHDL que veremos en la sección Resultados con mayor detalle.

- `ConnectionStateModel`: clase que internamente contiene una transición de maquina de estados. Más adelante se especificará como está compuesta una máquina de estados y como se transforma la misma a VHDL.

II-C. Secuencia de operaciones en una conexión

Para realizar una conexión se debe utilizar la interfaz visual de manipulación directa, clickeando un slot (o espacio de conexión), se genera un evento que es capturado por el framework visual Qt [14] para comenzar creando una conexión a través de la llamada que ya hemos visto `makeConnection` (ver Fig. 3). Luego se debe arrastrar la conexión creada hasta soltar sobre un segundo slot. Los pasos que realiza el sistema al soltar la conexión se detallan a continuación y se visualizan en el diagrama de secuencia de la Fig. 5:

El método `1:mouseReleaseEvent()` es llamado por el Framwork Qt cuando el usuario suelta el objeto visual `ConnectionView`. Luego en la llamada al método `2:getSlotEdit()` que se refiere a la vista del slot (objeto de la clase `SlotView`) sobre el que se encuentra el usuario al soltar el objeto visual de conexión `ConnectionView`; se debe tener en cuenta que al soltar la conexión esta debe unir dos slots, con lo cual la llamada `2:getSlotEdit()` devuelve el segundo slot a conectar; tener en cuenta que el primer slot es almacenado por el modelo de conexión al principio en la creación de la misma. Al objeto retornado `SlotView` se le asigna un nombre

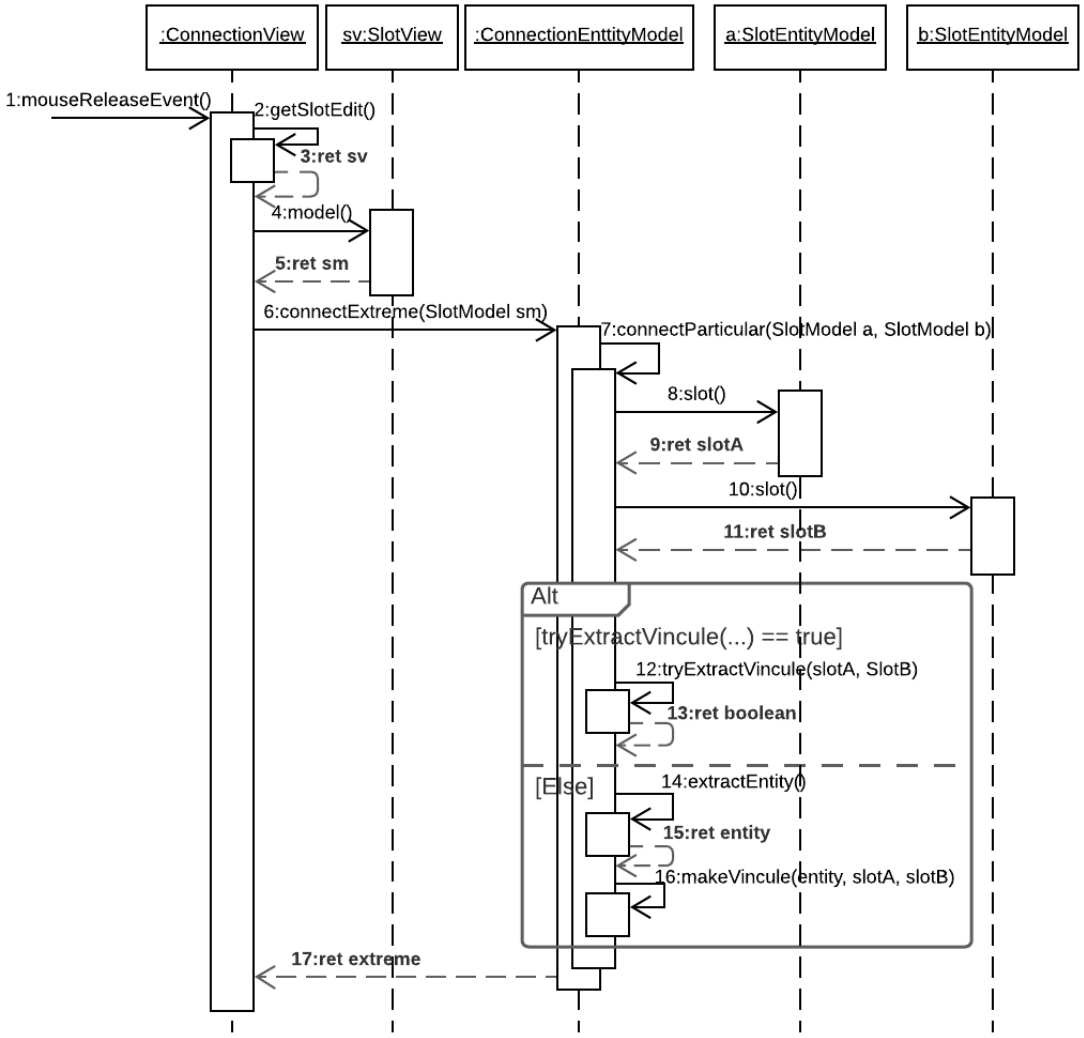

Figura 5: Diagrama de secuencia para la finalización en la realización de una conexión de puertos VHDL.

local `sv`. A través de la llamada al método `4:model()`, se retorna el modelo del `SlotView`, que es un objeto del tipo `SlotModel` que es asignado a un nombre local `sm`.

Con la llamada al método `6:connectExtreme` se pueden observar las propiedades que brinda la Programación Orientada a Objetos en C++: el objeto que representa la vista de conexión (instancia de la clase `ConnectionView`), solo conoce su modelo de datos como un objeto de tipo abstracto `ConnectionModel` y no como su instancia particular verdadera, en este caso la clase concreta del tipo `ConnectionEntityModel` (ver Fig. 4), con lo cual al llamar a `6:connectExtreme` este se implementa verdaderamente en la clase particular `ConnectionEntityModel`. Obsérvese que si el objeto del modelo de conexión (`ConnectionModel`) hubiera sido en verdad una instancia del tipo concreto `ConnectionStateModel`, la funcionalidad hubiera sido completamente diferente (relacionada a una máquina de estados).

Al método `6:connectExtreme` se le pasa como parámetro el objeto `sm` correspondiente a un `SlotModel`; como se ha explicado anteriormente `ConnectionEntityModel` ya cuenta con el otro modelo de slot `SlotModel` almacenado del momento de que el usuario comenzó la conexión visualmente; se llama a método `7:connectParticular` con ambos modelos de slots, estos deben ser del tipo `SlotEntityModel` para poder hacer la conexión. En caso de ser así, se procede a extraer los slots de conexión VHDL específicos `vhdl::Slot`, a través de la llamada `8:10:slot` asignándolo localmente a slotA y slotB respectivamente.

De estos slots se intenta extraer un vinculo (objeto de la clase `vhdl::Vincule`) y realizar la conexión; en caso de no poder se crea un nuevo vinculo con ambos slots, llamando en principio a la función `extractEntity()` la cual devuelve la entidad VHDL actual (objeto de la clase `vhdl::Entity`) sobre la cual se está editando, se almacena localmente con el nombre `entity`. Se devuelve el extremo de la conexión sobre el cual se ha realizado la operación: `Extreme::a`, `Extreme::b` o `Extreme::none` si no se puedo realizar.

II-D. Validaciones

En el núcleo del sistema se encuentran las validaciones que se realizan en los nombres de los componentes, para ello los mismo pertenecen a un ámbito `Scope` (ver Fig. 6). Se puede observar que, el ámbito esta compuesto por múltiples elementos, la estructura de datos interna de la del ámbito `Scope` es una estructura de datos `std::unordered_map` que contiene una clave de búsqueda eficiente y un valor, la clave será la cadena de caracteres de identificación dentro de ese ámbito, denominada en el sistema `uid` (Unique Identifier), y el valor será el objeto del elemento en particular (objeto del tipo `ScopeElement`).

Al ingresar cualquier objeto que herede de `ScopeElement` (como puede ser `vhdl::Entity`, `vhdl::Component` o `vhdl::Port` entre otros), se verificara si su nombre es único dentro del ámbito correspondiente, o sea al objeto `ScopeContainer` al cual se lo ingresa. De la misma forma se observa que

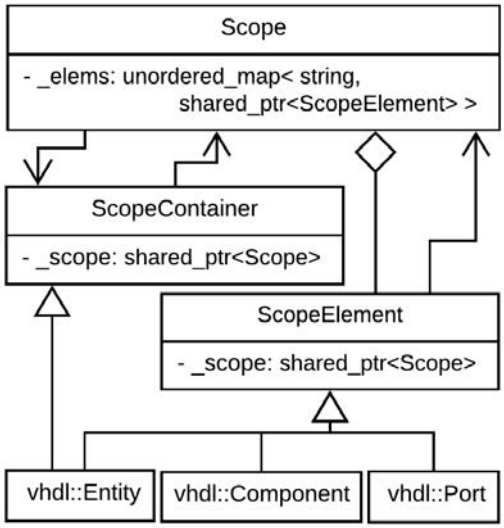

Figura 6: Diagrama del Sistema.

el objeto que contiene un ámbito `ScopeContanier` y el elemento del ámbito `ScopeElement`, contienen un atributo que es un puntero al ámbito `Scope`. Esto permite que al actualizarse el nombre de una elemento del ámbito se genere una validación inmediata del mismo. Se puede observar que `Entity` también hereda de `ScopeContainer` ya que una entidad tiene un ámbito de elementos.

Se realizan otras validaciones, como la correctitud en la construcción de la estructura de la máquina de estados, o la equidad en la cantidad de conexiones que se realizan a un puerto de un componente. Actualmente no se puede verificar si el código VHDL que se ingresa a mano (por ejemplo en las máquinas de estado) es válido.

III. Resultados

III-A. Conexión de componentes

En la Fig. 7 se observa dos entradas de la entidad padre conectadas a las entradas de un subcomponente. En la funcionalidad referente a VHDL se verifica si es necesario o no generar una señal interna para realizar un conexión. Para éste caso no es necesario generar una señal interna.

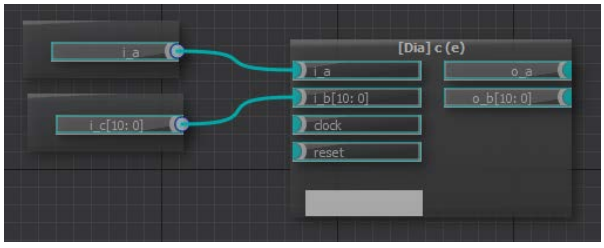

Figura 7: Conexión a entradas de componente.

En caso de que se realizen conexiones de uno a múltiples slots, como se ve en la Fig. 8, automáticamente se generará la inserción de código de la señal interna junto a su uso en la definición de mapeo de puertos VHDL.

De igual forma, se creará una señal interna VHDL, para conectar puertos de dos componentes internos, como se puede observar en la Fig. 9.

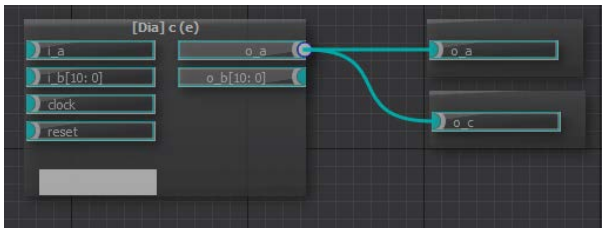

Figura 8: Conexión a múltiples salidas.

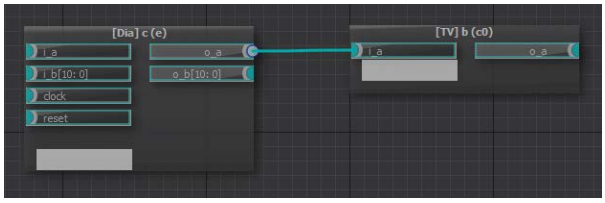

Figura 9: Conexión entre dos componentes.

III-B. Tablas de Verdad

Con Espresso es posible, a partir de tablas de verdad, generar funciones lógicas normalizadas. Estas funciones son devueltas en un formato diferente a VHDL, con lo cual es necesario que se transformen a través del uso de la herramienta de expresiones regulares. En la Fig. 10 se observa parte de la ventana de edición de una tabla de verdad; luego, en Fig. 11, se observa su correspondiente código para la generación VHDL.

Entradas (Autocompletar) - Salidas -

	g[1]	g[0]	f[0]	f[1]
0	0	0	1	0
1	0	1	0	1
2	1	0	0	1
3	1	1	0	1

Figura 10: Editor de tabla de verdad.

III-C. Inserción de código VHDL

Es posible también que se seleccione un archivo VHDL que contenga una Entidad para ser cargada en la interfaz

```
ENTITY f1 IS
   PORT (
      g: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
      f: OUT STD_LOGIC_VECTOR(0 TO 1)
   );
END ENTITY f1;

ARCHITECTURE arch OF f1 IS
BEGIN
  f[0] <= (NOT(g[1]) AND NOT(g[0]));
  f[1] <= (g[1]) OR (g[0]);
END ARCHITECTURE arch;
```

Figura 11: Código VHDL resultante de la tabla de verdad

```
entity FullAdder is
   generic (
      n: integer := 8 );
   port (
      cin: in std_logic;
      a: in std_logic_vector( n-1 downto 0 );
      b: in std_logic_vector( n-1 downto 0 );
      r: out std_logic_vector( n-1 downto 0
         );
      cout: out std_logic );
end FullAdder;
```

Figura 12: Código VHDL a importar

gráfica: el nombre, los atributos genéricos y los puertos de entrada y salida se verán gráficamente. Se puede observar la cabecera de una Entidad VHDL en Fig 12. Luego, en la Fig. 13 se observa el resultado en la interfaz al importar el archivo y la edición del valor genérico `n`, asignándole `16`, y modificando así los extremos máximos de los vectores `a`, `b` y `r`; siendo los mismos `n-1`.

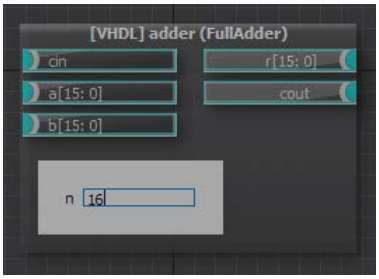

Figura 13: Componente creado en la importación Listing.12

III-D. Máquinas de estado y el lenguaje genHDL

Por otro lado, se puede realizar la creación de máquinas de estado desde la interfaz gráfica. De éste modo, la creación de máquinas de estado es facilitada ampliamente en comparación al uso exclusivo de VHDL con el mismo fin. Para la máquina de estado de la Fig. 14, en la aplicación se debió generar visualmente lo que se ve en la Fig. 15.

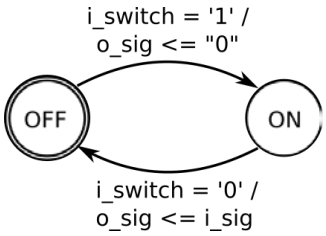

Figura 14: Máquina de estado a representar

Internamente se crea un código intermedio denominado genHDL (ver Fig. 16), este es útil para representar las máquinas de estado, cuenta con puntos de inserción VHDL. A partir de éste código intermedio se genera el código VHDL resultante de la máquina de estados, como se ve en la Fig. 17.

III-E. Nodos jerárquicos

Es posible utilizar nodos jerárquicos en el sistema, de tal forma de ir recorriendo el proyecto fácilmente entre los

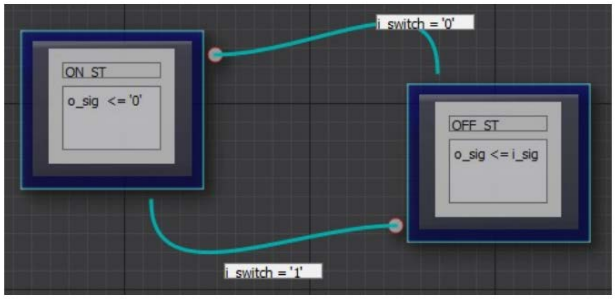

Figura 15: Máquina de estado a representar

```
statemachine sm_switch (clock: i_clk, reset:
    i_rst)
   state ON_ST
      %vhdl{
         o_sig <= '0'
      }
      transition
         with ( %vhdl{ i_switch = '0' } )
            to OFF_ST

   state OFF_ST
      %vhdl{
         regfin <= '1'
         o_sig <= i_sig
      }
      transition
         with ( %vhdl{ i_switch = '1' } )
            to ON_ST
```

Figura 16: Código parcial intermedio genHDL

subnodos y los nodos padre.

IV. Conclusiones

Se ha logrado crear un lenguaje de programación visual con algunas ventajas y potencialidades con respecto a otros lenguajes visuales. Entre sus mayores ventajas se encuentra la simplicidad de su utilización y la diversidad de herramientas con las que cuenta. También la abstracción en forma de framework visual que además de poder utilizarse para este

```
sm_switch: process( i_clk, i_rst )
begin
  if i_rst = '1' then
    state <= ON_ST;
  elsif i_clk'event and i_clk = '1' then
    case state is
      when ON_ST =>
        if i_switch = '0' then
          state <= OFF_ST;
        end if;
      when OFF_ST =>
        if i_switch = '1' then
          state <= ON_ST;
        end if;
    end case;
  end if;
end process;
```

Figura 17: Código parcial VHDL generado

proyecto, puede ser utilizado para otros sistemas que así lo requieran, siendo mucho más flexible que su predecesor NodeEditor.

La inserción de la representación intermedia genHDL brindó facilidades a la hora de interpretar textualmente y generar máquinas de estado. Brindando la posibilidad de ser comparado con el código VHDL resultante. De ésta forma es posible exponer desde el concepto visual hacia el textual simplificado, y de allí al especifico en VHDL. Se busca de ésta forma mejorar el aprendizaje de los conceptos de construcción de máquinas de estado en el diseño digital.

El lenguaje intermedio genHDL cuenta con mayor expresividad que únicamente la de máquinas de estado, teniendo la posibilidad de representar componentes, conexiones entre ellos y tablas de verdad fácilmente; con lo cual se está evaluando el uso de genHDL para realizar el almacenamiento de un proyecto entero FlowHDL.

Aún resta realizar mejoras para contar con una mayor completitud en términos de expresividad del lenguaje visual, como por ejemplo la generación de hardware dinámico a través de parametros genéricos; así como también una validación en los puntos de inserción de código VHDL (y posiblemente en el futuro Verilog). Interactuando con el compilador de código abierto Yosys [15] se pueden lograr ésta última mejora al sistema, ya que es posible desde Yosys generear descripciones JSON y FlowHDL tiene la posibilidad de recibir este tipo de entradas.

Referencias

[1] J. R. L. Vizcaíno and J. P. Sebastiá, *LabVIEW: Entorno gráfico de programación*. Marcombo, 2011.

[2] vortexmakes, "Rkh: State machine framework for reactive embedded systems," https://github.com/vortexmakes/RKH, 2019.

[3] J. Arroyo, "icestudio: Experimental graphic editor for open fpgas," https://github.com/bqlabs/icestudio, 2016.

[4] D. Gajski, T. Austin, and S. Svoboda, "What input-language is the best choice for high level synthesis (hls)?" in *Design Automation Conference*, 2010, pp. 857–858.

[5] M. Haldar, A. Nayak, N. Shenoy, A. Choudhary, and P. Banerjee, "Fpga hardware synthesis from matlab," in *VLSI Design 2001. Fourteenth International Conference on VLSI Design*, 2001, pp. 299–304.

[6] K. Takano, T. Oda, and M. Kohata, "Design of a dsl for converting rust programming language into rtl," in *International Conference on Emerging Internetworking, Data & Web Technologies*. Springer, 2020, pp. 342–350.

[7] G. Wang, H. Lam, A. George, and G. Edwards, "Performance and productivity evaluation of hybrid-threading hls versus hdls," in *2015 IEEE High Performance Extreme Computing Conference (HPEC)*, 2015, pp. 1–7.

[8] K. Marriott, B. Meyer, and K. B. Wittenburg, "A survey of visual language specification and recognition," in *Visual language theory*. Springer, 1998, pp. 5–85.

[9] B. A. Myers, "Taxonomies of visual programming and program visualization," *Journal of Visual Languages & Computing*, vol. 1, no. 1, pp. 97–123, 1990.

[10] M. Fayad and D. C. Schmidt, "Object-oriented application frameworks," *Commun. ACM*, vol. 40, no. 10, p. 3238, Oct. 1997. [Online]. Available: https://doi.org/10.1145/262793.262798

[11] R. L. Rudell, "Multiple-valued logic minimization for pla synthesis," California Univ Berkley Electronics Research Lab, Tech. Rep., 1986.

[12] D. P. et al, "Qt5 node editor," https://github.com/paceholder/nodeeditor, 2017.

[13] G. Kellogg, P.-A. Champin, and D. Longley, "JSON-LD 1.1 – A JSON-based Serialization for Linked Data," W3C, Technical Report, Dec. 2019. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02141614

[14] G. Lazar and R. Penea, *Mastering Qt 5*. Packt Publishing Ltd, 2016.

[15] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. Milanovic, "Yosys+nextpnr: an open source framework from verilog to bitstream for commercial fpgas," *CoRR*, vol. abs/1903.10407, 2019. [Online]. Available: http://arxiv.org/abs/1903.10407

APÉNDICE A
NODOS

Se presenta a continuación la especificación de la clase que representa los nodos. En la Fig. 18 se puede observar la clase abstracta `NodeModel`, nodos con sus componentes conectables: éstos pueden ser puertos de entradas y salidas en un componente VHDL, o directamente un nodo que tiene la posibilidad de ser conectado a otro, como en un estado de una máquina de estados (sabiendo que ésta conexión representa una transición); se procede a detallar algunos de sus métodos concretos y abstractos, en estos últimos, la funcionalidad se define por las clases concretas heredadas:

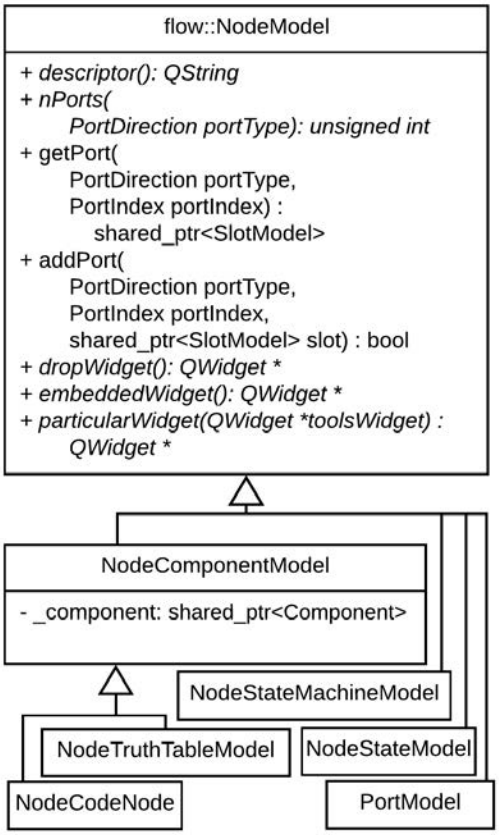

Figura 18: Diagrama de clases para el modelo de nodos.

- `descriptor`: método abstracto que se ocupará en devolver el nombre visible en el sistema visual.

- `nPorts`: método abstracto que dado una dirección de puerto (como puertos de entrada o de salida), da la cantidad de puertos con la cual se cuenta.

- `getPort`: método concreto con el cual dado una dirección y una posición (valor numérico entero) devuelve, de ser posible un modelo de una ranura o slot (espacio de conexión abstracto).

- `setPort`: método concreto con el cual dado una dirección, una posición (valor numérico entero) y el modelo de ranura (slot) lo agrega al nodo, se retorna un valor booleano (verdadero o falso) con lo cual se puede verificar si fue posible agregar el modelo del slot.

- `dropWidget`: método abstracto que retorna la funcionalidad visual habitual (widget o artilugio) que se muestra en una ventana al momento de ingresar un objeto visual nuevo.

- `embeddedWidget`: método abstracto que retorna el widget que se muestra dentro del nodo visual. Por ejemplo en un nodo VHDL este widget se mostrará por cada elemento genérico de la Entidad, pudiendo editar el valor de estos elementos.

- `particularWidget`: método abstracto que retorna el widget que se despliega al querer editar el nodo.

Se observa también en la Fig. 18 que `NodeTruthTableModel`, `NodeStateModel` y `PortModel` son clases concretas que heredan directamente de `NodeModel`; además `NodeComponentModel` también hereda de `NodeModel` pero sigue siendo abstracta ya que `NodeCodeModel` y `NodeStateMachineModel` implementan los métodos abstractos restantes para ser clases concretas. Se procede a detallar cada una de ellas:

- `NodeComponentModel`: clase que internamente agrega un componente proveniente de una entidad VHDL.

- `NodeTruthTableModel`: representa un nodo que internamente contiene una tabla de verdad. El método `embeddedWidget` que implementa ésta clase devuelve el widget necesario para editar la tabla de verdad.

- `NodeCodeModel`: representa un nodo que internamente contiene una entidad VHDL, generando así su interfaz de puertos y elementos genéricos. Esta clase hereda directamente, al igual que `NodeTruthTableModel`, de `NodeComponentModel` ya que también agrega un componente que utiliza la entidad VHDL como base.

- `PortModel`: representa un puerto de entrada o salida de la entidad jerárquica que se está editando en ese momento.

- `NodeStateMachineModel`: representa un nodo jerárquico que contiene una máquina de estados.

- `NodeStateModel`: representa un estado de una maquina de estados, solo puede aparecer dentro de un nodo jerárquico de tipo `NodeStateMachineModel`. La conexión entre dos de estos nodos corresponde a la semántica de cambio de estados; con lo cual este nodo solo contiene un espacio de conexión sin dirección (como se verá próximamente tendrá un `SlotModel` con `PortDirection` igual a `PortDirection::none`).