UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

APLICACIÓN WEB PARA DISEÑAR BASES DE DATOS

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

MATÍAS ARTURO LÓPEZ GALLARDO

PROFESOR GUÍA:
AIDAN HOGAN

PROFESOR CO-GUÍA:
SEBASTIÁN FERRADA ALIAGA

MIEMBROS DE LA COMISIÓN:
WILLY MAIKOWSKI CORREA
MATÍAS TORO IPINZA

SANTIAGO DE CHILE
2024

Resumen

Los diagramas entidad-relación (ER) son una herramienta fundamental en el diseño de bases de datos. Permiten visualizar la estructura, restricciones y entidades participantes del modelo conceptual de una base de datos.

En la actualidad, la mayoría de las herramientas disponibles en la web para crear diagramas ER son de pago, o bien, no poseen todas las características adecuadas para crear diagramas de forma sencilla. Esto se ve reflejado en la dificultad que tienen los alumnos, del curso Bases de Datos del Departamento de Ciencias de Computación de la Universidad de Chile, para crear diagramas ER adecuados, sin errores.

Con esta problemática en mente, y con el objetivo de facilitar la creación de diagramas ER, en la presente memoria se desarrolló un lenguaje de marcado, denominado ERdoc, para representar modelos ER. Se implementó un parser para la gramática del lenguaje junto a un sistema de detección de errores, tanto sintácticos como semánticos.

Adicionalmente, se desarrolló ERdoc Playground, una aplicación web, gratuita y de código abierto, que permite crear diagramas ER por medio del lenguaje ERdoc. Esta aplicación genera los diagramas en tiempo real, y además, soporta tres notaciones comúnmente utilizadas. Junto a la aplicación, se implementó un algoritmo de *layout* de grafos que intenta posicionar los elementos del diagrama ER de forma visualmente atractiva. Se verificó que este algoritmo es rápido para diagramas de hasta 139 entidades, 20 relaciones y 5 agregaciones, permitiendo su ejecución cada vez que una entrada del usuario modifica el diagrama.

Se evaluaron dos aspectos de ERdoc Playground: la usabilidad y el rendimiento de la aplicación. Se verificó que la aplicación es lo suficientemente rápida para generar diagramas en tiempo real. Además, se obtuvo una buena evaluación de la usabilidad por parte de los usuarios. Así, se concluye que se cumplieron los objetivos planteados, obteniéndose una aplicación web funcional, que se encuentra disponible para ser utilizada por estudiantes.

Dedicado a mis padres.

Agradecimientos

Agradezco a mi familia. A mis padres, Katiuska y Mauricio, por todo su esfuerzo para que nunca me faltara nada. A mi abuela, Patricia, por todo el cariño que me ha entregado a lo largo de mi vida.

A mis profesores guía, Aidan Hogan y Sebastián Ferrada, por todos sus consejos, disposición y apoyo durante el desarrollo de este trabajo.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

En la actualidad la mayoría de sistemas informáticos utilizan bases de datos para administrar y almacenar información. Una de las primeras etapas en el diseño de una base de datos corresponde al modelado conceptual, etapa en la cual se desarrolla un modelo de alto nivel que satisface los requerimientos del problema a resolver.

Una de las opciones existentes para realizar modelos conceptuales y diseñar la estructura de una base de datos es el modelo entidad-relación (ER), propuesto originalmente por Peter Chen [3], que representa la estructura de un modelo de datos como un conjunto de entidades y la forma en que estos interactúan a través de relaciones. Para representar gráficamente el modelo ER se utilizan los diagramas ER. Estos resultan útiles en las primeras etapas del diseño de una base de datos, ya que permiten la visualización de la estructura del modelo de datos de una forma clara y estandarizada.

A pesar de la importancia del modelo ER, se ha observado que los estudiantes del curso CC3201 Bases de Datos, del Departamento de Ciencias de la Computación de la Universidad de Chile, tienen dificultades para comprender y crear correctamente diagramas ER al momento de enfrentarse a este problema en cursos posteriores o prácticas profesionales a pesar de que el modelo ER (junto con los diagramas ER) es parte de los contenidos del curso CC3201.

Por otro lado, en la web existen varias aplicaciones para generar diagramas ER, pero la mayoría de ellas son de pago y las que son gratuitas tienen limitaciones en cuanto a su funcionalidad, puesto que gran parte de estas herramientas fueron diseñadas para crear diagramas de propósito general, por lo que carecen de características específicas al diseño de diagramas ER. De esta forma, se identifica la poca disponibilidad de herramientas gratuitas en la web para diseñar modelos ER a través de diagramas, por lo que se propone que una de las posibles causas de las dificultades de los estudiantes es que están usando las herramientas inadecuadas para crear diagramas ER.

Del mismo modo, se observa la ausencia de lenguajes, o formatos de texto, que permitan representar modelos ER en un formato semi-estructurado de forma directa. Las soluciones existentes entregan representaciones demasiado verbosas y no soportan todas las características del modelo ER. Desarrollar un lenguaje que permita representar el modelo ER con todas

sus características permitiría, por ejemplo, diagnosticar problemas en un modelo a través de errores semánticos, aportando así a la comprensión de los conceptos del modelo ER.

Finalmente, se propone diseñar e implementar una aplicación que permita a sus usuarios crear diagramas ER, por medio de un lenguaje a desarrollar, con el fin de facilitar la creación de diagramas ER para los estudiantes, evitar errores de notación y así mejorar la comprensión del modelo ER.

1.1. Objetivos

Objetivo General

El objetivo general de la memoria es desarrollar una aplicación web gratuita, de código abierto, que permita crear diagramas ER interactivos a través de un lenguaje de marcado, utilizando un lenguaje de definición propio. La herramienta debe utilizar la notación de Chen para visualizar los diagramas.

Objetivos Específicos

1. Diseñar e implementar un lenguaje de marcado que permita representar modelos ER en forma de texto de manera precisa.
2. Implementar un parser del lenguaje de marcado para lograr que la aplicación genere un diagrama ER a partir de la sintaxis definida.
3. Implementar un corrector de sintaxis que le pueda mostrar al usuario los errores sintácticos en su definición del modelo ER.
4. Implementar un corrector en la semántica del modelo definido por el usuario, que pueda detectar y destacar problemas, como una relación que referencia una entidad que no existe, una entidad sin una llave o sin atributos, entre otros errores que el modelo ER no permite.
5. Desarrollar una aplicación web a través de la cual los usuarios puedan ingresar sus modelos ER, descritos en el lenguaje de marcado desarrollado para este propósito, y puedan visualizar los diagramas respectivos.
6. Extender la aplicación web para que los usuarios puedan interactuar con los diagramas generados, por ejemplo, puedan arrastrar los elementos para moverlos de posición, cambiar el tamaño de los elementos, etc.
7. Para el final de la memoria, dejar la aplicación disponible en un ambiente de producción para que los usuarios puedan acceder a ella. Además dejar disponible el código fuente junto con su documentación en un repositorio público para que otros desarrolladores puedan contribuir al proyecto.

1.2. Estructura de la memoria

En la presente memoria se presenta el trabajo realizado para cumplir los objetivos anteriores. La memoria se estructura en 6 capítulos:

- En el Capítulo 2 se introducen los conceptos relacionados al trabajo realizado: el modelo ER, diagramas ER y otros conceptos utilizados durante el trabajo. Adicionalmente, se presentan las soluciones existentes de aplicaciones web para crear diagramas ER y posibles lenguajes que se podrían utilizar para representar modelos ER.
- En el Capítulo 3 se presenta el lenguaje ERdoc, lenguaje de marcado para representar modelos ER que fue desarrollado durante esta memoria.
- En el Capítulo 4 se presenta ERdoc Playground, aplicación web desarrollada durante esta memoria, su funcionamiento y sus principales características.
- En el Capítulo 5 se presenta la evaluación de dos aspectos de ERdoc Playground: su funcionamiento y su usabilidad.
- En el Capítulo 6 se discuten las conclusiones del trabajo. También, se presenta un resumen del trabajo realizado, se revisa el cumplimiento de los objetivos propuestos y posibles trabajos futuros.

Capítulo 2

Estado del Arte

En este capítulo se describe el modelo entidad-relación (ER) junto a sus conceptos, además se introducen los diagramas ER y dos de las notaciones más utilizadas en la actualidad para realizar los diagramas. Luego, se presentan las soluciones existentes en la web para crear diagramas ER y se realiza una comparación entre las opciones actuales para representar modelos ER por medio de lenguajes. Por último, se revisan algoritmos de dibujo de grafos y el concepto de *Parsing expression grammar*, un tipo de gramática utilizada durante el presente trabajo.

2.1. Modelo Entidad Relación

El modelo entidad-relación (ER), propuesto originalmente por Peter Chen [3], es un modelo conceptual para representar la estructura de un modelo de datos en términos de entidades y relaciones. Hoy en día, el modelo ER se usa principalmente para diseñar modelos de datos que logren capturar los requerimientos que un sistema debe satisfacer.

2.1.1. Conceptos del modelo ER

Entidad: Chen [3] define a una entidad como una "cosa" que puede ser identificada de manera distintiva. Por ejemplo, en un modelo ER podrían existir las entidades "Estudiante" y "Universidad". Esta definición resulta bastante amplia, ya que dependerá mucho del dominio del problema que se quiera modelar. Las entidades se caracterizan por sus atributos, así dos entidades serán distintas si tienen al menos un atributo distinto.

Una instancia de una entidad corresponde a la asignación de valores específicos a los atributos de dicha entidad; los atributos que permiten diferenciar las instancias de una entidad se conocen como atributos identificativos o llaves.

Relación: En el modelo ER, una relación corresponde a una asociación entre entidades necesaria para representar bien el problema que se está modelando; en una relación pueden participar una o más entidades. Las relaciones también tienen atributos, pero no identificativos. Las relaciones con dos entidades participantes se conocen como relaciones binarias; en general, una relación con N participantes se conoce como relación n-aria.

En un modelo de datos pueden existir restricciones en la forma en que interactúan las distintas entidades. Esto se representa a través de restricciones de cardinalidad y participación en las relaciones [6]: la cardinalidad de una relación indica el número máximo de instancias de una relación en las que puede participar una entidad. Por ejemplo, supongamos la relación "Estudia_en" en la que participan las entidades "Estudiante" con cardinalidad 0 o 1 y "Universidad" con cardinalidad N. Esto es equivalente a decir que un estudiante no puede pertenecer a más de una universidad y que en una universidad pueden estudiar varios estudiantes.

A su vez, está la restricción de participación, la que especifica si una entidad puede existir sin participar de una relación. En nuestro ejemplo, si nuestro modelo de datos requiere que cada Universidad debe tener al menos un estudiante, esto se vería reflejado en el modelo ER diciendo que la entidad Universidad tiene participación total en la relación Estudia_en, esto es que toda instancia de Universidad debe participar en una instancia de la relación Estudia_en. Por el otro lado, está la participación parcial, que significa que una instancia de una entidad puede existir sin participar en la relación.

El modelo ER planteado por Chen ha sido extendido con varios conceptos que permiten una representación más precisa de los datos a modelar. En la actualidad se usa el modelo ER con varias de sus extensiones. A continuación se revisarán algunas de estas extensiones.

2.1.2. Extensiones al modelo ER

Agregación: La agregación es una forma de agrupar a una relación y a sus entidades participantes en una nueva entidad, conocida como entidad de alto nivel o entidad virtual. El concepto de agregación fue introducido por Smith y Smith [13].

Jerarquía de clases: Se han aplicado los conceptos del paradigma de programación orientada a objetos, herencia y jerarquía de clases, al modelo ER. De esta forma, se tiene la relación "isA" que indica que una entidad es una especialización de otra. Se dice que una subclase hereda de una superclase cuando se tienen dos entidades relacionadas por isA. Además, la subclase heredará los atributos de la superclase.

Entidades débiles: Una entidad débil corresponde a una entidad que no tiene atributos identificativos, por lo que su llave dependerá de la llave de otra entidad. La relación entre una entidad débil y su entidad identificativa se conoce como relación débil.

2.2. Diagramas ER

Los diagramas ER son una forma visual de representar modelos ER. A lo largo del tiempo se han desarrollado distintas notaciones para representar los conceptos del modelo ER. A continuación se presentan dos notaciones que se siguen usando en la actualidad:

Notación de Chen: La notación de Chen es la notación original, propuesta en [3]; en esta notación, las entidades se representan como rectángulos y las relaciones como rombos; para mostrar que una entidad participa en una relación, se unen los símbolos con una línea. Los atributos de una entidad se representan como óvalos agrupados cerca de la entidad. Los atributos identificativos van subrayados. Las restricciones de cardinalidad se representan como una letra o número cercano posicionado en la entidad opuesta (notación *look across*), mientras que las restricciones de participación se representa como un círculo con su interior vacío (participación parcial) o relleno de negro (participación total). En la figura 2.1 se ejemplifica un diagrama ER que usa la notación de Chen. En este diagrama la entidad Estudiante tiene participación parcial y cardinalidad 1. La entidad Universidad tiene cardinalidad N y participación total en la relación.

La notación de Chen ha sido extendida para soportar:

- Entidades débiles y la relación en la que participan, que se denotan con sus respectivos símbolos pero con doble borde. Los atributos de una entidad que corresponden a llaves parciales se subrayan con una línea punteada.
- Agregación. Las entidades virtuales se representan encerrando la relación que encapsula y sus entidades participantes dentro de un rectángulo.

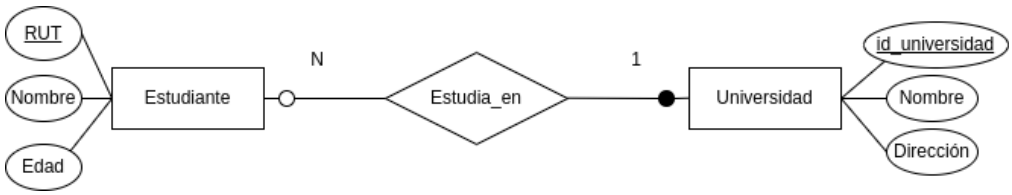

Figura 2.1: Diagrama ER que usa la notación de Chen

Song et al. [14] realizaron una comparación entre 10 distintas notaciones existentes en la época (1995) y categorizaron las distintas notaciones según el tipo de relaciones que permiten y en la forma en que se representan las restricciones de cardinalidad. De estas, la notación de Chen se encuentra dentro del grupo de las que soportan relaciones n-arias y es de las notaciones que más se encuentran en la actualidad en la literatura y en recursos en la Web. De hecho, la notación de Chen es la que se usa en el curso Bases de Datos (con algunas modificaciones) cuando se enseñan los diagramas ER. Por lo que se propone usar la notación de Chen en la herramienta a desarrollar.

Notación Crow's Foot: Otra notación que se encuentra en varios recursos en la Web es la notación "*Crow's Foot*", observada por primera vez en [7]; aquí las entidades se representan

mediante un rectángulo en cuya parte superior va el nombre de la entidad. Dentro del mismo rectángulo se anotan los atributos, las llaves primarias se identifican con un asterisco. Las relaciones se representan simplemente como una línea que une a las relaciones. Para representar las restricciones de cardinalidad, se ocupa una línea perpendicular a la de la relación para la cardinalidad de 1. Para la de N se agregan dos líneas al final de la relación. Los símbolos usados para mostrar las restricciones de participación son un círculo vacío para la participación parcial, mientras que para la participación total se usa una línea vertical. En la figura 2.2 se muestra el mismo diagrama ER que en la figura 2.1, pero esta vez utilizando la notación *Crow's Foot*.

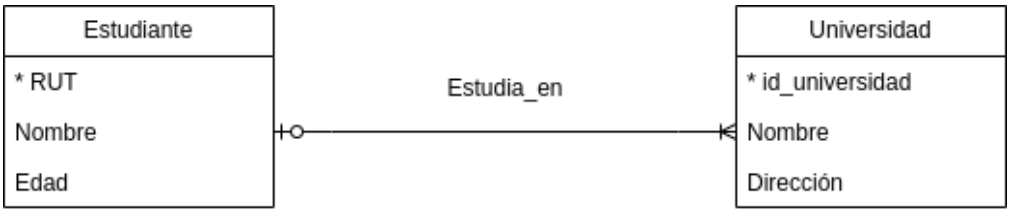

Figura 2.2: Diagrama ER usando la notación *Crow's Foot*. Estudiante tiene participación parcial y cardinalidad 1. Universidad tiene participación total y cardinalidad N.

La notación *Crow's Foot* no soporta relaciones n-arias ni los conceptos de entidad débil y agregación, por lo que no es una opción para realizar diagramas ER con requerimientos más complejos.

2.3. Soluciones Existentes

Hoy en día existen varias soluciones para crear diagramas de modelos de bases de datos, la mayoría disponibles como aplicaciones web; sin embargo, estas no son gratuitas o poseen limitaciones. Una de las herramientas más populares para diseñar bases de datos es *dbdiagram.io*[1], no obstante, esta aplicación no usa el modelo ER, por lo que no es útil para cuando es necesario crear diagramas ER. Además, permite crear solo 10 diagramas por usuario para luego solicitar una suscripción mensual y no es de código abierto. Una herramienta similar a *dbdiagram.io* es la librería de javascript *Mermaid*[2], que permite generar diagramas y gráficos a partir de texto, mediante un lenguaje de marcado. Esta librería cuenta con un editor web que permite visualizar los diagramas a medida que se escribe el código; *Mermaid* cuenta con una sintaxis para generar diagramas ER, pero esta solo permite representar entidades y relaciones binarias, no así relaciones n-arias, agregación, jerarquía ni entidades débiles. Por otro lado, la notación usada al generar los diagramas ER corresponde a la notación *Crow's Foot*.

Otra herramienta comúnmente usada es *draw.io*[3], que es una herramienta gratuita y de código abierto para crear diagramas de propósito general, lo que se refleja en la falta de funcionalidades específicas para modelar bases de datos y soporte parcial de símbolos de la

[1]https://www.dbdiagram.io
[2]https://mermaid.js.org/
[3]https://www.draw.io

notación de Chen. Finalmente, el resto de las herramientas disponibles en la web corresponden a aplicaciones de pago o siguen el modelo de ofrecer una cantidad limitada de diagramas gratuitos, por lo que no son una opción para los estudiantes.

En conclusión, ninguna de las aplicaciones web disponibles en la actualidad cumple con los requisitos de ser gratuita, de código abierto y usar el modelo ER siguiendo una notación estandarizada, en particular la notación de Chen.

2.4. Lenguajes para representar modelos entidad relación

Hoy en día existen varios lenguajes de programación y de marcado que se usan para representar modelos de bases de datos. Por otro lado, también existen formatos de texto para serializar datos estructurados; sin embargo, ninguna de estas opciones funcionan muy bien para representar específicamente modelos ER. A continuación se presentan algunas de las alternativas existentes, sus ventajas y desventajas:

2.4.1. DBML

DBML (*Database Markup Language*) es un lenguaje de marcado creado por la empresa Holistics [11]; este lenguaje es usado para definir la estructura de una base de datos siguiendo el modelo relacional. DBML permite definir esquemas, tablas, columnas de las tablas con sus tipos de datos, llaves primarias y llaves foráneas entre las tablas, junto con restricciones para estas: *one-to-many*, *many-to-one*, *many-to-many*, *one-to-one*. El principal uso que se le da al lenguaje DBML es en la aplicación web *dbdiagram.io*, donde es la forma en que los usuarios ingresan sus modelos de datos.

Sin embargo, dado que el lenguaje DBML utiliza conceptos del modelo relacional para representar bases de datos, no es posible describir el modelo ER de manera directa. Por lo que para usar DBML para representar el modelo ER, habría que crear una convención para hacer la diferencia entre entidades y relaciones, lo que dificultaría la comprensión del modelo ER.

2.4.2. SQL

SQL (*Structured Query Language*) es un lenguaje creado para modelar, administrar y realizar consultas en sistemas de bases de datos. Al igual que DBML, SQL está basado en el modelo relacional de datos por lo que tampoco permite representar directamente el modelo ER, y habría que definir un estándar para hacer la diferencia entre entidades y relaciones (por ejemplo, usar un esquema para entidades y otro para relaciones).

2.4.3. JSON

JSON (*Javascript Object Notation*) es un lenguaje pensado para el intercambio de datos estructurados. Su principal ventaja es que es sencillo de leer y escribir para humanos, a su vez es fácil de interpretar y generar por computadores. El lenguaje JSON permite estructurar datos por medio de pares llave/valor, comúnmente conocido como *objetos*, y listas ordenadas de valores. Aunque JSON no fue diseñado específicamente para representar la estructura de una base de datos, su estructura permite describir completamente un modelo ER, sin embargo, el resultado sería una representación bastante extensa y verbosa.

2.4.4. YAML

YAML es un lenguaje de serialización de datos comúnmente usado para crear archivos de configuración. YAML fue diseñado para ser legible por humanos y soporta tipos de datos como escalares, listas y objetos. Estas propiedades hacen posible representar modelos ER en YAML, pero al igual que JSON, la representación resulta verbosa y repetitiva.

2.4.5. Comparación

En el anexo A.1 se muestra la representación de un modelo ER consistente de dos entidades y una relación en los distintos lenguajes revisados. Se observa que las representaciones en JSON y YAML contienen demasiado texto repetido, ya que es necesario explicitar el nombre de cada campo, como *name* o *isKey*. Por el otro lado, en SQL y DBML se observa que para representar el modelo ER hay que razonar en términos del modelo relacional: esquemas, tablas y llaves foráneas, además de tener que especificar el tipo de dato al que pertenecen los atributos. Todos estos conceptos no existen en el modelo ER.

Finalmente, se identifica la necesidad de tener un lenguaje de marcado que permita describir el modelo ER de forma más directa.

2.5. Parsing Expression Grammar (PEG)

Las *parsing expression grammars*, introducidas por Ford en 2004 [8], corresponden a un tipo de gramática que describe una sintaxis a través de un conjunto de reglas siguiendo un modelo basado en reconocimiento, donde estas reglas deciden si el string está o no en la sintaxis definida. Una de las diferencias entre las PEG y las gramáticas libres de contexto (GLC) es que las GLC siguen un modelo generativo: su conjunto de reglas se aplica para generar strings del lenguaje.

Por otro lado, en las PEG el operador de elección es determinista y sigue un orden. Esto es, si se tiene la regla: $R = a|b$, esta primero intentará hacer match entre a y el string de entrada, si no es posible, intentará hacer match con b. Por el otro lado, en una GLC esta

misma regla podría generar dos strings distintos que pertenezcan al lenguaje. El eliminar la ambigüedad hace más fácil parsear expresiones que sigan una PEG. En la actualidad, el parser de la versión 3.10 del lenguaje de programación Python está basado en PEGs [10].

2.6. Dibujo de Grafos

El dibujo de grafos es un área de las ciencias de la computación que busca producir representaciones visuales de grafos que sean fácilmente entendibles y visualmente atractivos. Algunos de los criterios utilizados para evaluar la calidad de un dibujo de un grafo son: evitar cruces entre aristas, el dibujo muestra simetría, el largo de las aristas se mantiene constante, los nodos se distribuyen de manera uniforme en el dibujo, entre otros [1]. En general, optimizar estos criterios es un problema *NP-Hard* [1].

Así, se han desarrollado distintos algoritmos para dibujar grafos, que aplican heurísticas para satisfacer algunos de los criterios mencionados. A continuación se introducen algunos de los algoritmos:

2.6.1. *Force-directed Layout*

Los algoritmos de tipo *Force-directed layout* se basan en simulaciones físicas para lograr una visualización atractiva de un grafo. El algoritmo original, propuesto por Eades [5], simula que las aristas del grafo corresponden a resortes; mientras que para nodos que no sean adyacentes se aplican fuerzas de repulsión. Inicialmente, los nodos del grafo se disponen en una configuración inicial para luego comenzar la simulación hasta que las fuerzas de los resortes, junto a las fuerzas de repulsión, lleven al sistema mecánico (el grafo) a una configuración de mínima energía.

2.6.2. Algoritmo Radial

El algoritmo radial, propuesto por Eades [4], es específico para árboles, es decir, grafos conexos y sin ciclos. Este algoritmo organiza a los nodos del grafo de manera radial alrededor de un nodo raíz (elegido arbitrariamente). Los nodos que no son la raíz se posicionan en circunferencias concéntricas alrededor de la raíz. Los nodos con la misma altura se encuentran en la misma circunferencia.

2.6.3. Algoritmo de Sugiyama

El algoritmo de Sugiyama et al. [16] permite dibujar grafos dirigidos minimizando la cantidad de cruces entre aristas. El algoritmo hace que las aristas del grafo apunten todas en la misma dirección. Esto se logra organizando los nodos del grafo en capas, o jerarquías, para luego ordenar los nodos de cada capa, minimizando el cruce de aristas.

Capítulo 3

El lenguaje ERdoc

A continuación se presenta el lenguaje ERdoc, un lenguaje de marcado para representar modelos ER desarrollado durante este trabajo. Se describe la gramática del lenguaje, la implementación de un parser para el lenguaje y, finalmente, un sistema de detección de errores sintácticos y semánticos para este lenguaje.

El lenguaje ERdoc es un lenguaje de marcado, que se propone en este trabajo, para representar modelos ER (de acuerdo a la definición de Chen). Permite definir entidades, junto con sus propiedades:

- Nombre de la entidad.
- Atributos, incluyendo llaves, llaves parciales y atributos compuestos.
- Si es débil o no.
- Si hereda de otra entidad por jerarquía de clases.

Igualmente, permite definir relaciones y sus propiedades:

- Nombre de la relación.
- Atributos.
- Si identifica a una entidad débil o no.
- Entidades que participan en la relación.
- Entidades que participan en la relación a través de roles.
- Restricciones de cardinalidad.
- Restricciones de participación.

Además, permite definir entidades virtuales agrupando a una relación utilizando el concepto de agregación.

Un ejemplo de modelo ER definido en el lenguaje es el siguiente:

```
ENTITY Producto {
    ID_producto key
    nombre
    precio
}

ENTITY Tienda {
    nombre key
    direccion key
    hora_apertura
    hora_cierre
}

RELATION Vende(Producto 1, Tienda N!) {
    cantidad
}
```

Aquí se representan las entidades "Producto" y "Tienda", junto con la relación "Vende", relación en la que participan las dos entidades. "Producto" con cardinalidad 1 y participación parcial, mientras que "Tienda" con participación total y cardinalidad N. A continuación, se describe la gramática del lenguaje.

3.1. Notación usada

En la sección que viene se describe la gramática del lenguaje. Para describir la gramática se usará la siguiente notación:

- Símbolos terminales: Se denotan entre comillas simples y en color azul. A menos que se especifique lo contrario, los símbolos terminales no hacen diferencia entre mayúsculas y minúsculas. Por ejemplo: 'ENTITY'.
- Símbolos no terminales y producciones: Los símbolos no terminales se escriben en *cursiva* y las producciones se denotan por el nombre de la producción seguido del símbolo «:=» y una o más alternativas de cadenas de terminales o no terminales (separadas por «|»), por ejemplo:

  AttributeList := *AttributeIdentifier* | *AttributeList* ','*AttributeIdentifier*

  Representa una producción con dos alternativas, la primera alternativa es el símbolo no terminal AttributeIdentifier y la segunda es el símbolo no terminal AttributeList seguido del símbolo terminal «,» y el símbolo no terminal AttributeIdentifier.
- Símbolos opcionales: Se usará el símbolo «?» para representar que un símbolo terminal o no terminal es opcional, así:

RelationshipAttribute := *AttributeIdentifier AttributeCardinality*?
será equivalente a:
RelationshipAttribute := *AttributeIdentifier AttributeCardinality* | *AttributeIdentifier*

3.2. Gramática del lenguaje

3.2.1. Expresión principal

Se denominará "documento ER" a la expresión que representa un modelo ER completo, un documento ER está compuesto por una o más expresiones (entidades, relaciones o agregaciones).

- *ERDocument* := *ERExpression MultilineDivider ERDocument* | *ERExpression*

- *ERExpression* := *WeakEntityExpression* | *EntityExpression* | *RelationshipExpression* | *AggregationExpression*

3.2.2. Entidades

La expresión principal de una entidad contiene su nombre identificador, sus atributos y si esta extiende a otra.

- *EntityExpression* := 'ENTITY' *EntityIdentifier ExtendsDecl*? '{' *EntityAttributeList* '}'

La expresión ExtendsDecl indica que una entidad hereda de otra (jerarquía de clases):

- *ExtendsDecl* := 'EXTENDS' *EntityIdentifier*

Los atributos de las entidades pueden ser simples o compuestos. Ambos tipos de atributos pueden ser declarados como llaves de la entidad con el token terminal "key". La expresión MultilineDivider consiste de uno o más caracteres de salto de línea.

- *EntityAttributeList* := *EntityAttribute MultilineDivider EntityAttributeList* | *EntityAttribute*

- *EntityAttribute* := *AttributeIdentifier CompositeDecl*? 'key'?

- *CompositeDecl* := ':' '[' *AttributeIdentifierList* ']'

- *AttributeIdentifierList* := *AttributeIdentifier* ',' *AttributeIdentifierList* | *AttributeIdentifier*

Las expresiones *EntityIdentifier*, *AttributeIdentifier* y *RelationshipIdentifier* corresponden a la misma expresión regular que acepta caracteres alfanuméricos (para las vocales se aceptan tildes) y guiones bajos (_), excluyendo a las palabras reservadas del lenguaje: "entity", "extends", "key", "pkey", "relation" y "aggregation".

Siguiendo estas producciones, una expresión para una entidad válida:

```
ENTITY Producto {
    producto_id key
    nombre
    marca
}
```

Finalmente, una expresión para una entidad que utiliza todas las producciones se podría ver así:

```
ENTITY Alimento EXTENDS Producto {
    ingredientes
    fecha_expiracion: [dia, mes, año]
    calorias
}
```

Notar que la entidad "Alimento" hereda de la entidad "Producto", por lo que no posee llave, ya que heredará la llave de la entidad padre.

3.2.3. Entidades débiles

La sintaxis para entidades débiles es muy similar a la de las entidades regulares. La razón de separarlas en producciones distintas es para usar distintos tokens terminales para indicar que un atributo es parte de una llave; en las entidades regulares se usa "key" mientras que para las débiles se usa "pkey". De esta forma, será un error sintáctico utilizar "key" en entidades débiles y viceversa.

La expresión principal para entidades débiles contiene su nombre identificador, las relaciones a través de las que depende de otra entidad y sus atributos.

- *WeakEntityExpression* := 'ENTITY' *EntityIdentifier* *WeakDecl* '{' *WeakEntityAttributeList* '}'

La siguiente expresión se usa para declarar a una entidad como débil. Se deben explicitar las relaciones a través de las cuales se depende de otra entidad:

- *WeakDecl* := 'DEPENDS ON' *DependenciesList*

- *DependenciesList* := *RelationshipIdentifier* ',' *DependenciesList* | *RelationshipIdentifier*

Los atributos de las entidades débiles funcionan de la misma forma que en las entidades regulares. La única diferencia es que se debe usar el token terminal "pkey" para declarar un atributo como llave parcial.

- *WeakEntityAttributeList* := *WeakEntityAttribute MultilineDivider WeakEntityAttributeList* | *WeakEntityAttribute*

- *WeakEntityAttribute* := *AttributeIdentifier CompositeDecl*? 'pkey'?

Siguiendo estas producciones, una expresión para una entidad débil válida:

```
ENTITY Seccion DEPENDS ON Tiene {
    numero pkey
    alumnos
}
```

3.2.4. Relaciones

La expresión para las relaciones contiene su identificador, sus entidades participantes, pudiendo estas tener arcos etiquetados, restricciones de participación y de cardinalidad.

- *RelationshipExpression* := 'RELATION' *RelationshipIdentifier* '(' *ParticipatingEntitiesList* ')' *RelationshipAttributesDecl*?

A diferencia de las entidades, los atributos para las relaciones son opcionales.

- *RelationshipAttributesDecl* := '{' *RelationshipAttributesList* '}'

- *RelationshipAttributesList* := *AttributeIdentifier MultilineDivider RelationshipAttributesList* | *AttributeIdentifier*

- *ParticipatingEntitiesList* := *ParticipatingEntity* ',' *ParticipatingEntitiesList* | *ParticipatingEntity*

- *ParticipatingEntity* := *ParticipatingLabeledEntity* | *ParticipatingSingleEntity*

- *ParticipatingLabeledEntity* := *EntityIdentifier* ':' '[' *ParticipatingSingleEntityList* ']'

- *ParticipatingSingleEntityList* := *ParticipatingSingleEntity* ',' *ParticipatingSingleEntity-List* | *ParticipatingSingleEntity*

- *ParticipatingSingleEntity* := *EntityIdentifier* *EntityConstraintsDecl*?

- *EntityConstraintsDecl* := *EntityCardinality* '!'?

EntityCardinality es una expresión regular que acepta una sola letra mayúscula entre A y Z, o uno más caracteres numéricos, ya que esta es la forma usual en la que se describen las cardinalidades. El carácter opcional "!" denota participación total. Si la expresión Entity-ConstrainsDecl no está presente se asume que la entidad tiene cardinalidad "N" (muchos) y participación parcial. Finalmente, una expresión para una relación que utiliza todas las expresiones anteriores:

```
RELATION Compra(Persona: [Vendedor 1!, Comprador N], Producto) {
    fecha_transaccion
    unidades_producto
}
```

Esta expresión representa la relación "Compra" en la que participan las entidades "Persona" y "Producto"; la entidad Persona participa por medio de los arcos etiquetados "Vendedor" y "Comprador", Vendedor con cardinalidad 1 y participación total, y comprador con cardinalidad N y participación parcial. Por el otro lado, en la entidad producto no se explicitan las restricciones, por lo que se asume cardinalidad N y participación parcial.

3.2.5. Agregaciones

La expresión para representar una agregación contiene su identificador y la relación que se desea encapsular.

- *AggregationExpression* := *AggregationIdentifier* '(' *RelationshipIdentifier* ')' *Empty-CurlyBlock*?

Se ha optado a dejar la posibilidad de agregar corchetes vacíos al final para mantener la consistencia con las expresiones anteriores.

- *EmptyCurlyBlock* := '{' '}'

Así, una expresión válida para una agregación:

```
AGGREGATION Cliente_compra_Producto(Compra)
```

3.2.6. Documento ER completo

De esta forma, un documento ER sintácticamente válido se forma por una o varias de las expresiones para entidades, relaciones y agregaciones, separadas por uno o más saltos de línea. Un documento ER completo se ve de la siguiente forma:

```
ENTITY Estudiante {
    RUT key
    nombre_completo: [nombre, apellido]
    edad
}

ENTITY Universidad {
    nombre
    dirección
    university_id key
}

RELATION Estudia_en(Estudiante 1, Universidad N!) {
    fecha_ingreso
}

AGGREGATION Estudiante_Estudia_en_Universidad(Estudia_en)
```

3.3. Implementación del parser

El parser de la gramática descrita se implementó en el lenguaje de programación Typescript y por medio de la librería PeggyJS[1], siguiendo la metodología *test-driven development*. El parser del lenguaje se implementó en Typescript debido a la manera en que será usado en este trabajo, el parser se utilizará en la aplicación web para generar diagramas que se verá en el capítulo 4.

En la aplicación web se necesitará parsear la entrada de los usuarios cada vez que ingresen un nuevo carácter, haciendo necesario ejecutar la función que realiza el parsing en el cliente, pues si se ejecutara en el servidor habría que enviar una petición por cada nuevo carácter ingresado por el usuario, lo que llevaría a problemas de rendimiento y latencia. Otra opción sería utilizar alguna tecnología como websockets para mantener una conexión constante con el cliente, sin embargo, aún así podrían existir problemas de latencia y, al haber muchos clientes simultáneos, demasiada carga en el servidor. De esta forma, para ejecutar un programa en el cliente resulta necesario utilizar Javascript.

La implementación del parser corresponde a un programa escrito en PeggyJS donde se declaran las *parsing expression grammars*, que representan las reglas de la gramática descritas

[1]https://peggyjs.org/

anteriormente, y la forma en que se resuelve cada regla. Este programa genera una función de Javascript que recibe como entrada un documento ER, en la forma de un string, y su salida es una estructura de datos (un objeto de Javascript) que contiene toda la información proporcionada por el documento (y necesaria para crear los diagramas ER más adelante). A su vez, se desarrolló un *wrapper* de esta función en Typescript para agregar chequeo de tipos durante el desarrollo. En el anexo A.2 se encuentra el objeto de Typescript generado tras parsear el documento ER de la sección 3.2.6, convertido a JSON para mostrarlo.

Además, se ha creado un conjunto de tests que prueban que todas las expresiones del lenguaje se parseen correctamente y que se manejen los errores para entradas inválidas.

3.3.1. Manejo de errores

Para mejorar la experiencia de los usuarios que utilicen este lenguaje, se implementó la detección de errores sintácticos y semánticos en documentos ER. A continuación se presenta la implementación de esta característica junto con los distintos tipos de errores que se pueden detectar.

Errores sintácticos

La detección de errores sintácticos está implementada en el mismo parser del lenguaje por medio del manejo de errores de Typescript. El parser está hecho de forma que si falla en procesar la entrada, este arrojará un error del tipo *SyntaxError*, que contiene toda la información acerca del error para generar un mensaje útil. Por ejemplo, una entrada inválida podría ser:

```
ENTITY {
    id_persona key
    nombre: [nombre, apellido]
}
```

Esta entrada es inválida debido a que falta el nombre, o identificador, de la entidad. El parser arrojará un *SyntaxError* con la siguiente información: el *token* que se esperaba, el *token* recibido y la ubicación del error (línea, columna). Así, para la entrada inválida anterior es posible generar el siguiente mensaje descriptivo del error:

```
Line 1, column 8: Expected entity identifier but "{" found.
```

Errores semánticos

Para la detección de errores semánticos se implementó un módulo donde se definen reglas que se ejecutan sobre el objeto generado por el parser. Estas reglas contienen la función que se usa para validar el objeto, su código de error y la información relevante que entregan acerca

del error. Al igual que los errores sintácticos, estas funciones entregan toda la información necesaria acerca de un error para generar un mensaje relevante, incluyendo la posición del error semántico en el texto de entrada. Usualmente se conoce como *linter* a la herramientas que detectan errores, problemas de formato y otras características de un programa, por lo que se identificará como el *linter* del lenguaje a este módulo en las siguientes secciones.

Actualmente hay definidos 19 tipos de errores semánticos:

1. Entidad duplicada.
2. Entidad con atributo duplicado.
3. Entidad sin llave.
4. Entidad hereda de otra que no existe.
5. Entidad hereda de una entidad que es hija de esta (herencia circular).
6. Entidad que hereda de otra tiene llave.
7. Entidad débil depende de una relación que no existe.
8. Entidad débil no participa en una de las relaciones de las que depende.
9. Entidad débil no tiene llave parcial (si alguna de sus entidades identificativas participa en una de sus relaciones débiles con cardinalidad mayor a 1).
10. Entidad débil no tiene participación total, o cardinalidad igual a 1, en una de las relaciones de las que depende.
11. Relación duplicada.
12. Relación tiene un atributo duplicado.
13. Relación tiene solo una entidad participante (considerando el caso de roles).
14. Relación tiene un participante duplicado.
15. Relación tiene un participante que no existe.
16. Agregación duplicada.
17. Agregación encapsula una relación que no existe.
18. Agregación encapsula una relación que ya fue encapsulada por otra.
19. Agregación utiliza el nombre de una entidad que ya existe.

Por ejemplo, para el siguiente documento ER:

```
ENTITY Persona {
    RUT key
    nombre
}

ENTITY Entrada DEPENDS ON Compra {
    id_ticket pkey
    asiento
}

RELATION Compra(Persona, Entrada)
```

La función que aplica todas las reglas encontraría un error del tipo "entidad débil no tiene participación total, o cardinalidad igual a 1, en una de las relaciones de las que depende". La posición retornada indica donde parte y termina el error. Esta posición corresponderá a los caracteres subrayados en el documento anterior, correspondiente al *token* "Entrada". Notar que este error se puede corregir agregando participación total y cardinalidad de 1 al participante "Entrada" en la relación "Compra"; para esto habría que explicitar sus restricciones. Así, por ejemplo, el error desaparece al reemplazar la línea de la relación "Compra" por:

```
RELATION Compra(Persona, Entrada 1!)
```

Finalmente, se implementó un conjunto de tests para verificar que todas las reglas funcionen correctamente.

Capítulo 4

Diseño e implementación de la aplicación web

En este capítulo se describe la aplicación web desarrollada para lograr los objetivos planteados en el Capítulo 1. En primer lugar, se identifican los requerimientos que debe satisfacer la aplicación. Luego, se presenta un mockup, desarrollado al comienzo del proyecto, de la interfaz principal de la aplicación. Posteriormente, se presenta la arquitectura de la solución junto con las tecnologías utilizadas. Finalmente, se presenta la aplicación web ya finalizada junto con los detalles de su implementación.

4.1. Requerimientos de la Aplicación Web

A continuación, se presentan los requerimientos funcionales y no funcionales que la aplicación web desarrollada debe satisfacer.

4.1.1. Requerimientos Funcionales

1. La aplicación debe permitir al usuario ingresar un modelo ER a través del lenguaje ERdoc, mediante un editor de texto.
2. La aplicación debe reportar errores sintácticos en el documento ER ingresado por el usuario.
3. La aplicación debe reportar errores semánticos en el documento ER ingresado por el usuario.
4. La aplicación debe destacar, en el editor de texto, los errores cometidos por el usuario.
5. La aplicación debe generar diagramas ER a partir del documento ER ingresado por el usuario.
6. El usuario debe poder interactuar con los diagramas generados, por ejemplo, arrastrar los elementos para moverlos de posición.

7. La aplicación debe permitir al usuario elegir la notación a utilizar para generar los diagramas, se debe soportar la notación de Chen y las utilizadas en el curso Bases de Datos del departamento de Ciencias de la Computación de la Universidad de Chile: la notación de flechas y la (mín, máx).
8. Debe ser posible exportar los diagramas a un archivo de imagen, PDF o SVG.
9. La aplicación debe ofrecer al usuario la opción de posicionar los elementos del diagrama de forma automática.

4.1.2. Requerimientos no Funcionales

1. La aplicación web debe funcionar en distintos tamaños de pantalla en dispositivos no móviles (computadores de escritorio, laptops, entre otros) y navegadores.
2. Los diagramas se deben generar en tiempo real, es decir, se deben ver los cambios en el diagrama a medida que el usuario escribe en el editor.

4.2. Mockup de la Aplicación Web

Durante la etapa inicial del proyecto se desarrolló un mockup (ver Figura 4.1) de la interfaz principal de la aplicación web que logre capturar todos los requerimientos definidos anteriormente. En la interfaz principal se encuentra un editor de texto y, a la derecha de este, se muestra el diagrama ER que se va generando a partir de la entrada del usuario. En la barra superior de la aplicación se tienen botones para exportar el diagrama a distintos formatos, cargar diagramas de ejemplo, entre otros. En cuanto al editor de texto, se puede apreciar que las palabras clave del lenguaje ERdoc se encuentran destacadas en distintos colores para facilitar el entendimiento de su sintaxis; a su vez, se observa que se subrayan los errores en la entrada del usuario. Por otro lado, en el mockup se observa que el diagrama generado utiliza la "Notación con flechas", esta es una notación utilizada en el curso Bases de Datos. Al mismo tiempo, el mockup da a entender que es posible seleccionar otras notaciones. El objetivo tras este diseño es dejar la mayor cantidad de espacio disponible a la visualización del diagrama ER, ya que este es el resultado final que le interesa a los usuarios.

Si bien el diseño final de la aplicación web no resultó idéntico al mockup presentado, la interfaz implementada retiene la estructura general del mockup y, además, este resultó bastante útil para guiar la implementación. Por otro lado, se decidió dejar fuera el botón para compartir diagramas y el historial de cambios. Esto pues los diagramas pueden ser compartidos mediante los archivos exportados o simplemente a través del documento ER; de la misma forma, el historial de cambios se eliminó puesto que el editor de texto permite realizar las acciones de deshacer y rehacer cambios.

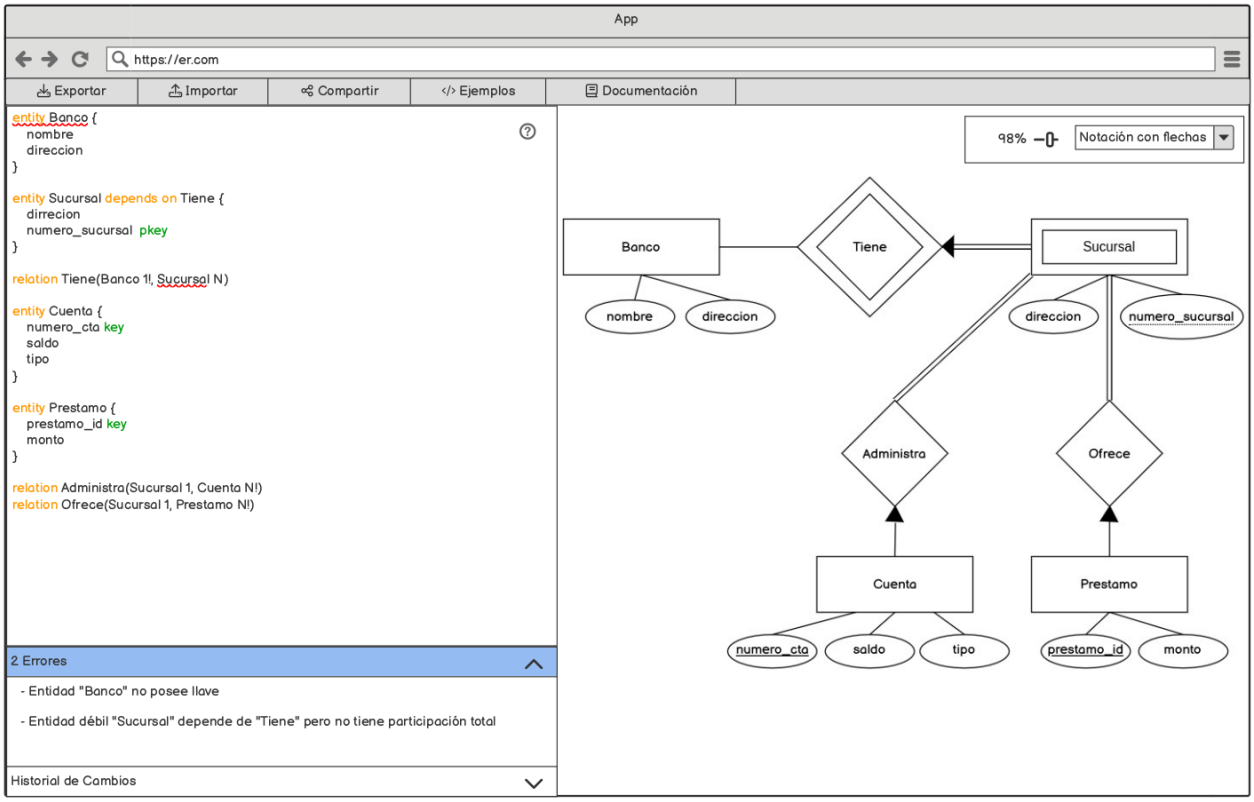

Figura 4.1: Mockup de la interfaz principal de la aplicación web.

4.3. Arquitectura y Tecnologías

En esta sección se presenta la arquitectura general de la aplicación web desarrollada, junto a las tecnologías utilizadas para su implementación.

Para implementar la solución se utilizó la arquitectura cliente-servidor, en la Figura 4.2 se tiene un diagrama de la arquitectura de la aplicación web junto a las principales tecnologías utilizadas. La aplicación web desarrollada corresponde a una que se ejecuta completamente en el lado del cliente. De esta forma, la principal responsabilidad del back-end de la aplicación es servir las páginas web correspondientes a la interfaz principal y a la documentación. Debido a esto, se optó por utilizar tanto para el back-end como para el front-end de la aplicación el framework Next.js[1] en conjunto al lenguaje de programación Typescript[2]. Adicionalmente, en el front-end se utilizó React[3] para desarrollar la interfaz principal.

Se eligió React por la naturaleza interactiva que tendrá la aplicación, ya que, por ejemplo, los usuarios podrán arrastrar y soltar elementos en la interfaz, y React es una librería que fue creada para manejar este tipo de interacciones, haciendo sencillo el manejo de ellas. Además, React es una tecnología muy popular con una gran cantidad de recursos y librerías que facilitan el desarrollo de la aplicación. Se eligió Next.js como framework para el front-end y back-end porque está fuertemente basado en React y funcionan muy bien juntos. Además,

[1]https://www.nextjs.org/
[2]https://www.typescriptlang.org/
[3]https://www.react.dev/

permite escribir tanto el front-end como el back-end en Typescript. Usar un solo lenguaje de programación para el front-end y el back-end es una ventaja, ya que se puede compartir código en ambos lados de la aplicación y, además, reduce la barrera de entrada para los desarrolladores que quieran contribuir al proyecto. Por último, se eligió usar Typescript en lugar de Javascript debido a que posee tipado estático, que permite escribir código más mantenible y extensible.

Por el lado del front-end, se utilizó la librería React Flow[4] para generar e interactuar con los elementos del diagrama ER. Adicionalmente, se integró el lenguaje ERdoc al editor de texto Monaco[5]. Se decidió usar este editor de texto gracias a la enorme cantidad de características que posee. Por ejemplo, permite implementar el destacado de errores y autocompletado de palabras de manera sencilla.

La aplicación fue puesta en producción en un servidor de la Universidad de Chile. El back-end se ejecuta en un cluster de PM2[6]. PM2 permite ejecutar el servidor en múltiples instancias y automatizar el despliegue de nuevas versiones.

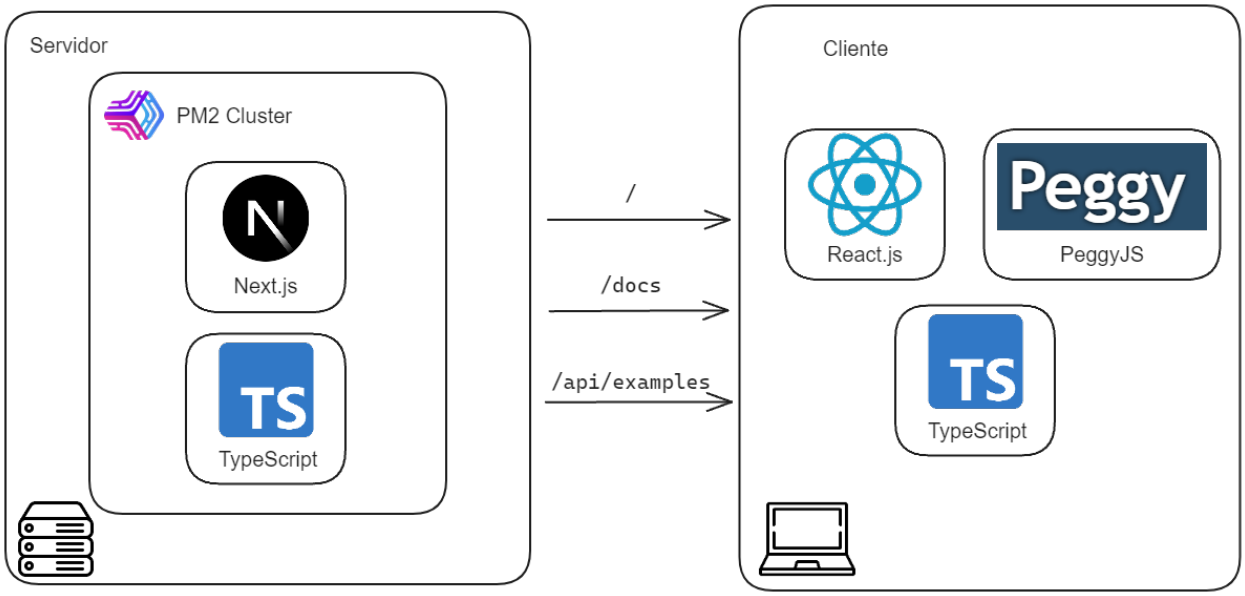

Figura 4.2: Arquitectura de la aplicación web.

[4]https://github.com/wbkd/react-flow/
[5]https://microsoft.github.io/monaco-editor/
[6]https://pm2.keymetrics.io/

4.4. Interfaces de la Aplicación

A continuación se muestran las interfaces de *ERdoc Playground*, la aplicación web desarrollada durante este trabajo.

4.4.1. Interfaz Principal

La interfaz principal de la aplicación (ver Figura 4.3) está formada por una barra de navegación, un panel para el editor de texto y otro panel para visualizar e interactuar con el diagrama ER generado a partir del contenido del editor de texto.

La barra de navegación contiene un botón para activar o desactivar el *layout* automático, característica que se discute en la siguiente sección. Además, se tiene un botón para cargar o importar diagramas a través de archivos JSON, un botón para exportar el diagrama a distintos formatos: PDF, SVG, JPEG y PNG; un botón que redirige a la documentación de la aplicación, un botón para cambiar el idioma de la aplicación y otros botones con información acerca de la aplicación (información acerca del autor y sobre el proyecto).

El panel con el editor de texto posee, además, una pestaña que muestra todos los errores en el documento ER presente en el editor y otra pestaña con botones para cargar distintos ejemplos de documento ER. Ambas pestañas se pueden abrir y cerrar.

En cuanto al panel que contiene al diagrama ER. Este cuenta con un botón que despliega el menú de configuración del diagrama (ver Figura 4.8), donde se puede elegir la notación a ser utilizada y el tipo de ruta que siguen las aristas.

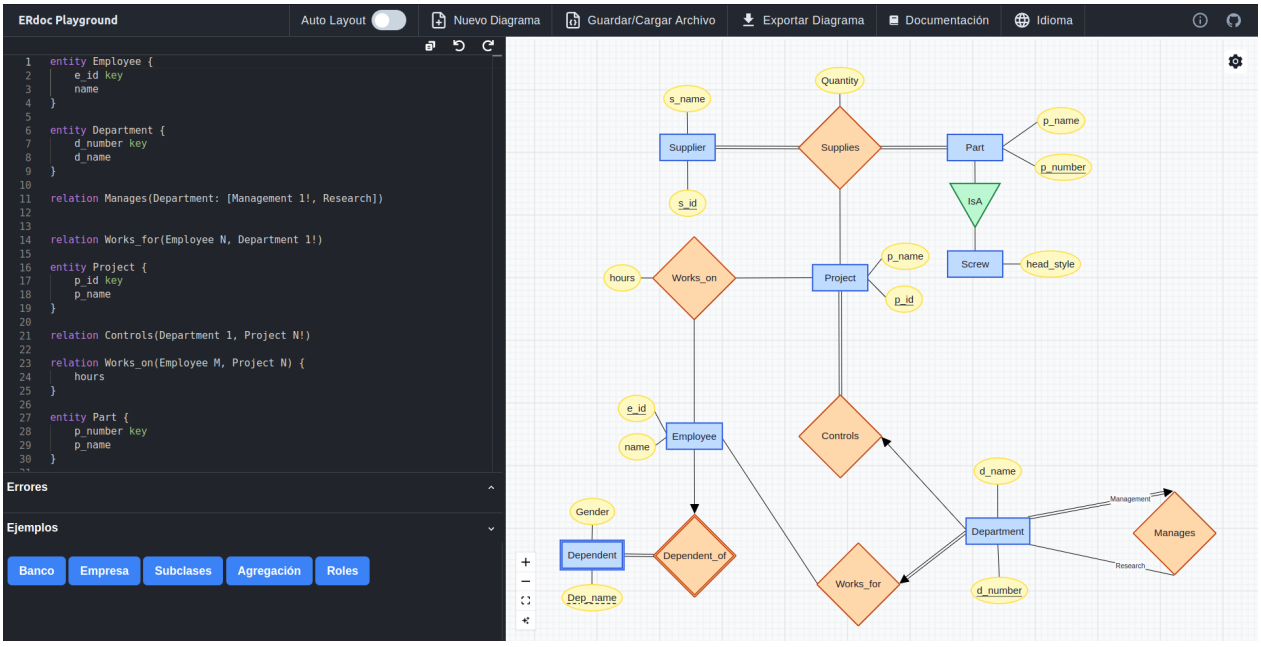

Figura 4.3: Interfaz principal de *ERdoc Playground*.

4.4.2. Documentación

Con el objetivo de introducir la aplicación, junto al lenguaje ERdoc, a nuevos usuarios, se desarrolló una sección de documentación. Esta sección contiene una breve descripción de la aplicación y una sección con la sintaxis del lenguaje ERdoc. En la sección de sintaxis se muestran todas las expresiones del lenguaje, ejemplos de documento ER e imágenes de los diagramas ER que se generan a partir de estos ejemplos. En las Figuras 4.4 y 4.5 se muestran algunas de las secciones de la documentación.

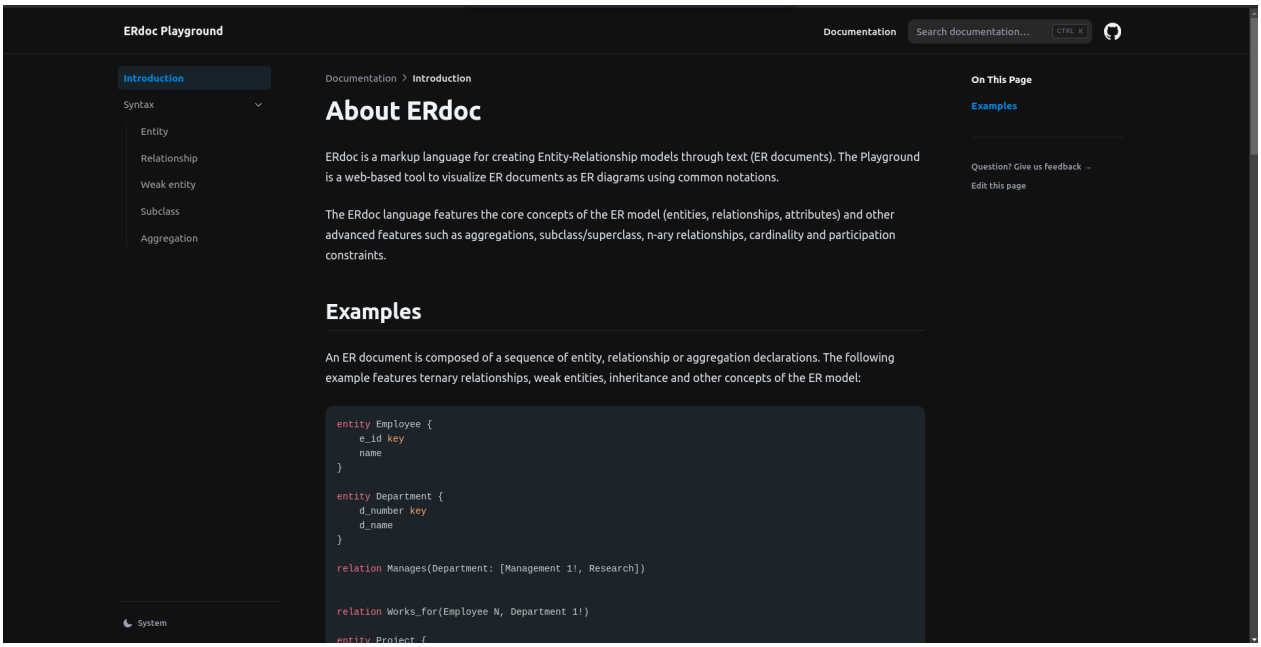

Figura 4.4: Sección "Introducción" de la documentación de *ERdoc Playground*.

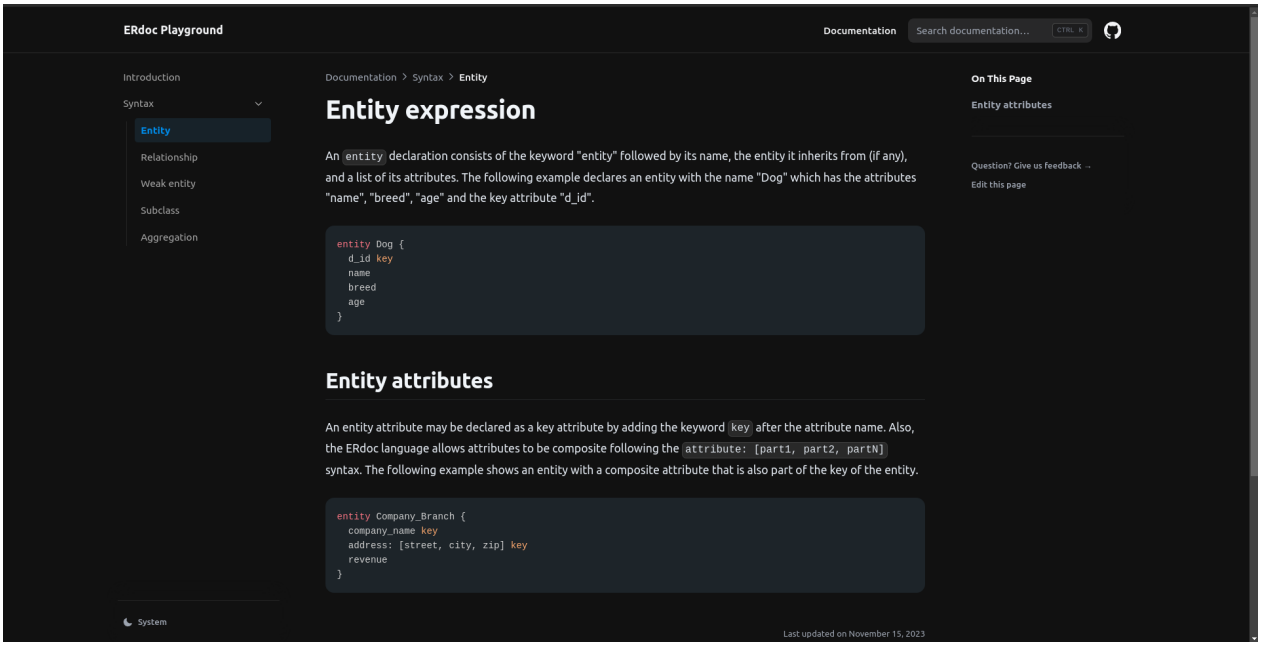

Figura 4.5: Documentación de la expresión para entidades del lenguaje ERdoc.

4.5. Flujo Principal de la Aplicación

En esta sección se presenta la implementación de la funcionalidad principal de la aplicación, que corresponde a convertir la entrada del usuario (en el lenguaje ERdoc) a diagramas ER. También se detalla la funcionalidad de cargar archivos de diagramas.

En primer lugar se tiene la Figura 4.6, un diagrama que muestra el flujo lógico de la aplicación. A continuación se detallará cada etapa del diagrama.

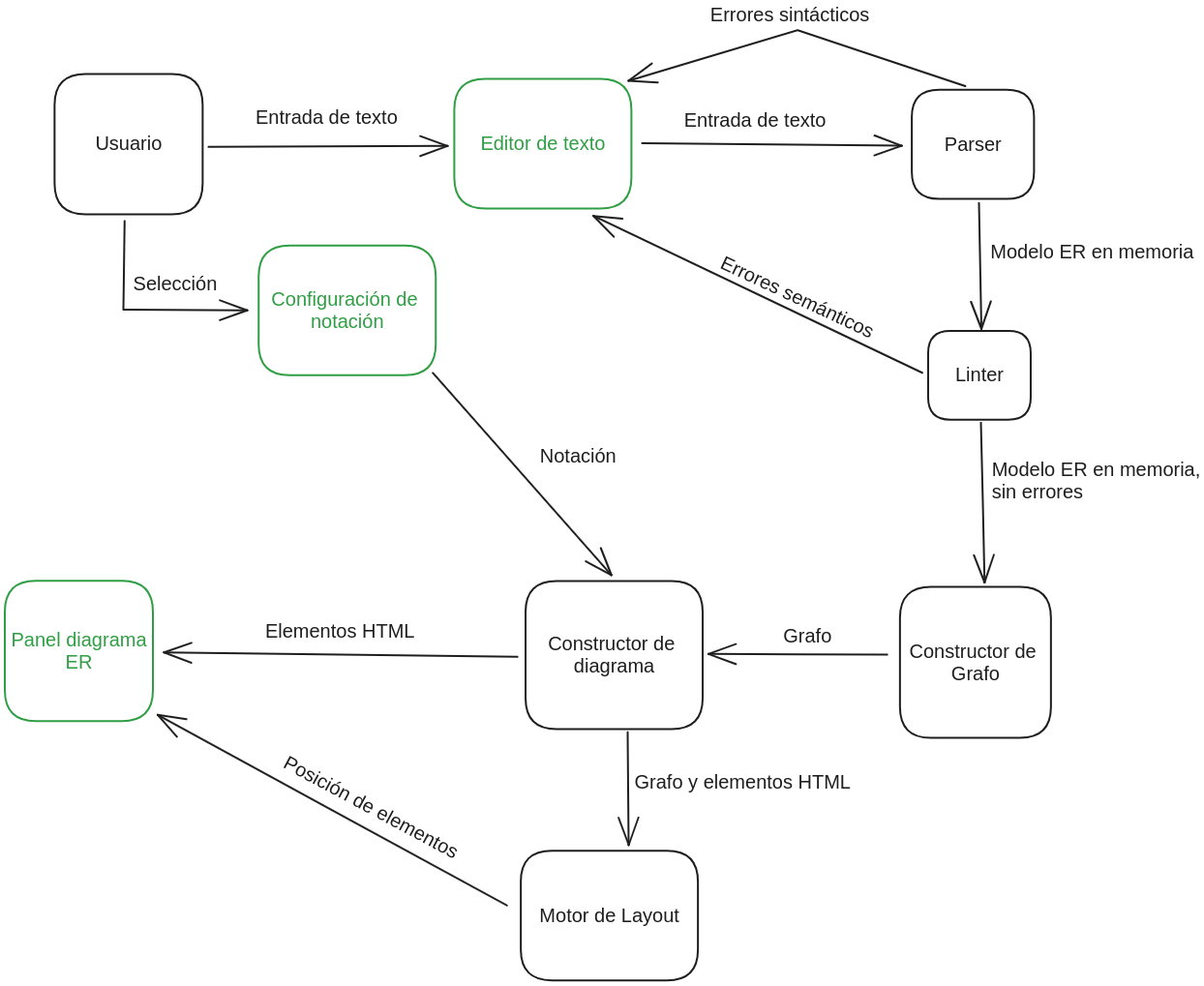

Figura 4.6: Flujo lógico de la aplicación, desde la entrada del usuario hasta la visualización de un diagrama ER. En verde se tienen los elementos que forman parte de la interfaz.

4.5.1. Editor de Texto, Parser y Linter

El primer paso para generar un diagrama ER corresponde a obtener la entrada del usuario. Como ya se mencionó, se utiliza el editor de texto Monaco como interfaz para que el usuario ingrese sus documentos ER. Para hacer de la aplicación una experiencia "en vivo", esto es, ir generando el diagrama ER a la vez que el usuario escribe, se le agrega un *EventListener*

al evento *onChange* del editor. Por consiguiente, el flujo de la Figura 4.6 se ejecuta cada vez que el usuario ingresa (o elimina) un carácter en el editor de texto.

Así, una vez ingresada una entrada por el usuario, se debe parsear y revisar errores en el texto. Para esto, se utilizan el parser y el linter del lenguaje ERdoc, descritos en la sección 3.3. Si la entrada del usuario corresponde a un documento ER válido y no posee ningún error se sigue con el flujo principal. Por el otro lado, de existir un error sintáctico o semántico en la entrada, se destacan los errores en el editor de texto como un subrayado de color rojo en la posición del error. El mensaje de error y la posición de este son provistos por ambos parser y linter. Esta información se utiliza para crear elementos del tipo *IMarkerData*, parte de la API que ofrece el editor Monaco para destacar errores, alertas y otros tipos de información, logrando así la visualización de errores en el editor de texto.

En la Figura 4.7 se observa el destacado de errores para una entrada válida, una entrada con error sintáctico y otra con error semántico.

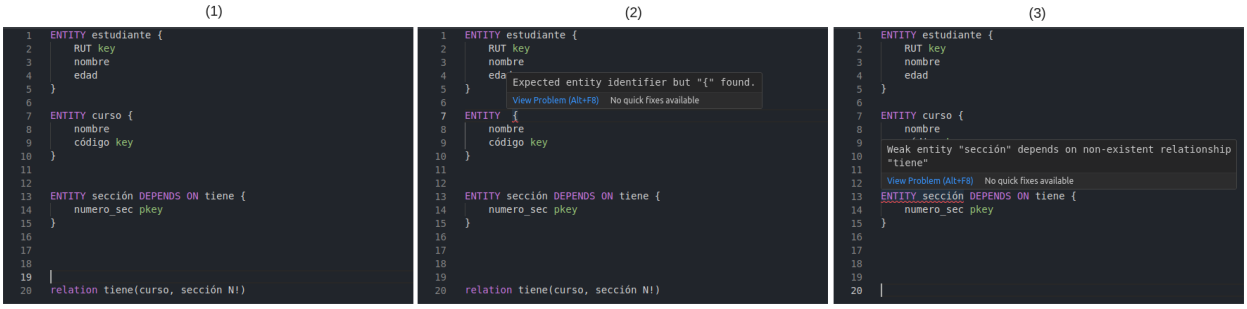

Figura 4.7: Destacado de errores para tres entradas distintas. (1) Entrada válida. (2) Entrada con error sintáctico. (3) Entrada con error semántico (entidad débil depende de una relación inexistente).

Por otro lado, en la Figura 4.7 también se observa que las palabras clave del lenguaje están destacadas en distintos colores. Las palabras clave *key* y *pkey* se destacan en un color verde claro, mientras que el resto de las palabras clave del lenguaje se destacan en un color similar al magenta. Esta característica, conocida como *syntax highlighting*, se implementó con el objetivo de mejorar la legibilidad del documento ER y facilitar el entendimiento de la gramática del lenguaje ERdoc. Esta característica se implementó utilizando Monarch[7], una librería que permite crear *syntax highlighters* para el editor Monaco, definiendo una gramática mediante expresiones regulares. Se desarrolló una simplificación de la gramática de ERdoc que logre hacer que se coloreen las palabras claves del lenguaje.

4.5.2. Construcción del Grafo

Una vez que la entrada del usuario ha sido parseada con éxito, esto es, no existen errores sintácticos ni semánticos, comienza el proceso de construcción del grafo. Esta etapa se encarga de convertir el modelo ER entregado por el parser al conjunto de nodos y aristas (un grafo) que conformarán el diagrama ER. Por ejemplo, la siguiente expresión para una entidad:

[7]https://microsoft.github.io/monaco-editor/monarch.html

```
ENTITY Libro {
    nombre key
    autor
    n_páginas
}
```

debería generar cuatro nodos, uno para la entidad y tres para sus atributos. Además, se deben generar tres aristas, una entre cada nodo de un atributo y el nodo de la entidad.

Adicionalmente, cada elemento del grafo debe almacenar información sobre el elemento del modelo ER que representa. Por ejemplo, el nodo de un atributo debe almacenar el nombre del atributo y si este es una llave. Ya que si es una llave, su nombre deberá ir subrayado al renderizar el nodo.

Con esta problemática en mente, se definieron cinco tipos de nodos:

1. **EntityNode**: nodo que representa una entidad. Almacena:
    - label: nombre de la entidad.
    - isWeak: atributo booleano que indica si la entidad es débil o no.
2. **EntityAttributeNode**: Nodo que representa un atributo de una entidad. Además contiene:
    - label: nombre del atributo.
    - isKey: atributo que indica si la entidad es parte de la llave de la entidad.
    - entityIsWeak: atributo que indica si la entidad a la que pertenece es débil o no.
3. **RelationshipNode**: nodo que representa una relación. Adicionalmente guarda:
    - label: nombre de la relación.
    - hasDependant: atributo que indica si alguno de sus participantes es una entidad débil dependiente de la relación representada por este nodo.
4. **RelationshipAttributeNode**: nodo que representa un atributo de una relación. Almacena:
    - label: nombre del atributo.
5. **AggregationNode**: nodo que representa una agregación. Este es un tipo particular de nodo, ya que al renderizarlo contendrá otros nodos dentro de él, guarda:
    - label: nombre de la agregación.
    - width: ancho en pixeles del nodo al renderizarlo. Este atributo se calcula dinámicamente al momento de renderizar el nodo, por la forma en que funciona React, es necesario incluirlo en esta etapa del flujo.
    - height: altura del nodo al renderizarlo. Funciona igual que width.

Gran parte de los datos que almacenan los distintos tipos de nodos tienen que ver con la semántica de un elemento del modelo ER a representar. Sin embargo, cada tipo de nodo también almacena una ID interna, la cual debe ser única, ya que con esta ID se identifican los nodos para generar las aristas entre ellos. Los nodos también guardan un atributo interno que indica si tienen algún nodo padre. Esto es útil para identificar a los nodos que deben ir dentro de una agregación.

En cuanto a las aristas del grafo, se define solamente un tipo de arista:

1. **ErEdge**: Arista que une dos nodos del grafo, internamente almacena:
    - id: ID única de la arista.
    - source: ID del nodo origen.
    - target: ID del nodo destino.
    - cardinality: se usa para aristas que unen una relación con uno de sus participantes, indica la cardinalidad del participante.
    - isTotalParticipation: se utiliza en aristas que unen una relación con uno de sus participantes. Indica si el participante tiene participación total en la relación.

Así, con los datos que guarda internamente cada arista, más adelante se le podrá asignar un estilo de CSS diferente de forma dinámica a cada una de ellas. De esta forma, se podrán representar las restricciones de participación y cardinalidad de los participantes de una relación.

4.5.3. Construcción del Diagrama ER y Notaciones

Ya teniendo el grafo que representa exactamente al modelo ER ingresado por el usuario, se deben convertir los nodos y aristas del grafo a elementos HTML. Es importante destacar que dependiendo de la notación seleccionada por el usuario, se deberán mostrar distintos tipos de aristas.

Esta etapa se implementó a través de clases de Typescript. Cada notación distinta es una clase que extiende la clase *AbstractNotation*, la cual contiene los elementos de React que por defecto se utilizan para renderizar un nodo o arista, obtenidos en la etapa anterior. ERdoc Playground soporta tres notaciones distintas: La notación de flechas, utilizada en el curso CC3201, a través de la clase *ArrowNotation*, la notación original de Chen, mediante la clase *ChenNotation* y la notación (mín, máx) (también utilizada en CC3201), representada por la clase *MinMaxNotation*. El anexo A.3 muestra tres diagramas ER generados con las distintas notaciones, para un mismo documento ER.

El trabajo que realiza cada clase representante de una notación es elegir el componente de React que se debe utilizar para renderizar un nodo o una arista en particular. Así, por ejemplo, el componente para las aristas en *ArrowNotation* decidirá agregar una punta de flecha en el punto de llegada para representar la participación total, mientras que la clase

ChenNotation utilizará un componente que agregará un círculo de color negro en el inicio de la arista para la misma situación.

Adicionalmente, en el constructor de cada clase representante de una notación se indica la propiedad booleana *isOrthogonal*, que controla la ruta que siguen las aristas para ir de un nodo a otro. Cuando *isOrthogonal* es verdadero, las aristas seguirán un camino formado por líneas ortogonales a los ejes del diagrama, por lo que la arista estará formada por segmentos verticales y horizontales. En el caso opuesto, las aristas unirán a dos nodos mediante la recta más corta entre los dos.

En la Figura 4.8 se observa el elemento de la interfaz que permite al usuario seleccionar la notación a utilizar para generar el diagrama ER, junto al elemento que permite seleccionar el tipo de camino que siguen las aristas.

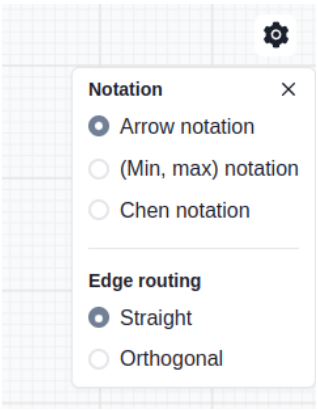

Figura 4.8: Panel de configuración del aspecto del diagrama ER.

4.5.4. Motor de Layout

El motor de *layout* de ERdoc Playground corresponde la funcionalidad encargada de posicionar los nodos del diagrama de forma visualmente atractiva con el objetivo de mejorar la experiencia de usuario, evitando que ellos deban posicionar manualmente los nodos cada vez que se cree uno nuevo. A continuación, se discuten distintos algoritmos y librerías que se probaron para lograr este objetivo.

Los criterios que se consideraron para identificar una visualización como "atractiva" fueron los siguientes:

1. Se minimiza la cantidad de cruces entre aristas: idealmente, en la visualización resultante del diagrama ER ninguna arista se cruza con otra. Esto se debe a que, la mayoría del tiempo, se observa esta característica en los diagramas ER creados por personas.
2. Se minimiza la cantidad de aristas que se cruzan con nodos: se busca que las aristas no pasen por encima de los nodos, ya que esto puede dificultar la visualización.
3. Se minimiza la cantidad de nodos que se superponen: se busca que los nodos no se superpongan entre sí. El algoritmo debería considerar el tamaño de los nodos para evitar posicionar un nodo sobre otro.

4. Se deben soportar agregaciones: el algoritmo debe considerar una agregación como un "subgrafo", entendiéndose como una región delimitada del diagrama resultante, que contenga los elementos del grafo correspondientes a la relación que se encapsula y sus entidades participantes.

5. Las subclases deben ir debajo de las superclases: en el caso de que exista una relación de superclase/subclase entre dos entidades, el algoritmo debe posicionar a la superclase por sobre la subclase. Notar que en casos de que exista herencia múltiple, esta disposición se asemeja a cómo usualmente se representan árboles.

Además, se tienen las restricciones de que el algoritmo de *layout* debe ejecutarse en el cliente y tiene que ser lo suficientemente rápido para ejecutarse en tiempo real, es decir, debería ser posible ejecutarlo cada vez que la entrada del usuario haga que cambie la cantidad de nodos o aristas en el diagrama, sin causar problemas de rendimiento en la aplicación.

Con estos criterios en mente, se probaron distintos algoritmos y librerías de *layout* para grafos en Javascript evaluando su desempeño y las visualizaciones que generan. Las figuras que se mostrarán a continuación corresponden a aplicar un algoritmo de *layout* al diagrama ER generado por el documento ER del anexo A.4.

Inicialmente, se utilizó la librería ELKjs[8] para usar su implementación del algoritmo *force-directed layout*. Este algoritmo logra distribuir los nodos de forma que se reduce en gran cantidad los cruces de aristas y el número de nodos superpuestos. Además, soporta el crear subgrafos para representar agregaciones. Sin embargo, no se puede mantener el orden de arriba hacia abajo para las superclases y subclases. Otra desventaja es que el algoritmo *force-directed layout* es un algoritmo iterativo, por lo que para conseguir una buena visualización se necesita una gran cantidad de iteraciones, haciendo que tome bastante tiempo, lo que no hace posible ejecutar el algoritmo cada vez que la entrada del usuario cambie el diagrama.

Las Figuras 4.9 y 4.10 muestran el resultado de aplicar el algoritmo *force-directed layout* de ELKjs con 2500 iteraciones y 5000 iteraciones, respectivamente. Se observa que con 2500 iteraciones algunos nodos quedan por sobre las aristas. Esto no ocurre con 5000 iteraciones. Sin embargo, duplicar la cantidad de iteraciones no ofrece grandes mejoras en cuanto a minimizar los cruces entre aristas.

[8]https://github.com/kieler/elkjs

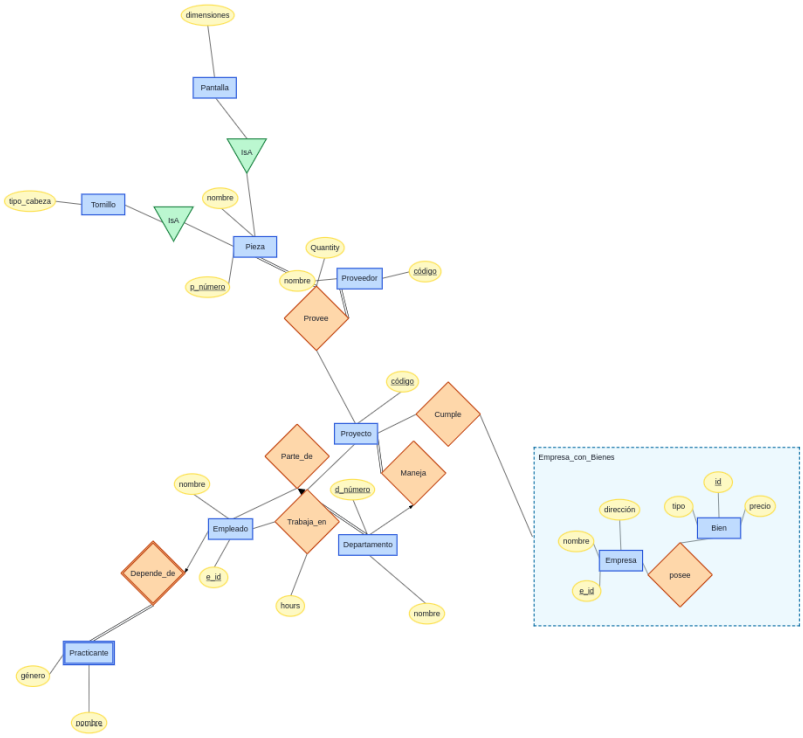

Figura 4.9: *layout* con el algoritmo *force-directed layout* de ELKjs con 2500 iteraciones.

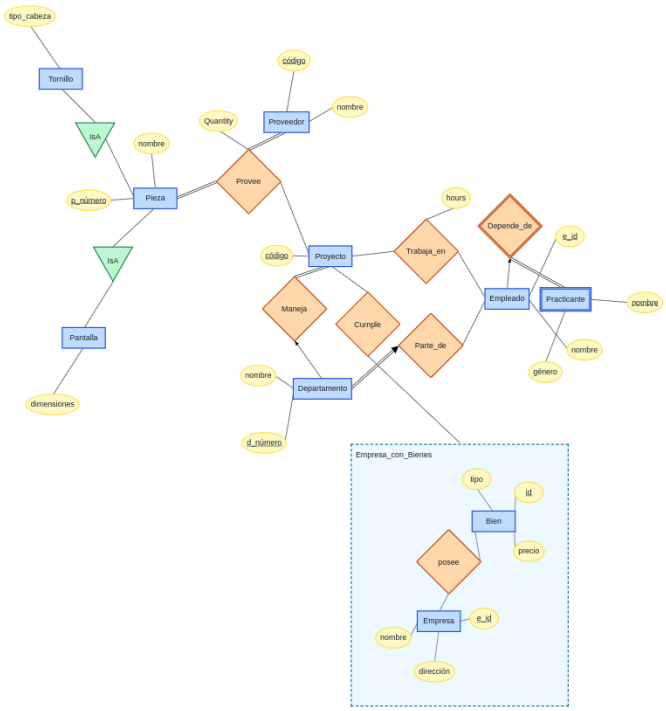

Figura 4.10: *layout* con el algoritmo *force-directed layout* de ELKjs con 5000 iteraciones.

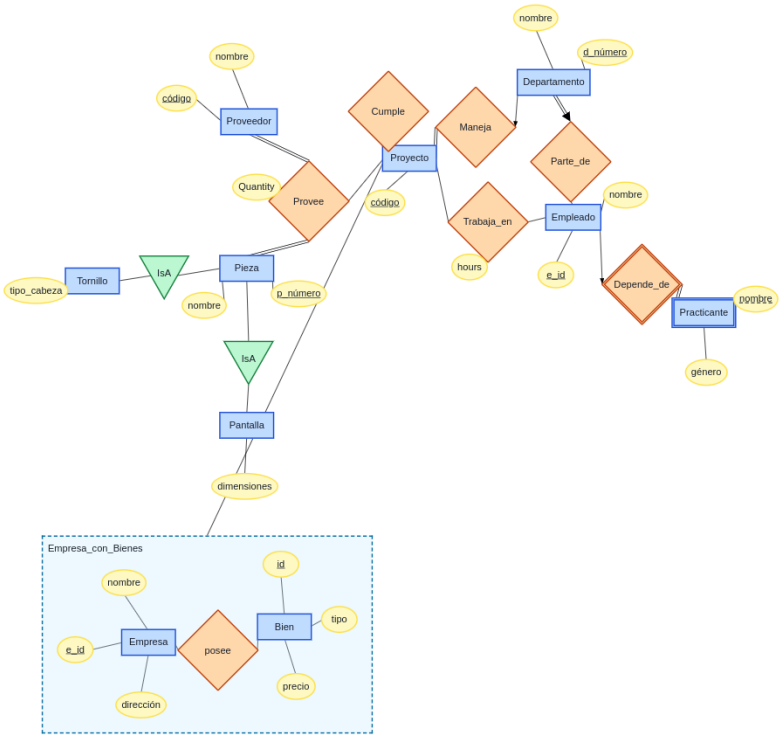

Figura 4.11: Diagrama ER tras aplicar el algoritmo *force-directed layout* de WebCola con *constraints* para subclases/superclases, con 1000 iteraciones.

Dado que el algoritmo *force-directed layout* de ELKjs no satisface los criterios definidos, se probó con la implementación de *force-directed layout* de la librería WebCola[9]. WebCola permite definir restricciones para el *layout* entre grupos de nodos específicos, permitiendo un mayor control sobre el resultado final del algoritmo. Por ejemplo, se puede definir que un grupo de nodos estén alineados verticalmente. Esta característica se conoce como *constraints*.

Los *constraints* de WebCola se aprovechan en las partes del diagrama en las que hay jerarquía de clases. Cada vez que se encuentren presentes una superclase y una subclase, se define un *constraint* de tipo *inequality* entre el nodo de la superclase, el nodo especial "isA" y la subclase. En esta restricción se indica que el nodo de la superclase debe estar por sobre el nodo "isA", mientras que la subclase por debajo del nodo "isA". Así, se mantiene el orden esperado.

El resultado de aplicar el algoritmo de WebCola se encuentra en la Figura 4.11. En la figura se observa que las relaciones de jerarquía de clases se posicionan mejor que con ELKjs entre ellas. Sin embargo, aún existen nodos que quedan por sobre las aristas y aristas que se cruzan entre ellas. En cuanto a las iteraciones necesarias, el algoritmo logra una buena visualización con 1000 iteraciones, bastante menos que al utilizar ELKjs. Sin embargo, aún no es lo suficientemente rápido como para ejecutarse cada vez que el usuario modifique el diagrama.

[9]https://ialab.it.monash.edu/webcola/

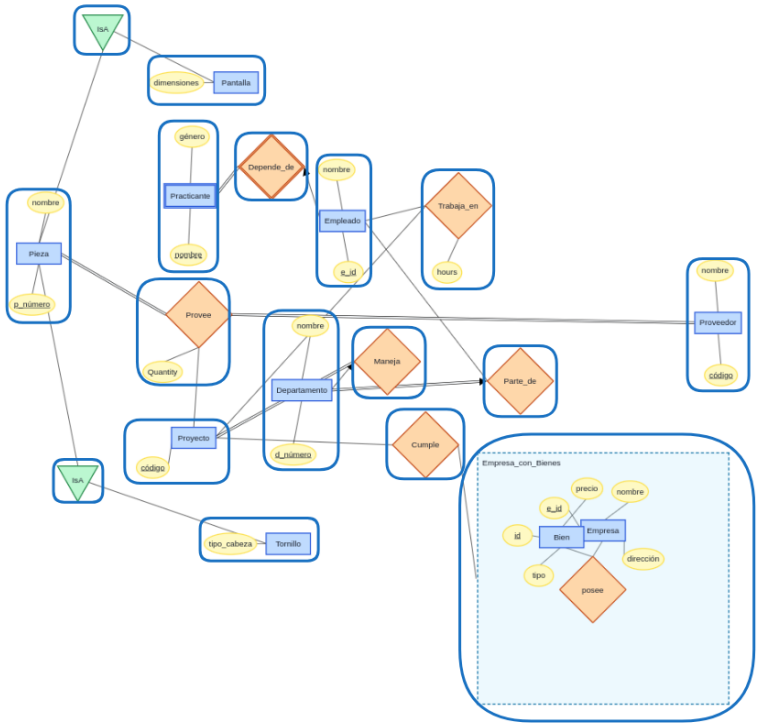

Figura 4.12: Primera etapa del algoritmo *multi-layout*. Encerrados en azul, los subgrafos que contienen elementos del diagrama.

Dado que las dos implementaciones de *force-directed layout* no lograron satisfacer con éxito los criterios definidos, se decidió por implementar un algoritmo propio utilizando ELKjs. El algoritmo aprovecha las distintas estrategias de *layout* que ofrece ELKjs junto a la noción de subgrafo. El algoritmo se denominó *multi-layout*. El resultado de aplicar el algoritmo de *multi-layout* se encuentra en la Figura 4.15.

Multi-layout descompone el grafo del diagrama en distintos subgrafos, en los cuales se ejecuta el algoritmo de forma independiente siguiendo distintas estrategias de *layout*. A continuación se detalla el funcionamiento del algoritmo.

En primer lugar, se crea un subgrafo para cada entidad, o relación, junto a sus atributos. Por otro lado, los nodos "isA" se consideran como un subgrafo que contiene solo a este nodo. Luego, se ejecuta un algoritmo de *layout radial* dentro de cada subgrafo. Esto logra posicionar los nodos de atributos alrededor (y cerca) de las entidades o relaciones a los que pertenecen. En el caso de que una relación se encuentre encapsulada por una agregación, los nodos de la relación y sus participantes se posicionan dentro de un subgrafo, las agregaciones siguen la estrategia de *force-directed layout* de ELKjs con 100 iteraciones.

En la Figura 4.12 se muestra un diagrama ER, inicialmente con sus nodos posicionados de forma aleatoria, luego de ejecutar la primera etapa del algoritmo. Notar que cada nodo dentro de un subgrafo ya se encuentra posicionado de forma correcta.

En este punto, cada subgrafo tiene a sus nodos internos posicionados de forma correcta.

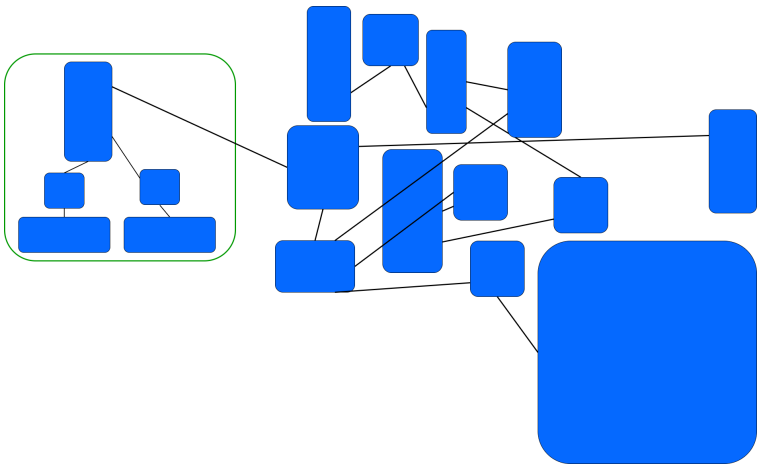

Figura 4.13: Segunda etapa del algoritmo *multi-layout*. En azul, los nodos del grafo. Encerrados en verde, los elementos del subgrafo correspondiente a entidades relacionadas por jerarquía de clases.

Por lo tanto, el siguiente paso es posicionar a los subgrafos entre ellos. Para esto, se crea un nuevo grafo donde cada nodo corresponde a un subgrafo de la etapa anterior.

Con el nuevo grafo construido, se crean nuevos subgrafos que contengan a entidades relacionadas entre sí por jerarquía de clases. Estos subgrafos contendrán, solamente, nodos representantes de entidades y de la relación "isA". Luego, en cada uno de los subgrafos se ejecuta el algoritmo de ELKjs *layered* con dirección hacia abajo, logrando una visualización de árbol para subclases y superclases.

En la Figura 4.13 se observa el grafo creado en esta etapa, donde cada uno de sus nodos corresponde a un subgrafo de la etapa anterior. Además, el grafo contiene al subgrafo (en verde) que se crea para el caso de jerarquía de clases, dentro de este los nodos están ordenados en forma de árbol.

Luego, los subgrafos representantes de jerarquía de clases son convertidos a un nuevo nodo, obteniéndose un nuevo grafo. En este grafo se ejecuta el algoritmo *force-directed layout* de ELKjs para posicionar a todos los subgrafos entre sí. Se observa que el grafo posee una menor cantidad de nodos que el diagrama ER inicial, ya que se están considerando grupos de múltiples elementos del diagrama como un solo nodo. De esta forma, gracias a la menor cantidad de nodos, se logran buenos resultados con solo 100 iteraciones del algoritmo. El resultado de aplicar *force-directed layout* a este grafo se encuentra en la Figura 4.14.

Una vez posicionados los nodos, se procede a reemplazarlos por los subgrafos que representan. A su vez, los subgrafos son reemplazados por los elementos que contienen, resultando los elementos originales del diagrama. El *layout* resultante para el diagrama ER se encuentra en la Figura 4.15.

El algoritmo *multi-layout* es más rápido que los anteriormente mencionados, permitiendo ejecutarlo cada vez que el usuario modifica el diagrama con su entrada. Además, se observa que produce una menor cantidad de cruces entre aristas y de nodos posicionados unos sobre otros. Cumpliendo en mayor medida los criterios definidos para un *layout* atractivo.

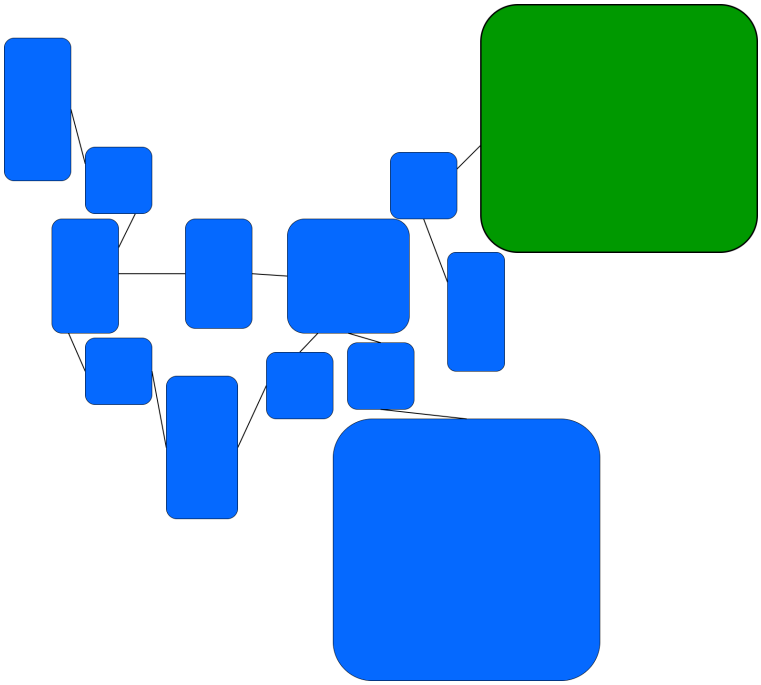

Figura 4.14: Tercera etapa del algoritmo *multi-layout*, tras aplicar *force-directed layout*. El nuevo grafo considera el subgrafo de jerarquía de clases como un nodo (en verde).

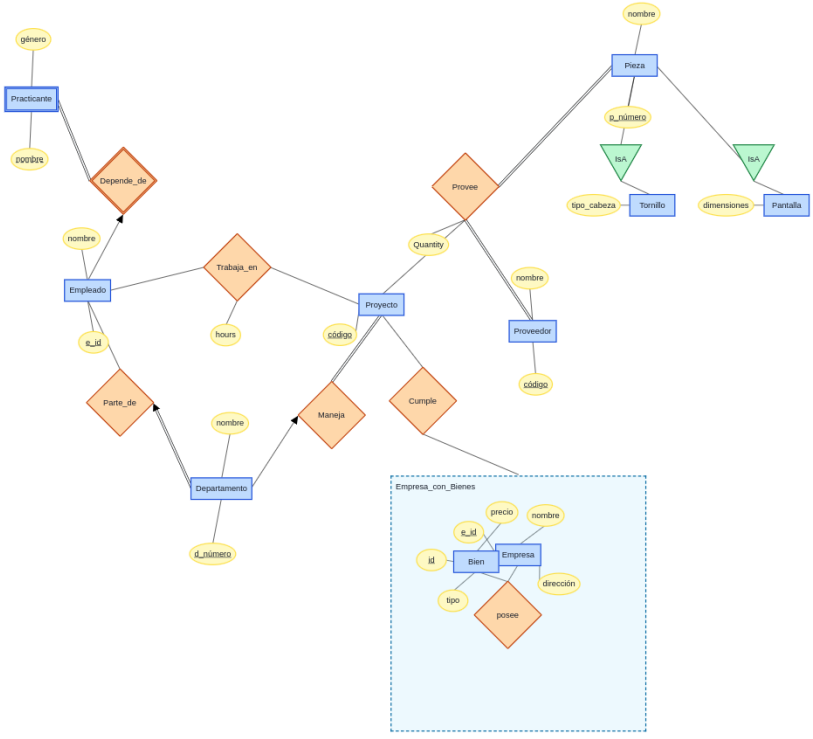

Figura 4.15: Diagrama ER tras aplicar el algoritmo *multi-layout*.

4.5.5. Carga de Archivos

Una funcionalidad importante de ERdoc Playground es la posibilidad de cargar y exportar el estado de la aplicación en la forma de un archivo JSON. Esto permite que los usuarios puedan compartir sus diagramas ER con otros. En el contexto de un curso de bases de datos, sería posible, por ejemplo, que los estudiantes entreguen sus tareas en este formato.

Los archivos contienen la información necesaria para reconstruir el diagrama ER, incluyendo el texto del documento ER junto a la posición de los nodos del diagrama. Así, la estructura del objeto JSON tiene las llaves:

- **erdoc**: contiene el texto del documento ER presente en el editor de texto al momento de exportar.
- **nodes**: contiene un arreglo de los nodos del diagrama; cada nodo contiene las llaves **id** y **position**. El ID único del nodo se guarda en **id**; mientras que **position**, a su vez, contiene las llaves **x** e **y**, que indican las coordenadas de la posición del nodo en el diagrama.
- **edges**: contiene un arreglo de las aristas del diagrama; cada arista almacena las llaves **id**, **source** y **target**, donde **source** y **target** corresponden a las IDs de los de nodos origen y destino de la arista, respectivamente.

Para reconstruir el diagrama, el documento ER almacenado en **erdoc** se ingresa al editor de texto. Así, se ejecuta el flujo principal de la aplicación, generando el grafo del diagrama ER. Luego, se reemplazan las posiciones de los nodos del grafo por las almacenadas en el archivo, para finalmente visualizar el diagrama ER reconstruido.

Capítulo 5

Evaluación

Para evaluar ERdoc Playground se midieron dos aspectos: el rendimiento de la aplicación y su usabilidad. De esta forma, se busca responder si la aplicación logra generar diagramas en tiempo real y si esta es usable.

5.1. Rendimiento

Con el objetivo de verificar si la aplicación logra generar diagramas en tiempo real, se midió el tiempo de ejecución de las distintas etapas del proceso de convertir documentos ER a diagramas ER: parser, linter y la construcción del grafo del diagrama. También se midió el tiempo de ejecución de los distintos algoritmos de *layout* mencionados en el capítulo anterior.

Todos los experimentos y mediciones presentes en esta sección fueron ejecutados en un computador de escritorio con el procesador AMD Ryzen 5 3600, memoria RAM de 16 GB, GPU Nvidia 1660 Super y un disco duro de estado sólido. El entorno de ejecución fue Node.js (versión 18.17.1) en el sistema operativo Linux Ubuntu 22.04. Se utilizó Node.js en lugar de un navegador, pues facilitó en gran medida la implementación de los experimentos; por ejemplo, permite guardar la gran cantidad de documentos ER generados como archivos, haciendo más sencilla su manipulación. Por otro lado, las funciones evaluadas no interactúan con el *Document object model* (DOM) del front-end, haciendo posible su ejecución en este entorno.

Para medir el tiempo de procesamiento de los documentos ER, se generaron múltiples documentos ER con distintas cantidades de elementos. Para lograr esto, se desarrolló un script que genera un documento con una cantidad (entregada como parámetro) fija de entidades, relaciones y agregaciones. La cantidad de atributos para las entidades es aleatoria, entre 1 y 4. El nombre de cada elemento también se asigna aleatoriamente, correspondiendo a una cadena de texto de largo 10.

El documento ER más grande que fue generado consiste de 139 entidades, 20 relaciones y 5 agregaciones, ya que se estima que, bajo el contexto de una herramienta de aprendizaje, los usuarios no crearán diagramas ER más grandes.

Cada documento ER fue procesado 100 veces, obteniéndose el promedio del tiempo de ejecución. Los resultados de las mediciones se encuentran en las Figuras 5.1, 5.2, 5.3 y 5.4. Se observa que el parser es la etapa que más tiempo demora al procesar documentos ER. Sin embargo, el tiempo total de procesamiento es menor a 6 milisegundos para todos los documentos ER probados. Para documentos ER más pequeños, se logran tiempos de procesamiento entre 0 y 3 milisegundos.

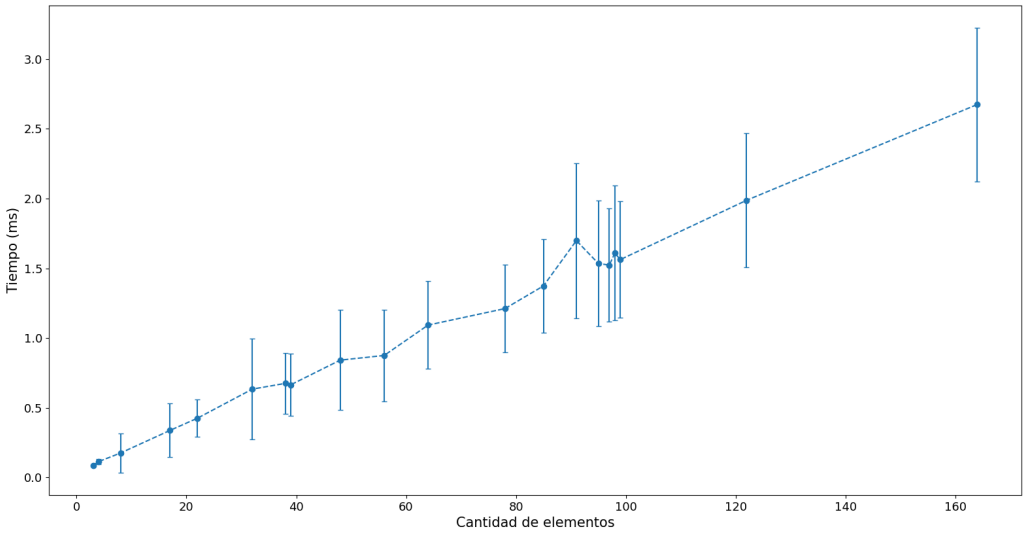

Figura 5.1: Tiempo (en milisegundos) de ejecución del parser de ERdoc, para documentos ER con distintas cantidades de elementos (entidades, relaciones y agregaciones).

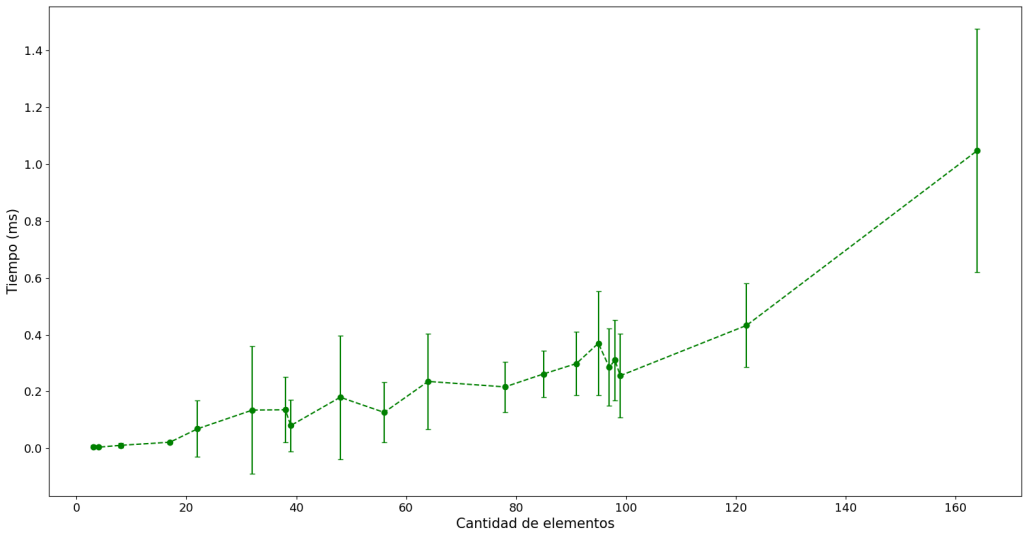

Figura 5.2: Tiempo (en milisegundos) de ejecución del linter de ERdoc, para documentos ER con distintas cantidades de elementos.

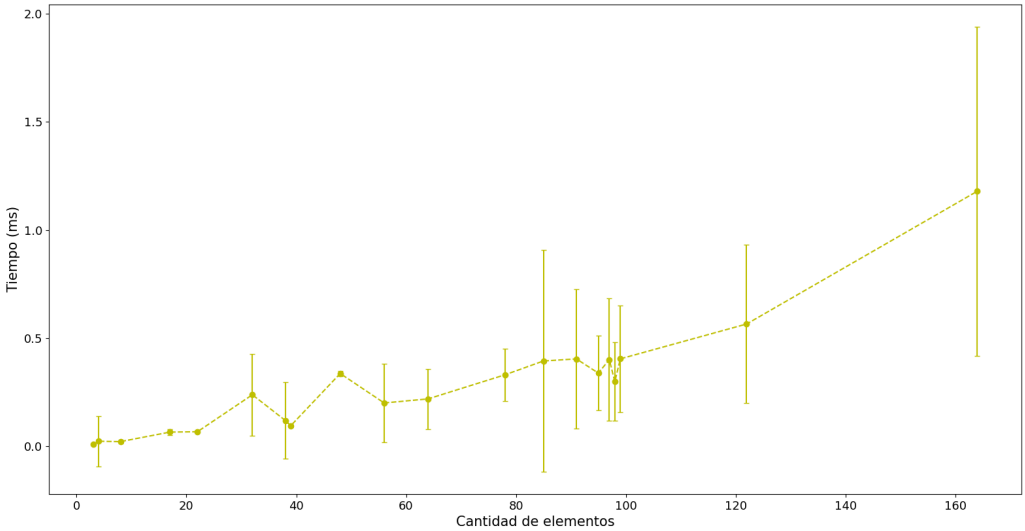

Figura 5.3: Tiempo (en milisegundos) de ejecución de convertir un documento ER a un grafo, para documentos ER con distintas cantidades de elementos.

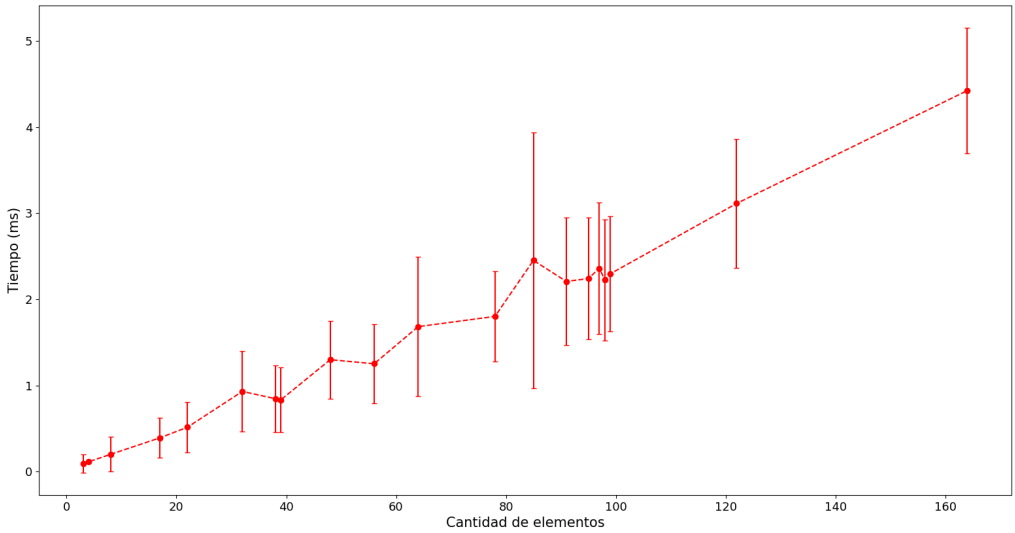

Figura 5.4: Tiempo (en milisegundos) total en procesar un documento ER: ejecutar parser, linter y convertir a grafo sucesivamente, para documentos ER con distintas cantidades de elementos.

El tiempo de ejecución de los algoritmos de layout se midió sobre los grafos generados a partir de los mismos documentos ER utilizados para evaluar el procesamiento de estos. Así, los algoritmos se evaluaron en grafos de tamaños más grandes a los que se espera que sean creados por los usuarios. Para obtener el tiempo promedio, cada algoritmo se evaluó 10 veces.

La Figura 5.5 muestra los tiempos de procesamiento (en escala logarítmica) para los algoritmos de *force-directed layout* de ELK y WebCola, junto al algoritmo *multi-layout*. Se verifica que *multi-layout* es mucho más rápido en comparación a los otros algoritmos, obteniendo un tiempo de ejecución máximo de 462 milisegundos, mientras que los otros algoritmos sobrepasan los 10 segundos de ejecución para grafos con una alta cantidad de elementos.

En conclusión, se verifica que la aplicación logra generar diagramas en tiempo real, convirtiendo documentos ER a diagramas ER en menos de 6 milisegundos y ejecutando el algoritmo *multi-layout* en menos de 500 milisegundos. Por lo tanto, la aplicación permite ejecutar el flujo principal con cada entrada del usuario sin causar problemas de rendimiento.

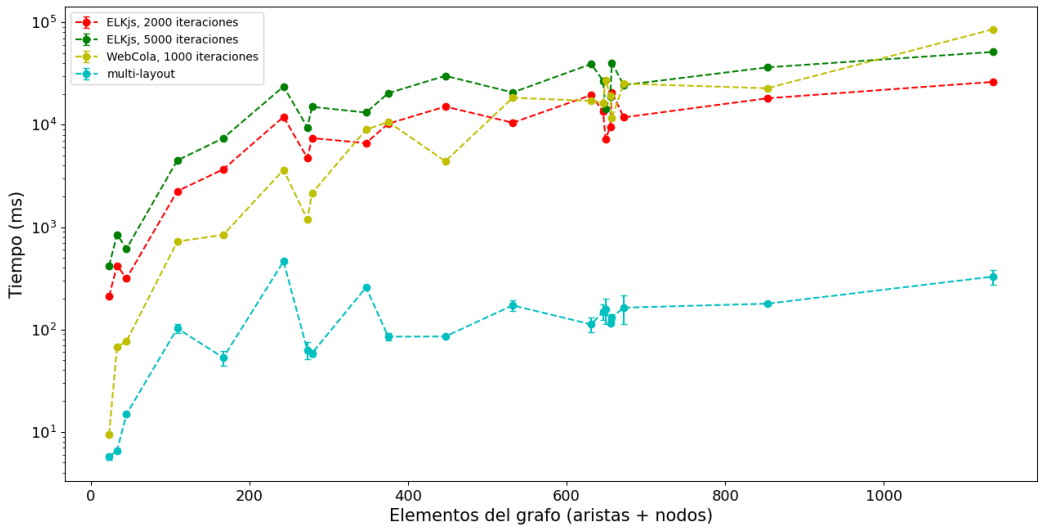

Figura 5.5: Tiempo (en milisegundos) en ejecutar los algoritmos de *layout*: *force-directed layout* de ELKjs con 2500 y 5000 iteraciones, *force-directed layout* de WebCola con 1000 iteraciones y *multi-layout*.

5.2. Usabilidad

5.2.1. Metodología de Evaluación

Para medir la usabilidad del sistema se aplicó una encuesta, consistente de una actividad, levemente guiada, seguido de una serie de preguntas acerca de la actividad desarrollada y la usabilidad de la aplicación. La encuesta se hizo en Google Forms[1].

La encuesta se difundió por el foro de la comunidad de estudiantes del departamento de Ciencias de la Computación de la Universidad de Chile (DCC), además se contactó directamente a ayudantes, auxiliares y profesores de cátedra del curso Bases de Datos del DCC.

La primera parte de la encuesta consiste de una caracterización del usuario, mediante las preguntas:

1. Indique el máximo rol alcanzado en relación a cursos del área de Bases de Datos.
2. ¿Cuándo fue su última participación en un curso del área de Bases de Datos?

Para la pregunta número 1 las alternativas de respuesta son:

- No he cursado cursos de bases de datos
- Estudiante
- Ayudante
- Profesor auxiliar
- Profesor de cátedra

Mientras que para la pregunta número 2, las alternativas de respuesta son:

- Este año
- Hace 1-3 años
- Hace más de 3 años

Luego, la encuesta continúa con la actividad a desarrollar por los usuarios. La actividad consiste en crear un diagrama ER utilizando ERdoc Playground. Para esto, en el formulario se le propone al usuario un modelo ER representando una universidad, el modelo consiste de dos entidades, una entidad débil y dos relaciones. La actividad presente en la encuesta es la siguiente:

[1] https://forms.google.com/

Actividad Propuesta

Comience con las entidades que forman parte del modelo ER:

1. Para comenzar, cree la entidad **Alumno**, con los atributos: "nombre", "email", "RUT" y "carrera". Notará que el diagrama no se ha creado, pues se deben respetar las reglas del modelo ER para visualizar el diagrama. En este caso, la entidad no tiene una llave. Arregle este problema haciendo que el atributo "RUT" sea la llave de **Alumno**.

2. Cree la entidad **Curso**, con los atributos "nombre", "código" y "sección". La llave está formada por los atributos "código" y "sección".

Seguimos con una relación entre las entidades del modelo:

1. Cree la relación **cursa**, entre **Alumno** y **Curso**. En este modelo, un curso tiene 1 o más alumnos, y un alumno cursa 0 o más cursos.

Finalmente, añadimos una entidad débil y una relación al modelo:

1. Cree la entidad débil **Evaluación**, esta depende de la entidad Curso a través de la relación **tiene**. Una evaluación posee los atributos "nombre" y "fecha". Su llave parcial corresponde al atributo "nombre". Observe que se muestra el error 'La entidad débil .ᴱᵛaluación"depende de la relación "tiene", que no existe'.

2. Para corregir el error, cree la relación **tiene**, en la cual participa **Evaluación** y **Curso**. En este caso, un curso debe tener una o más evaluaciones, mientras que una evaluación pertenece a exactamente un curso. Note que, al ser Curso una entidad débil, este debe poseer participación total en la relación, sino se mostrará un error.

Luego de la actividad, se le pide al usuario ingresar el documento ER que desarrolló para cumplir el objetivo. Junto a esto, se pregunta si se encontró con algún problema u otra complicación durante el desarrollo de la actividad.

La encuesta prosigue con el cuestionario *System Usability Scale* (SUS) [2]. SUS es un cuestionario de 10 preguntas, que tiene por objetivo medir la usabilidad de un sistema. Cada una de las 10 preguntas tiene cinco opciones de respuesta, puntuaciones de 1 a 5, donde 1 representa "Totalmente en desacuerdo" y 5, "Totalmente de acuerdo". Es importante mencionar que las preguntas originales fueron traducidas al español. A continuación, las 10 preguntas de esta etapa de la encuesta:

1. Creo que me gustaría usar este sistema con frecuencia.
2. Encontré el sistema innecesariamente complejo.
3. El sistema me pareció fácil de usar.
4. Creo que necesitaría el apoyo de un técnico para poder usar este sistema.
5. Encontré que las diversas funcionalidades de este sistema estaban bien integradas.

6. Encontré que había demasiada inconsistencia en este sistema.
7. Pienso que la mayoría de las personas aprenderían a usar este sistema muy rápidamente.
8. Encontré que el sistema era muy engorroso de usar.
9. Me sentí muy seguro al usar el sistema.
10. Necesité aprender muchas cosas antes de poder empezar a usar este sistema.

Para calcular la puntuación de una respuesta al cuestionario SUS, se suman las puntuaciones individuales de cada pregunta. Para calcular la puntuación de una respuesta a las preguntas 1, 3, 5, 7 y 9, se le resta 1 a la respuesta. Para las demás preguntas la puntuación corresponde a 5 menos la respuesta. Por último, la suma de las puntuaciones se multiplica por 2,5 para obtener la puntuación final. La puntuación final corresponde a un valor entre 0 y 100.

Una puntuación sobre el promedio corresponde a 68 [12], mientras que con una puntuación de 80,3, es probable que los usuarios recomienden el sistema a un amigo [12]. Por otro lado, Tullis y Stetson [15] encontraron que desde una cantidad de 12 participantes SUS obtiene resultados confiables.

Finalmente, la encuesta termina con las preguntas:

1. Si tuviera que hacer un diagrama ER, usaría esta aplicación (respuesta de 1 a 5, al igual que en la sección anterior).
2. En caso de preferir otra aplicación, indíquela acá (pregunta abierta).
3. Comentarios positivos acerca de la aplicación (pregunta abierta).
4. Comentarios negativos acerca de la aplicación (pregunta abierta).

5.2.2. Resultados

La encuesta fue respondida por 21 personas. 10 estudiantes del DCC, además de 3 ayudantes, 5 profesores auxiliares y 3 profesores de cátedra, roles en relación a cursos de Bases de Datos. Los profesores guía y co-guía de este trabajo no respondieron la encuesta.

En cuanto a la actividad, 9 personas la respondieron de manera completamente correcta. Mientras que 11, de manera parcialmente correcta o incorrecta (entendido como que el documento ER que ingresaron para generar el diagrama ER no representa exactamente el modelo ER planteado en la actividad). Una persona tuvo problemas para copiar su respuesta, por lo que no se puede saber si su respuesta fue correcta o no. Para las respuestas entregadas, todos lograron ingresar documentos ER sintácticamente válidos e ingresar de forma correcta las entidades y entidades débiles, declarando bien los atributos que correspondían a llaves y llaves parciales. Para las respuestas con errores, todos los errores ocurrieron al declarar las restricciones de cardinalidad y participación de las relaciones presentes en el modelo.

Las principales problemáticas con las que se encontraron los usuarios durante el desarrollo de la actividad fueron:

- Al hacer clic en los botones para cargar ejemplos, el documento ER que los usuarios habían escrito en el editor se reemplazaba por el ejemplo; además, un *bug* en la aplicación no hacía posible volver al documento ER anterior (ya sea con la combinación de teclas $Control + Z$ o el botón de revertir último cambio). Este *bug* fue solucionado una vez finalizada la encuesta.
- Algunos usuarios tuvieron problemas para entender la sintaxis de las restricciones de cardinalidad y participación.

Los resultados de cada pregunta del cuestionario SUS se muestran en la Tabla 5.1. La aplicación obtuvo una puntuación final promedio de 83,45, con una desviación estándar de 12,73. Esto indica que ERdoc Playground es usable de acuerdo a la percepción de los encuestados.

Tabla 5.1: Respuestas promedio a cada pregunta del cuestionario SUS. Las puntuaciones van desde 1 a 5. Para preguntas impares, un valor más alto es mejor. Para preguntas pares, un valor más bajo es mejor.

Pregunta	**Respuesta Promedio**	**Desviación estándar**
1	4,10	0,83
2	1,57	0,60
3	4,62	0,59
4	1,48	0,81
5	4,71	0,56
6	1,57	0,81
7	4,10	0,89
8	1,67	0,80
9	4,24	0,89
10	2,10	1,18

En la última sección de la encuesta, la pregunta "Si tuviera que hacer un diagrama ER, usaría esta aplicación" obtuvo una respuesta promedio de 4,33, con una desviación estándar de 1,06. Evaluándose de buena forma la percepción de la aplicación con respecto a las existentes.

Con respecto a los comentarios generales y positivos de la aplicación, algunos que se destacan son:

- Varios usuarios valoran el editor de texto en el que se ingresan los documentos ER y sus características: *syntax highlighting*, detección de errores y autocompletado.
- Se menciona varias veces la facilidad de uso de la aplicación, junto a la fluidez del flujo de trabajo para generar diagramas.
- En varias respuestas se valora el que los diagramas se generen en tiempo real.

Por otro lado, entre los comentarios generales y negativos sobre la aplicación, se repiten:

- A algunos usuarios les gustaría personalizar los elementos del diagrama, cambiando su color, tamaño, fuente de las letras, etc.
- Varios destacan el *bug* relacionado a cargar ejemplos, mencionado anteriormente.
- Un usuario mencionó que no pudo crear una entidad débil que dependiera de más de una relación. Inicialmente, el lenguaje ERdoc no permitía tal expresión. Una vez finalizada la encuesta, se agregó al lenguaje una expresión para definir múltiples relaciones débiles para una entidad débil.

Finalmente, mediante los resultados obtenidos en SUS, los comentarios de los usuarios y el bajo tiempo de respuesta en las funciones ocupadas en el flujo principal de la aplicación, se puede concluir que ERdoc Playground es una aplicación usable y con un buen rendimiento, permitiendo generar diagramas ER en tiempo real.

Capítulo 6

Conclusión

Durante esta memoria se diseñó ERdoc, un lenguaje de marcado para representar modelos entidad-relación mediante documentos ER (una cadena de texto válida, de acuerdo a la gramática del lenguaje). ERdoc permite representar entidades, entidades débiles, relaciones, junto a los conceptos de agregación y jerarquía de clases. A su vez, se implementó un parser para el lenguaje. También, se desarrolló un linter para ERdoc, módulo que detecta múltiples tipos de errores semánticos en documentos ER.

Por otro lado, se desarrolló una aplicación web, denominada ERdoc Playground, que permite crear diagramas ER en tiempo real a partir de documentos ER. Esta aplicación permite visualizar diagramas ER con la notación de Chen, y con tres notaciones de multiplicidades: la notación de Chen original, la notación (mín, máx) y la notación de flechas. Se implementó una funcionalidad para posicionar automáticamente los elementos del diagrama, de manera que la visualización sea atractiva. Los diagramas generados pueden ser exportados a distintos formatos, como imágenes, PDF o JSON. Por último, se creó una documentación para introducir el lenguaje ERdoc.

ERdoc Playground es gratis, de código abierto y se encuentra disponible en la dirección `https://erdoc.dcc.uchile.cl/`. El código fuente se encuentra en `https://github.com/matias-lg/er/`. La aplicación está licenciada bajo la licencia MIT.

Para validar el rendimiento de ERdoc Playground, se hicieron mediciones a las principales funcionalidades de la aplicación. Se obtuvo que el tiempo total de procesar un documento ER para generar un diagrama ER toma menos de diez milisegundos, mientras que el algoritmo de *layout* desarrollado es lo suficientemente rápido para ejecutarse cada vez que un usuario modifica su entrada para diagramas con hasta 1138 elementos (nodos y aristas), generando un retraso de menos de 500 milisegundos. Así, se verifica que la aplicación permite generar diagramas ER a medida que el usuario ingresa un documento ER de forma eficiente, en tiempo real.

Otro aspecto que se validó fue la usabilidad de la aplicación mediante una encuesta. La encuesta fue respondida por estudiantes y miembros del cuerpo docente del curso CC3201 Bases de Datos, del departamento de Ciencias de la Computación de la Universidad de Chile. Los resultados que se obtuvieron fueron positivos, logrando una buena evaluación de

la usabilidad de la aplicación a través del cuestionario *System usability scale*.

Así, se da por cumplido el objetivo general de la memoria: "desarrollar una aplicación web gratuita, de código abierto, que permita crear diagramas ER interactivos a través de un lenguaje de marcado, utilizando un lenguaje de definición propio. La herramienta utiliza la notación de Chen para visualizar los diagramas". En cuanto a los siete objetivos específicos que se definieron en la Sección 1.1, los objetivos 1, 2 y 3 se cumplieron mediante el diseño del lenguaje ERdoc y la implementación de su parser, el cual posee detección de errores sintácticos. El objetivo 4, relacionado a la detección de errores semánticos, se cumplió con la implementación del linter de ERdoc, pudiendo detectar 19 tipos distintos de errores semánticos. A través del desarrollo de ERdoc Playground, se cumple el objetivo 5. En cuanto al objetivo 6, relacionado a que los diagramas ER generados por la aplicación sean interactivos, se cumple parcialmente, ya que solamente se puede interactuar con los elementos del diagrama arrastrándolos, para cambiarlos de posición. El objetivo 7 se cumplió, pues tanto ERdoc Playground como su código fuente, han quedado disponibles en URLs públicas.

Como trabajo futuro, se podrían realizar extensiones y mejoras a ERdoc Playground. En particular, se podrían agregar formas de personalizar los diagramas ER, como cambiar el color, tamaño o fuente de elementos individuales de un diagrama. También, se podría mejorar el algoritmo de *layout* para que posicione los elementos del diagrama de forma más "natural", logrando disposiciones más parecidas a los ejemplos disponibles en ERdoc Playground (los cuales están posicionados manualmente). Otro aspecto a mejorar en la aplicación es el funcionamiento en dispositivos móviles; actualmente es difícil utilizar el editor de texto y visualizar los diagramas al mismo tiempo.

Por el lado del lenguaje ERdoc, se podrían crear nuevas herramientas que lo utilicen, como una utilidad para generar modelos relacionales a partir de documentos ER.

Finalmente, se espera que ERdoc Playground y el lenguaje ERdoc sean herramientas de utilidad para estudiantes que se encuentren aprendiendo sobre modelado entidad-relación, facilitando la creación de diagramas ER y evitando problemas de notación.

Bibliografía

[1] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry*, 4(5):235–282, 1994.

[2] John Brooke. Sus: A quick and dirty usability scale. *Usability Eval. Ind.*, 189, 11 1995.

[3] Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, mar 1976.

[4] P. Eades. *Drawing Free Trees*. IIAS-RR-. International Institute for Advanced Study of Social Information Science, Fujitsu Limited, 1991.

[5] Peter Eades. A heuristic for graph drawing. *Congressus numerantium*, 42(11):149–160, 1984.

[6] Ramez El-Masri and Gio Wiederhold. Properties of relationships and their representation. In *Proceedings of the May 19-22, 1980, National Computer Conference*, AFIPS '80, page 319–326, New York, NY, USA, 1980. Association for Computing Machinery.

[7] Gordon Everest. Basic data structure models explained with a common example. In *Proceedings of the October 18-19, Fifth Texas Conference on Computing Systems October 18-19.*, pages 39–46, Austin, TX, USA, 10 1976.

[8] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. *SIGPLAN Not.*, 39(1):111–122, jan 2004.

[9] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, page 111–122, New York, NY, USA, 2004. Association for Computing Machinery.

[10] Guido van Rossum, Pablo Galindo and Lysandros Nikolaou. Pep 617 – new peg parser for cpython. `https://peps.python.org/pep-0617/`.

[11] Holistics. Dbml (database markup language). `https://github.com/holistics/dbml`.

[12] Jeff Sauro. Measuring usability with the system usability scale (sus). `https://measuringu.com/sus/`.

[13] John Miles Smith and Diane C. P. Smith. Database abstractions: Aggregation. *Commun. ACM*, 20(6):405–413, jun 1977.

[14] Il-Yeol Song, Mary Evans, and Eui Kyun Park. A comparative analysis of entity-relationship diagrams. *Journal of Computer and Software Engineering*, 3, 01 1995.

[15] Jacqueline N Stetson and Thomas S Tullis. A comparison of questionnaires for assessing website usability. *UPA Presentation*, 2004.

[16] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981.

Anexo

A.1. Representación de modelo ER

El modelo ER a representar consiste de:

- La entidad **Producto** con los atributos: nombre (llave), precio, marca
- La entidad **Tienda** con los atributos: nombre (llave), dirección, email
- La relación **Vende** en la que participa **Producto** y **Tienda**

1. DBML:

```
Table entity.Producto {
  nombre string [primary key]
  precio int
  marca string
}

Table entity.Tienda {
  nombre string [primary key]
  direccion string
  email string
}

Table relation.Vende {
  nombre_producto string
  nombre_tienda string
}

Ref: relation.Vende.nombre_producto > entity.Producto.nombre
Ref: relation.Vende.nombre_tienda - entity.Tienda.nombre
```

2. SQL (PostgreSQL):

```
CREATE SCHEMA entity;
CREATE SCHEMA relation;

CREATE TABLE entity.Producto (
  nombre VARCHAR PRIMARY KEY,
  precio INTEGER,
  marca VARCHAR
);

CREATE TABLE entity.Tienda (
  nombre VARCHAR PRIMARY KEY,
  direccion VARCHAR,
  email VARCHAR
);

CREATE TABLE relation.Vende (
  nombre_producto VARCHAR REFERENCES entity.Producto(nombre),
  nombre_tienda VARCHAR REFERENCES entity.Tienda(nombre)
);
```

3. JSON:

```
{
    "entities": [
        {
            "name": "Producto",
            "attributes": [
                {"name": "nombre", "isKey": true},
                {"name": "precio", "isKey": false},
                {"name": "marca", "isKey": false}
            ]
        },

        {
            "name": "Tienda",
            "attributes": [
                {"name": "nombre", "isKey": true},
                {"name": "direccion", "isKey": false},
                {"name": "email", "isKey": false}
            ]
        }
    ],
    "relations": [
        {
            "name": "Vende",
            "members": [
                {"name": "Producto", "cardinality": "N"},
                {"name": "Tienda", "cardinality": "0 or 1"}
            ]
        }
    ]
}
```

4. YAML:

```
entities:
- name: Producto
  attributes:
  - name: nombre
    isKey: true

  - name: precio
    isKey: false

  - name: marca
    isKey: false
- name: Tienda
  attributes:
  - name: nombre
    isKey: true

  - name: direccion
    isKey: false

  - name: email
    isKey: false

relations:
- name: Vende
  members:
  - name: Producto
    cardinality: N

  - name: Tienda
    cardinality: 0 or 1
```

A.2. Resultado de parsear un documento ER

El resultado de parsear el documento ER mostrado en la sección 3.2.6 es el siguiente objeto de Typescript:


```
{
  "entities": [
    {
      "type": "entity",
      "name": "Estudiante",
      "attributes": [
        {
          "name": "RUT",
          "location": {
            "start": {
              "offset": 21,
              "line": 3,
              "column": 1
            },
            "end": {
              "offset": 28,
              "line": 3,
              "column": 8
            }
          },
          "isKey": true,
          "isComposite": false,
          "childAttributesNames": null
        },
        {
          "name": "nombre_completo",
          "location": {
            "start": {
              "offset": 29,
              "line": 4,
              "column": 1
            },
            "end": {
              "offset": 64,
              "line": 4,
              "column": 36
            }
          },
          "isKey": false,
          "isComposite": true,
          "childAttributesNames": [
            "nombre",
```

```
        "apellido"
      ]
    },
    {
      "name": "edad",
      "location": {
        "start": {
          "offset": 65,
          "line": 5,
          "column": 1
        },
        "end": {
          "offset": 69,
          "line": 5,
          "column": 5
        }
      },
      "isKey": false,
      "isComposite": false,
      "childAttributesNames": null
    }
  ],
  "hasParent": false,
  "parentName": null,
  "hasDependencies": false,
  "dependsOn": null,
  "location": {
    "start": {
      "offset": 1,
      "line": 2,
      "column": 1
    },
    "end": {
      "offset": 18,
      "line": 2,
      "column": 18
    }
  }
},
{
  "type": "entity",
  "name": "Universidad",
  "attributes": [
    {
      "name": "nombre",
      "location": {
        "start": {
```

```
      "offset": 93,
      "line": 8,
      "column": 1
    },
    "end": {
      "offset": 99,
      "line": 8,
      "column": 7
    }
  },
  "isKey": false,
  "isComposite": false,
  "childAttributesNames": null
},
{
  "name": "dirección",
  "location": {
    "start": {
      "offset": 100,
      "line": 9,
      "column": 1
    },
    "end": {
      "offset": 109,
      "line": 9,
      "column": 10
    }
  },
  "isKey": false,
  "isComposite": false,
  "childAttributesNames": null
},
{
  "name": "university_id",
  "location": {
    "start": {
      "offset": 110,
      "line": 10,
      "column": 1
    },
    "end": {
      "offset": 127,
      "line": 10,
      "column": 18
    }
  },
  "isKey": true,
```

```
        "isComposite": false,
        "childAttributesNames": null
      }
    ],
    "hasParent": false,
    "parentName": null,
    "hasDependencies": false,
    "dependsOn": null,
    "location": {
      "start": {
        "offset": 72,
        "line": 7,
        "column": 1
      },
      "end": {
        "offset": 90,
        "line": 7,
        "column": 19
      }
    }
  }
],
"relationships": [
  {
    "type": "relationship",
    "name": "Estudia_en",
    "participantEntities": [
      {
        "entityName": "Estudiante",
        "isComposite": false,
        "cardinality": "1",
        "participation": "partial",
        "location": {
          "start": {
            "offset": 150,
            "line": 12,
            "column": 21
          },
          "end": {
            "offset": 162,
            "line": 12,
            "column": 33
          }
        }
      },
      {
        "entityName": "Universidad",
```

```
    "isComposite": false,
    "cardinality": "N",
    "participation": "total",
    "location": {
      "start": {
        "offset": 164,
        "line": 12,
        "column": 35
      },
      "end": {
        "offset": 178,
        "line": 12,
        "column": 49
      }
    }
  }
],
"attributes": [
  {
    "name": "fecha_ingreso",
    "isComposite": false,
    "childAttributesNames": null,
    "location": {
      "start": {
        "offset": 182,
        "line": 13,
        "column": 1
      },
      "end": {
        "offset": 195,
        "line": 13,
        "column": 14
      }
    }
  }
],
"location": {
  "start": {
    "offset": 130,
    "line": 12,
    "column": 1
  },
  "end": {
    "offset": 149,
    "line": 12,
    "column": 20
  }
```

```
          }
        }
      ],
      "aggregations": [
        {
          "type": "aggregation",
          "name": "Estudiante_Estudia_en_Universidad",
          "aggregatedRelationshipName": "Estudia_en",
          "location": {
            "start": {
              "offset": 198,
              "line": 15,
              "column": 1
            },
            "end": {
              "offset": 255,
              "line": 15,
              "column": 58
            }
          }
        }
      ]
    }
```


A.3. Diagrama ER con distintas notaciones

Un ejemplo de documento ER de un banco es el siguiente:

```
ENTITY banco {
    código key
    nombre
    dirección
}

ENTITY sucursal DEPENDS ON tiene_suc {
    dirección
    no_suc pkey
}

RELATION tiene_suc(banco 1!, sucursal N!)

ENTITY cuenta {
    no_cuenta key
    saldo
    tipo
```

```
}

ENTITY préstamo {
    no_prest key
    monto
    tipo
}

RELATION tiene_cta(banco_con_sucs 1, cuenta N!)
RELATION tiene_prest(banco_con_sucs 1, préstamo N!)

ENTITY cliente {
    RUT key
    nombre
    dirección
    teléfono
}

ENTITY clienteVIP EXTENDS cliente {
    descuento
}

RELATION a_c(cliente N, cuenta M!)
RELATION l_c(cliente N, préstamo M!)

aggregation banco_con_sucs(tiene_suc)
```

Utilizando ERdoc Playground se crean los siguientes diagramas, seleccionando la notación de flechas (figura A.1), la notación de Chen (figura A.2) y la notación (mín, máx), en la figura A.3.

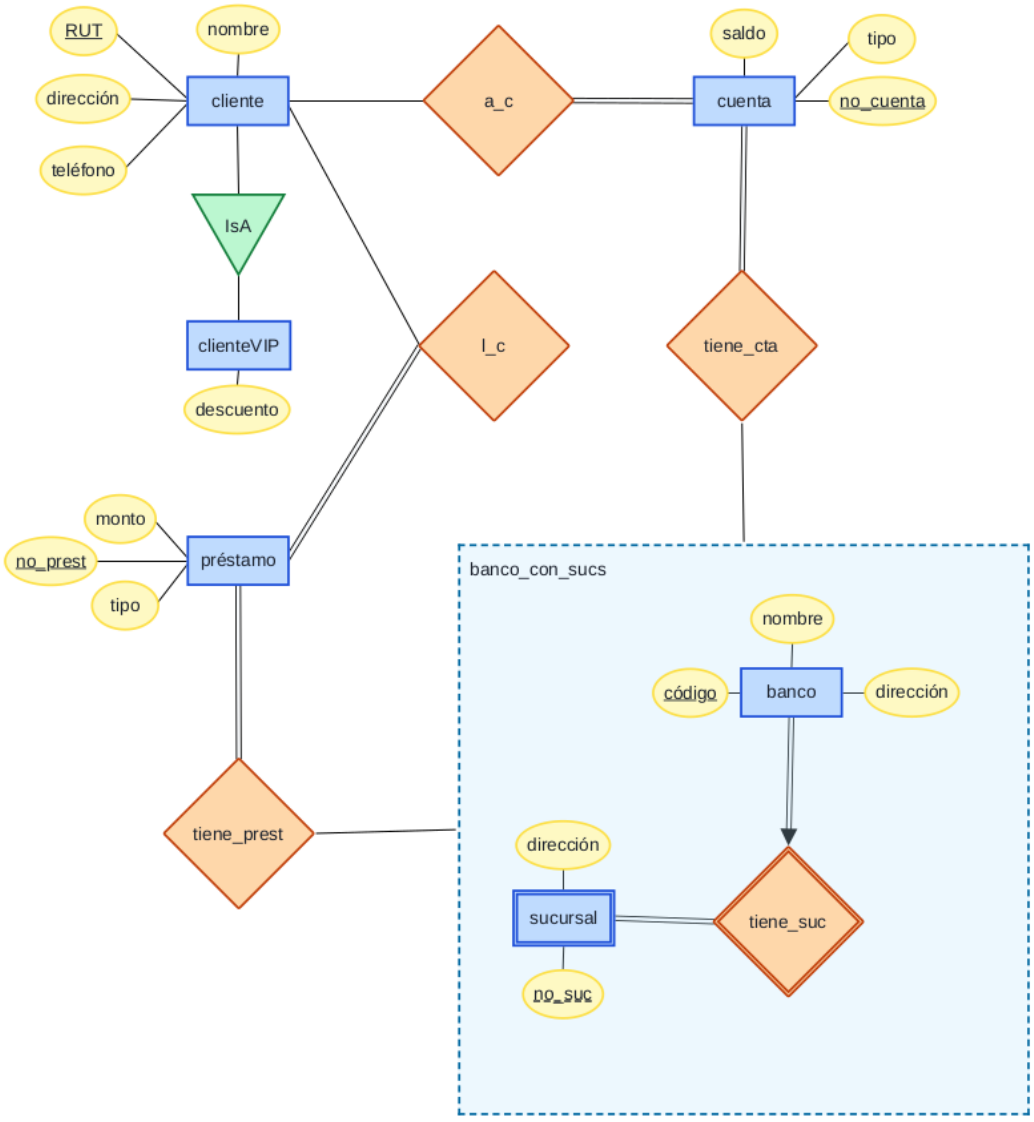

Figura A.1: Diagrama ER generado con la notación de flechas.

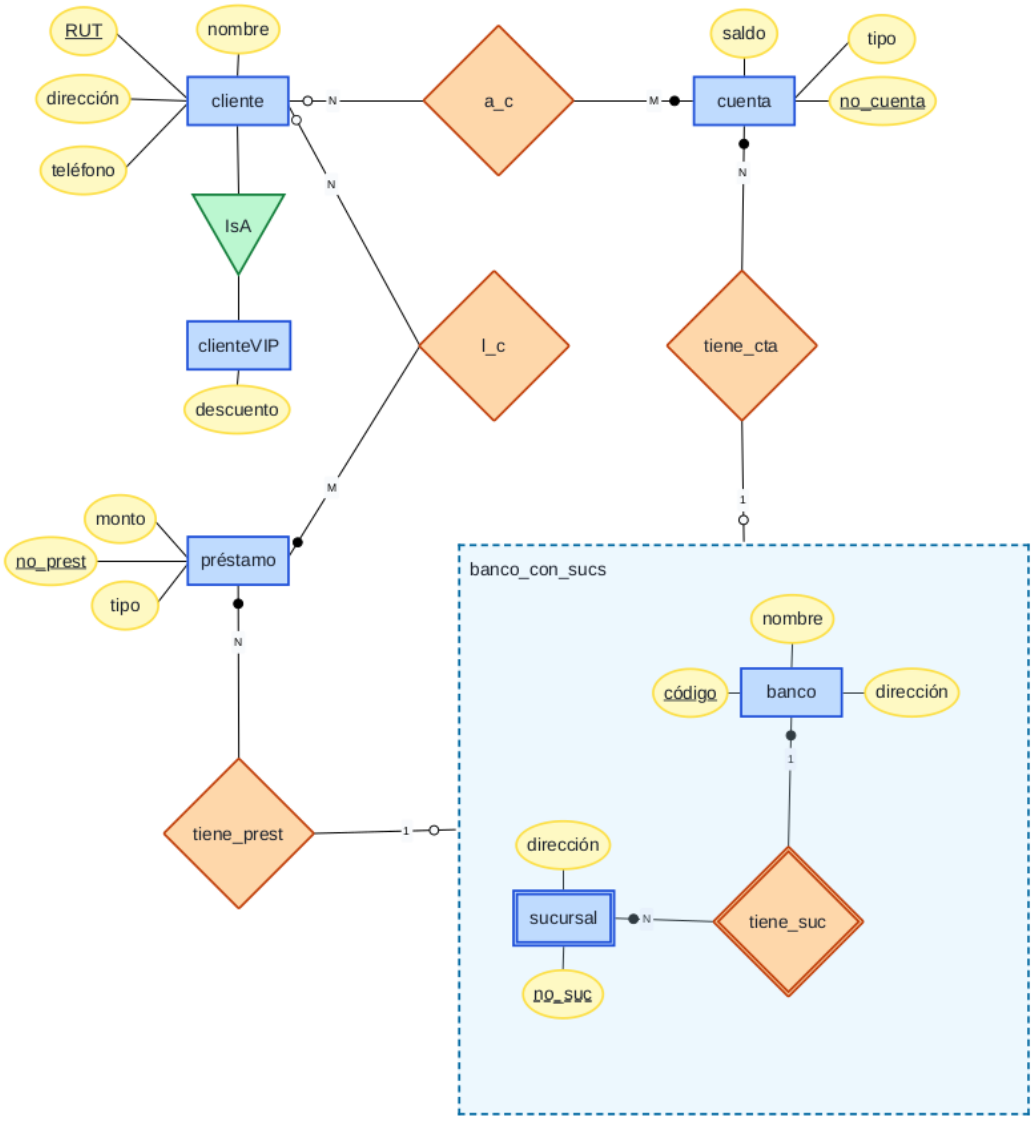

Figura A.2: Diagrama ER generado con la notación de Chen.

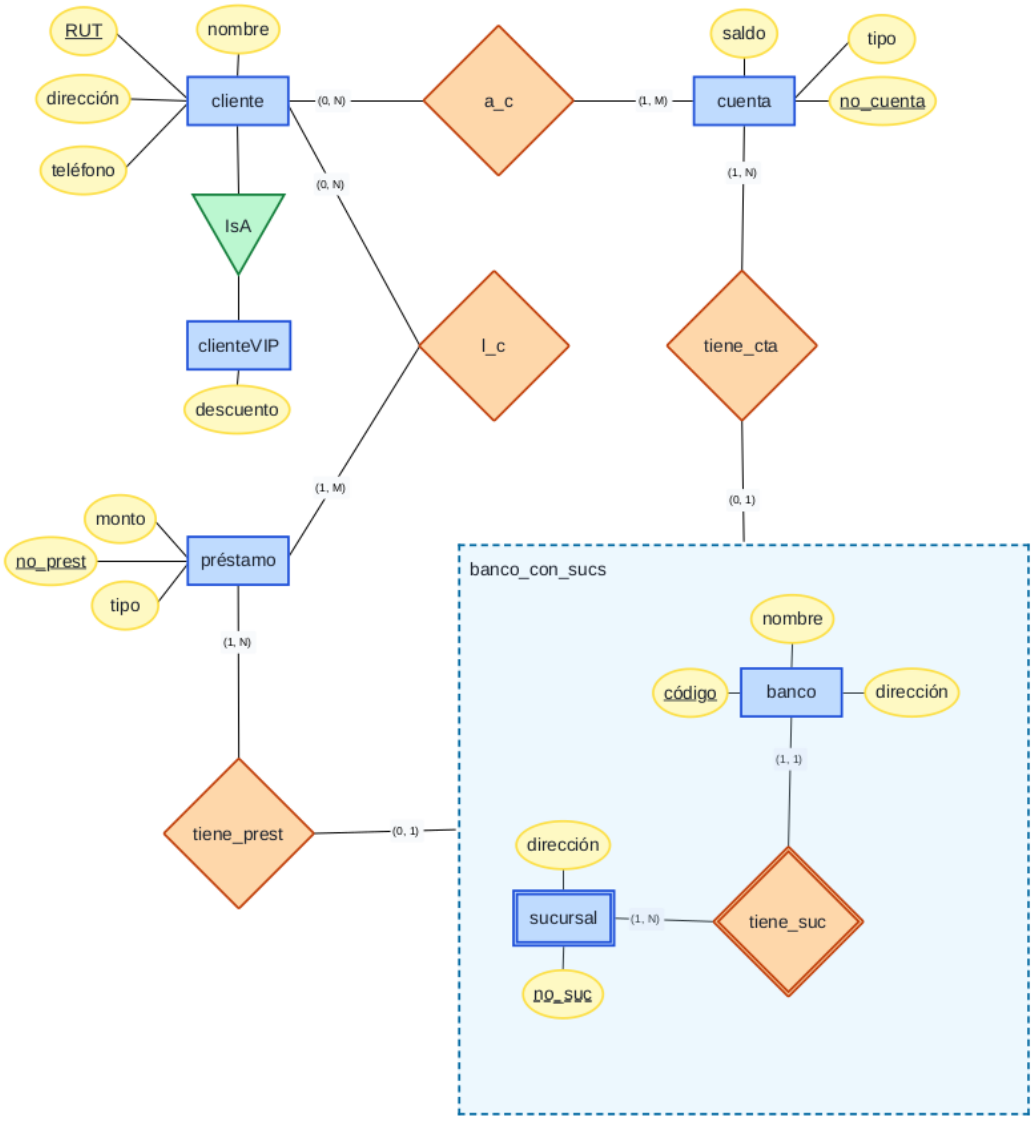

Figura A.3: Diagrama ER generado con la notación (mín, máx).

A.4. Documento ER de ejemplo para visualizar layouts

```
ENTITY Empleado {
    e_id key
    nombre
}

ENTITY Departamento {
    d_número key
    nombre
}


ENTITY Empresa {
    e_id key
    nombre
    dirección
}


ENTITY Bien {
    id key
    tipo
    precio
}

RELATION posee(Empresa, Bien)
aggregation Empresa_con_Bienes(posee)


RELATION Cumple(Empresa_con_Bienes,Proyecto)

RELATION Parte_de(Empleado N, Departamento 1!)

ENTITY Proyecto {
    código key
}

RELATION Maneja(Departamento 1, Proyecto N!)

RELATION Trabaja_en(Empleado M, Proyecto N) {
    hours
}

ENTITY Pieza {
    p_número key
```

```
    nombre
}

ENTITY Tornillo EXTENDS Pieza {
    tipo_cabeza
}


ENTITY Pantalla EXTENDS Pieza {
    dimensiones
}


ENTITY Proveedor {
    código key
    nombre
}

RELATION Provee(Proyecto M, Pieza N!, Proveedor P!) {
    Quantity
}

ENTITY Practicante DEPENDS ON Depende_de {
    nombre pkey
    género
}

RELATION Depende_de(Empleado 1, Practicante N!)
```