F_RULE: Un nuevo lenguaje de programación creado para evaluar reglas de flujo en una plataforma de desarrollo de sistemas de gestión de procesos

F_RULE: A new programming language created to evaluate flow rules in a process management systems development platform

Fabián Silva Pavez[1,2*] Marco Mora Cofré[2]

Recibido 6 de Enero de 2021, aceptado 12 de Mayo de 2021
Received: January 6, 2021 Accepted: May 12, 2021

RESUMEN

Este trabajo presenta un nuevo lenguaje de programación, integrado en una plataforma que permite crear aplicaciones de gestión de procesos en donde es utilizado para interpretar instrucciones que representan reglas de flujo asociados a cada uno de los procesos. Se aborda el origen del problema, alternativas de solución, y la motivación para la propuesta. Se describen las características del lenguaje y los elementos que lo componen. Posteriormente se valida la propuesta a través de una serie de casos de prueba en los que se da solución a la problemática mediante la creación de programas escritos en F_RULE. Finalmente desde los casos de prueba se observa que la propuesta realizada se adapta correctamente a diferentes situaciones típicas de trabajo en sistemas de gestión de procesos de negocios.

Palabras clave: lenguaje de programación, reglas de flujo, BPMS.

ABSTRACT

This work presents a new programming language, integrated in a platform that allows creating process management applications where it is used to interpret instructions that represent flow rules associated with each of the processes. The origin of the problem, alternative solutions, and the motivation for the proposal are addressed. The characteristics of the language and the elements that compose it are described. Subsequently, the proposal is validated through a series of test cases in which the problem is solved by creating programs written in F_RULE. Finally, from the test cases, it can be seen that the proposal made adapts correctly to different typical work situations in business process management systems.

Keywords: programming language, flow rules, BPMS.

INTRODUCCIÓN

La facilidad que ofrece un computador como herramienta para resolver problemas abstraídos de la realidad definiendo un conjunto de datos que represente la situación real, y omitiendo aquellos que son irrelevantes hacen que sea imprescindible su utilización para automatizar cada vez más procesos de diversa índole [1]. Actualmente los lenguajes de programación mediante la escritura de programas permiten la comunicación usuario-máquina. Un lenguaje de programación representa una máquina abstracta capaz de interpretar los términos utilizados por un lenguaje formal [2], y permite que sean

[1] E-Process, Talca, Chile, fabian.silva@e-process.cl
[2] Universidad Católica del Maule, Talca, Chile, marcomoracofre@gmail.com
* Autor de correspondencia: f.silva.pavez@gmail.com

desarrollados algoritmos como programas que al ser ejecutados solucionan la problemática para la cual han sido creados.

Un sistema de gestión de procesos de negocios (BPMS) es un software empresarial para diseñar y ejecutar la automatización de los procesos. Una regla de negocio es una declaración que define o restringe algún aspecto del negocio, de tal forma que permite asegurar su estructura, controlar o influir en su negocio [3]. La estructura de las reglas de negocio está basada en el paradigma Evento-Condición-Acción (ECA) [4], las cuales se pueden clasificar en [5]: reglas de inicialización, eventos o reglas del proceso, reglas de flujo y reglas de término.

F_RULE es un lenguaje desarrollado para resolver la problemática de evaluar reglas de flujo de un BPMS en una plataforma de software llamada Dynamics[3] creada para este fin, y su intérprete como cualquier otra solución debe considerar las siguientes condiciones y restricciones impuestas por la plataforma:

i) Su manipulación no debe alterar el código fuente del sistema.
ii) Debe ser capaz de evaluar condiciones con tipos numéricos, texto y lógicos.
iii) Su entrada y salida debe ser automática, sin interacción.
iv) La solución debe ser robusta.

Aproximaciones previas

Antes de desarrollar el lenguaje y crear el intérprete, se evaluaron otras alternativas:

i) Evaluación de reglas en el propio sistema: Esto consiste en incorporar la lógica como parte del código fuente. En versiones anteriores de la plataforma las reglas de flujo y su evaluación eran parte del código fuente. Esta solución no cumple con el punto i) de las restricciones.
ii) Motor de reglas de negocios: un motor de reglas de negocio permite evaluar reglas que se encuentran codificadas con cierto formato establecido por el propio motor. Se analizaron algunos como por ejemplo ACTICO[4], pero el proceso de integración con la plataforma, además de los costos y el modelo de negocio de su empresa no es compatible con Dynamics.
iii) Motor de inferencia para reglas de negocios y procesos: La idea es codificar las reglas y utilizar un motor de inferencia para la evaluación. En este trabajo [6] se propone una forma en que las reglas se codifican como una tabla de decisión, y el motor de inferencia se basa en jBPM [7]. Se optó por no considerarla como opción porque requería la instalación de una máquina virtual para la ejecución del motor de inferencia.
iv) Uso de librerías de terceros: Por ejemplo Roslyn [8], una librería que permite interpretar instrucciones escritas en c#[5]. Esta aproximación es interesante, pero se desechó finalmente porque su licencia MIT [9] se utiliza bajo el concepto software-provided-as-is, es decir sin ningún tipo de garantía, en caso de fallas; y por supuesto sin posibilidad de corregirlo.
v) Incrustación de un lenguaje: Es posible incrustar un lenguaje de tal forma que sea posible ejecutar código directamente en la aplicación, por ejemplo Python. En C++ es nativa la incrustación [10] pero fue descartado porque para c# era necesario instalar un paquete como Python.NET[6] o IronPython[7] y la adición de librerías en el proyecto.
vi) Otras alternativas pensadas pero no consideradas: ejecución de intérpretes externos como Julia[8], Rust[9] o Perl[10]. No es adecuada la solución porque ante la necesidad de que sea automática su ejecución, funcionan como cajas negras. Finalmente se pensó en usar código SQL[11]. Esta opción agrega una complejidad innecesaria al tener que adaptarse a un lenguaje que claramente no fue pensado para este tipo de problemas.

La motivación para desarrollar este lenguaje es que no existe un producto con las características necesarias para cumplir con los requisitos funcionales y no funcionales en el mercado, además que desarrollar una solución propia en un ambiente controlado garantiza su robustez.

La estructura de este artículo es la siguiente: en la sección *F_RULE: lenguaje para evaluación de*

[3] Dynamics es el nombre de la plataforma de gestión de procesos de negocios desarrollada y explotada por la empresa E-Process. (www.e-process.cl)
[4] https://www.actico.com/platform/business-rules/
[5] Lenguaje C Sharp. http://csharp.net/
[6] Python.NET (http://pythonnet.github.io/)
[7] IronPython (https://ironpython.net/)
[8] The Julia programming language (https://docs.julialang.org/en/v1/)
[9] RUST (https://www.rust-lang.org/)
[10] PERL (https://www.perl.org)
[11] SQL (structured query language) estándar: https://www.iso.org/standard/63555.html

reglas de flujo se presentan de manera pedagógica las características del lenguaje F_RULE a través de ejemplos y la especificación de la sintaxis del lenguaje a través de las gramáticas libres del contexto (GLC), a continuación en la sección *Ejemplos de uso de F_RULE en problemas tradicionales de programación* se presentan 3 problemas tradicionales de programación y sus respectivas soluciones usando el lenguaje F_RULE, luego en la sección *Aplicación de F_RULE en la evaluación de reglas de flujo* se presentan 5 casos de prueba en el ámbito para el cuál fue concebido, finalmente se presentan las conclusiones y mejoras futuras al lenguaje F_RULE.

F_RULE: LENGUAJE PARA EVALUACIÓN DE REGLAS DE FLUJO

Características del lenguaje

F_RULE es un lenguaje de alto nivel, imperativo (sus instrucciones se ejecutan una tras otra en forma secuencial), con un sistema de tipos dinámico (la comprobación de tipos se realiza durante la ejecución), y de propósito general. Su conjunto de instrucciones permite realizar programas que resuelvan problemas de diversa índole, tal y como se mostrará más adelante en esta sección. A continuación se presenta el lenguaje a través de ejemplos para cada tipo de instrucciones, además de todos sus operadores, tipos de datos, formato de expresiones, y elementos de entrada y salida.

Código fuente 1. Programa *Hola mundo* en F_RULE.

```
frule holamundo
start
        out << “hola mundo”
end
```

Tipos de datos

Los tipos de datos soportados son: int (entero), str (string), real (real), bool (lógico).

Operadores

Tabla 1. Operadores de F_RULE

Aritméticos	Relacionales	Lógicos
+ suma	< menor que	&& and
* multiplicación	< = menor o igual que	|| or
- resta	> mayor que	~ not
/ división	> = mayor o igual que	
% módulo	= = igual a	
^ potencia	~ = distinto de	

Sintaxis de expresiones

Las expresiones de cualquier tipo se escriben en notación prefija, es decir, primero el operador y luego entre paréntesis y separados por coma los operandos.

Por ejemplo, la fórmula de la energía cinética:

$E_c = \frac{m \cdot v^2}{2}$, en F_RULE se escribe como:

Ec = /(*(m, ^(v, 2)), 2)

Entrada y salida de datos

El ingreso de datos se realiza con la instrucción in >> seguida por una lista de nombres de variables separadas por coma. Las variables se le inyectan en formato JSON [11], antes de ejecutarlo. La salida de datos corresponde a una variable llamada out, a la que se le pueden enviar expresiones de cualquier tipo, los que se irán concatenando. El siguiente ejemplo muestra un JSON de entrada, la forma de declararla en el código, y una salida con distintas expresiones.

Código fuente 2. Ejemplo de parámetros de entrada en formato JSON.

```
{params: [{“name”: “edad”, “value”: “45”, “type”:
            “int”},
            {“name”: “nombre”, “value”: “Dave Gahan”,
            “type”:”str”}] }
```

Código fuente 3. Ejemplo de entrada y salida en F_RULE.

```
frule testentrada
start
        in >> nombre, edad
        out << “Hola”, nombre, “usted tiene, edad,
        “años”
end
```

Instrucción condicional:

La instrucción condicional tiene 2 formas: if-end, e if-else-end, y es posible anidarla, tal como se muestra en el siguiente ejemplo:

Código fuente 4. Ejemplo de instrucción condicional en F_RULE.

```
frule bisiesto
start
    in >>   year
    if ==   (% (year, 400), 0)
```

```
            out << “es bisiesto”
    else
            if &&(==(%(year, 4), 0), ~= (%(year, 100), 0))
                out << “es bisiesto”
            else
                out << “no es bisiesto”
            end
    end
end
```

Bucle while-end:
El bucle while-end permite ejecutar instrucciones mientras se cumple una condición.

Código fuente 5. Ejemplo de bucle en F_RULE.

```
frule suma
start
        in >> n
        suma=0
        i=1
        while <=(i, n)
            suma= +(suma, i)
            i= +(i, 1)
        end
        out << “la suma de 1 hasta “, n, “ es: “, suma
end
```

Retorno de variables:
El retorno de variables se realiza con la instrucción result << más una lista de variables separadas por coma. Además de la variables out y error.

Código fuente 6. Ejemplo de uso de *result* en F_RULE.

```
frule retornos
start
      valor= 2.71828
      ok= true
      out << “Constante de Napier: ”, valor
      result << ok, valor
end
```

Código fuente 7. Ejemplo de salida result en formato JSON.

```
{result: [{“name”: “ok”, “value”: “true”, “type”:
          “bool”},
          {“name”: “valor”, “value”: “2.71828”,
          “type”:”real”}],
out:      [“Constante de Napier: 2.71828”],
          error: {“exitCode”: 0, “error”: 0, “message”:
          “”, “errorline”: 0} }
```

Comentarios
Desde su aparición hasta el fin de la línea es omitido en la interpretación o ejecución.

Código fuente 8. Ejemplo de comentarios en F_RULE.

```
# Comentario de línea completa
z = +(x, y) # Comentario de resto de línea
```

Sintaxis de F_RULE
La especificación de la sintáxis del lenguaje F_RULE expresada en notación Backus normal form (BNF) [12] es la siguiente:

<altern1>::= **if** < expr bool > <CRLF > < bloque > < CRLF > **end**
< altern2 > ::= **if** < expr bool > < CRLF > < bloque > **else** <bloque> <CRLF> **end**
<asig> ::= <identificador> = < expr > [<comentario>]
<bloque> ::= <linea> {< CRLF> <linea>}
<bucle> ::= **while** <expr bool> <CRLF> <bloque> <CRLF> **end**
<caracter de subrayado> ::= _
<coma> :: = ,
<comentario> ::= # {<caracter>}
<CRLF> : = ascii(0x0A) ascii(0x0D)
<digito> :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<entrada> :: = **in** >> <varaible> {, <variable>}
<expr> :: = <literal> | <identificador> | <expr1> | <expr2>
<expr1> : = <op1> <parentesis1> <expr bool> <parentesis2>
<expr2> :: = <op2> <parentesis1> <expr> <coma> <expr> <parentesis2>
<frule> :: = **frule** <identificador>
<identificador> :: = <letra>{<letra> | <digito>}
<letra> :: = <mayuscula> | <minuscula> | <caracter de subrayado>
<linea> :: = (<asig> | <altern1> | <altern2> | <bucle> | <out>)
[<comentario>]
<litbool> ::= **true** | **false**
<literal> ::= <litint> | <litreal> | <litbool> | <litstr>
<litint> ::= <signo><digito>{<digito>}
<litreal> ::= <signo><digito>{<digito>}.<digito >{<digito>}
<litstr> ::= **“**{<caracter>}**”**
<main> ::= <entrada> <CRLF> <bloque> <CRLF> <salida>
<mayuscula> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<minuscula> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<op1> ::= <oplogico1>
<op2> ::= <oplogico2> | <oparitmetico> | <oprelacional>

<oparitmetico> ::= + | * | / | - | % | ^
<oplogico1> ::= ~
<oplogico2> ::= && | ||
<oprelacional> ::= < | <= | > | >= | == | ~=
<out> ::= out << expr [, expr]
<parentesis1> ::= (
<parentesis2> ::=)
<prog> ::= <frule> <CRLF> <start> <CRLF> <main> **end**
<signo> ::= | + | -
<start> ::= **start**

Metodología de desarrollo del lenguaje F_RULE

A continuación se explica el desarrollo del lenguaje y de su intérprete.

i) Desarrollo de la idea, requisitos y restricciones: La idea original era evaluar diferentes tipos de condiciones incluyendo expresiones que contengan operadores aritméticos. Luego se estableció que el lenguaje tome los argumentos en formato JSON, y que la salida de variables fuera en el mismo formato ya que son utilizadas para realizar determinadas acciones posteriores a la ejecución.
ii) Selección de simbología y palabras reservadas: Se seleccionaron palabras reservadas y simbología de operadores que son comunes en la mayoría de los lenguajes de programación utilizados[12].
iii) Desarrollo de la GLC: Se desarrolló la GLC, considerando una instrucción por línea y notación prefija en la evaluación de expresiones.
iv) Implementación del intérprete: Luego de haber definido el lenguaje, se comenzó a crear el intérprete en lenguaje c#, sin incluir librerías adicionales. Para la tokenización se utilizó la clase Regex que se encuentra en el namespace System.Text.RegularExpressions, que es estándar en el framework .NET en su versión 4.6.1[13].
v) Finalmente el código se encapsuló en una clase llamada Frule, de tal forma que para ejecutar un programa, basta crear una instancia y llamar el método execute. Finalmente, para obtener la salida se debe llamar a la property (propiedad) result, tal y como se muestra en el siguiente código:

Código fuente 9. Extracto de integración de F_RULE en la plataforma, código en C#.

```
var frule = new Frule(<código fuente>,
argumentos);
frule.execute();
var salida = frule.result;
```

La salida result es una lista de objetos que contienen 3 atributos: name, type, value. Dichos atributos se pueden utilizar para realizar diversas operaciones de forma automática, una vez finalizada la ejecución.

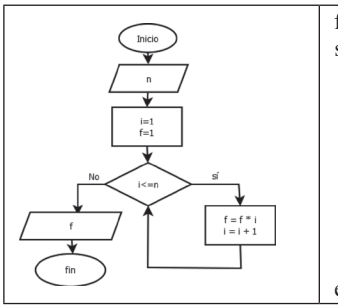
EJEMPLOS DE USO DE F_RULE EN PROBLEMAS TRADICIONALES DE PROGRAMACIÓN

Es posible utilizar F_RULE para resolver algunos problemas tradicionales de programación, utilizando su capacidad para interpretar condiciones e iteraciones. A continuación se mostrará la implementación de la solución de los siguientes problemas: cómputo de factorial de un número, serie de Fibonacci iterativa y divisores de un número.

Cómputo del factorial de un número

A continuación se presenta el algoritmo iterativo del cómputo de n!. El diagrama de flujo y el código se muestra a continuación.

Código fuente 10. Diagrama de flujo y código para computar n! en F_RULE.




```
frule factorial
start
      in >> n
      i=1
      f=1
      while <=(i, n)
            f= *(f, i)
            i= +(i, 1)
      end
      out << f
end
```

Como se puede apreciar las instrucciones del diagrama de flujo son convertibles de forma intuitiva al lenguaje F_RULE, lo que facilita su codificación.

Otros problemas tradicionales

A continuación se presenta el algoritmo iterativo que determina la serie de Fibonacci de n términos, y la

[12] "Reserved Key Words". Recopilación en varios lenguajes de programación. https://github.com/AnanthaRajuCprojects/Reserved-Key-Words-list-of-various-programming-languages
[13] Clase Regex (AssemblySystem.Text.RegularExpressions) Documentación en castellano. https://docs.microsoft.com/en-us/dotnet/api/system.text.regularexpressions.regex

lista de divisores de un número n. Ambos números son enteros positivos.

Código fuente 11. Otros programas solucionados con código F_RULE.

```
frule fibonacci
start
        in >> n
        a= 0
        b= 1
        i= 1
        while <=(i, n)
            c= +(a, b)
            a= b
            b= c
                out << a, " "
                i= + (i, 1)
        end
end
```

```
frule divisores
start
        in >> n
        i=1
        while <=(i,/(n,2))
            if ==(%(n, i), 0)
                out << a, " "
            end
            i= +(i, 1)
        end
        out << n
end
```

Aplicación de F_RULE en la evaluación de reglas de flujo

Un flujo representa la lógica en un proceso de negocio, o simplemente en un proceso automatizado. El cambio desde una etapa a otra depende de una función de transición que evalúa un conjunto de condiciones y determina la siguiente etapa y/o el responsable. A este conjunto de condiciones se les llama regla de transición.

Definición del caso de estudio

Para realizar los casos de estudio, se utilizará un proceso de pago. La información sobre los antecedentes ingresados referentes al pago no es relevante para efectos de la evaluación de las reglas de flujo y solo se considera el monto de forma genérica porque en una de las condiciones se utiliza.

Proceso de pago

La descripción del flujo asociado al proceso de pago es la siguiente:

i) El flujo inicia con el registro de un pago en la etapa "Registro". La única opción desde esta etapa es enviarlo a la etapa "VºBº de la Jefatura Directa".

ii) Desde la etapa "VºBº de la Jefatura Directa" existen dos opciones:
   a. Enviar a etapa "Aprobación del Jefe de Departamento", cada registro de pago depende de una clasificación del proveedor, por lo que el cálculo del jefe de departamento debe realizarlo a partir del parámetro respectivo del proveedor.
   b. Devolver a etapa "Registro" para corrección.

iii) Desde Aprobación del Jefe de Departamento, se pueden cambios a 3 etapas:
   a. Aprobar el pago y enviar a la etapa "Realizar Pago", si es que el monto no supera las 10 UF.
   b. Si el monto es mayor que 10UF la aprobación la deberá realizar el rol "Gerente General" en la etapa "VºBº Gte General".
   c. También existe la posibilidad de que se devuelva a la etapa anterior, para rectificar montos o recabar más antecedentes.

iv) Desde la etapa "VºBº Gte General" también hay 3 alternativas:
   a. Aprobar el pago y enviar a la etapa "Realizar Pago", para que el rol "Finanzas" lo gestione.
   b. Devolver a la etapa "Aprobación del Jefe de Departamento".
   c. Rechazar pago, y archivar.

v) Finalmente se establece que en la etapa "Realizar Pago", se recibe la solicitud (ya aprobada), y simplemente se gestiona y se archiva.

Representación del flujo del proceso de pago

A continuación se muestra el flujo del proceso de pago, primero se muestra la información referente a los datos, y luego la representación gráfica en la figura 1.

Las siguientes tablas representan las etapas y roles utilizados en los casos de prueba.

Casos de prueba

A continuación se presentan los casos de prueba, que corresponden a problemas reales de cambios de etapa. El cambio de etapa en el sistema se realiza de forma automática, luego de evaluar el código de F_RULE que se presenta a continuación.

Código fuente 12. Extracto de código para realizar automáticamente la operación enviar (send) en la plataforma. Código en C#.

```
var frule = new Frule (codigofuente, parámetros);
try {
      frule.run();
      if (!frule.exitCode==0) {
          var result = f.result.ToDictionary (
                        p = >p. name,
                        p = > p. value);
      var m = Dynamics.currentMessage;
      try {
```

Tabla 2. Tablas Role (izq.) y Step (der)

role_id	rolename	step_id	stepname
101	Originador	1001	Registro
201	Jefe Directo	1002	VºBº de la Jefatura Directa
301	Jefe Departamento 1	1003	Aprobación del Jefe de Depto
302	Jefe Departamento 2	1004	VºBº del Gerente General
303	Jefe Departamento 3	1005	Realizar Pago
401	Gerente General		
501	Finanzas		

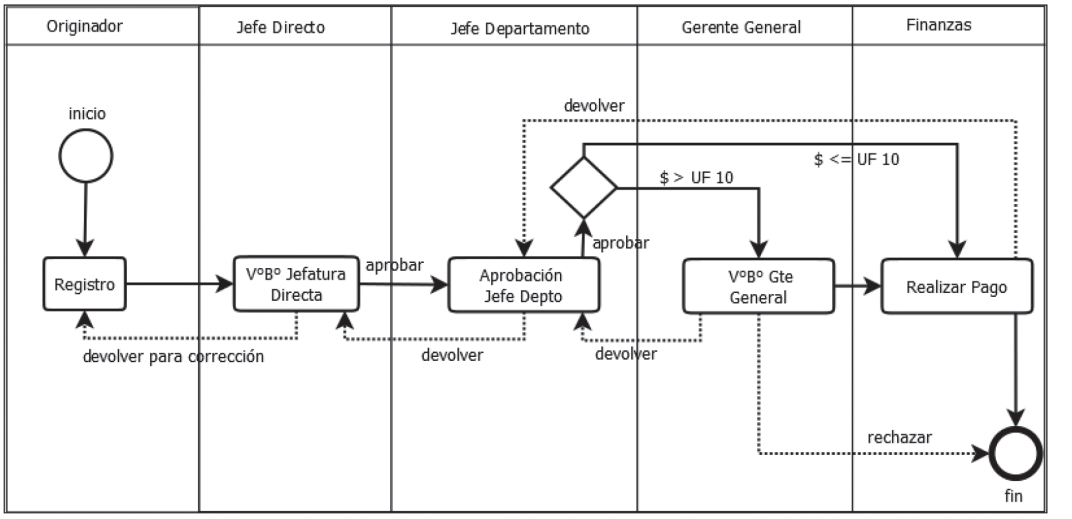


Figura 1. Flujo del proceso de pago.

```
        m.role_id = int.Parse (result [“nextRole”]);
        m.step_id = int.Parse (result [“nextStep”]);
        m.getBitacora(). AddEvent(new Bitacora
Event(...)));
        m.save();
      }
      catch (Exception) {
        // Procesar exception
        }
      }
      else {                          // procesar error
      }
}
catch (Exception) { }
```

En los 3 primeros casos de prueba, el código F_RULE está relacionado con la acción (o transición). O sea, desde cada transición existirá un código que será ejecutado, pero sólo uno de los códigos indicará la siguiente etapa. Para asegurar esto, el código de F_RULE debe ser determinista.

En el caso 4 el código F_RULE está relacionado a la etapa, es decir, es capaz de determinar la etapa siguiente desde la etapa actual, usando un solo código F_RULE.

En el último caso, experimental, se utiliza un código F_RULE por todo el flujo por lo que es necesario pasar como argumento la etapa actual y el responsable actual de la etapa.

Cada caso de prueba plantea una problemática desde lo más simple a complejo, y se mostrará el código necesario para que se realice el cambio de etapa.

Caso de prueba 1

El caso consiste en crear el código F_RULE necesario para avanzar a la siguiente etapa. No hay condiciones de por medio, sólo se asigna la etapa siguiente y el responsable.

– Etapa actual: Registro
– Etapa siguiente: VºBº de la Jefatura Directa
– Elementos a evaluar: no tiene

Código fuente 13. Solución al caso de prueba 1, el código F_RULE está asociado a la transición [Registro] → [VºBº de la Jefatura Directa].

```
frule enviarVBJefaturaDirecta
start
      nextRole = 201
      nextStep = 1002
      notificationId = 1
      result << nextRole, nextStep
end
```

Caso de prueba 2.
La diferencia en este caso es que como hay 3 departamentos, si se envía a aprobación, se deberá elegir el departamento entre el 1 y el 3, y por supuesto se asignará como responsable al jefe del departamento seleccionado.
– Etapa actual: VºBº de la Jefatura Directa
– Etapa siguiente: Aprobación Jefe Depto.
– Elementos a evaluar: accion, numdepto

Código fuente 14. Solución al caso de prueba 2. El código F_RULE está asociado a la transición [Vº Bº de la Jefatura Directa] → [Aprobación Jefe Depto].

```
frule enviarAJefeDepartamento
start
      in >> numdepto, accion
      nextStep = 0 #no viaja
      if && (> = (numdepto, 1), < = (numdepto, 3))
            nextRole = +(300, numdepto)
      end
      if ==(accion, “aprobar”)
            nextStep = 1003
      end
      result << nextRole, nextStep
end
```

Caso de prueba 3
Al igual que en el caso de prueba 2, la condición tiene que ver sólo con la acción porque el código F_RULE está asociado a la transición que permite devolverse a la etapa anterior.
– Etapa actual: VºBº de la Jefatura Directa
– Etapa siguiente: Registro
– Elementos a evaluar: accion

Código fuente 15. Solución al caso de prueba 3. El código F_RULE está asociado a la transición [Aprobación Jefe Depto] → [Registro].

```
frule devolverARegistro
start
      in >> accion
      nextStep = 0 #no viaja
      nextRole = 0
      if == (accion, “rectificar”)
            # etapa anterior
            nextStep = 1001
            nextRole = 101
      end
      result << nextRole, nextStep
end
```

Caso de prueba 4
En este caso, la siguiente etapa puede ser una de las 3: (1) Enviar a Aprobación VºBº Jefatura Directa (2) VºBº Gerente Gral o (3) Pago o Devolver. Por lo tanto el código F_RULE está relacionado a la etapa.
– Etapa actual: Aprobación Jefe Depto.
– Etapa siguiente: VºBº Gte General | Realizar Pago | VºBº Jefatura Directa
– Elementos a evaluar: accion, monto

Código fuente 16. Solución al caso de prueba 4. El código F_RULE está asociado a la etapa [Aprobación Jefe Depto].

```
frule enviarAVBGteGralOPagoODevolver
start
      in >> accion, monto, valor UF, rjefedepto
      nextStep = 0 # no viaja
      if == (accion, “aprobar”)
            if <= (monto, *(valor UF, 10))
                  nextStep = 1005 # pago
                  nextRole = 501 #finanzas
            else
                  nextStep = 1004 #vb gte gral
                  nextRole = 401 #gte gral
            end
            else
            if == (accion, “devolver”)
                  nextStep = 1003
                  nextRole = rjefedepto
            else
                  # acción no reconocida, no viaja
                  nextRole = 0
            end
      end
      result << nextRole, nextStep
end
```

Caso de prueba 5
El último caso de prueba permite incorporar la lógica completa del flujo en un solo código F_RULE, por eso se dice que este código está asociado al flujo.

– Etapa actual: Cualquier etapa
– Etapa siguiente: Cualquier etapa
– Elementos a evaluar: etapaActual, accion, numdepto, valoruf

Código fuente 17. Solución al caso de prueba 5. El código F_RULE está asociado al flujo completo.

```
frule flujoproceso
start
      in >> etapaActual, accion, monto, valorUF, rjefedepto
      deptoroles = [301, 302, 303]
      nextStep = 0 #no viaja
          status = "activo" # valor por defecto
      # etapa actual: registro
          if ==(etapaActual, 1001)
              nextRole = 201 # jefe directo
              nextStep = 1002 # VB de la jefatura directa
      end
      # etapa actual: VB de la jefatura directa
      if == (etapa Actual, 1002)
          if == (accion, "aprobar")
              if && (>=(numdepto, 1), <= (numdepto, 3))
                    # Jefe departamento 1, 2 o 3
              nextRole = +(300, numdepto)
              end
              nextStep = 1003 # Aprob. jefe depto
          end
          if ==(accion, "devolver")
              nextRole = 101 # originador
              nextStep = 1001 # registro
          end
      end
      # etapa actual: # Aprob. jefe depto
          if == (etapaActual, 1003)
              if == (accion, "aprobar")
          if <= (monto, *(valoruf, 10))
              nextStep = 1005 #pago
              nextRole = 501 #finanzas
          else
              nextStep = 1004 #vb gte gral
              nextRole = 401 #gte gral
      end
if == (accion, "devolver")
      nextRole = 201 # jefe directo
          nextStep = 1002 # VB de la jefatura directa
      end
      end
      if ==(etapaActual, 1004)
      if ==(accion, "aprobar")
          nextStep = 1005 #pago
          nextRole = 501 #finanzas
          end
          if ==(accion, "devolver")
              nextStep = 1003 # Aprob. jefe depto
              nextRole = rjefedepto
          end
          if ==(accion, "rechazar")
              status = "rechazado"
          end
      end
      if == (etapa Actual, 1005)
          if == (accion, "pagar")
              status = "archivado"
          end
          if == (accion, "devolver")
              nextStep = 1004 #vb gte gral
              nextRole = 401 #gte gral
          end
      end
      result << nextRole, nextStep, status
end
```

Como se puede apreciar, en este último caso de prueba es posible incorporar la lógica completa del flujo en un solo código F_RULE, de esta forma se hace más simple su mantenimiento.

CONCLUSIONES

En este trabajo se ha presentado un nuevo lenguaje de programación denominado F_RULE y la implementación de un intérprete escrito en C# que se encuentra integrado en la plataforma Dynamics.

F_RULE es un lenguaje de alto nivel, con un sistema de tipos dinámico, de propósito general y automático en el sentido de su ejecución.

Aunque su origen se debe a la necesidad de automatizar la evaluación de reglas de flujo en el contexto de la plataforma Dynamics, el intérprete presentado en este trabajo permite su ejecución autónoma, como un software independiente, o que puede ser usado para interpretar código formando parte de otra aplicación.

A través del documento se pudo apreciar los elementos que componen el lenguaje, su estructura y una serie de ejemplos de aplicación en problemas tradicionales de programación.

Respecto de las restricciones impuestas por la plataforma Dynamics, el lenguaje F_RULE y su intérprete las cumplen cabalmente:

i) No es necesario recompilar la plataforma cada vez que se realiza una modificación en las reglas.
ii) Permite evaluación de condiciones con valores de tipo numérico, texto y lógico.
iii) Su entrada y salida es automática en formato JSON.
iv) La implementación de su intérprete es robusta porque en caso de errores, es posible modificarla,

además si se produce algún error en el código fuente, el lenguaje entrega información del error y su ubicación. Esto es detectable y además corre de forma segura en bloques de código protegidos por cláusulas try-catch (que capturan excepciones), y la plataforma no corre peligro de desestabilizarse o caerse.

Respecto de las otras aproximaciones analizadas antes de desarrollar el lenguaje e implementar el intérprete se puede apreciar que en comparación con todos los casos, es preferible la solución propuesta en este trabajo:

i) En el primer caso respecto de la evaluación de las reglas en el mismo sistema, utilización de F_RULE resulta más apropiado porque no se requiere modificar código fuente.
ii) Respecto de los motores de reglas de negocio, implicaba costo adicional, integración con una herramienta que acopla la plataforma, ya que se establece una dependencia con este producto. Además se analizó la factibilidad de crear un propio motor de reglas, pero su diseño e implementación se estimó en un tiempo mayor que desarrollar F_RULE.
iii) Para el caso del motor de inferencia, la implementación implicaba la instalación de una máquina virtual. Aunque esto no fuera complejo, la codificación de las reglas y el mantenimiento de la base de reglas implica un esfuerzo adicional. Para el caso de la codificación de reglas en el lenguaje F_RULE es simple en el sentido de que su código es muy intuitivo respecto de la forma en la que se acostumbra describir una regla bajo el paradigma ECA.
iv) A diferencia del uso de librerías compiladas de terceros, el lenguaje F_RULE es transparente en su ejecución, y en caso de falla, simplemente se puede corregir.
v) En comparación con la incrustación de un lenguaje en la plataforma, para su utilización como un único proceso (o servicio), el código fuente del lenguaje F_RULE es reducido, pues incluyendo todas las clases, bordea las 1000 líneas de código, y prácticamente no tiene dependencias, pues en su desarrollo no se incluyen librerías externas a las provee el framework de .NET.
vi) Finalmente, a diferencia del uso otros lenguajes que permiten ser llamados desde la plataforma, el lenguaje F_RULE no es una caja negra, pues su funcionamiento interno es conocido, es trazable y las salidas son verificables respecto del código fuente que las generó.

A través de los casos se prueba se pudo constatar que el lenguaje es capaz de soportar distintas configuraciones de reglas de flujo, y que es posible implementarlo a nivel de acción, etapa, o flujo completo. Esto añade flexibilidad, que permite realizar diferentes implementaciones de acuerdo a las necesidades propias.

En el mismo contexto, F_RULE puede ser utilizado también en otros aspectos que utilicen el paradigma ECA, como por ejemplo: reglas de inicialización, reglas del proceso, reglas de término, notificaciones con condiciones, etc.

TRABAJOS FUTUROS

Los futuros desarrollos para el mejoramiento del lenguaje de programación F_RULE son los siguientes:

1. Incorporar arrays y tipos estructurados.
2. Incorporar un bucle for-end.
3. Crear biblioteca de funciones.
4. Control de errores.
5. Crear un portal web de prueba del lenguaje.

AGRADECIMIENTOS

El presente artículo se llevó a cabo gracias al soporte de la empresa E-Process (www.e-process.cl), en donde se desarrolló la plataforma Dynamics, sobre la que está integrado el intérprete de F_RULE, en la que se realizaron las pruebas y casos de estudio.

REFERENCIAS

[1] N. Wirth. “Algorithms + data structures = Programs”. Prentice Hall PTR, USA. 1978.
[2] D.C. Kozen. “Automata and computability”. Springer-Verlag, Berlin, Heidelberg. First Edition. 1997.
[3] B. Hitpass. “Business process Management (BPM) Fundamentos y Conceptos de Implementación”. Fourth Edition. 2017.
[4] F. Bry, M. Eckert, P. Patranjan and I. Romanenko. “Realizing business processes with ECA Rules: Benefits, Challenges,

limits". Lecture Notes in Computer, pp. 48-62. 2006. DOI:10.1007/11853107_4.

[5] E. Kanana, V. Vassilev and K. Ouazzane. "Two-level architecture for rule-based business process management". 2018.

[6] N. Grzegorz, K. Krzysztof and K. Kaczor. "Proposal of an Inference Engine Architecture for Business Rules and Processes", pp. 453-464. 2013. DOI: 10.1007/978-3-642-38610-7_42.

[7] jBPM: un juego de herramientas para crear aplicaciones comerciales que ayuden a automatizar los procesos y las decisiones comerciales. https://www.jbpm.org/

[8] NET Compiler Platform. Roslyn. https://github.com/dotnet/roslyn

[9] The MIT licence. https://opensource.org/licenses/MIT

[10] Embedding Python in another application. Python 3.8 online official documentation. https://docs.python.org/3/extending/embedding.html

[11] Especificación JSON. http://www.json.org/json-es.html

[12] D. McCracken and E. Reilly. "Backus-Naur form (BNF)". Encyclopedia of Computer Science. John Wiley and Sons Ltd, pp. 129-131. 2003.