UNIVERSIDAD POLITÉCNICA DE VALENCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

PROYECTO FINAL DE CARRERA

INGENIERO EN INFORMÁTICA

DAFRULE: UN MODELO DE REGLAS ENRIQUECIDO MEDIANTE FLUJOS DE DATOS PARA LA DEFINICIÓN VISUAL DE COMPORTAMIENTO EN ENTORNOS REACTIVOS

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria Informàtica

PROYECTO REALIZADO POR:
Patricia Pons Tomás

DIRECTORES:
Dr. Francisco Javier Jaén Martínez
Alejandro Catalá Bolós

JUNIO 2012

Agradecimientos

A mis padres, por haber sido siempre tan comprensivos, pacientes y atentos, por su apoyo incondicional, y por muchos otros motivos que jamás podría expresar con palabras.

A mi familia, por todo el cariño que me dan cada día, y porque pase lo que pase, siempre están ahí.

A mis compañeros durante estos 5 años de carrera, por todos los buenos momentos que hemos compartido.

A Javier y Alejandro, por todas las oportunidades que me han brindado, por todo su apoyo, ayuda y dedicación.

Índice de Capítulos

1. Introducción *1*

1.1. **Motivación** 1

1.2. **Juegos reactivos** 3

1.3. **Inteligencia ambiental** 7

1.4. **Presentación del proyecto** 11

2. DaFRule: Reglas enriquecidas mediante flujos de datos *13*

2.1. **Entorno como Ecosistema** 13

2.2. **Reglas** 16

2.3. **Flujos de Datos** 20

2.4. **Evaluación sobre la comprensión de los flujos de datos** 25

3. Implementación del editor WIMP *27*

3.1. **Área de selección de elementos de la regla** 28

3.2. **Selección de vista** 29

3.3. **Área de edición** 29

3.4. **Barra de herramientas** 31

3.5. **Área de comprobación de errores** 32

3.5.1. **Algoritmo DFS adaptado** 34

3.6. **Consola de resultados** 36

3.7. **Ejemplo completo de edición de una regla** 37

4. Motor de procesamiento de reglas *42*

4.1. **Procesador de eventos** 42

4.2. **Implementación del procesador de eventos** 43

4.3. **Cálculo del valor objetivo de un proceso de datos** 45

4.4. **Depurador WIMP** 49

5. Validación del procesador de eventos *51*

5.1. **Evaluación del proceso de *matching*** 51

5.1.1. **Tiempo medio de establecimiento de un *matching* en base al número de entidades en el ecosistema** 51

5.1.2. **Tiempo medio del proceso de *matching*** 53

5.2. **Evaluación del mecanismo de ejecución de reglas** 57

5.2.1. **Optimización sobre el filtrado de la población destino** 57

5.2.2. **Tiempo de cálculo de los procesos de datos** 60

5.3. **Comparativa: procesamiento secuencial vs. procesamiento paralelo** 61

5.3.1. **Paralelización del proceso de *matching*** 62

5.3.2. **Paralelización del proceso de evaluación** 64

6. Conclusiones *68*

6.1. **Trabajo Futuro** 69

7. Bibliografía 71
8. Anexo A: Gramática ANTLR para la definición textual de reglas 73

Índice de Figuras

Figura 1. Ejemplo de edición de comportamiento en Scratch. 4
Figura 2. Ejemplo de edición de comportamiento en Alice. 5
Figura 3. Interfaz de Nexel para la personalización del comportamiento de un dispositivo 8
Figura 4. Diagrama de clases UML para dar soporte a entornos basados en entidades. 15
Figura 5. Estructura de una regla. 17
Figura 6. Ejemplos textuales de reglas soportadas por el editor. 18
Figura 7. Diagrama de clases UML para dar soporte a las reglas. 20
Figura 8. Ejemplos visuales de flujos de datos simples soportados por el editor. 23
Figura 9. Diagrama de clases UML para soportar transformaciones mediante flujos de datos. 24
Figura 10. Interfaz del editor de reglas 27
Figura 11. Exploración de la colección de entidades para seleccionar la población fuente de la regla. 28
Figura 12. Desplegable para la selección de la vista. 29
Figura 13. Ejemplo de proceso de datos correctamente editado. 30
Figura 14. Selección de un procesador de datos. 32
Figura 15. Selección de valor a asignar a un nodo proveedor de constantes 32
Figura 16. Selección de una propiedad de cualquier entidad del ecosistema. 32
Figura 17. Ejemplo de proceso de datos con flujos de datos incorrectos, y procesadores de datos todavía por completar. 33
Figura 18. Grafo dirigido con una arista hacia atrás. 34
Figura 19. Pseudocódigo del algoritmo DFS adaptado para la detección de ciclos en un proceso de datos. 35
Figura 20. Creación de una nueva regla. 37
Figura 21. Interacción para añadir al área de edición un valor constante. 38
Figura 22. Interacción para añadir al área de edición un operador. 38
Figura 23. Edición de la precondición. 39
Figura 24. Edición de la condición de filtrado. 39
Figura 25. Interacción para añadir a un proceso de datos una propiedad de cualquier entidad del ecosistema. 40
Figura 26. Edición del proceso de datos asociado al parámetro de una acción. 41
Figura 27. Edición correcta del proceso de datos asociado al parámetro de una acción. 41
Figura 28. Diagrama de clases UML del procesador de eventos. 42
Figura 29. Pseudocódigo para el procesamiento de ocurrencias de evento. 44
Figura 30. Pseudocódigo para el *matching* de ocurrencias de evento con reglas. 44
Figura 31. Ejemplo del uso de tablas *hash* para almacenar valores de propiedades y atributos. 45
Figura 32. Pseudocódigo del algoritmo para el cálculo del valor a asignar al nodo objetivo del proceso de datos. 47
Figura 33. Ejemplo del cálculo del valor objetivo de un proceso de datos. 48
Figura 34. Visualización de las propiedades de las entidades de la simulación. 50
Figura 35. Traza de ejecución tras haber lanzado dos ocurrencias de evento. 50

Figura 36. Tiempo medio de establecimiento de un *matching* según el número total de entidades en el ecosistema. 52
Figura 37. Tiempo medio de establecimiento de un *matching* según el número de tipos de entidad. 52
Figura 38. Ejemplo del uso de tablas *hash* con doble indexación para detectar las reglas aplicables. 54
Figura 39. Tiempo medio del proceso de *matching* según el número de tipos de evento (sin optimización). 55
Figura 40. Tiempo medio del proceso de *matching* según el número de tipos de evento (con optimización). 56
Figura 41. Tiempo medio del proceso de *matching* según el número de tipos de evento (con optimización, rangos alterados). 57
Figura 42. Tiempo medio de evaluación de la condición de filtrado (sin aplicar optimización) 59
Figura 43. Tiempo medio de evaluación de la condición de filtrado (aplicando optimización) 59
Figura 44. Tiempo medio de evaluación de un proceso de datos en base al número de operadores. 60
Figura 45. Detalle de la Figura 44. 61
Figura 46. Aceleración en función del número de hilos empleados, para distinto número de reglas en el ecosistema. 63
Figura 47. Aceleración en función del número de hilos empleados, para distinto número de niveles del proceso de datos. 65
Figura 48. Tiempo medio del proceso de evaluación en base al número de hilos empleados, para distinto número de niveles del proceso de datos. 66
Figura 49. Aceleración según el número de hilos disponibles, para distinto número de entidades por tipo. 67

Resumen

La definición de reglas de comportamiento para entornos reactivos es una tarea compleja para aquellos usuarios no expertos o sin conocimientos de programación previos. Las herramientas disponibles hasta la fecha para facilitar esta tarea o bien son poco flexibles y expresivas en cuanto a las reglas que permiten definir, o bien la expresividad es adecuada pero la herramienta es compleja de entender por usuarios no programadores. Dada la necesidad de proveer mecanismos que faciliten la tarea de edición de reglas de comportamiento, se ha desarrollado un lenguaje de reglas expresivo y genérico aplicable a diversos ámbitos. Se ha desarrollado un editor que hace uso de dicho lenguaje, así como un motor de procesamiento de reglas que permite emplear las reglas definidas con este lenguaje en la simulación de entornos reactivos. Se han llevado a cabo experimentos para validar la corrección del lenguaje propuesto, y la escalabilidad del motor de procesamiento de reglas.

Palabras Clave

Reglas, ECA, Eventos, Flujos de datos, Lenguaje visual, Entidades virtuales, Comportamiento, Personalización.

1. Introducción

En este capítulo se introduce el presente proyecto final de carrera, DaFRule, centrado en proporcionar un modelo de reglas enriquecidas con expresiones de flujos de datos e implementar un prototipo que permita su edición sin la necesidad de introducir código mediante teclado. A continuación se exponen los problemas que han motivado el desarrollo de este proyecto y posibles puntos a solucionar de dichos problemas. Seguidamente se enfocan estos problemas en dos áreas de aplicación que podrían beneficiarse de estas mejoras. Para cada área de aplicación se reporta la situación actual del campo, y finalmente se expone la solución que aborda este proyecto, junto a los objetivos que pretende cubrir.

1.1. Motivación

Las nuevas generaciones están muy familiarizadas con la tecnología y los avances en computación. La mayoría de los jóvenes de hoy en día son capaces de manejar multitud de aplicaciones de ordenador con fluidez, especialmente juegos educativos interactivos o videojuegos, y más aún, comprender el comportamiento de los mismos. Esta comprensión provoca el desarrollo del pensamiento computacional, habilidad que permite resolver problemas lógicos de diferentes disciplinas mediante la abstracción de los mismos en términos computacionales. El pensamiento computacional es aplicable a prácticamente cualquier campo, y gracias a la abstracción de conceptos en la que se fundamenta, es posible establecer relaciones entre diferentes ámbitos o áreas. Es precisamente por su implicación en diferentes áreas de forma generalista por lo que se considera el pensamiento computacional como una habilidad de suma importancia a ser explotada para el desarrollo humano [1].

Sin embargo, a pesar de estas habilidades y de la amplia literatura disponible, programar como tal todavía resulta complicado para principiantes o no-programadores. Se han llevado a cabo numerosos estudios para determinar los motivos que provocan esta dificultad [2], y normalmente radican en los mismos lenguajes de programación, ya que éstos no suelen seguir la misma lógica que sigue el razonamiento humano. La solución que un programador piensa para dar respuesta a un determinado problema no tiene una correspondencia directa con la solución que debe ser expresada mediante un lenguaje de programación. Hay que llevar a cabo un proceso de traducción desde el conocimiento del programador hacia la sintaxis y semántica del lenguaje de programación correspondiente, y es durante este proceso en el cual se producen los errores o se encuentran las dificultades. Cuanto mayor sea la diferencia entre nuestra forma de expresar conocimiento y la forma en la que los lenguajes de programación permiten expresar este conocimiento, mayor dificultad habrá para aprender a programar. Es necesario, por tanto, buscar lenguajes de programación que permitan reducir estas diferencias.

Los lenguajes de programación deben aproximarse al pensamiento humano lo máximo posible. Para ello se ha estudiado la manera en la que los no-programadores expresan sus soluciones a problemas computacionales en lenguaje natural [2][3]. A raíz de estos estudios se ha hecho patente que uno de los mecanismos mejor comprendidos por gente joven o no-programadores es el

comportamiento reactivo basado en reglas. Este es un punto favorable, ya que la mayoría de aplicaciones multimedia, así como los videojuegos, tienen un comportamiento reactivo basado en reglas Evento-Condición-Acción (ECA). Sin embargo, ser capaces de comprender este comportamiento no es equivalente a poder plasmarlo de manera textual mediante lenguajes de programación. La especificación textual de estas reglas requiere aprender la sintaxis y semántica del lenguaje para saber cómo expresar cada comportamiento y los mecanismos de los que disponemos para ello. Además, se ha determinado que uno de los errores más comunes en la especificación de reglas de comportamiento es el de omisión: los usuarios no expertos encuentran dificultades a la hora de reunir todos los elementos que intervienen en las reglas, y de esquematizar estas reglas de manera lógica.

En el caso de pretender que las reglas de comportamiento sean editadas por usuarios convencionales, y no predefinidas por el desarrollador de software, es obvio que la manera más adecuada no debería ser la especificación textual de dichas reglas, ya que ello conllevaría para el usuario un sobreesfuerzo al tener que aprender a un lenguaje específico para cada contexto de edición. Por otra parte, sería provechoso para aquellos que se inicien en la programación, disponer de un paso intermedio para comenzar a entender cómo especificar comportamiento y desarrollar el pensamiento computacional, sin tener que verse ligados inicialmente a ningún lenguaje en particular.

Esto nos lleva a buscar nuevas formas alternativas a la especificación textual del comportamiento. A raíz de las dificultades observadas para iniciarse en la programación, el campo de los lenguajes visuales está principalmente centrado en facilitar esta tarea. Este tipo de lenguajes pretenden abstraer la sintaxis frecuentemente tediosa de los lenguajes textuales, empleando metáforas visuales cercanas al usuario, que hagan de los conceptos de programación algo más familiar y cercano. Los mecanismos visuales que estos lenguajes proveen juegan un papel importante, facilitando la comprensión y reduciendo la diferencia conceptual entre el razonamiento lógico y su especificación. Sin embargo, aunque estos lenguajes suelen ser adecuados para el contexto en el que se definen, muchas veces la expresividad que proveen queda limitada, o bien por el propio dominio del lenguaje, o bien si se trata de un lenguaje genérico, por la simplicidad de las construcciones que permite. Una tarea no trivial a llevar a cabo sería encontrar la relación correcta entre la simplicidad de un lenguaje visual y la expresividad que éste permita.

Por tanto, dado que la especificación de comportamiento reactivo mediante reglas está presente en numerosas áreas de la computación, y puesto que es un mecanismo fácilmente comprensible por usuarios no expertos, se requiere la construcción de un editor de reglas apropiado y fácil de usar, lo cual supone una tarea realmente desafiante. Este editor debe evitar la codificación textual de las reglas, ya que es la tarea que mayor dificultad supone para alguien no experto, y el principal foco de errores para un desarrollador. De este modo, la especificación de comportamiento se debe llevar a cabo empleando mecanismos visuales más intuitivos y comprensibles. Por otro lado, la expresividad del lenguaje visual basado en reglas propuesto debe permitir cálculos no triviales. Usualmente, conforme aumenta la expresividad permitida por un lenguaje visual, mayor es la dificultad que experimentan los usuarios al lidiar con dicho lenguaje. Pero los

usuarios que diseñen reglas para un entorno reactivo deben ser capaces de crear expresiones en ocasiones complejas para definir valores de dicho entorno, por lo que los mecanismos visuales ofrecidos por el editor deberán ser lo más intuitivos posible para facilitar esta tarea, permitiendo a su vez construcciones complejas. Este lenguaje visual podría ser aplicado a muchos de los entornos en los que se requiriera definir comportamiento reactivo de las entidades que intervienen, proveyendo, pese a la utilización de metáforas visuales, una alta expresividad, de la que habitualmente carecen los entornos basados en técnicas visuales en comparación a otros basados en lenguajes textuales.

A continuación se presentan dos áreas dispares que podrían beneficiarse de un editor de reglas de estas características: video juegos e inteligencia ambiental. En la primera, el editor se emplearía para la especificación de comportamiento de las entidades en los entornos virtuales. En la segunda, resulta de especial interés el uso de reglas para la personalización de la inteligencia en el ambiente con el propósito de suplir con el conocimiento específico de los usuarios las carencias de los sistemas inteligentes autónomos.

1.2. Juegos reactivos

Una de las áreas de interés donde sería beneficioso poder definir reglas de comportamiento es el ámbito de los juegos. Ya hemos comentado la capacidad de los jóvenes de comprender el comportamiento reactivo de los juegos y aplicaciones, además de ser un área muy motivadora para fomentar el aprendizaje. Pero incluso más motivador que aprender jugando es el hecho de aprender creando. De momento, la mayoría de juegos están pensados para seguir una secuencia preestablecida de acciones y hechos, pero dar un paso más allá supondría que jóvenes sin ningún conocimiento de programación fueran capaces de crear sus propios juegos e historias y posteriormente simularlos y jugar con ellos.

Para ello, se deberían proveer mecanismos para la creación y animación de las entidades del ecosistema, o bien proveer entidades genéricas que se puedan personalizar. Sin embargo, para la progresión de la historia y la interacción entre personajes, es necesario definir el comportamiento de las entidades en tiempo de edición, y ello sin tener que pasar por la programación como tal. Es bastante común que el comportamiento de entidades en juegos 2D se base en reglas reactivas, por lo que una buena manera de definir este comportamiento sería disponer de un editor que permitiera especificar las reglas de comportamiento de manera visual. La representación visual de las reglas permitiría al usuario razonar sobre las mismas, y es una manera de aprender conceptos de programación de un modo diferente, con un enfoque más motivador e interactivo. Estas reglas de comportamiento definidas durante la edición serían como una plantilla que, durante la simulación del juego, permitiría desencadenar las acciones e interacciones que dan vida a la historia.

Actualmente podemos encontrar diversos entornos basados en juegos que emplean cierto tipo de lenguaje de programación para expresar el comportamiento de sus personajes o entidades virtuales, o incluso la evolución de un mundo virtual al completo. En estos entornos varían los objetivos, tecnologías o paradigmas de programación empleados, y más variable es todavía la expresividad permitida por

cada uno de ellos. No obstante, todos están orientados a no-programadores o adolescentes que inician sus estudios en informática.

Entre estos entornos se encuentra Scratch [4]. Enfocado a la producción multimedia e inspirado en el lenguaje visual LogoBlocks [6], Scratch pretende hacer la programación más atractiva permitiendo a los jóvenes contar historias de manera sencilla con una interfaz tradicional de ventanas. Las complejas construcciones sintácticas de los lenguajes de programación se simplifican empleando una especie de plantillas con forma de bloques. La propia forma de estos bloques orienta al usuario sobre las posibles relaciones que se pueden establecer entre ellos. Sin embargo, a pesar de que es uno de los entornos más visuales, su expresividad está más limitada que aquellos entornos que no son tan visuales o que recaen en lenguajes puramente textuales, ya que las combinaciones de elementos permitidas, aunque potentes para el dominio, son sencillas para aquellos que busquen algo más generalista. Un ejemplo de definición de reglas de comportamiento con Scratch se puede ver en la Figura 1: a la derecha se muestra el escenario actual, a la izquierda se encuentran los bloques constructivos agrupados por categorías, y en el panel central aparece la zona de edición.

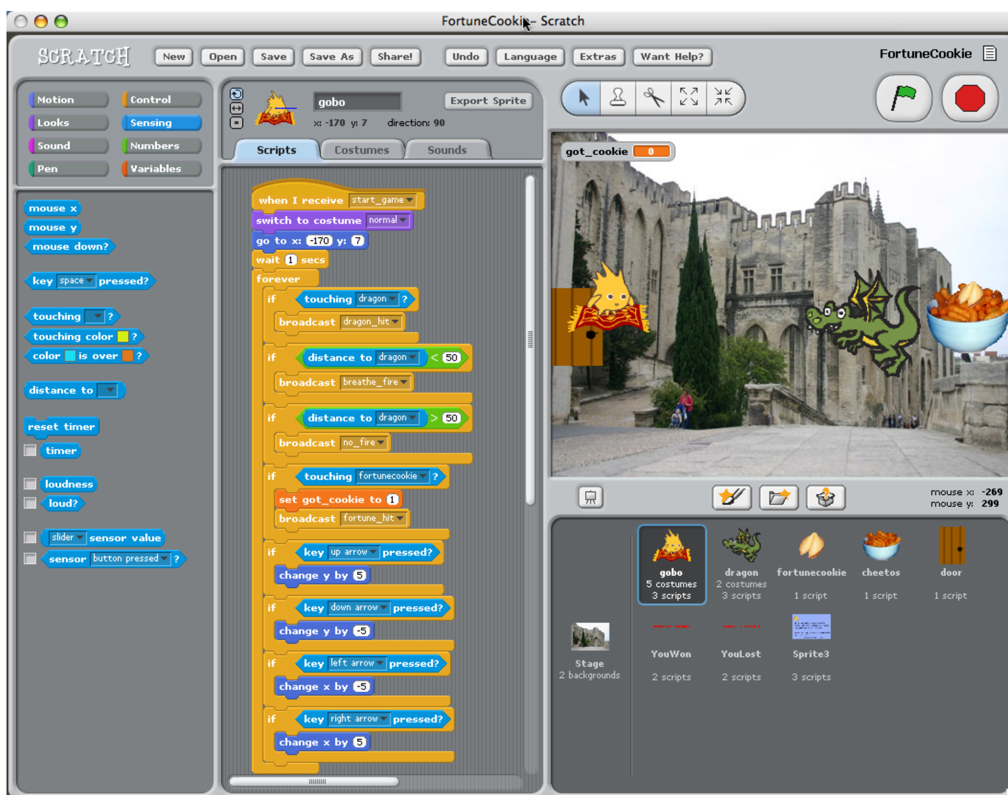


Figura 1. Ejemplo de edición de comportamiento en Scratch.

Otro de los entornos más destacados es Alice [7], que emplea un lenguaje altamente expresivo mediante el paradigma de programación estructurada por selección. Alice también se enfoca a la producción de videos, animaciones y juegos esta vez en 3D mediante una interfaz *drag-and-drop* que puede observarse en la Figura 2. Los usuarios disponen de un conjunto de objetos en 3D, y de una serie de acciones predefinidas (por ejemplo, mover, rotar, etc.) que pueden aplicarse a dichos objetos. La especificación de comportamiento se realiza seleccionando, arrastrando y posicionando elementos en los huecos correspondientes, mientras que se puede ver el resultado de cada acción a medida que se va construyendo el

programa. El objetivo fundamental de Alice es ofrecer a los jóvenes una primera experiencia programando más satisfactoria y amena. Se ha demostrado que con este método los usuarios sin conocimientos de programación son capaces de programar porciones de código bastante extensas.

En el ámbito de la simulación, uno de los trabajos relevantes es AgentSheets [8], que emplea un lenguaje basado en reglas, AgenTalk, para especificar el comportamiento de agentes simulables. AgentSheets permite a los usuarios definir de manera gráfica el comportamiento de agentes, y esto se traduce internamente a sentencias IF-THEN que contienen condiciones y acciones especificando este comportamiento. AgentSheets resultó ser fácil de utilizar, ya que incluso niños eran capaces de crear animaciones simples de manera rápida y sin ningún conocimiento previo de programación. En cambio, las reglas definidas de manera gráfica no son tan potentes como las que se pueden especificar de manera textual con AgenTalk.

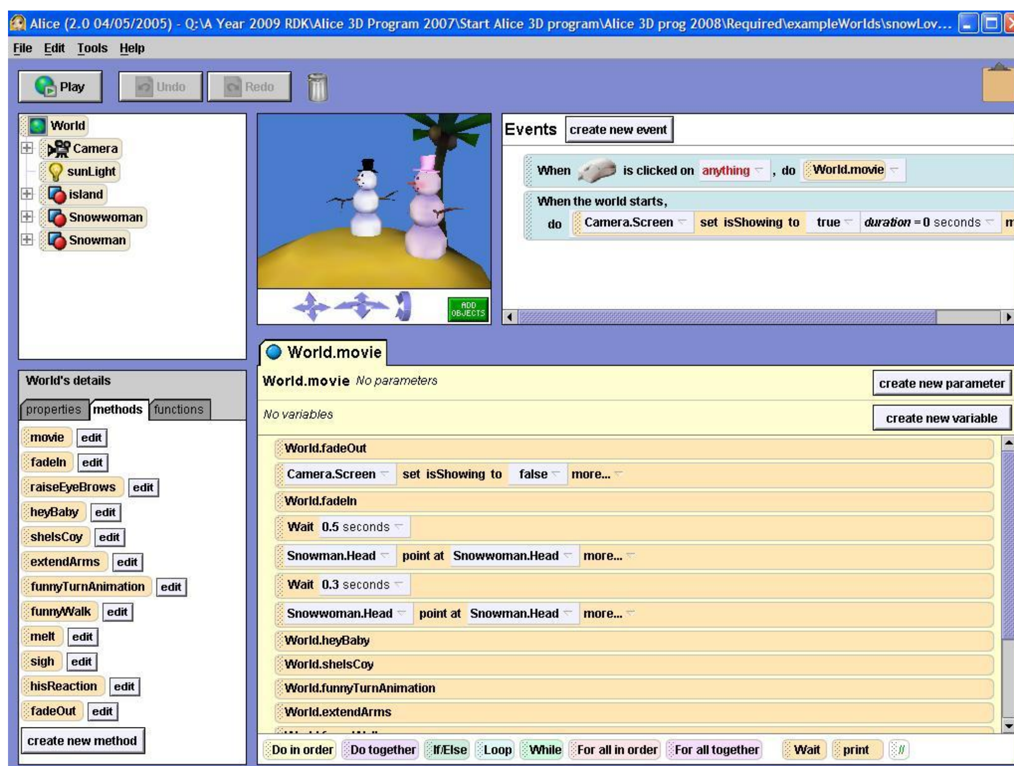


Figura 2. Ejemplo de edición de comportamiento en Alice.

A continuación, se presentan dos lenguajes visuales destinados a ayudar a programar el *LEGO Programmable Brick* [5]. Este dispositivo es un pequeño computador capaz de controlar cuatro motores y seis sensores, que se emplea para construir y manejar robots formados con piezas de LEGO. El primer lenguaje es LogoBlocks [6], alternativa gráfica a la programación textual con BrickLogo, que a su vez es una extensión del lenguaje Logo, al cual se le añadieron primitivas para controlar los sensores y motores del dispositivo. Con LogoBlocks se dispone de una interfaz tradicional de ventanas, en la que se arrastran bloques desde una paleta al área de edición, para formar las instrucciones que controlarán el dispositivo. Al estar orientado a principiantes, las construcciones permitidas evitan cometer errores sintácticos, pero la expresividad queda limitada. Además, los usuarios

experimentados encuentran en ocasiones más sencillo escribir las sentencias de manera textual que de manera gráfica.

En segundo lugar encontramos LEGOsheets [9], basado en AgentSheets. LEGOsheets es un entorno educativo que pretende facilitar la tarea de programar y/o simular el comportamiento del *LEGO Programmable Brick*. Tanto los sensores como los motores del dispositivo pueden conectarse al computador donde se ejecute LEGOsheets, de modo que las entradas de los sensores de LEGOsheets serán las recibidas a través de los sensores del mundo real, mientras que el comportamiento que se defina con esta aplicación tendrá consecuencias inmediatas en los motores reales del dispositivo. La idea es permitir a los niños interactuar primero con la interfaz, de manera que se familiaricen con ella y vean las consecuencias que sus acciones tienen sobre el dispositivo, para posteriormente pasar a especificar comportamiento mediante reglas. A cada motor se le puede asociar una regla de comportamiento del tipo IF-THEN, además de un valor por defecto que se emplea al iniciar la simulación o bien en caso de que la condición de la regla no se cumpla. Las condiciones de las reglas son muy sencillas, y las acciones se limitan a dar valor a los motores, pero lo importante es que la transición paulatina desde la especificación manual hacia la especificación mediante reglas hace que esta última tarea sea fácilmente comprensible.

Estos entornos, basados en aplicaciones de escritorio, y con métodos de entrada conocidos y básicos (teclado y ratón) son más fáciles de desarrollar. Sin embargo, hay otros trabajos relacionados que exploran nuevos métodos de interacción, y que son por tanto de interés. AlgoBlock [11] es un lenguaje de programación tangible dirigido a mejorar y facilitar la interacción entre los usuarios. El objetivo de AlgoBlock es lograr dirigir un submarino dentro de un laberinto, empleando piezas físicas con forma de bloques que pueden ser conectadas entre sí. Los movimientos que se programan para el submarino mediante los bloques físicos pueden verse en un monitor.

TurTan [10] es un lenguaje de programación tangible que aprovecha las capacidades de interacción de las superficies interactivas. Mediante tangibles que representan instrucciones, TurTan pretende ayudar en el aprendizaje de conceptos de programación utilizando dichos tangibles para indicar los movimientos de una pequeña tortuga situada inicialmente en el centro de la mesa interactiva. Cada tangible representa un movimiento, que puede ser parametrizado rotando dicho tangible al ponerlo sobre la superficie de la mesa. De este modo, se logra que la tortuga se desplace más o menos, o que tome una u otra orientación. Combinando adecuadamente los tangibles se pueden llegar a crear figuras geométricas asombrosas.

Finalmente, Tern [12] es otro lenguaje de programación tangible, esta vez sin dispositivos electrónicos. Las instrucciones son piezas de plástico o de madera que se conectan entre sí, y que posteriormente son escaneadas para su procesamiento. Las piezas están diseñadas de manera que ofrecen orientación sobre las posibles conexiones entre ellas, pero en caso de cometer errores sintácticos en la construcción del programa, éstos se muestran tras analizar la imagen escaneada. El factor colaborativo no es tan fácil de conseguir en aplicaciones de escritorio, pero en cambio estos lenguajes tangibles permiten hacer hincapié en la comunicación y la colaboración como forma de aprendizaje.

Hasta el momento tenemos que, en ocasiones, la especificación de reglas de manera visual enmascara completamente que se está definiendo una regla de comportamiento, como es el caso de AgentSheets, lo cual limita notablemente la expresividad permitida, y no nos acerca demasiado a la metodología de programación. En el otro extremo, si mostramos al usuario que está definiendo una regla de comportamiento, puede que el mecanismo de especificación empleado sea demasiado complejo o requiera de entrenamiento previo, como en el caso de Alice. Además, es frecuente que las construcciones permitidas se limiten al dominio, como es el caso de TurTan o AlgoBlock. Para facilitar la tarea de especificación de comportamiento en forma de reglas, es necesario encontrar un lenguaje que no restrinja dramáticamente las construcciones a ningún dominio concreto, sino que pueda ser adaptable fácilmente a un amplio abanico de juegos reactivos.

1.3. Inteligencia ambiental

Otra área de interés en las que los usuarios requerirían introducir reglas de comportamiento es el área de la Inteligencia Ambiental (AmI, del inglés *Ambient Intelligence*) [13]. La inteligencia ambiental es un área de la informática que pretende hacer los entornos que nos rodean sensibles a nosotros, de manera que se adapten para hacer nuestro día a día más fácil, y sin que notemos la presencia de dichos sistemas [14]. Por tanto, para que estos sistemas inteligentes sean prácticos, se necesita la mayor automatización y transparencia posible en la adaptación de los mismos. Pero debido a la gran variedad de situaciones posibles, es probable que las soluciones tomadas por los desarrolladores para una situación concreta no sean del agrado de todos los usuarios. Por ello, sería ventajoso disponer de mecanismos para que los usuarios finales pudieran personalizar o configurar a su gusto el comportamiento de dichos sistemas.

Hasta ahora, los avances en inteligencia ambiental se han centrado principalmente en mejorar la transparencia y la automatización de las tareas. Para ello se ha tratado de desarrollar la inteligencia de los subsistemas de sensores, así como mejorar la ubicuidad, entre otros. Estos avances, aunque han sido sumamente importantes y han conseguido altos grados de automatización, han dejado de lado la interacción explícita del usuario con el sistema, pretendiendo la transparencia absoluta de las tareas. Sin embargo, debido a la variedad de usuarios, dispositivos, ámbitos de aplicación y escenarios, parece una tarea casi imposible desarrollar infraestructuras capaces de adaptarse de manera autónoma a las necesidades reales de cada circunstancia. Se hace patente, por tanto, que no todas las situaciones pueden ser aprendidas, ni las respuestas ofrecidas por los sistemas pueden ser del agrado de todos los usuarios, ya que incluso para una misma situación o escenario, distintos usuarios probablemente tendrán distintas preferencias sobre las acciones a llevar a cabo.

Pero son precisamente estas preferencias las que pueden marcar la diferencia, ya que se pueden convertir en una gran fuente de conocimiento a ser aprendido por los sistemas en las etapas iniciales de configuración. Sin dejar de buscar la autonomía y la transparencia, hay que tener en cuenta el conocimiento que puede ser captado a través de la personalización de los usuarios, tal y como se hace ya en otras áreas de la informática.

Algunos autores ya han visto la necesidad de personalización para facilitar la adaptación y la configuración de sistemas en entornos inteligentes en los que la variabilidad la marca el usuario. Sin embargo, en este tipo de estudios, las herramientas proporcionadas para la personalización se enfocaban a diseñadores, desarrolladores, o usuarios finales bien preparados, pero no para usuarios finales. Además, las interfaces proporcionadas con estas herramientas de personalización o bien permiten el desarrollo rápido de prototipos, o bien permiten la expresión visual de reglas simples, que aunque sean suficientemente potentes en algunos casos, de algún modo están limitadas al dominio de aplicación.

Por tanto, supondría una ventaja involucrar explícitamente al usuario final, y permitirle definir parte de la configuración deseada para la adaptación del sistema, ya que de este modo el usuario se sentiría familiarizado con el entorno y sus requisitos. Además, el sistema no tendría que depender exclusivamente de información totalmente aprendida o automática, o de la información que ha sido codificada por los desarrolladores y diseñadores.

A continuación se presentan diversos estudios orientados a proveer mecanismos para especificar el comportamiento por parte de los usuarios, para que el sistema se adapte mejor a sus necesidades.

Uno de estos trabajos es Nexel [15], una herramienta gráfica para la personalización. Nexel permite personalizar y adaptar las aplicaciones a necesidades específicas como parte del proceso de desarrollo de aplicaciones ubicuas basadas en componentes auto-configurables. El editor de Nexel permite al usuario especificar el comportamiento a llevar a cabo cuando se producen ocurrencias de evento por parte de los dispositivos. Como parte del comportamiento, se pueden introducir bifurcaciones, estructuras de repetición y secuencias de instrucciones de control. Para definir visualmente el comportamiento, Nexel dispone de un área de especificación a la que se pueden arrastrar componentes de diversa índole y con diferente semántica, conectándolos secuencialmente entre sí, como se puede observar en la Figura 3 (extraída de [15]). Sin embargo, el proceso de configuración está diseñado para dar soporte a desarrolladores u otros usuarios en el contexto de desarrollo de productos, no para usuarios finales. De hecho, con Nexel se requiere tener ciertos conceptos de programación, aunque no avanzados, para llevar a cabo esta tarea de personalización.

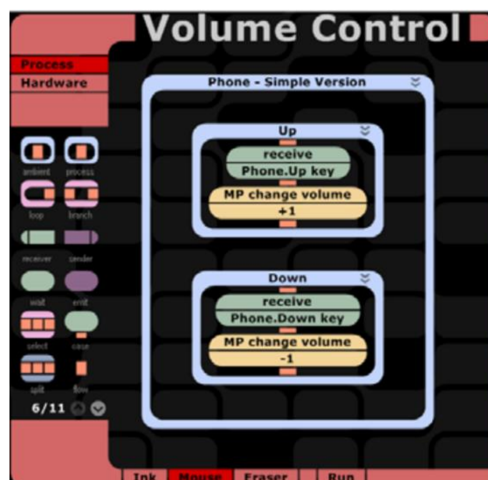


Figura 3. Interfaz de Nexel para la personalización del comportamiento de un dispositivo.

iCAP [16] es una herramienta para el prototipado de aplicaciones sensibles al contexto. Con iCAP no se necesita escribir ninguna línea de código para desarrollar un prototipo, ya que emplea una interfaz gráfica basada en ventanas y un lápiz para definir las reglas de comportamiento de los dispositivos. iCAP permite definir las entradas y salidas del sistema, para posteriormente utilizarlas en la especificación de las reglas. Las reglas en sí son del tipo IF-THEN, y al igual que en Nexel, se construyen arrastrando elementos al área de edición. En iCAP, esta área está dividida en dos partes, IF y THEN, para especificar por un lado las condiciones, y por otro lado, las acciones. La expresividad de estas reglas sólo permite operadores lógicos o relacionales, mientras que el ámbito de las reglas, al ser definido por los usuarios, no está limitado al dominio. Aunque en principio iCAP está enfocado a permitir a los desarrolladores crear rápidamente prototipos simulables, se pretende también que pueda ser empleado por usuarios finales para que estos no necesiten tener nociones de programación para crear sus propias aplicaciones.

Otro de los trabajos que se han llevado a cabo para controlar el comportamiento de aplicaciones sensibles al contexto es el presentado en [17]. Esta vez, las reglas para especificar comportamiento son reglas ECA más expresivas que en iCAP. Se han desarrollado dos editores de reglas destinados a usuarios finales: uno basado en una interfaz de ventanas, y el otro para dispositivos PDA. En el editor basado en ventanas, se permite o bien crear nuevas reglas, o bien modificar las ya existentes. Los mecanismos empleados para la edición y modificación de reglas son menos intuitivos (pestañas, listas, *checkbox*…) pero permiten expresar reglas más complejas. Además, mientras se está editando la regla, se muestra de manera textual el contenido de la misma. En el segundo editor para PDA, tan sólo se pueden llevar a cabo modificaciones de reglas ya existentes, de manera que al comenzar a modificar una regla ya se puede observar un ejemplo de regla completamente construido, lo cual facilita la tarea. Los mecanismos visuales de representación de las reglas son una especie de piezas de puzle conectadas entre sí para definir las condiciones y las acciones, y por tanto resultan más intuitivos que los del primer editor. Se han llevado a cabo estudios con ambos editores, determinándose que el editor gráfico para PDA permitía definir reglas más rápidamente y cometiendo menos errores.

El trabajo descrito en [18] presenta un prototipo de interfaz visual para la especificación de reglas de comportamiento, completamente enfocada a usuarios no expertos. Al igual que las dos herramientas anteriores, la especificación de la regla se realiza arrastrando los componentes a las áreas de condición y acción. A diferencia de iCAP, donde las reglas eran del tipo IF-THEN y nada permitía distinguir los eventos de las condiciones, este prototipo añade a las construcciones IF-THEN dos bloques opcionales, WHEN y OR-IF. Un bloque WHEN nos permitiría establecer condiciones sobre los eventos que se indiquen en la condición, mientras que un bloque OR-IF nos permite especificar alternativas. La aplicación todavía no ha sido validada con usuarios no expertos, pero pretende facilitar de cualquier manera la tarea de edición de una regla mediante ayudas visuales o sugerencias.

SiteView [19] es una herramienta para la personalización que explora una nueva forma de interacción: las reglas son creadas mediante tangibles. SiteView dispone de varios componentes, todos ellos orientados a facilitar la tarea de edición de reglas a los usuarios finales. Por un lado, se encuentra un visor de reglas, que muestra la regla actual en edición, o bien las reglas aplicables a determinadas

condiciones. Por otro lado, también dispone de un visor del entorno, que muestra cómo se comporta el entorno al activar una regla. Las reglas son del tipo IF-THEN, por tanto, SiteView provee un área para definir las condiciones y otra área para especificar las acciones. Las condiciones se definen posicionando elementos tangibles predefinidos sobre unos sensores, mediante los cuales se forma la expresión condicional resultante realizando la conjunción de los distintos elementos situados en estos sensores. Las acciones se determinan posicionando sobre una representación a escala del entorno, los tangibles que indican cambios en dichos elementos. Tanto los tangibles para las condiciones, como los tangibles para las acciones, están predefinidos en base a comportamientos o situaciones comunes, por lo que las capacidades de las reglas quedan limitadas a los elementos tangibles disponibles. La forma en que se definen las condiciones también impone restricciones a los usuarios, ya que no se pueden usar tangibles de la misma clase en una única condición (por ejemplo, si tenemos tangibles para indicar días de la semana, y tangibles para representar horas, no podríamos poner reglas del tipo "Si es martes y miércoles…", pero sí podríamos poner "Si es martes y son las 8h…").

Un ejemplo de herramienta de personalización no basada en reglas es aCAPella [20], que emplea la programación por demostración para definir comportamientos en entornos sensibles al contexto. Para especificar el comportamiento a tomar dada una situación, el sistema debe ser primero entrenado. Se debe preparar para captar datos de diferentes sensores, y posteriormente los usuarios seleccionan qué entradas son las que determinan el contexto actual, y qué acciones se deben tomar dadas dichas entradas. Esta herramienta está orientada a usuarios finales, ya que ellos son quienes mejor conocen las características particulares para las que desean ejecutar alguna acción.

Por último, el trabajo presentado en [21] permite a los usuarios finales especificar el comportamiento de sistemas ubicuos mediante una herramienta basada en reglas ECA. El lenguaje textual que se emplea para definir las reglas permite una alta expresividad, ya que pueden escribirse expresiones complejas empleando caracteres comodín para filtrar elementos, así como elementos TIMER que permiten realizar acciones en función del tiempo (acciones que se ejecutan en un determinado momento, acciones que deben durar un determinado tiempo, etc). Sin embargo, el uso de estas expresiones complejas requiere de cierta experiencia, por lo que se han desarrollado varias interfaces gráficas que permitan a usuarios finales llevar a cabo dichas tareas sin que necesariamente tengan que tener conocimientos de programación. No obstante, cada interfaz presenta tanto puntos fuertes como desventajas.

La mayoría de los trabajos en el campo de personalización de entornos inteligentes se basan en reglas para especificar el comportamiento, ya que tal y como se ha visto en [2] y [3], estas construcciones son mejor comprendidas por no-programadores. Pero hay una gran variedad de mecanismos de visualización y edición de estas reglas, todos ellos pretendiendo facilitar la tarea de edición, sin perder expresividad. Sin embargo, como ocurría en el entorno de juegos, cuanto más simple y sencilla sea la interfaz, menor expresividad permite, mientras que si la interfaz es más compleja y menos intuitiva, se pueden construir reglas más expresivas. El mejor ejemplo lo tenemos en [17], donde se han presentado dos interfaces visuales, y se ha podido observar cómo afecta el factor de la usabilidad a la expresividad de la regla. Por tanto, es necesario en el campo de AmI, encontrar

un editor de reglas de comportamiento adecuado cuya interfaz encuentre el equilibrio entre sencillez y potencia expresiva. Este editor de reglas proveería al área de la inteligencia ambiental de una herramienta genérica y potente capaz de permitir la personalización de dispositivos y tareas por parte de no-programadores de manera sencilla.

1.4. Presentación del proyecto

Una vez observadas las dificultades existentes para lograr la especificación de reglas de comportamiento en dos áreas de interés, los requisitos que debería tener una herramienta de edición de reglas para paliar estas dificultades son los siguientes:

- La especificación de las reglas debe ser visual.
- La capacidad expresiva de las reglas debe ser potencialmente alta, ya que hasta la fecha, la especificación visual de reglas de comportamiento no es tan potente ni permite tanta flexibilidad como la especificación textual de dichas reglas.
- Las reglas deben permitir especificar comportamiento de manera genérica, para poder emplear el mismo editor en diferentes contextos, sin tener que aprender un lenguaje de especificación distinto en cada contexto.
- La edición debe ser sencilla, y estar al alcance de los usuarios más noveles. Los usuarios sin ningún conocimiento previo de programación deben poder definir reglas de comportamiento sin excesiva dificultad.

El presente proyecto consiste en la implementación de un editor visual de reglas basadas en un modelo Evento-Condición-Acción (ECA) simplificado, en el que la especificación de los parámetros implicados se llevará a cabo por medio de flujos de datos de transformación. Este editor permitirá a usuarios no expertos desarrollar las reglas necesarias para expresar comportamiento reactivo en diversos ámbitos de manera sencilla. Las reglas se definen en tiempo de edición, pero es posible que determinadas acciones sólo deban ser llevadas a cabo bajo ciertas condiciones. Así pues, las mismas reglas deben definir las condiciones bajo las cuales serán instanciadas en tiempo de ejecución, dependiendo de los eventos que ocurran y del estado del ecosistema. Por tanto, una regla puede entenderse como una plantilla genérica que es instanciada en tiempo de ejecución mediante los valores obtenidos a partir del estado del ecosistema. Por otro lado, la expresividad del lenguaje basado en reglas debe permitir cálculos no triviales, como ya hemos comentado. Por tanto, los usuarios que diseñen un ecosistema reactivo deben ser capaces de crear expresiones complejas para definir los valores de los parámetros de la regla. Para lograr este objetivo, se ha optado por enriquecer la especificación de comportamiento mediante flujos de datos. Estos flujos de datos deben contener elementos capaces de transformar y procesar datos de entrada, relacionándolos entre sí, dando lugar a construcciones más complejas que nos conduzcan a reglas más expresivas.

Los principales puntos en los que se estructura el presente documento son, en primer lugar, la formalización de las reglas ECA en las que se basa el lenguaje, junto a la exposición del metamodelo de reglas ECA basado en flujos de datos que da soporte al mismo. A continuación se detallará la implementación de un editor

basado en una interfaz WIMP que soporte la especificación visual de reglas y de los flujos de datos implicados en ellas. Más adelante se expone el procesador de eventos que se ha implementado para permitir la activación y ejecución de las reglas durante la simulación del ecosistema. Para la validación tanto del editor como del procesador de eventos, se ha desarrollado un depurador con interfaz WIMP que se expone también en este documento. Además, se presentan una serie de experimentos realizados para determinar la escalabilidad y el buen rendimiento del metamodelo y el procesador de eventos implementados. Finalmente, se expondrán las conclusiones y se detallará el trabajo futuro a desarrollar.

2. DaFRule: Reglas enriquecidas mediante flujos de datos

Como se ha revisado en el capítulo anterior, y tal y como se demuestra en [22], las aplicaciones y sistemas orientados a no-programadores cuyo objetivo es emplear la programación para lograr otros medios, están en su mayoría basados en eventos. Eventos y reglas son, por tanto, los componentes básicos necesarios que permiten expresar comportamientos en dichos sistemas. Una aproximación basada en eventos hace más comprensible la definición de las reglas que establecen dicho comportamiento, tal y como demuestran los resultados de los estudios experimentales en [2][3]. Es por ello que en el presente trabajo se ha adoptado esta aproximación, ya que parece la más razonable para expresar comportamiento en un entorno reactivo.

2.1. Entorno como Ecosistema

Ya que se pretende construir un editor de reglas genérico que pueda ser aplicado en diversos ámbitos, el modelo subyacente no puede recaer en conceptos específicos del dominio de aplicación. Por tanto, es necesario llevar a cabo un modelado del entorno más genérico y flexible, que permita describir elementos comunes presentes en los entornos reactivos, que tal y como se explicará más adelante puede requerir la adaptación de datos manejados por el editor al modelo conceptual que finalmente lleva a cabo la animación y gestión del entorno.

Para modelar entornos de manera genérica, sin recaer en elementos específicos de cada dominio, se ha definido el concepto de ecosistema. Un ecosistema permitirá representar los elementos del entorno necesarios para especificar comportamiento, de manera más global. Principalmente, un ecosistema se caracteriza por estar poblado de entidades. Dichas entidades son instancias de una definición de entidad, que especifica tanto su comportamiento, como otros aspectos relacionados con el dominio específico en el que se encuentra. Si fuera el caso de juegos, los tipos de entidades podrían dar idea de la apariencia visual de sus instancias, mientras que si se tratase de un entorno AmI, los tipos de entidades representarían tipos de dispositivos cuyas funciones e interfaces quedan definidas en el tipo. En otras palabras, las entidades conforman a un tipo de entidad determinado. En cierto modo, estas entidades del ecosistema han sido dotadas de naturaleza orientada a objetos, por lo que cada tipo de entidad tendrá un conjunto de acciones disponibles, definidas en base a sus parámetros de entrada, así como una serie de propiedades. Una instancia de un determinado tipo de entidad dispondrá de aquellas acciones que provea su tipo. Además, dispondrá de un conjunto de pares propiedad-valor, donde las propiedades de este conjunto serán las que se especifican en el tipo, y los valores de dichas propiedades indican el estado del ecosistema para cada entidad durante la simulación o evolución del mismo. Por ejemplo, en el ámbito de juegos, podríamos tener un tipo de entidad *Elfo*, y dos instancias de dicho tipo, *elfo1* y *elfo2*. Imaginemos que el tipo *Elfo* define 3 propiedades: *salud*, *experiencia* y *ataque*, representando respectivamente el porcentaje de vida del personaje, su experiencia acumulada, y la cantidad de daño que puede hacer al atacar. Pero tanto *elfo1* como *elfo2* tendrán distintos valores asociados a dichas propiedades en cada momento de la simulación. Por otra parte,

en el ámbito de AmI, un tipo de entidad podría ser *AireAcondicionado*, cuya propiedad sería la *temperatura*, y se podrían tener tantas instancias de este tipo como dispositivos de aire acondicionado existan en una oficina o casa. Cada instancia de tipo *AireAcondicionado* tendría un valor propio asociado a la temperatura, y se podrían regular estos valores tanto de manera independiente como de manera conjunta, tal y como se verá en los siguientes apartados.

Tanto las propiedades de una entidad, como los parámetros de las acciones se refieren a elementos que en tiempo de ejecución contendrán un valor que deben conformar a un tipo determinado. Estos elementos podrán representar valores básicos, como pueden ser valores numéricos, cadenas de caracteres, valores booleanos, etc. pero también pueden referirse a entidades, que servirían para expresar relaciones entre elementos del ecosistema en forma de características específicas de dichos elementos. Por ejemplo, en el ejemplo de los elfos, supongamos que existe en el ecosistema un nuevo tipo de entidad, *Orco*, y que existen dos instancias de este tipo, *orco1* y *orco2*. Supongamos que cada elfo tiene como objetivo en el juego matar a un determinado orco. Puede herir al resto de orcos, pero gana el juego tan sólo si consigue eliminar al orco correspondiente. Para saber qué orco debe eliminar cada uno de los elfos, el tipo *Elfo* podría disponer de una propiedad llamada *enemigoPrincipal*. Dicha propiedad será de tipo *Orco*, y de este modo podríamos asociar las instancias de tipo *Orco* a las propiedades de las instancias de tipo *Elfo*, de modo que *elfo1.enemigoPrincipal = orco1*, y *elfo2.enemigoPrincipal = orco2*.

Por otra parte, durante el tiempo de vida del ecosistema, las entidades son capaces de inyectar ocurrencias de eventos en el sistema a medida que éste evoluciona. Dichas ocurrencias conforman a una definición de un tipo de evento. Las ocurrencias de evento lanzadas por las entidades durante la simulación harán evolucionar el estado del ecosistema a medida que las reglas especificadas se instancien en consecuencia. Cada tipo de evento tendrá un conjunto de atributos que permitirán, de manera similar a lo que ocurría con las propiedades de una entidad, caracterizar las ocurrencias de evento. De este modo, cada ocurrencia de evento que conforme a un determinado tipo de evento, dispondrá de un conjunto de pares atributo-valor, cuyos atributos serán los especificados en la definición del evento, y donde los valores representan la instanciación de dicho evento para un instante de simulación concreto. Por ejemplo, en el caso de entornos inteligentes, supongamos que tenemos un tipo de entidad llamado *SensorTemperatura*, cuyas instancias se encargan de captar la temperatura ambiente del lugar donde se encuentran ubicadas. Supongamos también que existe un tipo de evento llamado *CambioTemperatura*, cuyo único atributo es *nuevaTemperatura*, que como su nombre indica, sirve para reflejar la nueva temperatura detectada. Cada vez que una instancia de tipo *SensorTemperatura* detecta un cambio en la temperatura, dicha instancia lanzará una ocurrencia de evento del tipo *CambioTemperatura*, cuyo atributo *nuevaTemperatura* tendrá asociado el valor correspondiente a la nueva temperatura detectada. De manera análoga a como sucedía con las propiedades de una entidad o con los parámetros de una acción, los atributos de un evento también representan elementos que en tiempo de ejecución tomarán un valor determinado para cada ocurrencia de evento. De nuevo podremos establecer como valores de un atributo tanto valores básicos como cualquier instancia de un tipo del ecosistema.

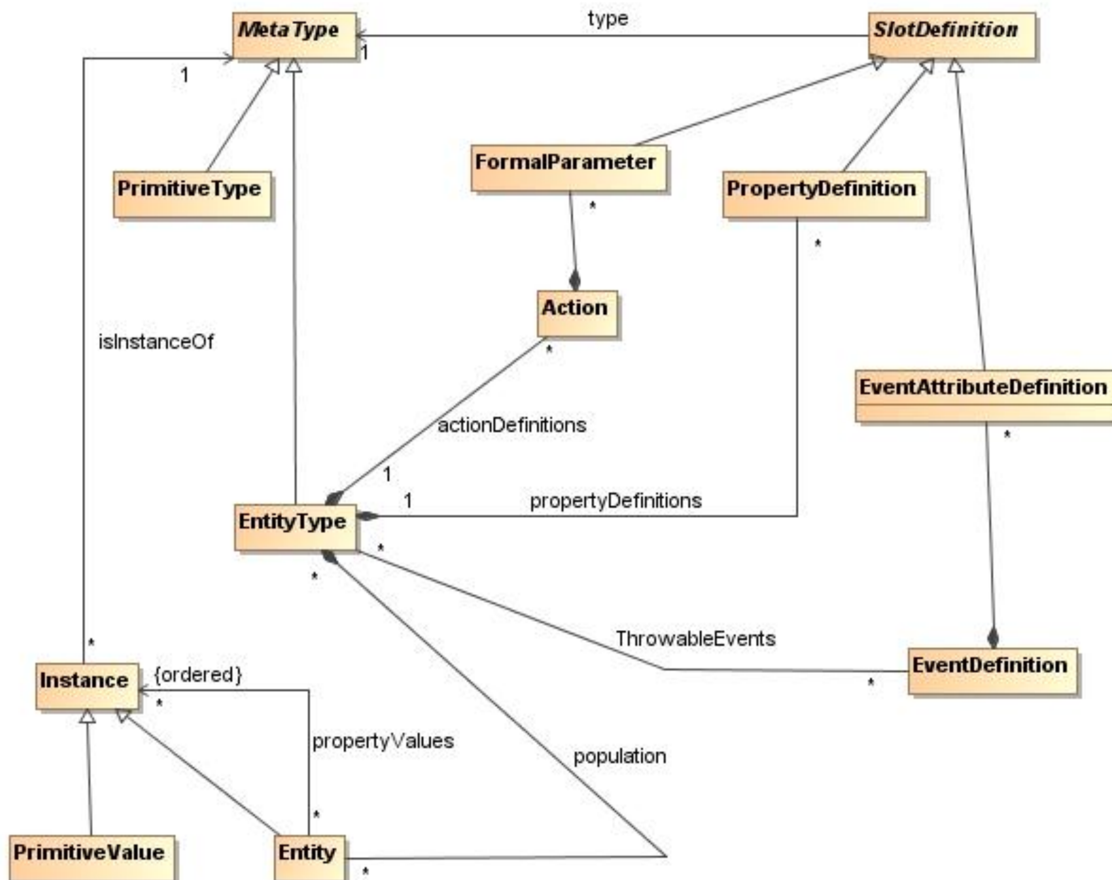


Figura 4. Diagrama de clases UML para dar soporte a entornos basados en entidades.

La descripción completa de tipos de entidades, acciones, propiedades, tipos de eventos e instancias de entidad se especifican mediante un metamodelo genérico, que puede verse en la Figura 4, de manera que el editor descrito en este trabajo permita cubrir una gran variedad de escenarios e incluso una gran variedad de ámbitos en los que se requiera definir o personalizar comportamiento. De este modo se logra mayor abstracción y flexibilidad, ya que los conceptos específicos pueden variar en cada escenario de aplicación, pero se mantienen comunes para la edición de reglas. No obstante, lo habitual es que sea necesario llevar a cabo una posterior proyección para adaptar las abstracciones con las que trabaja el editor con los elementos específicos del middleware de destino que se utilice en cada caso [23]. Por ejemplo, en el dominio de AmI sería necesario proyectar la terminología del ecosistema modelado a la ontología u ontologías sobre las que descanse el middleware de gestión del ambiente inteligente.

Otra de las ventajas del metamodelado es que se facilita el mantenimiento y la evolución de los sistemas, ya que los dominios específicos de aplicación del editor pueden sufrir cambios, mientras que el metamodelo se mantiene. Concretamente, la aparición de nuevos sensores en el campo AmI no requeriría más que introducir un nuevo tipo de entidad en el editor, y definir para ello sus propiedades y acciones, pero el metamodelo en sí no se vería afectado. Lo mismo ocurre en el entorno de juegos, donde añadir nuevos tipos de personajes o nuevas características de los mismos no requiere actualizar el metamodelo.

2.2. Reglas

Hemos visto que la abstracción principal sobre la cual se centra un ecosistema son las entidades y que en su especificación son necesarias ciertas definiciones asociadas como son los tipos de entidad, que indican qué propiedades tendrán las entidades y qué acciones serán capaces de realizar. De forma similar a como se conceptualizaban estas características básicas de las entidades, se podría considerar que las reglas de comportamiento debieran ser artefactos especificados a nivel de entidad o incluso en su tipo. Sin embargo, esa elección hubiera confinado la expresión de reglas a que dicha entidad interviniera necesariamente en las mismas, limitando, por tanto, la expresividad. Para evitar esta limitación, las reglas se han considerado a nivel del ecosistema, no ligadas a ninguna entidad específica. De este modo, se pueden definir reglas en las que intervienen o bien individuos específicos, o bien poblaciones de entidades de manera más natural, en lugar de tener que definir una regla para cada una de las entidades de una colección si quisiéramos llevar a cabo una operación sobre ellas.

En este contexto consideramos reglas definidas formalmente como un par ordenado R = <P, Q>, donde P es el antecedente y Q el consecuente. El antecedente, P, se define como P = (E, S, C) donde E es la definición de un evento que debe ser lanzado por una fuente S (p.ej. una entidad), y C es la condición que deben cumplir ciertos datos (p.ej. propiedades y atributos) del evento E y la fuente S para que la regla se instancie.

El consecuente, Q, se define como Q = (T, O, F, {DP}), donde T es la población destino a la que afecta la regla, O es la operación a ejecutar sobre cada una de las entidades de la población destino, F es una condición que filtra las entidades de la población destino que se verán afectadas por la operación anterior. Finalmente, {DP} es un conjunto de procesos de datos que especifican cómo se establecen los parámetros de la operación antes de su ejecución.

La operación O del consecuente de la regla podrá ser, o bien ejecutar una acción, o bien asignar valor a una propiedad de la población destino. Para el caso en el que se pretenda asignar valor a una propiedad, tan sólo es necesario un único proceso de datos DP que especifique cómo calcular el valor a establecer en la propiedad. En cambio, si la operación es la invocación de una acción, habrá tantos procesos de datos en el conjunto {DP} como parámetros tenga la acción. Cada uno de estos procesos de datos especificará cómo debe establecerse el valor de cada parámetro de la acción, antes de que esta se ejecute.

Las condiciones C y F se pueden expresar también mediante procesos de datos en los cuales el valor resultante debe ser booleano, y se empleará en simulación para determinar si la condición se satisface o no. Por un lado, dados una fuente S y un evento E que activen la regla, la condición C involucrará tanto propiedades de la fuente como atributos del evento, entre otros, realizando transformaciones y operaciones con dichos datos para obtener finalmente un valor booleano que determine el cumplimiento de esta condición. Por otro lado, dada una población destino T, la condición de filtrado F involucrará principalmente propiedades de la población destino de la regla, y operando con dichos datos, se establecerá la condición que se debe cumplir para que una entidad de T se vea afectada por la operación O.

La semántica para una regla R es la siguiente: si una entidad perteneciente a S lanza un evento de tipo E, y se cumple la condición C, entonces la regla es instanciada y lanzada. La operación O se ejecutará sobre aquellas entidades de T que cumplan la condición F, estableciendo los parámetros de O conforme a la especificación de los procesos de datos {DP}. La estructura de la regla formalizada se muestra en la Figura 5.

SI S: ___________ **LANZA** una ocurrencia de evento de tipo E: ___________
[**Y SE SATISFACE** la condición C: ____]
ENTONCES PARA CADA entidad en T: _______________
[**QUE CUMPLA** la condición de filtrado F: _________________________________]
EJECUTAR O según los valores especificados en {DP}

Figura 5. Estructura de una regla.

Uno de los aspectos más destacables del modelo de reglas es la posibilidad de establecer tanto la fuente como el destino de manera concreta o de manera genérica. Esto es, indicando una entidad en particular, o por contra, un tipo de entidad que denotará a todas las entidades que se instancien a partir de dicho tipo y estén en simulación. El significado de la especificación de las poblaciones de manera individual (es decir, la fuente y/o el destino de la regla son entidades concretas) es el mismo significado que en los sistemas de reglas tradicionales. En cambio, hay que tener en cuenta el significado de utilizar poblaciones fuente y/o destino utilizando un tipo de entidad, ya que la semántica difiere ligeramente con respecto a las poblaciones individuales, e incluso entre población tipo como fuente o como destino. Si se define la fuente como un conjunto de entidades de un tipo, la regla se instanciará en caso de que cualquier entidad de dicho tipo cause el evento de tipo E. Por tanto, si todas las entidades de un mismo tipo lanzan una ocurrencia de evento de tipo E, se instanciará una regla por cada entidad, sin necesidad de haber definido una regla específica para cada una de ellas.

Por otra parte, si se define la población destino de la regla como un conjunto de entidades de un tipo, entonces habrá una instanciación de la operación de la regla para cada una de las entidades de la población destino. En cada instanciación se comprobará la condición de filtrado sobre la entidad destino correspondiente, y si se cumple, dicha entidad se verá afectada por la operación O. En caso contrario, la operación no se instancia para dicha entidad. Este mecanismo es ventajoso y permite simplificar la especificación del comportamiento cuando las instancias del tipo de entidad son desconocidas a priori, o cuando existen varias instancias de un tipo de entidad que deben comportarse exactamente de la misma manera. La Figura 6 muestra algunos ejemplos textuales de reglas soportadas por el editor que se propone. Esta representación en lenguaje natural es tan sólo una representación textual de la capacidad expresiva que el metamodelo implementado permite.

SI S: *robot1* **LANZA** una ocurrencia de evento de tipo E: *AtravesarPortal*
ENTONCES PARA CADA entidad en T: {*robot1*}
EJECUTAR O: Asignar Propiedad (*robot1.posición* ← *AtravesarPortal.posición_salida*)

(a)

SI S: *avión* de *TipoAvión* **LANZA** una ocurrencia de evento de tipo E: *MisilLanzado*
[Y SE SATISFACE la condición C: *MisilLanzado.lugar = "Tierra"* **]**
ENTONCES PARA CADA entidad en T: *tanque* de *TipoTanque*
[QUE CUMPLA la condición de filtrado F: *distancia (tanque.posición, MisilLanzado.posición_colisión) < 10)***]**
EJECUTAR O: Asignar Propiedad (*tanque.defensa* ← *tanque.defensa - MisilLanzado.daño*)

(b)

SI S: *televisor* de *TipoTelevisor* **LANZA** una ocurrencia de evento de tipo E: *Encender*
ENTONCES PARA CADA entidad en T: *radio* de *TipoRadio*
[QUE CUMPLA la condición de filtrado F*: radio.estado == Encendido AND radio.ubicación == televisor.ubicación* **]**
EJECUTAR O: Ejecutar Acción *radio.Apagar ()*

(c)

SI S: *persona* de *TipoPersona* **LANZA** una ocurrencia de evento de tipo E: *UbicaciónCambiada*
[Y SE SATISFACE la condición C: *UbicaciónCambiada.destino = "Salón"* **]**
ENTONCES PARA CADA entidad en T: *persiana* de *TipoPersiana*
[QUE CUMPLA la condición de filtrado F:
persiana.ubicación == UbicaciónCambiada.destino **]**
EJECUTAR O: Ejecutar Acción *persiana.Levantar* (*altura* ← *5 * sensorLuzSalón.luminosidad / 20*)

(d)

Figura 6. Ejemplos textuales de reglas soportadas por el editor.

La regla de la Figura 6 (a) serviría para especificar el comportamiento de una entidad concreta en un juego de plataformas, donde la población fuente y destino son la misma entidad, y no existen condiciones asociadas a la regla. De este modo, cada vez que la entidad *robot1* lance una ocurrencia de evento del tipo especificado, la regla se instanciará, y se cambiará la posición del robot según el atributo *posición_salida* del evento. La regla de la Figura 6 (b) define el comportamiento típico de un juego de guerra, donde las poblaciones son genéricas, y se ha definido una condición sobre el evento de la regla, así como un filtrado sobre la población destino. Para cada una de las entidades de la población destino que cumplen la condición de filtrado, se lleva a cabo la asignación de un valor a la propiedad *defensa* de dicha entidad. Antes de llevar a cabo esta asignación es necesario calcular el valor a asignar mediante una operación aritmética, en este caso una resta entre el valor de la propiedad *defensa* de la entidad destino en cuestión, y el valor del atributo *daño* del evento que activó la regla. En la regla de la Figura 6 (c) se encuentra un ejemplo de regla para personalizar un entorno inteligente. En esta regla se emplean de nuevo poblaciones genéricas, pero en este caso la condición de filtrado sobre la población destino es una condición compuesta, ya que hace uso de un operador de conjunción para unir dos comparaciones sobre el estado de los elementos de la regla. Además, la operación involucrada en esta regla es la ejecución de una acción sin parámetros de entrada. Por último, la regla de la Figura 6 (d) es de nuevo aplicable al ámbito de la inteligencia ambiental. Se emplean poblaciones genéricas, hay una condición asociada al evento, y una condición de filtrado sobre la población destino, pero en

este caso cabe destacar que la operación a realizar es la ejecución de una acción con un parámetro de entrada. Para establecer el valor que debe tomar este parámetro se llevan a cabo varias operaciones aritméticas que además involucran una de las propiedades de otra entidad del ecosistema que no pertenece ni a la población destino ni a la población fuente.

Teniendo en cuenta la formalización de las reglas propuesta anteriormente, así como los ejemplos de reglas textuales, la expresividad requerida para el lenguaje de reglas debe al menos soportar asignaciones, operadores aritméticos y lógicos. Una gramática que describe de manera textual la expresividad buscada puede ser definida como sigue:

```
Instrucción := Variable '←' Expresión
Operando    := Variable | Constante | Expresión | Función
Operando_parentizado  := '(' Operando ')' | Operando
Operador    := '+' | '-' | 'x' | 'AND' | 'OR' | 'NOT' | '<'
            | '>' | '==' | '<=' | '>='
Expresión   := Operando_parentizado[OperadorOperando_parentizado ]
Función     := Nombre_función '(' Parámetros ')'
Parámetros := NIL | '(' Expresión [, Expresión]* ')'
Nombre_función  := 'MAX' | 'MIN' | 'ABS' | …
```

La gramática propuesta tan solo ofrece operaciones básicas, pero ya que la implementación de las reglas se basa en un metamodelo, resulta sencillo extender esta gramática con nuevas operaciones, o con funciones específicas del ámbito de aplicación, como por ejemplo *Distancia*, *Colisión*, etc. La manera de realizar esa extensión se verá en el siguiente apartado.

Al igual que con las entidades y eventos, las reglas y los procesos de datos van a describirse también por medio de un metamodelo que permita mayor flexibilidad, y que recoja toda la expresividad planteada en este capítulo. La Figura 7 representa el diagrama de clases UML que da soporte al modelo conceptual de reglas que se ha formalizado. En primer lugar, una regla reactiva (*ReactiveRule*) consta de un tipo de evento (*EventType*), una población fuente (*SourcePopulation*) y una población destino (*TargetPopulation*). Además, la regla reactiva debe contener una precondición en forma de proceso de datos (*PreconditionDataProcess*) que permita establecer las condiciones que se deben cumplir para activar la regla, así como una condición de filtrado (*FilterPreconditionDataProcess*) para determinar qué instancias de la población van a verse afectadas por el consecuente de la regla. Finalmente, el tipo de operación a realizar (*OperationBinding*) determina si se realizará la asignación de una propiedad o si se ejecutará una acción sobre la población destino. En función del tipo de operación, se tendrán más o menos procesos de datos (*BindingDataProcess*) para especificar o bien la propiedad a asignar, o bien los parámetros de la acción a ejecutar. Uno de los puntos importantes a tener en cuenta es la especificación tanto de la fuente como del destino de dos maneras posibles: o bien como una entidad específica del ecosistema (*EntitySourcePopulation* o *EntityTargetPopulation*), o bien como tipos de entidades, indicando una población de entidades del mismo tipo (*EntityTypeSourcePopulation* o *EntityTypeTargetPopulation*).

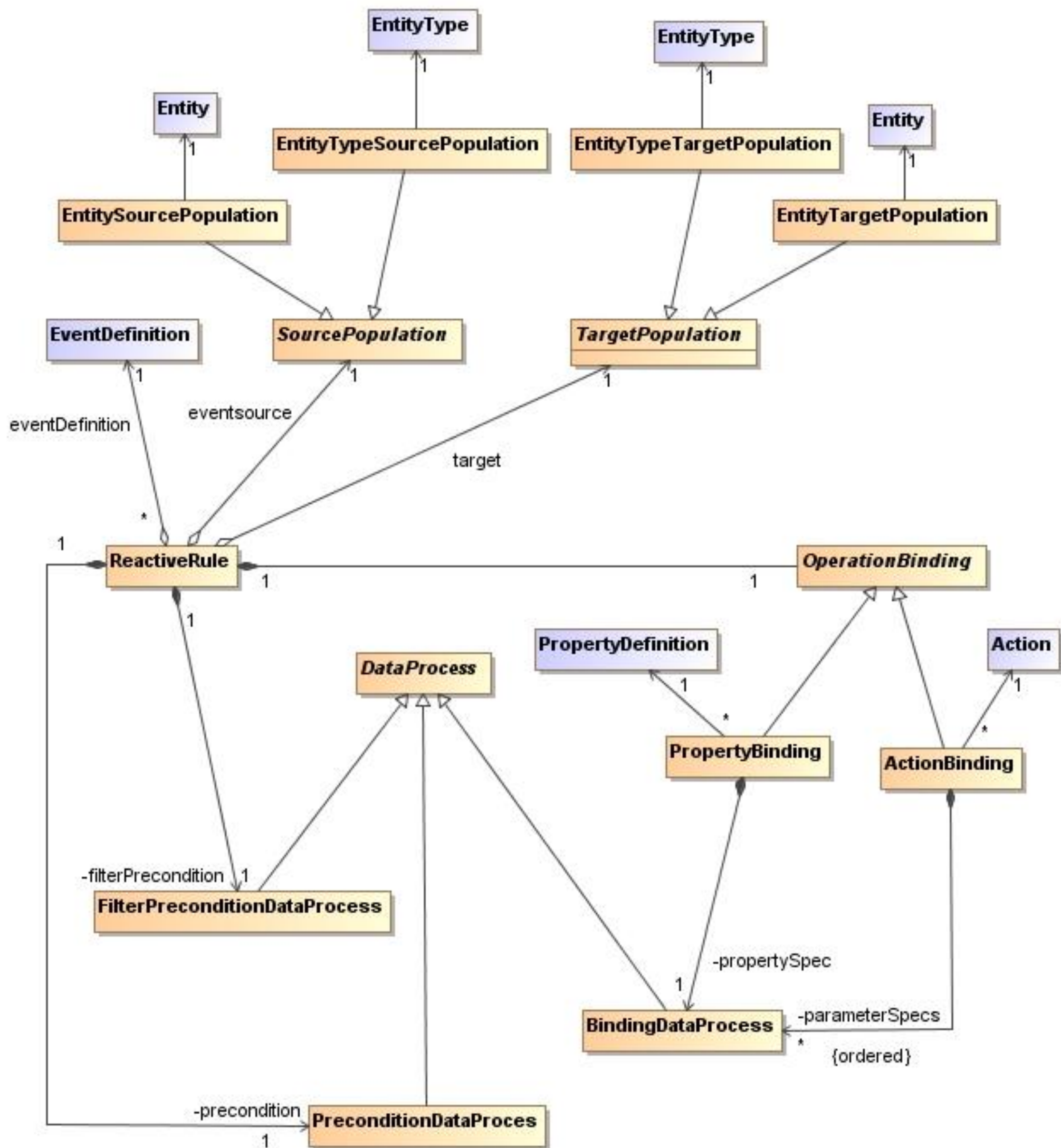


Figura 7. Diagrama de clases UML para dar soporte a las reglas.

2.3. Flujos de Datos

En las reglas definidas en el apartado anterior se disponía de diversos procesos de datos para expresar o bien condiciones o bien la especificación del consecuente de la regla. Se han dado algunos ejemplos textuales de reglas de comportamiento, y en este apartado se pretende exponer cómo se pueden expresar ciertas partes de dichas reglas de manera no textual mediante mecanismos sencillos que no limiten la expresividad. En el presente trabajo se ha considerado de sumo interés proveer mecanismos visuales sencillos e intuitivos que no limiten la expresividad deseada para el lenguaje de definición de reglas propuesto. Esta limitación de la expresividad se observa en otros sistemas de definición de comportamiento de manera visual, en los cuales las reglas se especifican de manera más simple, no permitiendo por ejemplo aplicar ninguna transformación a los datos antes de ser establecidos como parámetros de la operación. Así, en lugar de forzar a que los

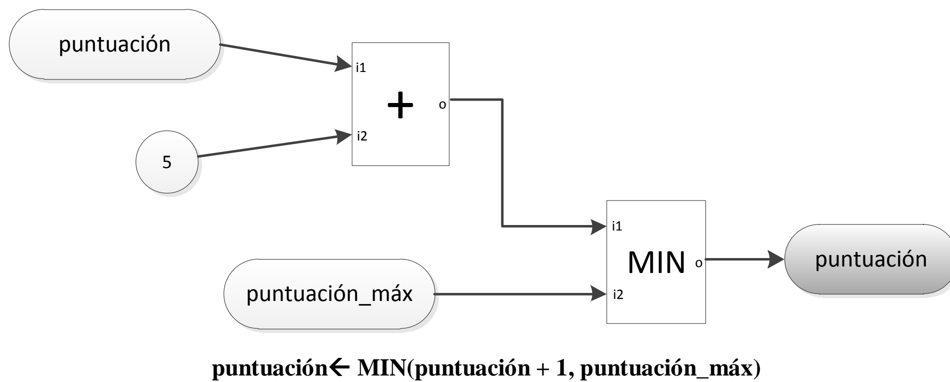
valores de los parámetros de una operación deban tomarse directamente de los valores que toman los atributos del evento que activó la regla, o de los valores de las propiedades de la entidad fuente, sin someterse a ningún tipo de transformación u operación, se ha optado por permitir, mediante flujos de datos, aplicar operaciones sobre los datos disponibles y combinarlos obteniendo construcciones mucho más complejas. De este modo se pretende combinar la sencillez de las estructuras de reglas típicas con la enorme expresividad que permiten las estructuras de flujos de datos, con lo que el lenguaje resultante será más potente y usable gráficamente.

Los procesos de datos presentes en una regla consistirán, por tanto, en un conjunto de flujos de datos y un conjunto de operadores, que se combinarán entre sí para realizar transformaciones sobre los datos de entrada, obteniendo como resultado final un valor que se asignará al consecuente del proceso de datos en curso (p.e. una propiedad, un parámetro de una acción, o el resultado de una condición). Las expresiones textuales que aparecían en las reglas presentadas en el apartado anterior son como muchas otras empleadas en cálculo y matemáticas en general, mientras que de manera visual, el proceso de datos asociado a una expresión tiene una estructura similar a una organización en árbol. La variable a ser calculada y asignada se encuentra en la parte derecha de la expresión, y los operadores o procesadores de datos, que realizan las transformaciones, se representan como cajas con entradas y salidas. De este modo, las variables y constantes involucradas en el proceso de transformación se conectan mediante flujos de datos a los operadores, quedando relacionadas entre sí. También es posible conectar las salidas de unos operadores con las entradas de otros, anidando operaciones o incluso reutilizando un mismo cálculo en diversos puntos de la expresión.

Las variables involucradas en un proceso de datos pueden ser, como ya se ha comentado, tanto atributos del evento que activa la regla, como propiedades de la población fuente o de la población destino. Pero además, se provee un mecanismo para involucrar fácilmente propiedades de cualquier entidad que esté siendo simulada en el ecosistema, con el fin de no imponer limitaciones en el alcance de las reglas. Por su parte, se proveen también mecanismos para involucrar constantes de cualquiera de los tipos definidos en el ecosistema, lo cual significa que podremos establecer como valor constante en una regla tanto valores simples de tipos primitivos (enteros, reales, cadenas…) como entidades de cualquiera de los tipos del ecosistema, gracias a las posibilidades que el metamodelo presentado en el apartado 2.1 nos ofrece. Sin embargo, a pesar de esta potencia expresiva, se sigue permitiendo definir reglas más sencillas que no involucren ni condiciones ni filtrados, e incluso sin ningún tipo de procesamiento ni transformación de datos, tan solo mediante asignaciones desde los datos del antecedente a los datos del consecuente, con el fin de facilitar el aprendizaje progresivo de la herramienta y garantizar la mayor flexibilidad posible.

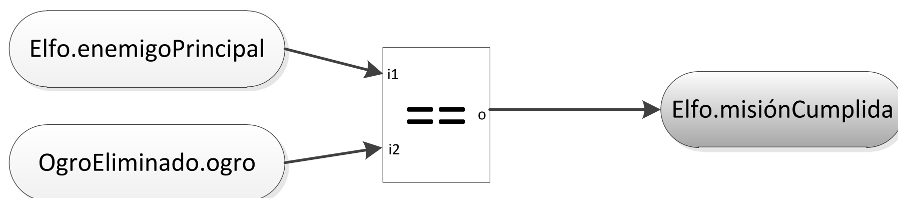
En la Figura 8 se muestran algunos ejemplos de expresiones sencillas empleando flujos de datos para ilustrar la representación visual propuesta, junto a la representación textual de dicha expresión. La expresión (a) ilustra un cálculo sencillo de la puntuación de un juego, en la cual se evita el desbordamiento asignando la máxima puntuación posible en caso de que esta sea superada. La expresión (b) muestra un ejemplo de condición para escenario definido en el

apartado 2.1 que trataba sobre la enemistad entre elfos y orcos. Suponiendo que en dicho contexto existe un evento *OgroEliminado*, que es lanzado por cualquier entidad de tipo *Elfo* que consigue derrotar a otra entidad de tipo *Orco*, podríamos establecer una regla para determinar qué elfo gana el juego (si recordamos, ganará aquel que derrote primero a su enemigo orco). En este caso, la condición de la regla establecerá que para cualquier entidad de tipo *Elfo*que lance el evento *OgroEliminado*, si el enemigo principal de dicho elfo es el orco que se indica en *OgroEliminado.ogro*, entonces el elfo ha cumplido su misión. Como puede observarse, en esta expresión se emplean poblaciones genéricas en lugar de definir una regla específica para cada entidad de tipo *Elfo*. La expresión (c) es una condición que podría emplearse en un sistema inteligente que disponga de sensores para captar la temperatura ambiente, de manera que con esta expresión lograríamos activar la calefacción de una oficina u hogar si la diferencia entre la temperatura ambiente y la última temperatura registrada difiere en más de 6 grados. Por último, la expresión (d) muestra cómo podría regularse la intensidad de luz de una bombilla en función de la luz ambiente y de la distancia entre el usuario y la bombilla.



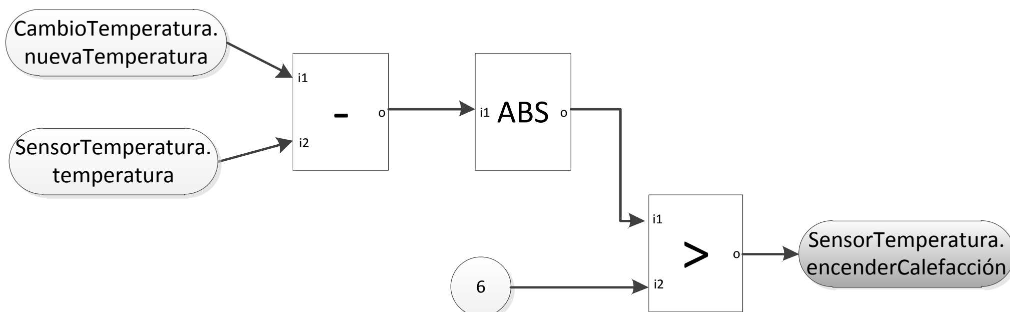
puntuación← MIN(puntuación + 1, puntuación_máx)

(a)



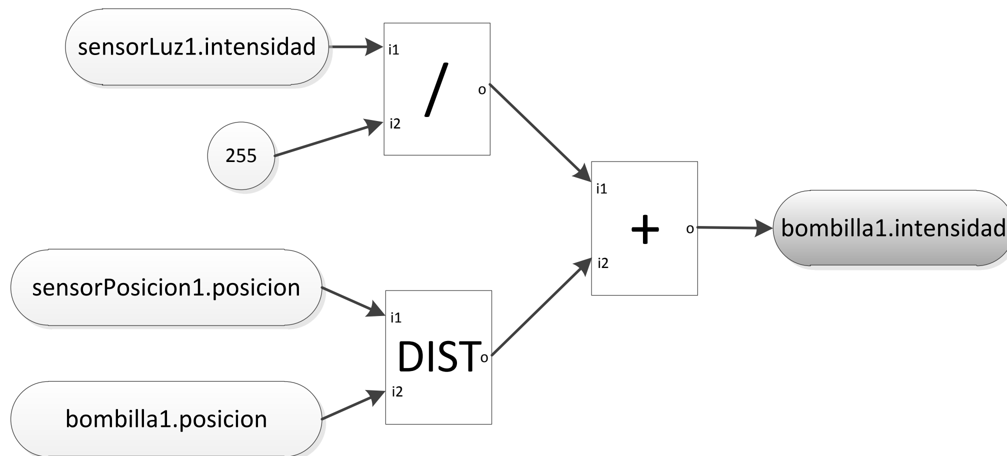
Elfo.misiónCumplida←Elfo.enemigoPrincipal ==OgroEliminado.ogro

(b)



SensorTemperatura.encenderCalefacción←ABS(CambioTemperatura.nuevaTemperatura – SensorTemperatura.temperatura) > 6

(c)



bombilla1.intensidad ← (sensorLuz1.intensidad / 255) + DIST (sensorPosicion1.posicion, bombilla1.posicion)

(d)

Figura 8. Ejemplos visuales de flujos de datos simples soportados por el editor.

El metamodelo que da soporte a los procesos de datos se muestra en la Figura 9. Para representar una expresión basada en flujos de datos el modelo sigue un enfoque tipo grafo, de modo que un proceso de datos (*DataProcess*) está formado, por un lado, por un conjunto de nodos (*DataProcessNode*) que tienen puertos de entrada y de salida (*Port*), y por otro lado, por un conjunto de flujos de datos (*DataFlow*) que permiten establecer conexiones entre los puertos de los nodos para formar expresiones.

Los nodos de un proceso de datos se especializan en diferentes tipos para representar los diversos conceptos necesarios en las expresiones que pretendemos definir. Por un lado, tendremos nodos proveedores de datos (*InputNode*), como pueden ser los proveedores de constantes (*ConstantProvider*). Estos nodos permiten incluir en el proceso de datos valores constantes de cualquiera de los tipos del ecosistema. Otros nodos proveedores de datos son aquellos nodos que permiten acceder a los atributos del evento que activó la regla (*EventAttributeDefinitionSource*), o bien los nodos que representan propiedades de las poblaciones fuente y destino de la regla (*PropertyDefinitionSource*). Un caso especial de nodo proveedor de datos es del tipo los *BoundedPropertyDefinitionSource*. Un nodo de este tipo permite referenciar una propiedad de una entidad concreta del ecosistema, que no sea ni fuente ni destino de la regla. Es necesario tener la referencia tanto de la entidad como de la propiedad deseada de dicha entidad, puesto que las propiedades de una entidad vienen determinadas por su tipo. Así, todas las entidades del mismo tipo tendrán la misma definición de propiedades, siendo únicamente posible diferenciarlas por sus valores asociados en la simulación del entorno. Para el caso de los nodos *PropertyDefinitionSource* se indica tan sólo la propiedad a la que se debe hacer referencia, puesto que el enlace con la entidad específica sobre la que se debe extraer la propiedad se hace de manera implícita al instanciar la regla para una determinada población fuente o destino.

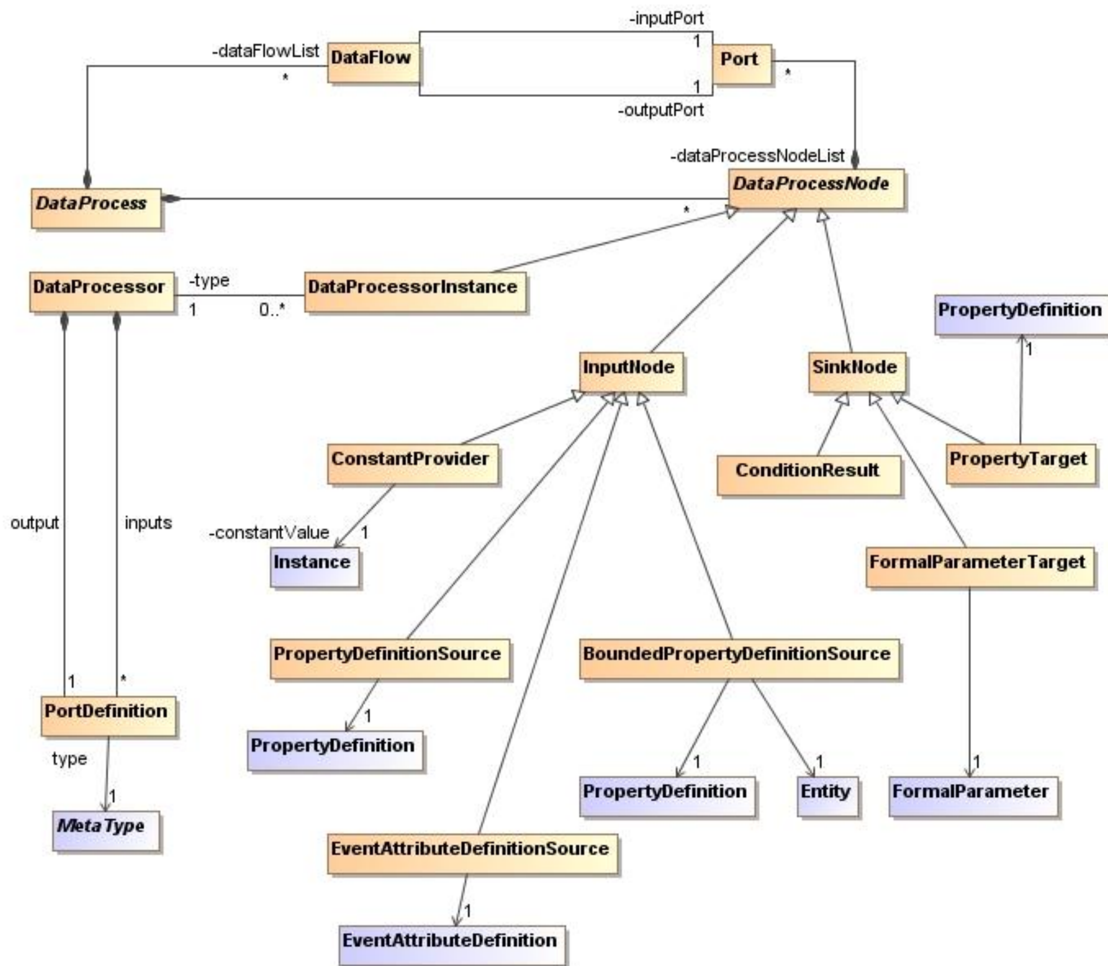


Figura 9. Diagrama de clases UML para soportar transformaciones mediante flujos de datos.

Por otra parte, existen tres tipos de nodos sumidero u objetivo. Estos nodos son sobre los que se realiza la asignación del valor calculado en la expresión del proceso de datos. Sólo puede existir un nodo de tipo sumidero en un proceso de datos, ya que sólo se puede realizar una asignación en cada proceso. Para los nodos sumidero de los procesos de datos que expresan la precondición y el filtrado será necesario un nodo que recoja el valor booleano que debe resultar de dicha expresión (*ConditionResult*). En cambio, para el cálculo del consecuente de la regla, el nodo sumidero del proceso de datos que expresa la operación a realizar podrá ser de dos tipos: *PropertyTarget* si la operación era una asignación a alguna propiedad de la población destino, o bien *FormalParameterTarget* si la operación era la invocación de una acción y se está definiendo uno de sus parámetros.

Una de las características más importantes del modelo y que le otorga gran capacidad expresiva es la presencia de operadores en los flujos de datos. Los nodos que definen procesadores dentro de un proceso de datos están representados por la clase *DataProcessorInstance*. No obstante, es frecuente tener una expresión en la cual el mismo operador se aplique varias veces con operandos distintos, de manera que el metamodelo debería soportar el hecho de tener un tipo de operador del cual obtener instancias distintas. Por ejemplo, se podría desear poner en una expresión varios operadores de tipo suma, pero cada uno de ellos sería una instancia distinta de este tipo ya que tendría entradas y salidas diferentes. Esto queda representado en el metamodelo mediante la clase *DataProcessor*, cuyas instancias van a

representar los tipos de operadores disponibles, especificando sus puertos de entrada y de salida, los tipos de valores que se espera recibir en dichos puertos (enteros, reales, etc.), y la operación que realiza dicho operador. Así, un nodo *DataProcessorInstance* representará una instancia de un tipo de operador determinado. Ampliando el repertorio de tipos de operadores disponibles (*DataProcessor*) se logra personalizar el editor para cada dominio específico de aplicación.

2.4. Evaluación sobre la comprensión de los flujos de datos

Para explorar la idoneidad de la utilización de un lenguaje de flujos de datos como el que se va a utilizar en las reglas definidas en DaFRule, se llevó a cabo un pequeño estudio preliminar acerca de la comprensibilidad del lenguaje visual de flujos de datos. El estudio involucró a alumnos de 1º de Bachiller (de 16 años de edad en promedio) que estaban cursando la asignatura de Informática. Estos alumnos precisamente iban a enfrentarse por primera vez a los conceptos de programación, por lo que los resultados darían una idea acerca de la facilidad para comprender y emplear este tipo de abstracciones y estructuras que representan conceptos computacionales.

De acuerdo al currículum de la asignatura, el profesor de la asignatura de Informática dedicó 4 sesiones a enseñar los conceptos básicos necesarios (variables, operadores, instrucciones de asignación, etc.). Para la enseñanza de estos conceptos se empleó un lenguaje textual que se corresponde con el de la gramática anteriormente presentada en el apartado 2.2, y además el lenguaje visual para flujos de datos equivalente, siguiendo una representación como en los ejemplos de la Figura 8. Ambas aproximaciones fueron impartidas dándoles la misma importancia, y sin dar preferencia a una u otra, y fueron debidamente ejemplificadas por el profesor. Así mismo, los alumnos resolvieron algunos ejercicios conjuntamente.

Los alumnos realizaron una prueba con ejercicios relacionados con el uso de los lenguajes textual y visual vistos. Tras la prueba, solamente 24 alumnos decidieron responder al cuestionario sobre preferencias de uso. En dicho cuestionario, los alumnos tuvieron que elegir, para cada cuestión, qué versión del lenguaje, textual o visual, preferían en relación a la sentencia que se les planteaba. En lugar de utilizar cuestiones con escalas Likert para valorar el grado de acuerdo y desacuerdo con la sentencia, se optó por formular las cuestiones de tal forma que tuvieran que elegir forzosamente entre textual y visual, siguiendo recomendaciones. Esta decisión fue tomada durante una reunión con los profesores, quienes sugirieron como mejor opción emplear una selección en lugar de escalas, ya que permitiría obtener una respuesta más directa de los alumnos aun en el caso de que el sujeto no tuviera su capacidad crítica muy desarrollada, como suele ocurrir en esas edades. De esta forma, se pretendió valorar si un lenguaje visual para flujos de datos parecía aportar alguna ventaja en cuanto a comprensibilidad frente al lenguaje textual, que se asemeja a las expresiones matemáticas que ya habían visto en las clases de matemáticas y que ahora estaban expandiendo con algunos conceptos computacionales.

La Tabla 1 resume los conteos del cuestionario. Es destacable que existe una notable mayor preferencia por el lenguaje visual en cuanto a la facilidad de uso, la

comprensión, el aprendizaje y la utilización para cómputo, aunque para la escritura de expresiones no muestra tanta ventaja. Por tanto, a la vista de los resultados, parece realmente prometedor el uso de flujos de datos para no-programadores, proporcionando una herramienta valiosa a tener en cuenta, y es por ello por lo que definitivamente se optó por incluirlas en las reglas como se propone en el presente trabajo. Además, se espera que la retroalimentación visual del editor de reglas realmente ayude a la escritura de expresiones.

Cuestión	**Textual**	**Visual**
Me ha resultado más fácil aprender el uso del lenguaje…	7	16
Entiendo mejor las expresiones escritas en el lenguaje…	7	15
Los diferentes elementos que componen el lenguaje visual (operadores, variables, etc.) los entiendo mejor en el lenguaje…	5	18
Aprendí más rápido el lenguaje…	6	17
Encuentro más fácil calcular los valores en el lenguaje…	4	19
Escribo más fácilmente expresiones en el lenguaje…	10	12

Tabla 1. Conteo para las cuestiones sobre preferencia de uso de los lenguajes.

3. Implementación del editor WIMP

Tras la definición del metamodelo que permite la especificación genéricamente de reglas de comportamiento, se ha desarrollado un prototipo de un editor de reglas que permita validar las ideas del modelo propuesto. Este editor ha sido desarrollado en C#, y está provisto de una interfaz tradicional basada en controles y ventanas. No obstante, se ha procurado que la interfaz del editor sea lo más intuitiva posible, ya que no se debe perder de vista que este editor está principalmente está orientado a no-programadores o no expertos que requieran especificar comportamiento en sistemas de diversa índole. Por tanto, se ha elegido como plataforma un HP TouchSmart IQ522, dotado de pantalla táctil de 22', para hacer las pruebas de concepto. Se ha decidido que la interacción con el editor sea táctil, ya que generalmente este tipo de interacción de los usuarios con el sistema empleando sus manos o dedos resulta más intuitiva y familiar que la interacción mediante el teclado y ratón tradicionales.

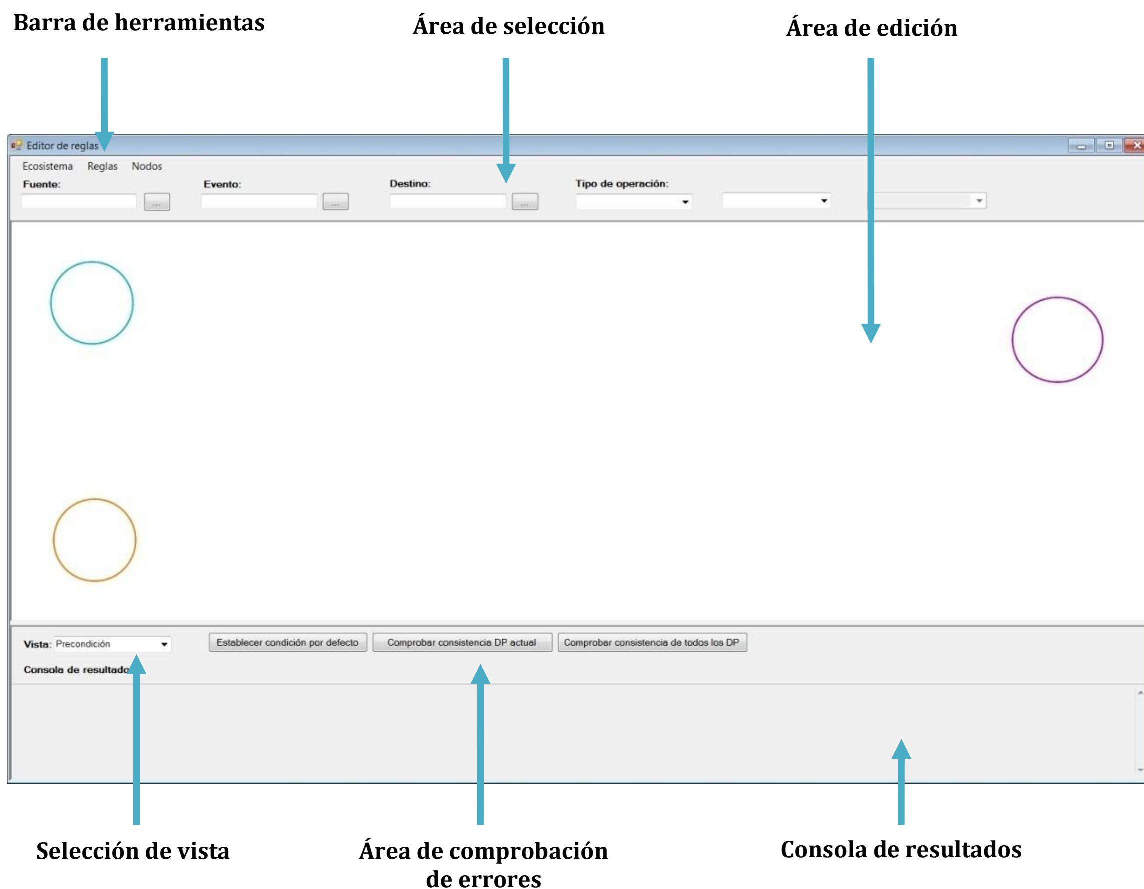


Figura 10. Interfaz del editor de reglas

En la Figura 10 se muestra la interfaz básica del editor de reglas desarrollado. La edición de una regla requiere la sistemática selección de diversos elementos, como son los eventos, entidades y operaciones involucradas, así como la edición de flujos de datos que representan las expresiones para el cálculo de las operaciones de la regla. Es por ello que la ventana principal del prototipo se divide, tal y como se aprecia en la figura, en varias áreas claramente diferenciadas. Entre estas áreas destaca el "canvas" para el área de edición de flujos de datos, el área para la selección de los elementos involucrados en la regla y el área para la comprobación

de errores de edición. Se dispone también de una barra de herramientas que permite diversas acciones, una lista desplegable para alternar entre las diversas vistas de edición de una regla y una consola en la parte inferior del editor que informa del resultado de cualquier acción.

3.1. Área de selección de elementos de la regla

En primer lugar, los elementos que se deben seleccionar para conformar la estructura básica de una regla son: 1) El evento que activa la regla; 2) la población fuente; 3) la población destino; 4) la operación a realizar; 4.a) la propiedad de la población destino a modificar, en caso de que la operación sea una asignación a propiedad; 4.b) la acción de la población destino a llevar a cabo, en caso de que la operación sea la ejecución de una acción.

Estos elementos deben poder seleccionarse o modificarse en cualquier punto de la edición de una regla (bien sea porque se seleccionan según se van necesitando, o bien porque llegados a un punto se ha cometido un error y se debe seleccionar una nueva población, un nuevo evento, etc.). La única restricción que se impone a la selección de estos elementos es la que aparece de manera natural: no se puede seleccionar una propiedad o acción, sin antes haber seleccionado la población destino y el tipo de operación a llevar a cabo.

Por otra parte, cada vez que se seleccione o modifique uno de estos elementos, la regla que está siendo editada debe remodelarse en consecuencia, eliminando cualquier rastro de los elementos modificados, o bien añadiendo los nuevos elementos, e incluso eliminando cualquier flujo de datos en el que estuviera implicado un elemento antiguo.

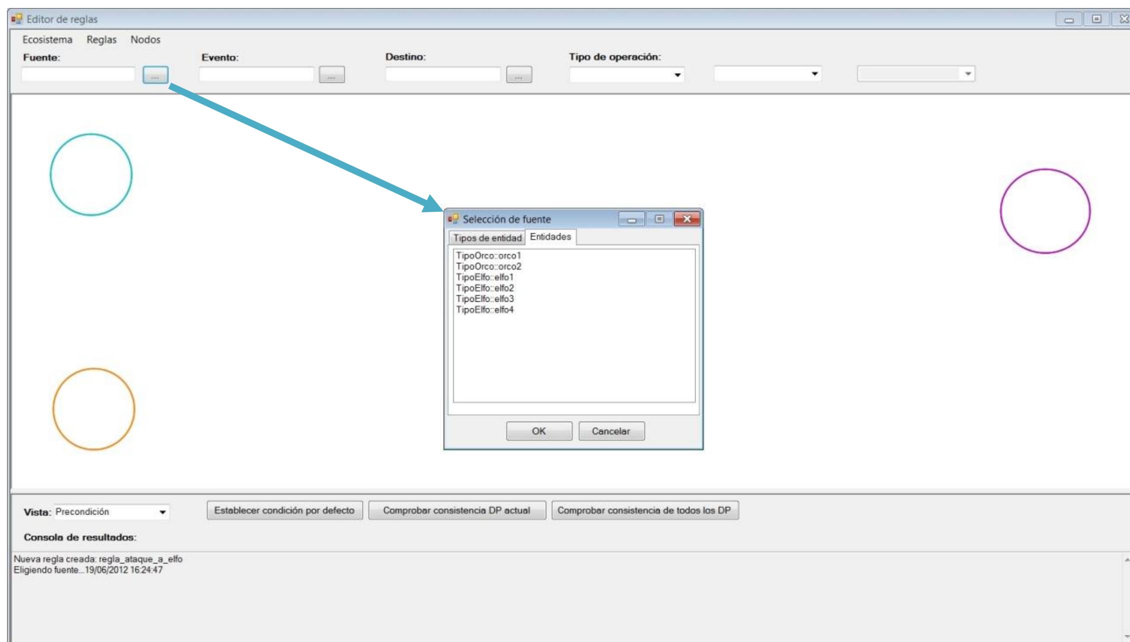


Figura 11. Exploración de la colección de entidades para seleccionar la población fuente de la regla.

Para llevar a cabo esta selección se debe llevar a cabo una exploración de colecciones. En la barra de herramientas de la parte superior del área de edición se dispone, para cada elemento, de una etiqueta que lo identifica, una caja de texto donde aparece el elemento seleccionado (si lo hay), y un botón que despliega la

colección de elementos correspondiente para realizar la selección. En la Figura 11 se puede observar un ejemplo de exploración de colecciones, donde se está explorando el conjunto de entidades del ecosistema para seleccionar la población fuente.

3.2. Selección de vista

Por razones de usabilidad sólo se permite editar un único proceso de datos a la vez, por lo que es necesario cambiar la vista de edición seleccionando el proceso de datos deseado en un menú desplegable. Por tanto, la regla completa no se muestra mientras se está editando, sino que se compone de un conjunto de vistas en función de los procesos de datos que la forman. Se requieren como mínimo dos vistas, una para la precondición y otra para la condición de filtrado. En función de la operación a llevar a cabo, se dispone de más o menos vistas: en el caso de una asignación a propiedad, se dispone de una vista más para editar este proceso de datos, mientras que en el caso de la ejecución de una acción, se dispone de tantas vistas adicionales como parámetros tenga la acción seleccionada. De este modo, solo la fuente, el evento y el destino se muestran en el área de edición de todas las vistas, ya que son elementos comunes a todos los procesos de datos. El resto de elementos que se muestran en el área de edición de cada vista son particulares de cada proceso de datos (procesadores, constantes, flujos de datos, etc.). En la Figura 12 se muestra el desplegable para seleccionar la vista de la regla a editar.

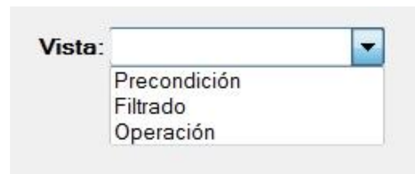


Figura 12. Desplegable para la selección de la vista.

3.3. Área de edición

El área de edición es la zona donde se definen los distintos procesos de datos que conforman la regla. El primer punto a resolver es cómo representar los conceptos que involucra una regla para que la edición sea intuitiva. Para ello, la distribución espacial de los elementos en el área de edición juega un papel fundamental a la hora de establecer un orden en la definición de un proceso de datos, a la vez que permite separar de manera visual los diferentes conceptos que involucra la regla. Por tanto, se ha optado por mostrar una disposición acorde al sentido natural de lectura que típicamente sigue el sentido izquierda a derecha.

Como se ha comentado al tratar el tema de las vistas, los elementos comunes a todos los procesos de datos involucrados en una regla son la población fuente y el evento, y por otro lado la población destino. Estos elementos se representan en el área de edición mediante zonas circulares coloreadas de forma específica. Para facilitar la edición, al seleccionar la población fuente y/o el evento, en el área de edición se muestran los elementos seleccionados, así como los nodos que representan las propiedades de la fuente y los atributos del evento respectivamente, junto a las áreas circulares que les corresponden.

La fuente y el evento se mostrarán a la izquierda del área de edición. Esto se debe a que las propiedades de la fuente y los atributos del evento son nodos proveedores de datos, y por tanto lo más probable es que se empleen a lo largo del proceso de datos para realizar operaciones sobre ellos a fin de obtener el valor a asignar al nodo sumidero u objetivo. Mientras tanto, este nodo objetivo del proceso de datos, así como la población destino, se mostrarán a la derecha del área de edición, ya que el resultado de todas las operaciones que se definan debe confluir en una asignación a este nodo objetivo, y resulta más lógico situarlo a la derecha indicando que el proceso de datos termina en ese punto. Esto ayudará a mantener cierto orden en la disposición de los elementos atendiendo a la característica cultural occidental en la que predomina la lectura de izquierda a derecha.

Por otra parte, en un proceso de datos también intervienen operadores para la transformación de los datos. Estos operadores se representan como cajas con entradas y salidas, y se deben ir situando a lo largo del área de edición, entre los nodos proveedores de datos y el nodo objetivo, de modo que la secuencia de operaciones aplicadas a los nodos fuente se produce de izquierda a derecha, confluyendo en el nodo objetivo. Así se ofrece una visión clara del orden en el que se aplican estas operaciones, y se pueden detectar errores en dicho orden de aplicación más fácilmente. Además, se permite mover los operadores por el área de edición, situándolos donde el usuario desee, para facilitar una mejor visión de los flujos de datos.

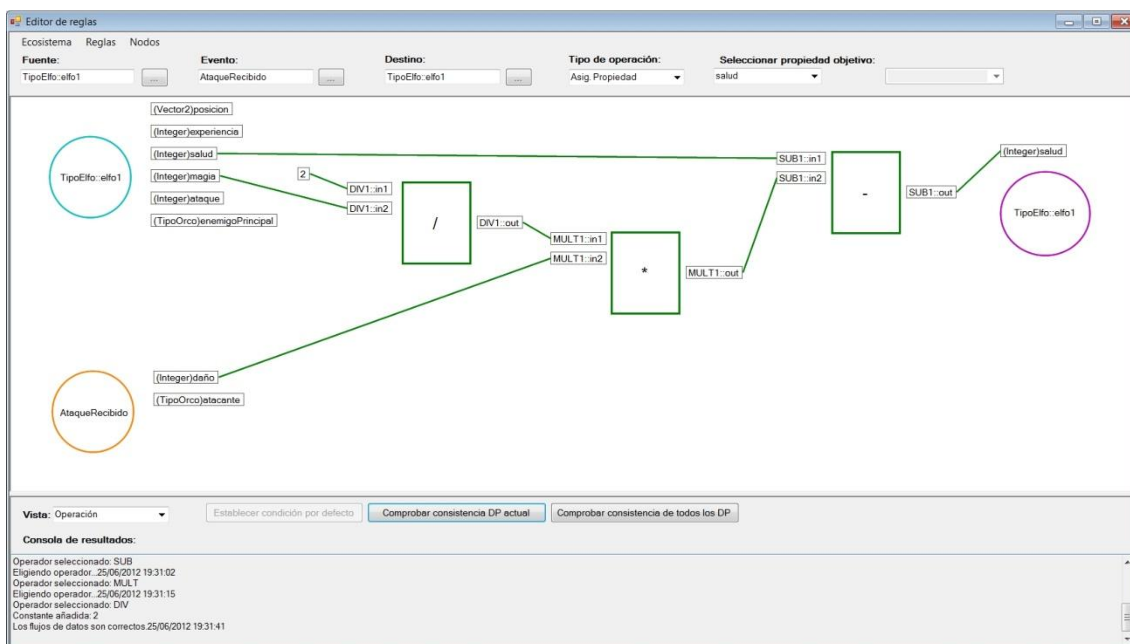


Figura 13. Ejemplo de proceso de datos correctamente editado.

Todos los puertos de salida de datos presentes en el área de edición (propiedades de la fuente, atributos del evento, valores constantes, o salidas de los operadores) se pueden unir mediante los dedos a los puertos de entrada (entradas de los operadores, o nodos objetivo de la regla) para crear los flujos de datos que definen cómo se realiza el cálculo del dato objetivo de la operación o de la correspondiente condición. De no ser correcta la especificación de la regla, se pueden borrar y reeditar flujos de datos según se desee. Se considera que una regla está correctamente definida cuando todos los procesos de datos, tanto de la operación como de las condiciones, son sintácticamente correctos. En la Figura 13

se muestra un proceso de datos correctamente editado. Este proceso de datos corresponde a la operación de asignación de un valor a una propiedad de la población destino, en el contexto de un juego de rol en el que los personajes pierden salud al recibir ataques. La expresión equivalente sería *elfo1.salud ← elfo1.salud – ((2 / elfo1.magia) * AtaqueRecibido.daño)*. Se puede observar que se han empleado tres procesadores de datos para realizar las operaciones correspondientes, y que dichas operaciones involucran un valor constante, dos propiedades de la población fuente, y un atributo del evento.

3.4. Barra de herramientas

En el área superior del editor de reglas se dispone de una barra de herramientas que ofrece diversas acciones. Por un lado, se permite crear nuevas reglas, eliminar reglas ya existentes, o seleccionar una de las reglas existentes para su edición.

Por otra parte, se ha definido una gramática en ANTLR [24] que permite guardar las reglas correctamente editadas con la herramienta en ficheros de texto, así como cargar reglas en el editor a través de la especificación textual de dichas reglas en un fichero. La gramática empleada para la especificación textual de reglas se define en el Anexo A. La posibilidad de almacenar las reglas editadas permite que estas reglas estén disponibles para otras herramientas como la que se explica en el capítulo 1.

En último lugar, la barra de herramientas permite añadir y borrar operadores y constantes al área de edición, así como propiedades de cualquiera de las entidades del ecosistema. Para añadir una instancia de un tipo de operador al proceso de datos que se encuentre activo en un determinado momento, se debe realizar una exploración de la colección de operadores que se han definido en la herramienta. Tras seleccionar el operador deseado, se debe situar la instancia de este operador en el lugar deseado dentro del área de edición. La Figura 14 muestra la exploración de la colección de operadores disponibles. En el caso de las constantes, se ha visto en el apartado 2.3 que se pueden añadir nodos al proceso de datos que contengan instancias de cualquiera de los tipos definidos en el ecosistema. Los tipos definidos en el ecosistema son, por un lado, los tipos de entidades, y por otro lado, los tipos primitivos de datos (enteros, reales, vectores, booleanos, etc.). Por tanto, al añadir una constante al área de edición, se debe seleccionar qué tipo de valor representa esta constante, así como indicar el valor que tomará en simulación. Por último, se debe situar el nodo proveedor de constante en el área de edición, al igual que se hace con los operadores. La Figura 15 muestra el menú que aparece tras indicar que se desea añadir un nodo constante. En este menú se determina qué valor constante se quiere añadir, pudiéndose elegir entre las distintas entidades del ecosistema, o bien indicar un valor de tipo primitivo (entero, real, booleano, etc). En el caso de querer añadir una propiedad de una entidad cualquiera del ecosistema para emplearla como nodo proveedor de datos en un proceso de datos, se debe seleccionar en primer lugar la entidad deseada, y en segundo lugar, la propiedad de dicha entidad que se desea añadir. Finalmente, al igual que con los operadores y las constantes, se debe añadir este nodo al área de edición para su posterior uso. En la Figura 16 se muestra la ventana emergente que permite seleccionar este tipo de nodos.

Figura 14. Selección de un procesador de datos.

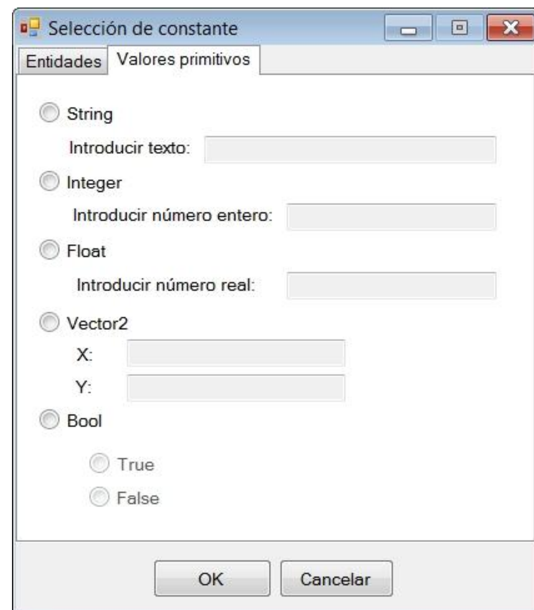


Figura 15. Selección de valor a asignar a un nodo proveedor de constantes

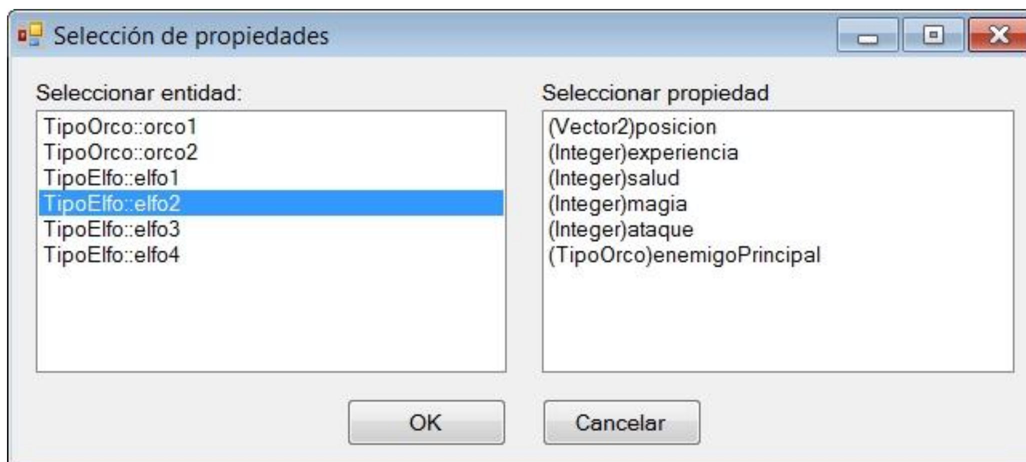


Figura 16. Selección de una propiedad de cualquier entidad del ecosistema.

3.5. Área de comprobación de errores

Para facilitar la localización de los errores cometidos en la edición, se dispone de dos mecanismos de comprobación de errores. Uno de ellos se realiza on-line, de forma que cada vez que se edita un flujo de datos desde un puerto de entrada a otro puerto de salida, se realiza la comprobación entre los tipos de los puertos. Si dichos tipos son compatibles, la línea que representa el flujo de datos aparece marcada con color verde, representando una correcta edición, mientras que si los tipos de datos de los puertos no coinciden o no son compatibles, la línea se muestra en rojo para indicar al usuario que ha cometido un error al establecer las conexiones.

Por otra parte, se dispone de un mecanismo de comprobación de errores off-line, que se encarga, cuando el usuario pulsa el botón correspondiente, de realizar un

chequeo completo o bien del proceso de datos actual, o bien de toda la regla. Las comprobaciones que se realizan son las siguientes:

a) El nodo objetivo del proceso de datos tiene asignado un flujo de datos de entrada.
b) Todos los operadores presentes en un proceso de datos tienen flujos de datos asociados a sus puertos, o lo que es lo mismo, no pueden haber operadores que no se encuentren bien definidos.
c) No existen ciclos, es decir, no hay ningún nodo p en el proceso de datos cuyo puerto de salida tenga algún flujo de datos conectado a algún puerto de entrada de otro nodo q, y q es proveedor de datos de p.

Para indicar visualmente que se han detectado errores en la comprobación b, los operadores correctamente editados se muestran en color verde, mientras que los operadores que no estén correctamente editados se muestran en color rojo.

Así, mediante códigos de colores se puede mostrar al usuario aquellos flujos de datos en los que los tipos no corresponden, o procesos de datos incompletos, con el objetivo de facilitar la edición y reducir la tasa de errores en la definición a medida que el usuario progresa en su edición. La Figura 17 muestra un ejemplo de regla con errores sintácticos: hay dos procesadores de datos a los que todavía se les deben asignar flujos de datos tanto de entrada como de salida; el nodo objetivo de la regla todavía no tiene asignado ningún flujo de datos de entrada; existe un flujo de datos incorrecto, ya que trata de conectar un valor de tipo vectorial con la entrada de un procesador de datos que opera con tipos enteros o reales, no con vectores.

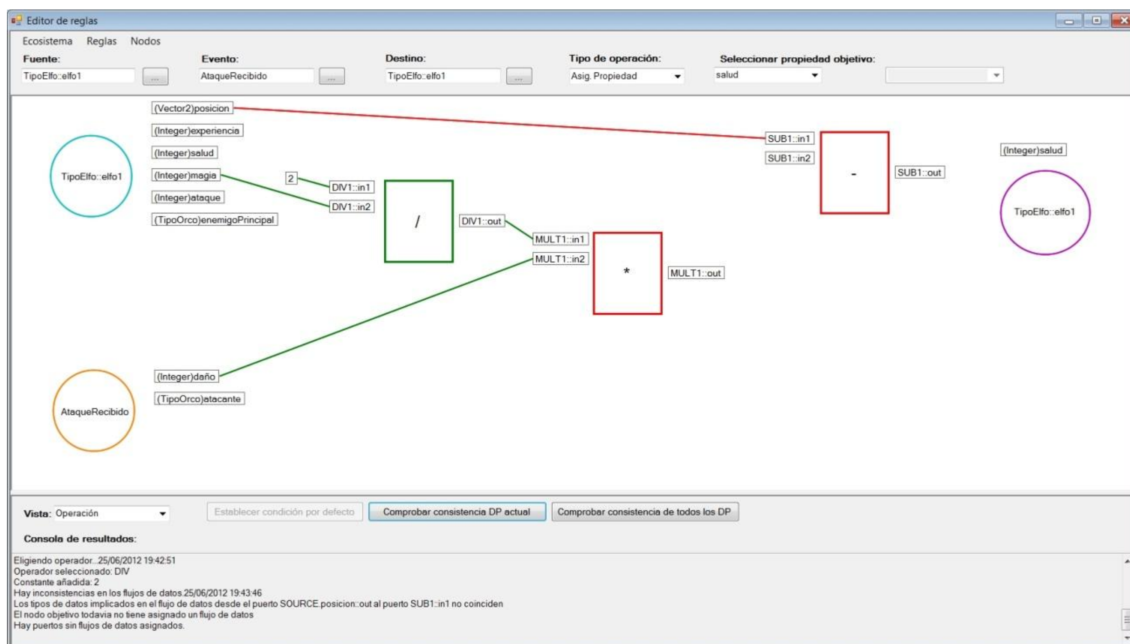


Figura 17. Ejemplo de proceso de datos con flujos de datos incorrectos, y procesadores de datos todavía por completar.

En cuanto a la comprobación c, la búsqueda de ciclos se realiza empleando el algoritmo DFS [25], que se explica a continuación.

3.5.1. Algoritmo DFS[1] adaptado

El algoritmo *DFS* (*Depth-FirstSearch*) es un algoritmo que realiza el recorrido en profundidad de un grafo. Dado un grafo $G = (V, A)$, donde V es el conjunto de vértices del grafo y A el conjunto de aristas, y dado un vértice v desde donde comenzar la búsqueda, este algoritmo explora sistemáticamente las aristas de G visitando primero los vértices adyacentes a los visitados más recientemente. De este modo, se va profundizando en el grafo, alejándose progresivamente de v.

Durante el recorrido del grafo, los vértices se van coloreando para indicar su estado. Un vértice blanco indica que todavía no ha sido explorado, un vértice gris indica que el vértice está siendo explorado, y un vértice negro indica que la exploración de este vértice y de todas las aristas que salen de él ha terminado.

A lo largo del recorrido también se realiza una clasificación de las aristas del grafo. Considerando un grafo dirigido, existen cuatro tipos de aristas:

- **Aristas del árbol (*tree edges*):** una arista *(u,v)* es de este tipo si el vértice *v* ha sido descubierto por primera vez a través de esta arista.
- **Aristas hacia atrás (*back edges*):** una arista *(u,v)* es una arista hacia atrás si el vértice *u* es descendiente del vértice *v*, es decir, si también existe una arista dirigida *(v,u)* que haya sido explorada con anterioridad.
- **Aristas hacia adelante (*forward edges*):** una arista *(u,v)* es una arista hacia adelante si el vértice *v* es descendiente del vértice *u*, pero ya había sido descubierto en algún instante de tiempo anterior.
- **Aristas de cruce (*cross edges*):** el resto de aristas, representando vértices que no tiene por qué ser descendientes unos de otros.

Una arista *(u,v)* puede clasificarse en uno de los cuatro tipos de aristas anteriores atendiendo al color del vértice *v* en el momento en el que explora la arista:

- Si el vértice *v* es blanco, se trata de una arista del árbol.
- Si el vértice *v* es gris, se trata de una arista hacia atrás.
- Si el vértice *v* es negro, se trata de una arista hacia adelante o de una arista de cruce.

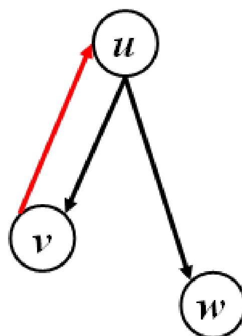


Figura 18. Grafo dirigido con una arista hacia atrás.

[1] El algoritmo DFS completo hace uso de estructuras para almacenar los instantes de tiempo en los que comienza y finaliza la exploración de cada vértice, además de almacenar para cada vértice del grafo a través de qué vértice se ha descubierto. Este tipo de información no es necesaria para el caso que se expone, por lo que no se han implementado estas características del algoritmo.

Según el algoritmo DFS, un grafo dirigido G contendrá un ciclo si se encuentra alguna arista hacia atrás. Por ejemplo, si tenemos un grafo como el de la Figura 18, al comenzar a explorar el vértice *u*, se marcaría este vértice en gris. Al pasar a explorar sus vértices adyacentes no visitados, se encontraría el vértice *v*, y se aplicaría DFS sobre dicho vértice. Al hacerlo, se marcaría *v* en gris, y se encontraría como vértice adyacente el vértice *u*, pero éste estaría marcado ya en gris, y por tanto la arista *(v,u)* sería una arista hacia atrás, que indicaría que hay un ciclo.

Para aplicar el algoritmo DFS a los procesos de datos de una regla, es necesario realizar algunas abstracciones. En primer lugar, los vértices del grafo serán los nodos que intervienen en el proceso de datos. Estos nodos están conectados entre sí mediante flujos de datos que conectan las salidas de unos nodos con las entradas de otros. En consecuencia, cada flujo de datos desde un nodo *u* a un nodo *v*, siendo el nodo *u* quien produce el dato, y el nodo *v* quien lo consume, se traduce en una arista dirigida *(v,u)*. Finalmente, lo que se obtiene al realizar esta abstracción es un grafo dirigido con forma de árbol, cuya raíz es el nodo objetivo del proceso de datos. A partir de este nodo raíz es desde donde se comienza a aplicar el algoritmo DFS adaptado, cuyo pseudocódigo se muestra en la Figura 19.

```
01: DataProcess::DFS() returns bool
02: BEGIN
03:    FORALL node: DataProcessNode IN dataProcessNodeList
04:       visited.Add(node, WHITE);
05:
06:    RETURN DFS_visit(targetDataProcessNode);
07: END
08:
09:
10: DataProcess::DFS_visit(node : DataProcessNode) returns bool
11: BEGIN
12:    visited[node] <- GRAY;
13:
14:    FORALL previousNode IN node.GetPreviousNodes()
15:       IF (visited[previousNode] == GRAY)
17:          return true;
18:       ELSEIF (visited[previousNode] == WHITE)
19:          cycleDetected <- DFS_visit(previousNode)
20:          IF (cycleDetected)
21:             return true;
22:          ENDIF
23:       ENDIF
24:    ENDFOR
25:
26:    visited[node] <- BLACK;
27:
28:    return false;
20: END
```

Figura 19. Pseudocódigo del algoritmo DFS adaptado para la detección de ciclos en un proceso de datos.

En las líneas 3 y 4 se establecen todos los nodos como no visitados, poniendo su color a blanco. En la línea 6 se realiza la invocación al algoritmo recursivo DFS desde el nodo objetivo del proceso de datos. El algoritmo DFS original realiza una invocación de la función *DFS_visit* para cada uno de los nodos del grafo, en caso de que dicho nodo no haya sido explorado todavía. Esto es debido a que no se conocen a priori las componentes conexas del grafo, y puede que haya más de una. Pero

para un proceso de datos correcto, se sabe que la estructura del proceso de datos tiene forma arbórea, y por tanto una única componente conexa, de modo que se puede llevar a cabo una simplificación, invocando la función recursiva *DFS_visit* una única vez desde la raíz de proceso de datos, que es el nodo objetivo. Por ello, la complejidad de este algoritmo DFS adaptado es *O(|DF|)*, siendo DF el número de flujos de datos dentro del proceso.

Cuando se invoca *DFS_visit* sobre un nodo, lo primero que hace esta función es indicar que el nodo está siendo explorado, poniendo su color a gris (línea 12). Posteriormente, se realiza un recorrido de todos los flujos de datos que conectan el nodo en cuestión con nodos anteriores, lo que equivale a la exploración de vértices adyacentes. Para cada nodo de este conjunto, se comprueba su estado o color. En caso de que se encuentre un nodo gris, se trata de una arista hacia atrás, y por tanto se ha detectado un ciclo (líneas 15 y 16). Si se trata de un nodo blanco (líneas 18 a 23) se trata de un nodo no visitado, y por tanto se invoca la función *DFS_visit* sobre dicho nodo de manera recursiva para realizar su exploración. A continuación, se espera a que la nueva invocación de *DFS_visit* o bien encuentre un ciclo, retornando inmediatamente, o bien alcance un caso base sin haber encontrado ciclos, y por tanto el algoritmo continuaría. Los casos base del algoritmo DFS original se dan cuando en la exploración de un nodo, éste no tiene aristas adyacentes, o bien todas sus aristas adyacentes están visitadas. En la adaptación del algoritmo DFS al caso de los procesos de datos, un caso base implicaría que se ha encontrado un nodo fuente, a saber, nodos de tipo *ConstantProvider*, *EventAttributeDefinitionSource* o *PropertyDefinitionSource*, o bien que se ha encontrado un nodo para el cual los nodos que le proveen los datos ya han sido explorados.

El algoritmo DFS termina devolviendo un valor booleano que indica si se ha detectado un ciclo en el grafo o no. En caso de que el algoritmo devuelva un valor booleano *true*, se está indicando que se ha detectado algún ciclo, por lo que en la consola de errores se reportará dicho problema. Sin embargo, el algoritmo implementado no da detalles al usuario sobre qué ciclo es el que se ha encontrado, de modo que una posible mejora a considerar sería extender esta implementación para ofrecer al usuario información más detallada sobre los ciclos detectados. Hay versiones del algoritmo DFS que emplean estructuras de datos para almacenar información asociada al resultado de la exploración de cada nodo, de modo que se pueda detectar posteriormente qué aristas en concreto son las que provocan el ciclo.

3.6. Consola de resultados

Mediante una consola se reportan tanto las selecciones o ediciones realizadas, como los errores encontrados en la definición de la regla referentes a los tipos de los elementos involucrados o a las conexiones establecidas. De este modo se reportan aquellas acciones que no tienen un *feedback* visual claro, se enfatizan las que sí lo tienen, y además se ofrece información más concreta acerca de los motivos de los errores sintácticos detectados.

3.7. Ejemplo completo de edición de una regla

En este apartado se muestra un ejemplo completo paso a paso de la edición de una regla. La regla a editar es la correspondiente a la Figura 6 (d). En primer lugar, se selecciona la opción de añadir una nueva regla al ecosistema, dándole el nombre adecuado, como se muestra en la Figura 20.

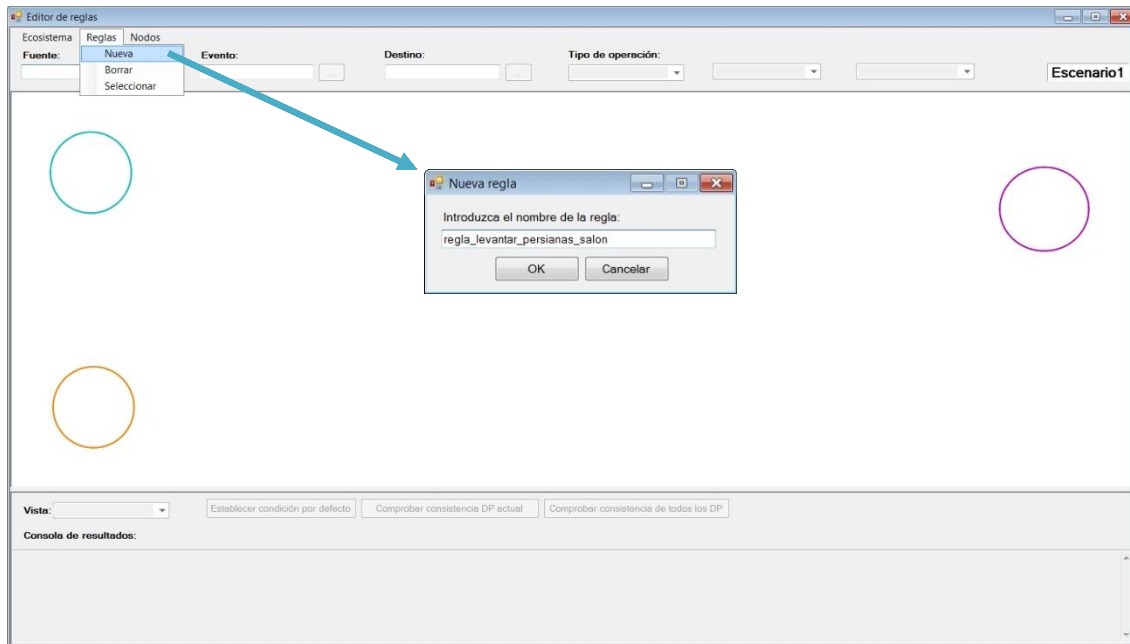


Figura 20. Creación de una nueva regla.

A continuación, se seleccionan la población fuente, el evento, y la población destino de la regla, explorando las distintas colecciones asociadas a estos elementos, tal y como se indicaba en la Figura 11. Una vez se ha definido la población fuente como *TipoPersona*, el evento como *UbicaciónCambiada*, y la población destino como *TipoPersiana*, se procede a la edición de la precondición. En la regla que se está definiendo, la expresión de la precondición establece que la regla sólo se activará si el atributo *destino* del evento *UbicaciónCambiada* es "*Salón*", es decir, si la persona que ha lanzado la ocurrencia de evento de tipo *UbicaciónCambiada* ahora se encuentra en el salón. Para editar esta expresión, se precisa añadir al área de edición una constante representando la cadena "*Salón*". En la Figura 21 se muestra la interacción para añadir dicha constante. Además, se debe añadir también al área de edición un operador de igualdad que compare el atributo *UbicaciónCambiada.destino* con la constante "*Salón*", como se muestra en la Figura 22. El resultado de esta comparación es un valor booleano que determina si la precondición se satisface o no. Dicho valor booleano debe ser asignado al nodo objetivo de la precondición, que en esta ocasión no es se representa de manera explícita. En todas las vistas se puede observar que a la derecha del área de edición aparece una circunferencia coloreada. En las vistas de precondición y filtrado, dicha circunferencia permanece vacía, representando el nodo objetivo de las condiciones. En cambio, en la vista de operación, el nodo objetivo de cada proceso de datos se representa explícitamente, y esta circunferencia se emplea para mostrar la población destino o la acción a ejecutar. De este modo, en la precondición que se está editando, la salida del operador de igualdad debe

conectarse con la circunferencia que representa el nodo objetivo de la precondición, tal y como se muestra en la Figura 23.

Cabe destacar que, en lugar de una cadena de texto, la ubicación podría haberse definido como un tipo de entidad en el ecosistema, por ejemplo *TipoUbicación*, y para este tipo se podrían haber definido distintas instancias representando las distintas ubicaciones posibles, como por ejemplo el *Salón*. Así, en lugar de comparar cadenas de texto, se compararían entidades entre sí.

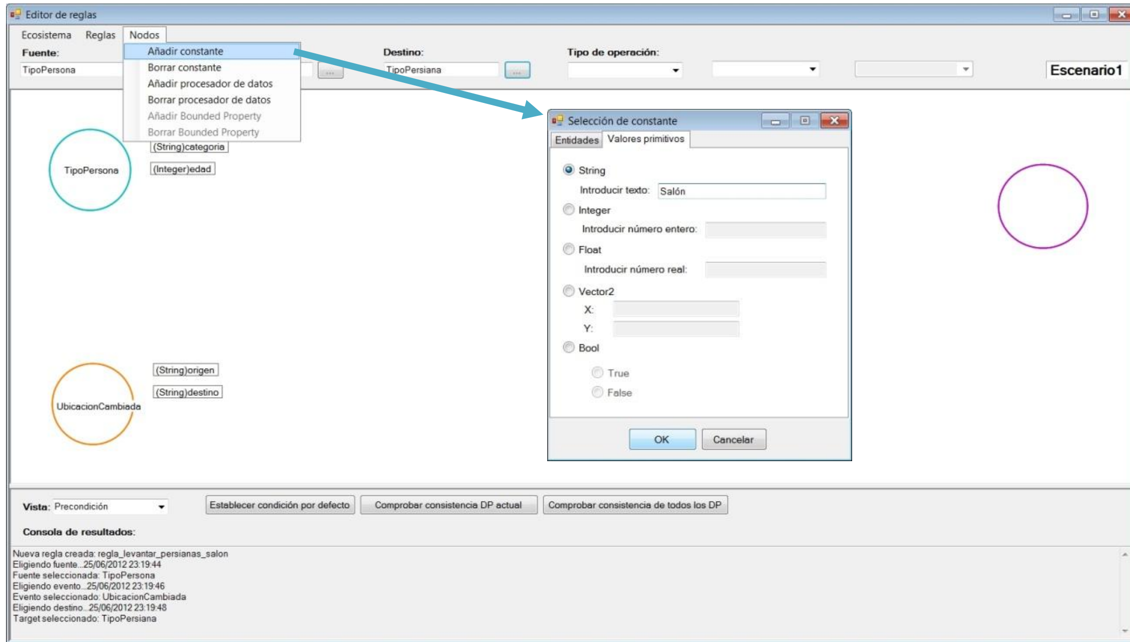


Figura 21. Interacción para añadir al área de edición un valor constante.

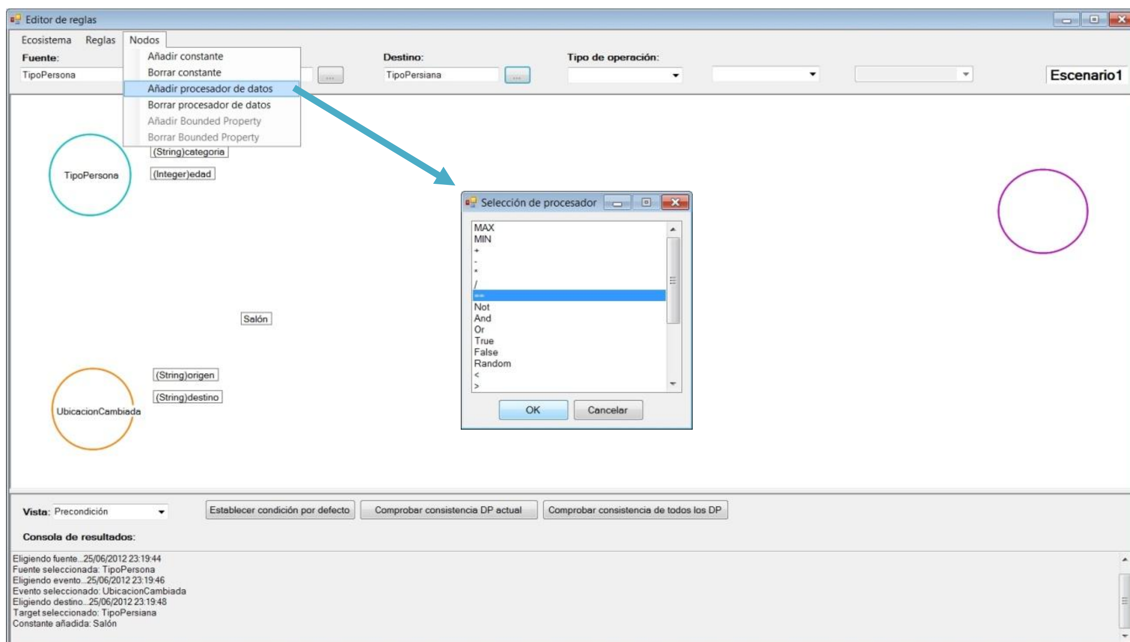


Figura 22. Interacción para añadir al área de edición un operador.

Una vez se editada la expresión de la precondición, se comprueba que no hayan errores en el proceso de datos comprobando la consola de resultados por un lado, y los colores tanto de los flujos de datos como de los operadores. La Figura 23

muestra que no se ha encontrado ningún error, y por tanto se continúa con la edición de la condición de filtrado, cambiando a la vista correspondiente.

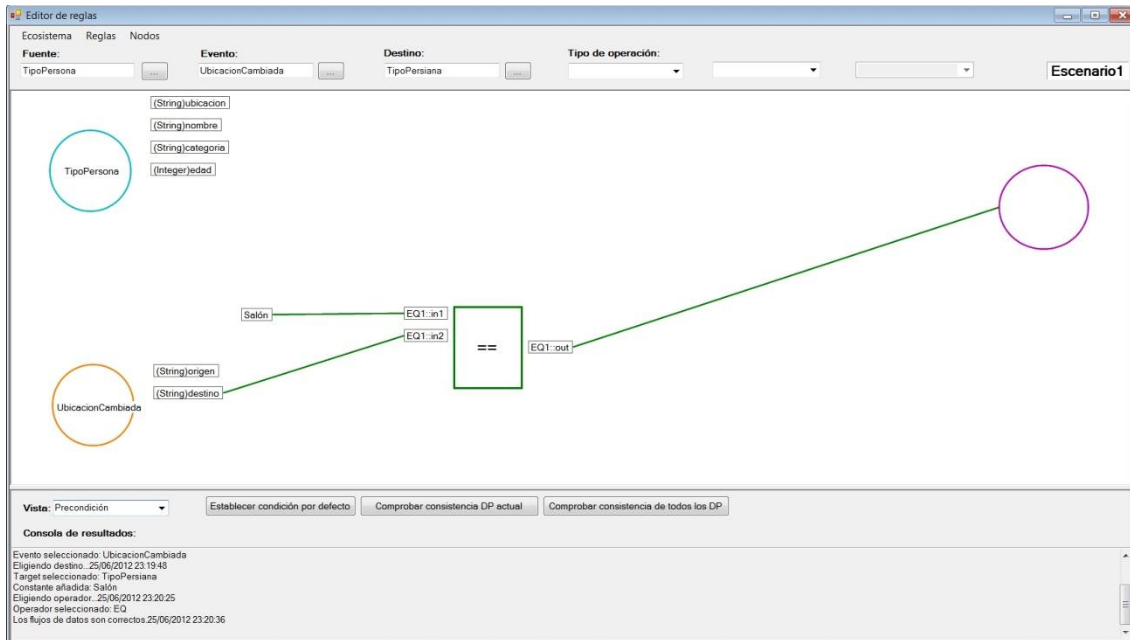


Figura 23. Edición de la precondición.

Para la vista de la condición de filtrado, el editor ofrece el conjunto de propiedades de la población destino como nodos proveedores de datos, ya que son las propiedades que deberían usarse para determinar si la operación se aplica sobre una entidad o no. En la Figura 24 se muestra que estas propiedades quedan situadas a la izquierda de la pantalla, junto a las propiedades de la población fuente y a los atributos de evento.

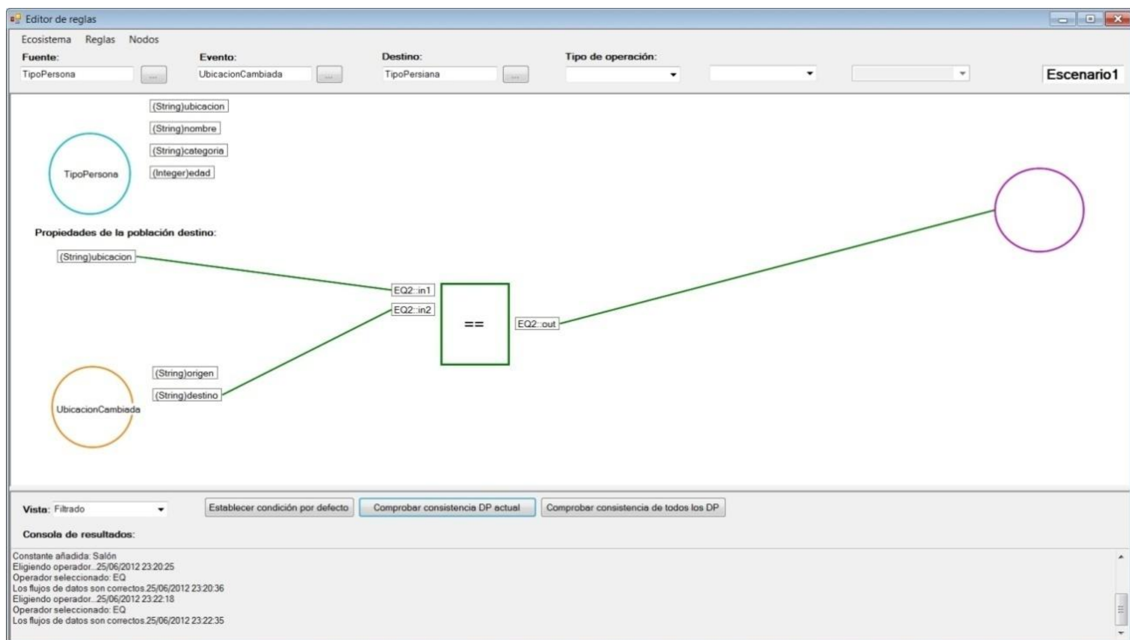


Figura 24. Edición de la condición de filtrado.

La expresión de la condición de filtrado de la regla en cuestión debe comparar el atributo *destino* del evento *UbicaciónCambiada*, con la propiedad *ubicación* de la

población destino. Para ello se debe añadir un operador de igualdad al área de edición, y asignar los flujos de datos como se indica en la Figura 24.

Tras la comprobación de errores sin que se detecte ningún problema, se procede a la edición de la operación a ejecutar. Para definir la operación a llevar a cabo, primero se debe seleccionar el tipo de esta operación. En esta ocasión lo que se pretende es realizar la ejecución de una acción, y por tanto, se debe seleccionar la acción deseada. Al seleccionar una acción, se muestra un desplegable con todos los parámetros de dicha acción, y se debe definir un proceso de datos para cada uno de estos parámetros. En el caso de la regla que se define en este ejemplo, la acción *Levantar* tan sólo dispone de un único parámetro, por lo que únicamente debe editarse un proceso de datos más.

La expresión a definir es *altura* ← *5 * sensorLuzSalón.luminosidad / 20*. Como se puede observar, en esta expresión se hace uso de una propiedad de la entidad *sensorLuzSalón*. Se debe añadir dicha propiedad al área de edición para poder usarla en la expresión. La interacción para ello se muestra en la Figura 25, donde se debe seleccionar el primer lugar la entidad deseada, y posteriormente la propiedad de dicha entidad que se quiere añadir al proceso de datos.

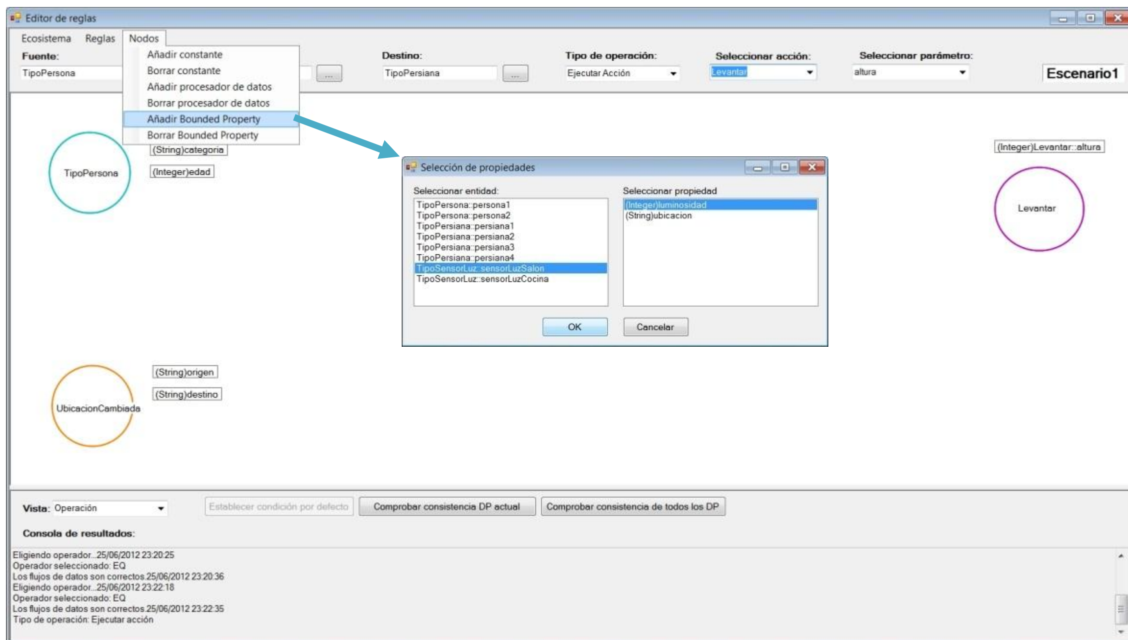


Figura 25. Interacción para añadir a un proceso de datos una propiedad de cualquier entidad del ecosistema.

Además de la propiedad *sensorLuzSalón.luminosidad*, se requieren dos valores constantes para completar las operaciones de la expresión. En la Figura 26 se observa la edición del proceso de datos del parámetro *altura* de la operación *Levantar*. En ese instante de la edición, únicamente se ha añadido al proceso de datos la constante entera 20, la propiedad *sensorLuzSalón.luminosidad*, y el operador divisor al que se le han provisto sus operandos. En la Figura 27 se muestra la expresión completa correctamente editada. Se ha precisado añadir una constante entera más para representar el valor 5 y un operador de multiplicación, y se han editado los flujos de datos conectando correctamente estas operaciones entre sí para conformar la expresión deseada. El resultado de la expresión se ha asignado al nodo objetivo, que en este caso es el parámetro *altura* de la operación *Levantar*.

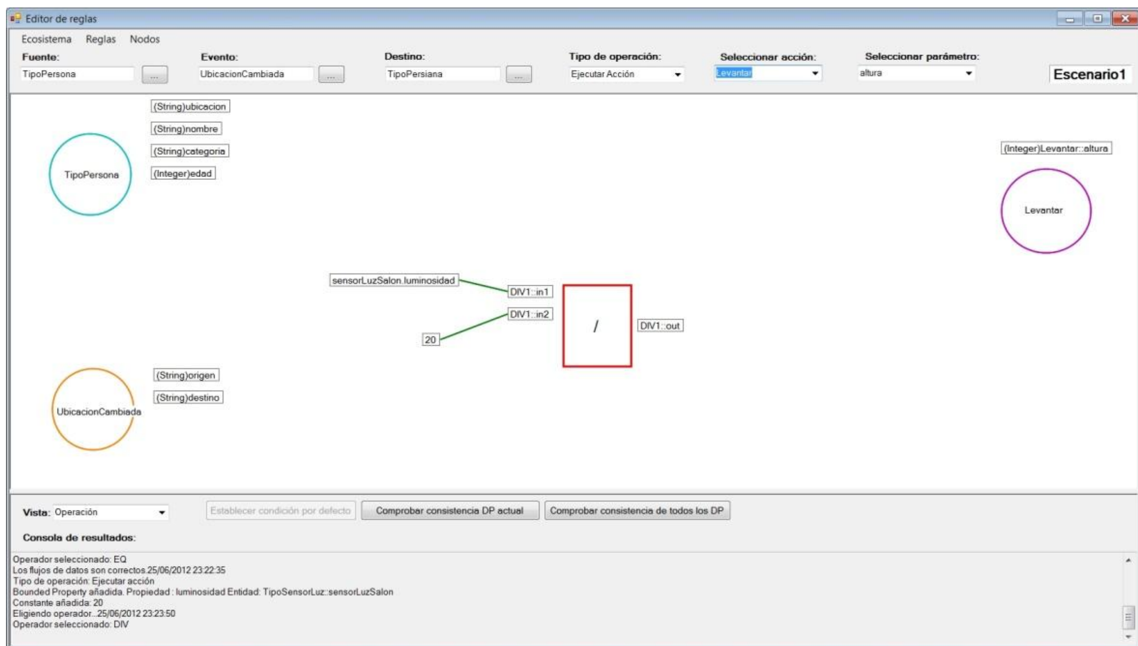


Figura 26. Edición del proceso de datos asociado al parámetro de una acción.

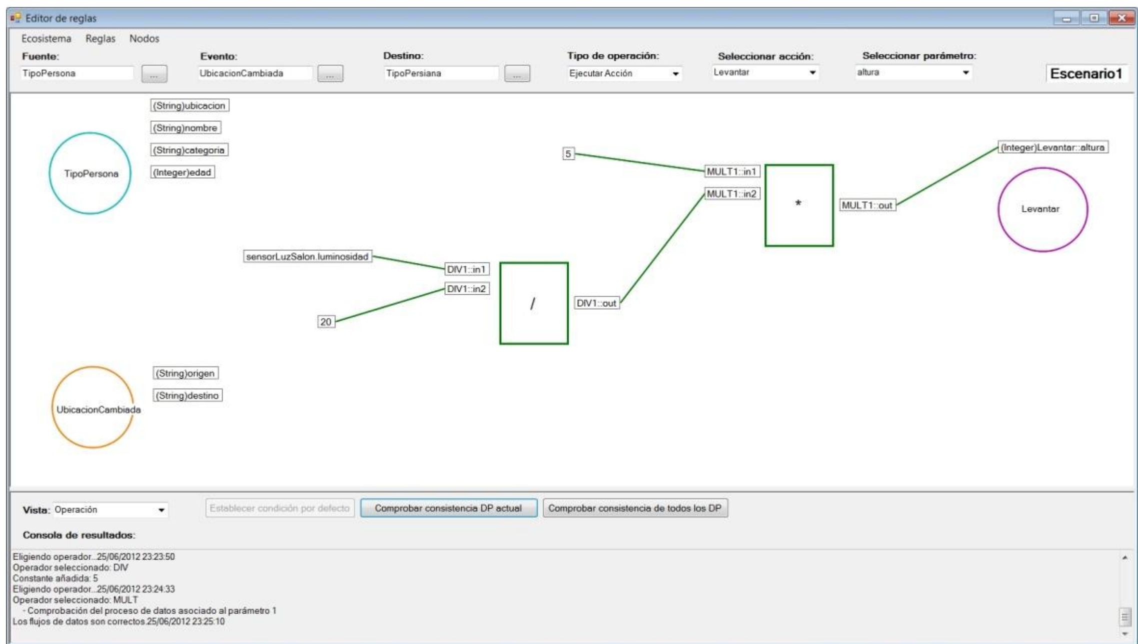


Figura 27. Edición correcta del proceso de datos asociado al parámetro de una acción.

4. Motor de procesamiento de reglas

Una vez realizada la edición de las reglas de comportamiento, el paso siguiente es poner en marcha el mecanismo de simulación del ecosistema para permitir la evolución del mismo. A medida que un ecosistema evoluciona, se producen ocurrencias de evento en el mismo que deben ser atendidas. Para ello, es necesario llevar a cabo el desarrollo e implementación de un procesador de eventos con el fin de permitir la activación e instanciación de las reglas pertinentes que den respuesta a estas ocurrencias de evento.

4.1. Procesador de eventos

Tras haber descrito el metamodelo que da soporte a la definición de reglas en el capítulo 2, a continuación se describe el procedimiento que permite la activación y procesamiento de las reglas. El motor de reglas que se ha construido se basa en un procesador de eventos que haya las correspondencias entre estos y las entidades que estén en el entorno. En definitiva, este procesador se encarga de determinar qué reglas deben ser instanciadas y activadas en función de los eventos que se produzcan en el entorno. En la Figura 28 se muestra el modelo correspondiente al procesador de eventos que se expone a continuación.

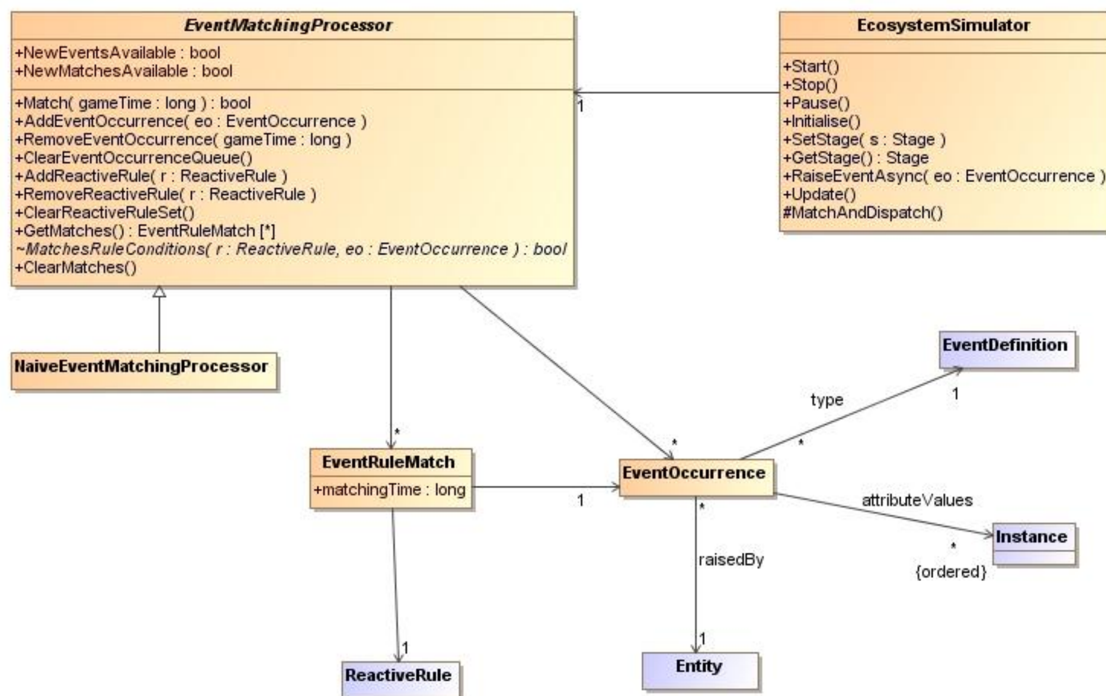


Figura 28. Diagrama de clases UML del procesador de eventos.

Una ocurrencia de evento (*EventOccurrence*) conforma a un tipo de evento (*EventDefinition*) que determina las características de esta ocurrencia, entre ellas, los atributos que la definen (*EventAttributeDefinition*). De manera análoga a como sucedía con las propiedades de las entidades, en este caso los atributos de evento se emplean para definir el tipo de evento, pero cada ocurrencia de un evento que conforme a un determinado tipo tendrá diferentes valores asociados a estos atributos. Por tanto, una ocurrencia de evento debe especificar un conjunto de

pares atributo-valor que representen los valores concretos de los atributos de esta ocurrencia, además de indicar la entidad que la ha provocado.

Cuando se produce una ocurrencia de evento en el sistema, de todas las reglas presentes en el sistema el procesador extrae el conjunto de reglas cuya definición de evento y cuya fuente se correspondan con el evento y la fuente que especifica la ocurrencia de evento. Para cada correspondencia que se encuentre, se instancia el antecedente de la regla con los valores que se obtengan de la ocurrencia de evento, y se evalúa la precondición. Si la precondición se cumple, el procesador de eventos crea una nueva instancia de la clase *EventRuleMatch*, que relaciona la ocurrencia de evento producida con la regla correspondiente que casa con ella.

Una vez obtenido el conjunto de *matchings* entre reglas y ocurrencias de evento cuyas precondiciones son ciertas, para cada una de estas reglas que han hecho *matching*, el procesador debe instanciarla y evaluar su consecuente. Para ello, primero se ejecuta el filtrado de la población destino, si procede, con lo que se obtiene el conjunto de entidades sobre las que aplicar la operación. A continuación, se procede a evaluar la operación para cada una de estas entidades.

4.2. Implementación del procesador de eventos

En el apartado anterior se ha explicado el metamodelo que permite la instanciación de las reglas en base a las ocurrencias de evento que se produzcan en el sistema durante la simulación o ejecución del mismo. En las siguientes figuras puede observarse el pseudocódigo correspondiente al proceso de *matching* que lleva a cabo el procesador de eventos para determinar qué reglas deben instanciarse y ejecutarse. Se asume que las reglas son adecuadamente incluidas durante la inicialización, y que las ocurrencias de evento son inyectadas en el ecosistema por medio del middleware en el momento en que se producen. El algoritmo que se muestra en la Figura 29 esquematiza el comportamiento de procesador de eventos, que consta de dos partes diferenciadas:

En primer lugar se obtiene el conjunto de *EventRuleMatch* con las correspondencias entre las ocurrencias de evento que se han producido y las reglas que se han activado en consecuencia (línea 5). En la Figura 30 se muestran los detalles del proceso de *matching* que lleva a cabo el procesador de eventos. Sea un conjunto de ocurrencias de evento que hayan sido inyectadas en el sistema, ordenado en base al instante temporal en el que se producen. Para cada una de estas ocurrencias, se busca en el conjunto de reglas disponibles aquellas reglas cuya definición de evento coincida con el tipo de la ocurrencia de evento considerada, y cuya población fuente incluya a la entidad que ha provocado la ocurrencia de evento (líneas 8 y 9). Si se encuentra una regla de estas características, se debe determinar si la precondición se cumple para la entidad y el evento considerados (línea 11). En ese caso, se añade el *matching* encontrado al conjunto de *matchings* pendientes de procesamiento (línea 12), y se prosigue con el algoritmo hasta haber procesado todas las ocurrencias de evento pendientes. Nótese que para una única ocurrencia de evento pueden producirse varios *matchings*, ya que es posible que haya varias reglas cuyo antecedente case con la ocurrencia de evento en cuestión. Tras este proceso, se obtiene el conjunto de reglas pendientes de ejecución, que deberán ser evaluadas por el simulador.

En segundo lugar, si se han encontrado correspondencias (*matchings*) entre las ocurrencias de evento producidas y las reglas definidas en el ecosistema, se procesan una a una estas correspondencias. Para cada *EventRuleMatch* encontrado se realiza el filtrado de la población destino tal y como se ha establecido en la regla (línea 11), seleccionando aquellas entidades de la población que cumplan con la condición de filtrado. Para cada una de estas entidades, se ejecuta la operación correspondiente definida en la regla (línea 12).

```
01: EcosystemSimulator::MatchAndDispatch()
02: BEGIN
03:    IF (eventMatchingProcessor.newEventsAvailable)
04:    THEN
05:       eventMatchingProcessor.Match(gameTime);
06:
07:       IF (eventMatchingProcessor.newMatchesAvailable)
08:       THEN
09:          FORALL match: EventRuleMatch IN
10:                 eventMatchingProcessor.GetMatches()
11:             population ← FilterTargetPopulation(match);
12:             ExecuteRule(match,population);
13:
14:             eventMatchingProcessor.ClearEventOccurrenceQueue();
15:             eventMatchingProcessor.ClearMatches();
16:          ENDFOR
17:       END IF
18:    END IF
19: END
```

Figura 29. Pseudocódigo para el procesamiento de ocurrencias de evento.

```
01: EventMatchingProcessor::Match(gameTime:long)
02: BEGIN
03:    newMatchesAvailable ← false;
04:
05:    FORALL eo: EventOccurrence IN pendingEventOccurrences
06:       FORALL rr: ReactiveRule IN currentRuleSet
07:
08:          IF (rr.evDefinition == eo. evDefinition &&
09:              eo.raisedBy BELONGS_TO rr.SourcePopulation &&
10:              MatchesRulePrecondition(rr, eo))
11:          THEN
12:             currentMatches.Add(new EventRuleMatch(eo, rr));
13:             newMatchesAvailable ← true;
14:          ENDIF
15:
16:       ENDFOR
17:    ENDFOR
18:
19:    pendingEventOccurrences.Clear();
20: END
```

Figura 30. Pseudocódigo para el *matching* de ocurrencias de evento con reglas.

4.3. Cálculo del valor objetivo de un proceso de datos

Para cada uno de los procesos de datos que forman la regla se deben llevar a cabo una serie de transformaciones en los datos de entrada para producir finalmente el valor que se deseaba calcular. Se ha visto anteriormente que el conjunto de flujos de datos que especifican las operaciones a realizar tiene una estructura en forma de árbol. Aprovechando esta estructura, el cálculo del valor objetivo de un proceso de datos se realiza explorando el conjunto de flujos de datos empezando desde el nodo objetivo, y navegando hacia atrás para obtener los valores de entrada que se conectan a este nodo. Se trata entonces de una recursión en la cual, para cada nodo que posea puertos de entrada, se debe calcular el valor que recibe dicho puerto, explorando los flujos de datos que entran en él hasta llegar a un caso base que pueda calcularse sin recursión. Los casos base son o bien nodos proveedores de constantes, o bien nodos que representan propiedades de entidades del ecosistema, o bien nodos que representan atributos del evento producido. Durante la edición, estos nodos eran meros representantes de dichos elementos, pero durante la simulación, estos elementos se instancian en la activación de la regla. Por ejemplo, un nodo que represente una propiedad de una entidad del ecosistema tomará su valor del conjunto de pares propiedad-valor de dicha entidad, empleando así el valor de esta propiedad en el momento actual de la simulación. Para facilitar la recuperación de estos valores de propiedades o atributos, se han empleado tablas *hash* como estructuras de datos. Cada entidad del ecosistema dispone de una tabla *hash* indexada por sus propiedades, que permite recuperar en tiempo *O(1)* el valor de una determinada propiedad para la entidad en cuestión. Lo mismo sucede con las ocurrencias de evento: cada ocurrencia de evento dispone de una tabla *hash* indexada por los atributos del tipo de evento al que conforma.

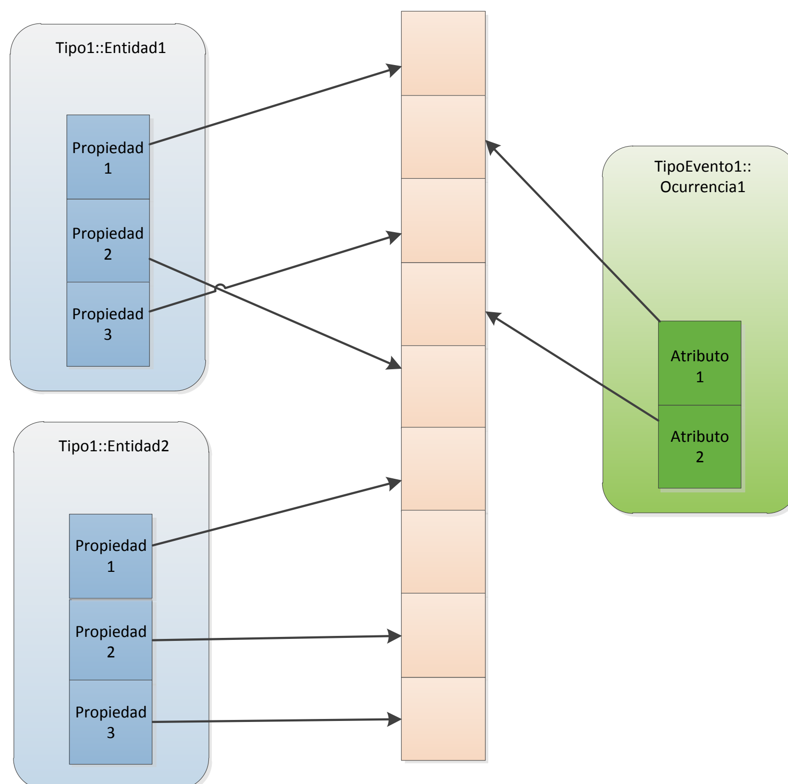


Figura 31. Ejemplo del uso de tablas *hash* para almacenar valores de propiedades y atributos.

La Figura 31 muestra cómo se emplean las tablas *hash* para almacenar los valores de las propiedades de las entidades del ecosistema, y de los atributos de las ocurrencias de evento que se van produciendo.

En la Figura 32 se muestra el pseudocódigo del algoritmo que realiza el cálculo del valor resultante del proceso de datos. En primer lugar, cada proceso de datos dispone de un método denominado *ComputeValue* (líneas 1 a 6) que se encarga de iniciar el cálculo. Para ello, se invoca al método recursivo *CalculateOutput*, que recibe como parámetro un nodo a partir del cual comenzar a calcular. Si se invoca este método desde el nodo objetivo del proceso de datos (línea 4), lo que se obtiene es el valor resultante del proceso de datos que debe ser asignado al nodo objetivo. El método *CalculateOutput* actúa de manera distinta en función del tipo del nodo que recibe como parámetro: en caso de que el nodo recibido como parámetro sea el nodo objetivo, se invoca *CalculateOutput* sobre el nodo antecesor a este, es decir, se navega hacia atrás por el flujo de datos que entra en el nodo objetivo, llegando al nodo anterior que le sirve el dato, y sobre él se prosigue el cálculo (líneas 13-15). Si se trata de un nodo de tipo *PropertyDefinitionSource* y haga referencia a una propiedad de la población fuente, se accede a las estructuras de datos de la población fuente para recuperar el valor de esta propiedad (líneas 17-20). Lo mismo sucede si se trata de una propiedad de la población destino (líneas 22-25). Para el caso en el que el nodo sea de tipo *BoundedPropertyDefinitionSource*, se accede a las estructuras de datos de la entidad del ecosistema a la que hace referencia el nodo, obteniéndose el valor de la propiedad requerida para esta entidad (líneas 27-30). En cambio, si nos encontramos con un nodo de tipo *EventAttributeDefinitionSource*, se accede a las estructuras de datos de la ocurrencia de evento que activó la regla para recuperar el valor del atributo involucrado (líneas 32-36). Para los nodos *ConstantProvider*, se recupera el valor que contiene dicho nodo (líneas 38-41). Finalmente, el caso más especial sucede cuando el nodo recibido es un *DataProcessorInstance*, es decir, un procesador de datos. En este caso, se inicializa una lista para contener los valores que se obtengan en todas las entradas del procesador (líneas 44-45). Para cada uno de los puertos de entrada del mismo (línea 46), se navega hacia atrás por el flujo de datos conectado al puerto de entrada, obteniendo uno de los nodos antecesores. Sobre este nodo antecesor se invoca recursivamente el método *CalculateOutput* (línea 49), siendo el resultado de esta invocación el valor del operando correspondiente a este puerto de entrada del procesador de datos. Cuando se tienen todos los valores de los operandos, se ejecuta la operación adecuada para el tipo de procesador de datos que representa este nodo (línea 53).

La Figura 33 muestra el proceso de cálculo del valor objetivo de la expresión *televisión.volumen ← MIN (televisón.volumen_máx, televisión.volumen + 1)*.

```
01: DataProcess::ComputeValue(source:Entity, event: EventOccurrence,
02: target: Entity) returns:Instance
03: BEGIN
04:    return CalculateOutput(targetDataProcessNode, source,
05:    event, target);
06: END
07:
08: DataProcess::CalculateOutput(node:DataProcessNode,
09: source:Entity, event:EventOccurrence, target:Entity)
10: returns:Instance
11: BEGIN
12:  SWITCH
13:    CASE node es el nodo objetivo del proceso de datos THEN
14:         ancestorNode : nodo antecesor del actual;
15:       return CalculateOutput(ancestorNode,source,event,target);
16:
17:    CASE node es un nodo que representa
18:         una propiedad de la entidad fuente THEN
19:       value : valor de la propiedad para la entidad fuente;
20:       return value;
21:
22:    CASE node es un nodo que representa
23:         una propiedad de la entidad destino THEN
24:       value : valor de la propiedad para la entidad destino;
25:       return value;
26:
27:    CASE node es un nodo que representa una propiedad
28:         de cualquier otra entidad del ecosistema THEN
29:       value : valor de la propiedad para la entidad en cuestión;
30:       return value;
31:
32:    CASE node es un nodo que representa
33:         uno de los atributos del evento THEN
34:       value : valor del atributo
35:             para la ocurrencia de evento actual;
36:       return value;
37:
38:    CASE node es un nodo que representa
39:         un valor constante THEN
40:       value : valor que indica el nodo constante;
41:       return value;
42:
43:    CASE node es un procesador de datos THEN
44:       inputs : lista de Instance para almacenar los valores
45:                de los operandos de entrada del procesador;
46:       FORALL p : Port IN puertos de entrada del procesador
47:         ancestorNode: nodo cuya salida está conectada al
48:                       puerto p por medio de un flujo de datos;
49:         value=CalculateOutput(ancestorNode,source,event,target);
50:         inputs.Add( value );
51:       ENDFOR
52:
53:       return node.ExecuteOperation( inputs );
54:
55:    ENDSWITCH
56: END
```

Figura 32. Pseudocódigo del algoritmo para el cálculo del valor a asignar al nodo objetivo del proceso de datos.

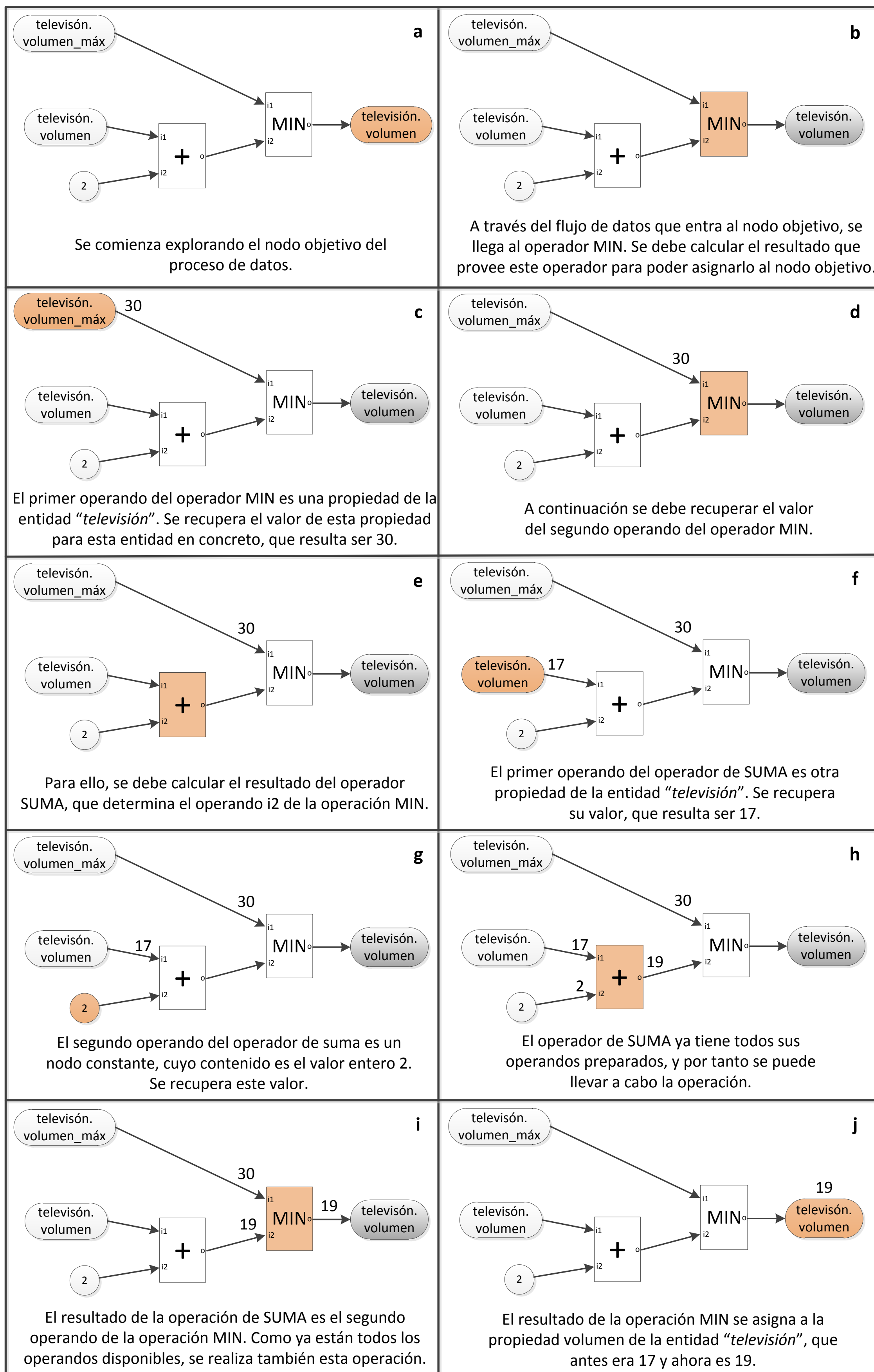


Figura 33. Ejemplo del cálculo del valor objetivo de un proceso de datos.

4.4. Depurador WIMP

Para validar que las reglas editadas con el editor presentado en el capítulo 3 y conforme al modelo propuesto permiten calcular correctamente los valores objetivo durante la simulación, se ha implementado un depurador. Este depurador permite cargar una definición de ecosistema que incluya las reglas correctamente editadas con el editor anterior, e inyectar una serie de ocurrencias de evento que ponen en marcha el procesador de eventos. Se puede llevar a cabo el procesamiento de estas ocurrencias de evento una a una, o bien de manera conjunta, y observar los resultados que producen estas ocurrencias en el ecosistema. Para el procesamiento de eventos se emplean los algoritmos detallados anteriormente en el apartado 4.1, mientras que para el cálculo de la condición, la precondición y el resultado de la operación se emplea el algoritmo de cómputo presentado en el apartado 4.3. En cualquier momento desde que se carga en el depurador un conjunto de reglas se permite comprobar el valor de las propiedades de cualquier entidad de la simulación, por lo que es posible visualizar los cambios que provocan en las entidades cada una de las operaciones ejecutadas debido a la instanciación de las reglas.

Tras la instanciación de una regla debido a una ocurrencia de evento que casa con ella, el depurador muestra una traza de salida que indica qué definición de regla se ha instanciado, qué entidades se han visto afectadas y de qué manera (asignación de valor a una propiedad, o invocación de un método de dicha entidad).

Este depurador ha permitido validar que el cálculo del consecuente de la regla se realiza correctamente conforme a la especificación de los flujos de datos de la regla, y que por tanto el modelo propuesto es válido y correcto. La Figura 34 y la Figura 35 muestran la funcionalidad básica que ofrece el depurador. En esta ejecución del depurador se ha cargado un ecosistema para el ámbito de juegos que dispone de varias reglas, entre ellas la editada en la Figura 13, que realizaba el cálculo del nuevo valor de la propiedad *daño* de una entidad *elfo1* de tipo *TipoElfo*. Otra de las reglas que se han cargado permite que una entidad que conforma a *TipoElfo* ejecute una de sus acciones, estableciendo cómo deben determinarse los parámetros que recibe esta acción. En la Figura 34 se permite comprobar el estado de las entidades, es decir, los valores de sus propiedades, tras la carga de los ficheros que contienen el ecosistema con las reglas y las ocurrencias de evento a lanzar. Conforme se van lanzando ocurrencias de evento, se puede ir comprobando cómo cambian estos valores. Tras haber lanzado dos ocurrencias de evento, en la Figura 35 se muestra la traza de ejecución con la información sobre las reglas que han sido instanciadas para cada ocurrencia de evento. Para la primera regla activada, cuya operación era una asignación a propiedad, se muestra el valor a ser asignado, resultado del cálculo del proceso de datos de la operación. Para la segunda regla, cuya operación era la ejecución de una acción, se muestran los valores de cada uno de los parámetros de entrada de la acción, calculados a partir de los procesos de datos que los definían.

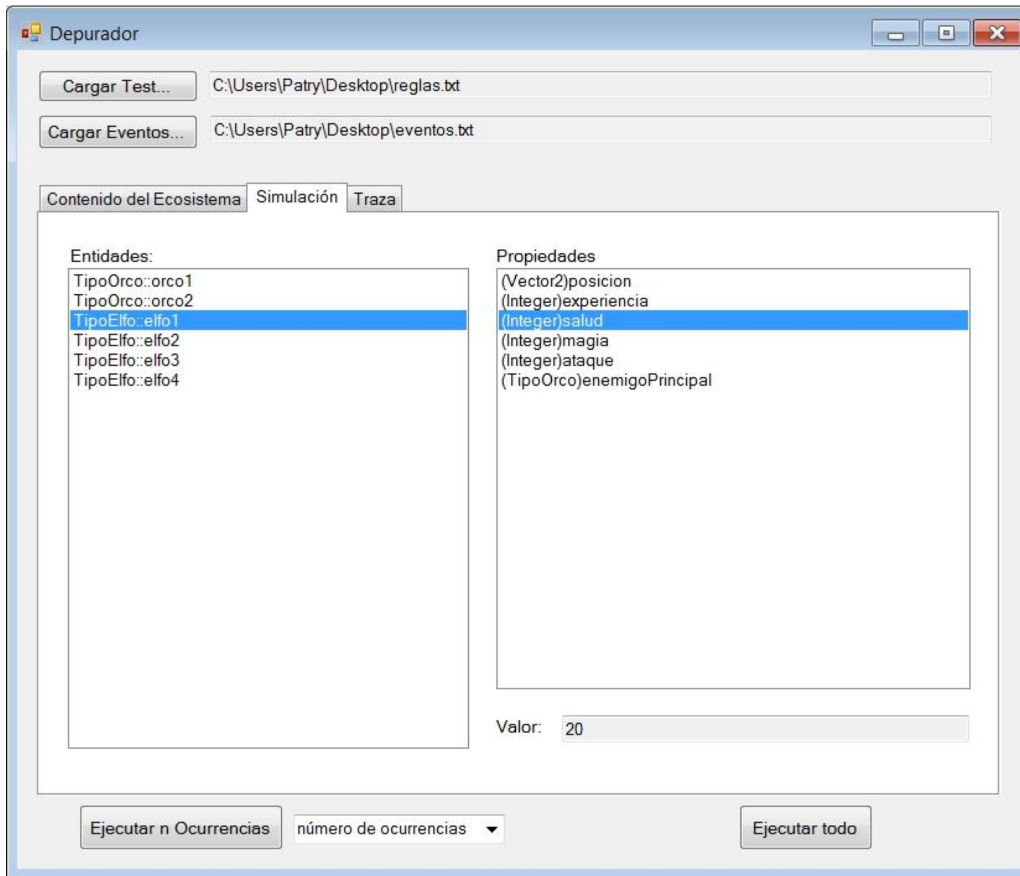


Figura 34. Visualización de las propiedades de las entidades de la simulación.

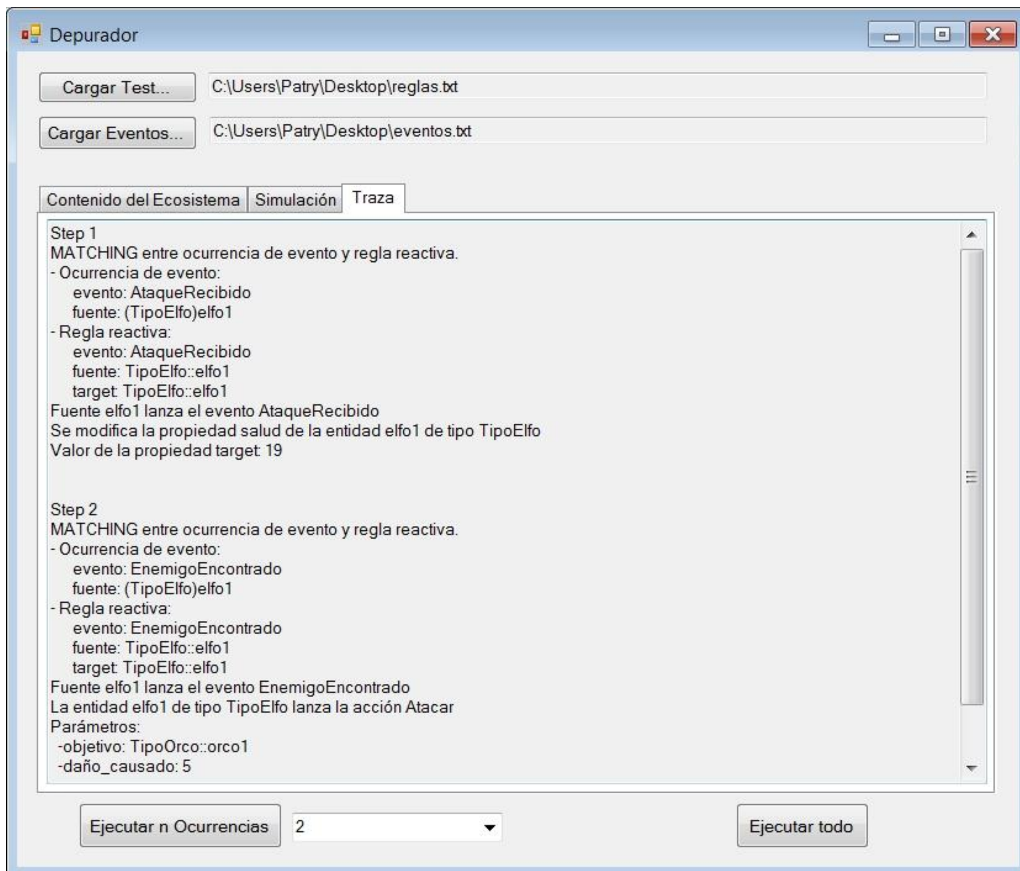


Figura 35. Traza de ejecución tras haber lanzado dos ocurrencias de evento.

5. Validación del procesador de eventos

Tras la implementación del metamodelo para la definición de reglas, el editor que soporta este metamodelo, y el procesador de eventos para la simulación de ecosistemas, se han llevado a cabo una serie de experimentos para determinar la validez del mecanismo de activación y ejecución de reglas presentado, en base a su productividad y escalabilidad. Los resultados de estos experimentos se exponen a lo largo del presente capítulo, dividiéndose en tres tipos diferenciados. En primer lugar, se evalúa el rendimiento del proceso de *matching*, variando diferentes parámetros y aplicando algunas optimizaciones. En segundo lugar, se evalúa el rendimiento del mecanismo de cómputo de valores en los procesos de datos. Finalmente, se realiza una comparación entre el procesamiento secuencial de ocurrencias de evento con una versión paralela del mismo proceso.

5.1. Evaluación del proceso de *matching*

Dado que se ha optado por un metamodelo genérico, que permite la definición de reglas para poblaciones genéricas, resulta necesario determinar si esta aproximación no introduce ninguna sobrecarga en el procesamiento de las ocurrencias de evento. Se han realizado dos experimentos al respecto.

5.1.1. Tiempo medio de establecimiento de un *matching* en base al número de entidades en el ecosistema

El proceso de *matching* consiste en determinar, para cada una de las ocurrencias de evento pendientes de ser procesadas, qué conjunto de reglas se activan. Para ello, para cada ocurrencia de evento que se produzca, se ha de recorrer el conjunto de reglas del ecosistema en busca de correspondencias. Dada una ocurrencia de evento y una regla, el establecimiento de un *matching* es la tarea de determinar si la ocurrencia en cuestión activa la regla. Esta activación se produce si el tipo de evento de la ocurrencia es igual al tipo de evento de la regla, y si la entidad que lanza la ocurrencia de evento pertenece a la población fuente de la regla. La comparación entre el tipo de evento de la ocurrencia y el tipo de evento de la regla resulta sencilla. El caso conflictivo se da con la población fuente de la regla, ya que puede estar definida de manera específica, como una entidad, o de manera genérica, como un tipo de entidad. Dada la entidad que lanza la ocurrencia de evento, si la regla es de tipo específico, se comparan las entidades entre sí, mientras que si la regla es de tipo genérico, se comprueba que el tipo de la entidad es igual al tipo de la fuente de la regla. Se desea comprobar que a pesar de que aumente el número de tipos de entidad y/o de entidades en el ecosistema, el establecimiento de un *matching* entre una ocurrencia de evento y una regla dadas se realiza en tiempo constante. De este modo, se pretende demostrar que el metamodelo empleado y los mecanismos de reflexión que provee permiten la escalabilidad del sistema en base al número de entidades presentes en el mismo.

Para el experimento en cuestión se han especificado reglas cuya población fuente está definida como un tipo de entidad. La precondición y condición de filtrado de estas reglas es trivial, la población destino es una entidad específica, y la operación a realizar se ha definido como una asignación a propiedad simple (asignación directa de un valor a una propiedad). Para las distintas iteraciones del

experimento se ha ido variando el número de tipos de entidades en el sistema entre 1 y 100, y para cada cantidad de tipos de entidad se ha ido variando el número de entidades que conforman a cada tipo, entre 1 y 40.000. De este modo, el número total de entidades en el ecosistema, teniendo en cuenta las entidades de todos los tipos, varía entre 1 y 4.000.000. Dicho número de entidades en cada iteración puede calcularse como T * I, siendo T el número de tipos de entidad, e I el número de entidades por tipo. El resto de parámetros se han mantenido constantes.

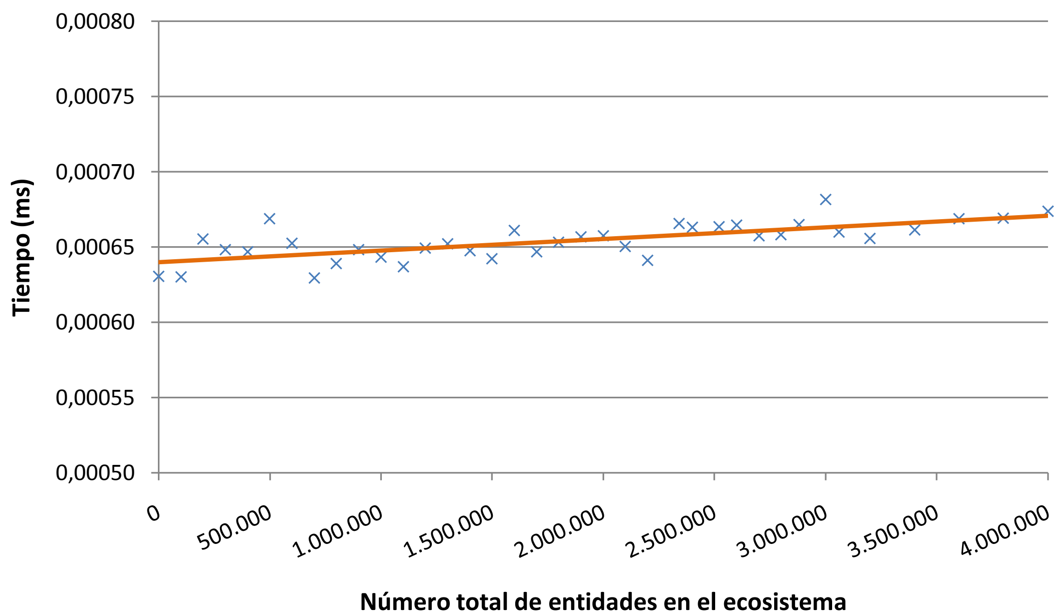


Figura 36. Tiempo medio de establecimiento de un *matching* según el número total de entidades en el ecosistema.

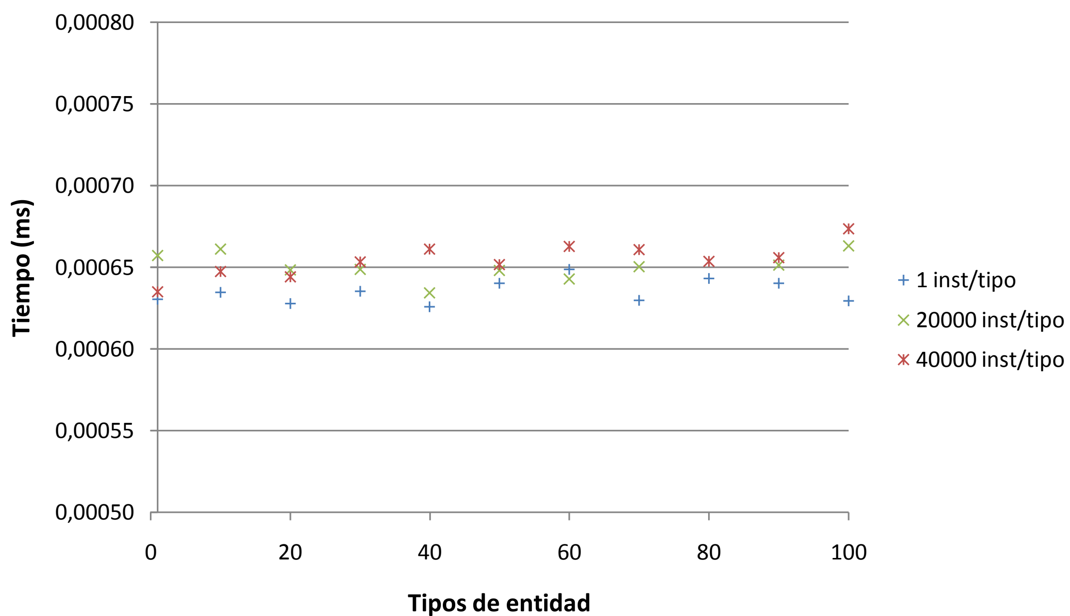


Figura 37. Tiempo medio de establecimiento de un *matching* según el número de tipos de entidad.

La Figura 36 muestra que el tiempo medio de establecimiento de un *matching* permanece prácticamente constante en 0,00065 milisegundos, aunque se incremente enormemente el número de entidades en el ecosistema. En la Figura 37 se muestra el tiempo medio del establecimiento de un *matching* según el número de tipos de entidades que se han considerado (de 1 a 100 tipos), para tres casos distintos: una entidad por cada tipo (en total de 1 a 100 entidades), 20.000 entidades por tipo (en total de 1 a 2.000.000 de entidades) y 40.000 entidades por tipo (en total de 1 a 4.000.000 de entidades). Se observa por un lado que el incremento del número de entidades que conforman a un tipo no afecta al tiempo de establecimiento del *matching*, y que tampoco afecta a este tiempo el incremento del número de tipos dentro del ecosistema.

5.1.2. Tiempo medio del proceso de *matching*

Dada una ocurrencia de evento, el proceso de *matching* completo para dicha ocurrencia supone recorrer todo el conjunto de reglas disponibles en el ecosistema en busca de aquellas reglas que casen con la ocurrencia, ya que una misma ocurrencia puede activar varias reglas. En este contexto, el coste computacional de procesar una ocurrencia de evento en busca de *matchings* se incrementa conforme aumenta el número de reglas en el sistema.

El objetivo de este experimento es determinar cómo evoluciona el tiempo medio del proceso de *matching* de una ocurrencia de evento a medida que aumenta el número de reglas a explorar en busca de coincidencias. Además, se pretende demostrar la ventaja que supondría emplear estructuras de datos que facilitaran el acceso al conjunto de reglas aplicables.

Dado que una ocurrencia de evento se procesa en base al tipo de evento al que conforma y a la entidad fuente que lo lanza, una buena manera de obtener una mejora en el tiempo de procesamiento de la ocurrencia sería disponer de estructuras de datos indexadas en base a estos dos parámetros, que permitieran obtener en tiempo prácticamente constante el conjunto de reglas potencialmente aplicables. Para ello se han empleado dos tablas *hash*, cada una con dos niveles de indexación. El primer nivel de indexación se realiza en base al tipo de evento de la regla. Una consulta sobre este primer nivel de indexación devuelve una segunda tabla *hash*, esta vez indexada en base a la población fuente de la regla. La consulta sobre el segundo nivel de indexación devuelve una lista con las reglas potencialmente aplicables para dicho evento y fuente, de modo que lo único que resta para determinar si cada una de estas reglas se activa o no sería comprobar su precondición.

Las dos tablas *hash* son necesarias ya que las ocurrencias de evento son lanzadas por entidades concretas del ecosistema, pero las reglas pueden definirse o bien para poblaciones específicas o bien para poblaciones genéricas. De este modo, una ocurrencia lanzada por una entidad específica podría activar tanto reglas cuya población sea específica y dicha población corresponda explícitamente con la entidad en cuestión, o bien reglas cuya población sea un tipo de entidad, y se trate del tipo de entidad al que conforma la entidad que lanzó la ocurrencia. Por ejemplo, una entidad *sensorLuz1*que conforme al tipo *SensorLuz* podría activar tanto reglas cuya fuente sea *sensorLuz1* (fuente específica) como reglas cuya fuente sea el tipo *SensorLuz* (fuente genérica). De este modo, se dispone de una tabla *hash*

que permite recuperar las reglas cuya fuente sea una entidad concreta, y otra tabla *hash* que permite recuperar las reglas cuya fuente sea un tipo de entidad.

El conjunto de reglas potencialmente aplicables dada una ocurrencia de evento se obtiene consultando ambas tablas *hash*, con coste *O(1)* cada una, y uniendo los dos conjuntos de reglas que se obtienen tras la consulta. La Figura 38 ejemplifica el uso de las dos tablas *hash* que se han descrito, donde en azul quedaría el primer nivel de indexación según el tipo de evento, en naranja el segundo nivel de indexación en función de la fuente, y en verde las reglas que se obtendrían tras la navegación de los dos niveles. Puede observarse que, para una ocurrencia de evento cuyo tipo de evento sea *TipoEvento1*, y cuya fuente sea la *entidad1* de *TipoEntidad1*, se activarían las reglas 1 y 2. Para una ocurrencia de evento de *TipoEvento2*, cuya fuente sea la *entidad1* de *TipoEntidad1*, tan sólo se activaría la regla 6 correspondiente a una fuente de tipo poblacional. Esta regla se activa ya que los eventos se corresponden, y la *entidad1* de *TipoEntidad1* conforma al tipo definido en la fuente de la regla. Para este caso no hay ninguna regla cuya población fuente sea una entidad que case con la ocurrencia de evento descrita. En cambio, para una ocurrencia de evento que conforme a *TipoEvento1*, y cuya fuente sea la *entidad1* de *TipoEntidad2*, se activarían las reglas 3 y 4; la regla 3 debido a que la población fuente de dicha regla es exactamente la *entidad1* de *TipoEntidad2*; la regla 4 debido a que la *entidad1* de *TipoEntidad2* conforma al tipo de la fuente de la regla.

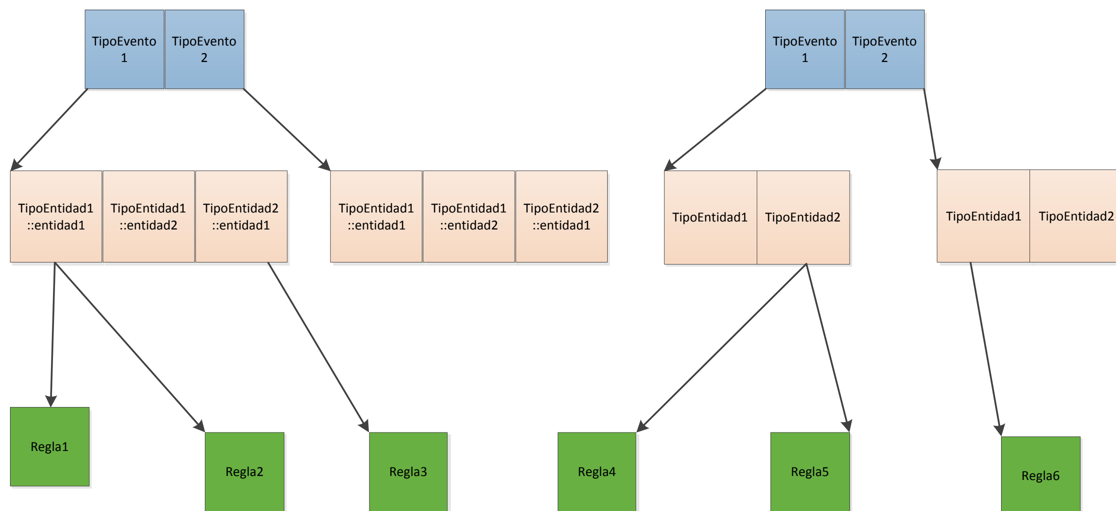


Figura 38. Ejemplo del uso de tablas *hash* con doble indexación para detectar las reglas aplicables.

Se han llevado a cabo dos experimentos, el primero de ellos realizando el proceso de *matching* de manera secuencial, recorriendo todo el conjunto de reglas, y el segundo experimento empleando las tablas *hash* descritas anteriormente. Para ambos experimentos se ha empleado un ecosistema con un número fijo de tipos de entidad y de entidades de cada tipo. Se han realizado diversas iteraciones aumentando el número de reglas presentes en el ecosistema, donde el número de reglas se calcula como E * I * R. El parámetro E indica la cantidad de tipos de eventos considerados, y se ha variado entre1 y 2.000; I indica el número de entidades en el ecosistema, y como se ha indicado este parámetro ha permanecido fijo en 10 entidades, todas del mismo tipo; R indica el número de reglas potencialmente aplicables para un par <Tipo de Evento, Fuente> determinado, y se

ha variado entre 1 y 10. En consecuencia, el número total de reglas en el ecosistema ha oscilado entre 1 y 200.000.

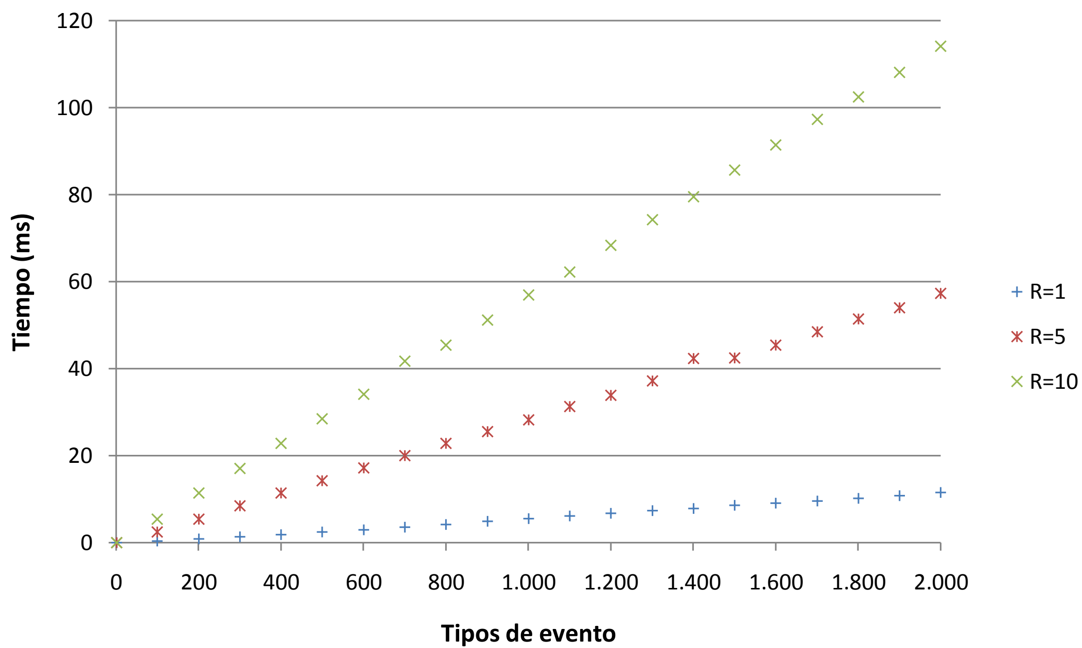


Figura 39. Tiempo medio del proceso de *matching* según el número de tipos de evento (sin optimización).

En la Figura 39 se pueden observar los resultados de la versión del experimento sin emplear las tablas *hash*, es decir, recorriendo todo el conjunto de reglas. Para un número determinado de repeticiones, entendiendo por repetición el número de reglas aplicables dado un par <Tipo de Evento, Fuente>, se observa que el tiempo medio del proceso de *matching* para una ocurrencia de evento aumenta linealmente con el número de tipos de evento, ya que a más tipos de evento, más reglas se incluían en el sistema. Además, este aumento es más pronunciado conforme se aumenta el número de repeticiones, ya que aumentar este valor, según se ha definido, implica tener más reglas en el sistema.

Por otro lado, en la Figura 40, se muestran los resultados de la versión en la que se han empleado las tablas hash para indexar los conjuntos de reglas aplicables. Para un número determinado de repeticiones, el tiempo medio del proceso de *matching* para una ocurrencia de evento se mantiene constante aunque aumente el número de tipos de evento (y por tanto el número de reglas) en el sistema. Lo único que se nota es un ligero aumento de este tiempo medio si se incrementa el número de repeticiones, ya que si aumenta el número de repeticiones, el tiempo de obtención del conjunto de reglas potencialmente aplicables es constante, pero se debe comprobar si cada una de las reglas de dicho conjunto cumple con la precondición. Por tanto, si el conjunto de reglas potencialmente aplicables aumenta, lo hace en consecuencia el tiempo medio del proceso debido al recorrido secuencial para evaluar la precondición sobre cada una de ellas.

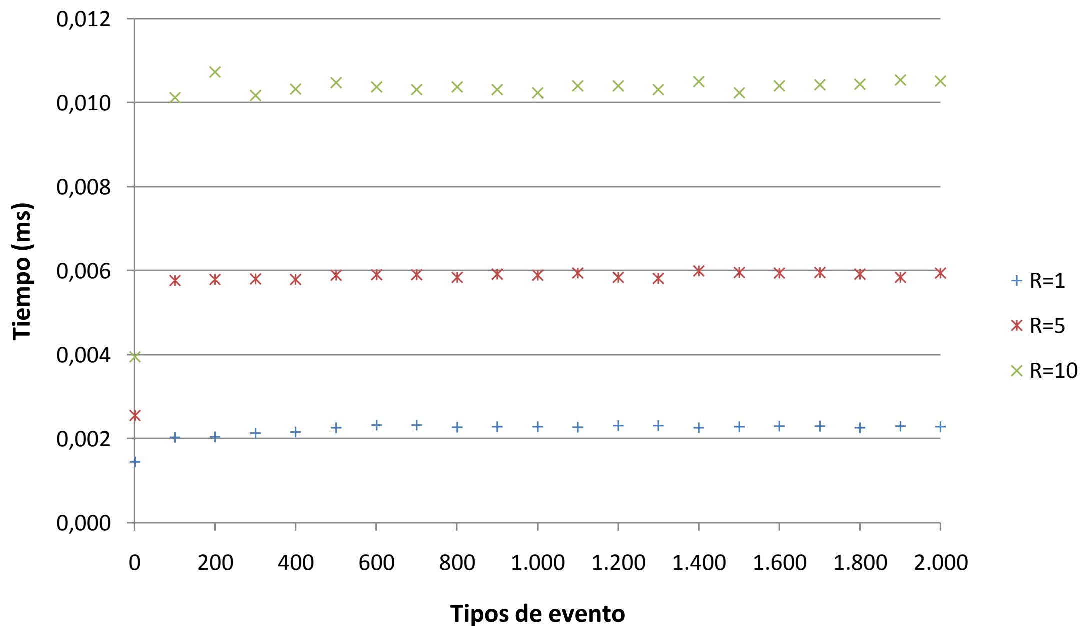


Figura 40. Tiempo medio del proceso de *matching* según el número de tipos de evento (con optimización).

Para comprobar que incluso en un caso extremo, la solución mediante indexación es más rápida que la secuencial, se muestra en la Figura 41 el resultado de un tercer experimento al respecto. En dicho experimento se ha mantenido la variación del número de reglas totales en el ecosistema entre 1 y 200.000, pero se ha optado por reducir el rango de tipos de eventos, entre 1 y 10, aumentando de esta forma el rango de repeticiones, entre 1 y 2.000. De este modo, cada acceso a las tablas hash para un par <Tipo de Evento, Fuente>, devolvería un conjunto de reglas potencialmente aplicables de cómo máximo 2.000 reglas. Para cada una de estas reglas se debería evaluar la precondición, lo cual implica tener que recorrer este conjunto de reglas una a una. Este recorrido secuencial es inevitable e introduce un sobrecoste en el tiempo medio del proceso de *matching*. Como se puede observar, para el peor caso contemplado, en el que se debe recorrer un conjunto de 2.000 reglas y evaluar la precondición sobre cada una de ellas, el tiempo medio asciende a 1,8 milisegundos. Este tiempo es mucho mayor que en el caso de tener únicamente una regla potencialmente aplicable, pero sin embargo se observa que el sobrecoste introducido es mucho menor que tener que recorrer el conjunto de reglas completo, con lo cual, incluso en una situación desfavorable, esta aproximación supone una mejora considerable.

Con estos experimentos se ha demostrado la ventaja que supone el uso de tablas *hash* para la indexación de reglas cuya fuente sea una entidad específica. Pero, como se ha comentado anteriormente, la entidad que lanza una ocurrencia de evento puede activar tanto reglas con poblaciones fuente específicas, como reglas con poblaciones fuente genéricas. Por tanto, se han llevado a cabo los mismos experimentos que se han descrito en este apartado, pero esta vez considerando la indexación, mediante tablas *hash,* de reglas con fuentes poblacionales en lugar de fuentes específicas. Se han mantenido los rangos de los parámetros E y R, mientras que en estos experimentos se ha optado por tener 10 tipos de entidad, con una entidad de cada tipo. De nuevo se han tomado los tiempos medios del proceso de *matching* para ambas versiones, la tradicional y la que conlleva indexación. Los

resultados son idénticos a los obtenidos en la Figura 39, Figura 40 y Figura 41, por tanto no se reportan.

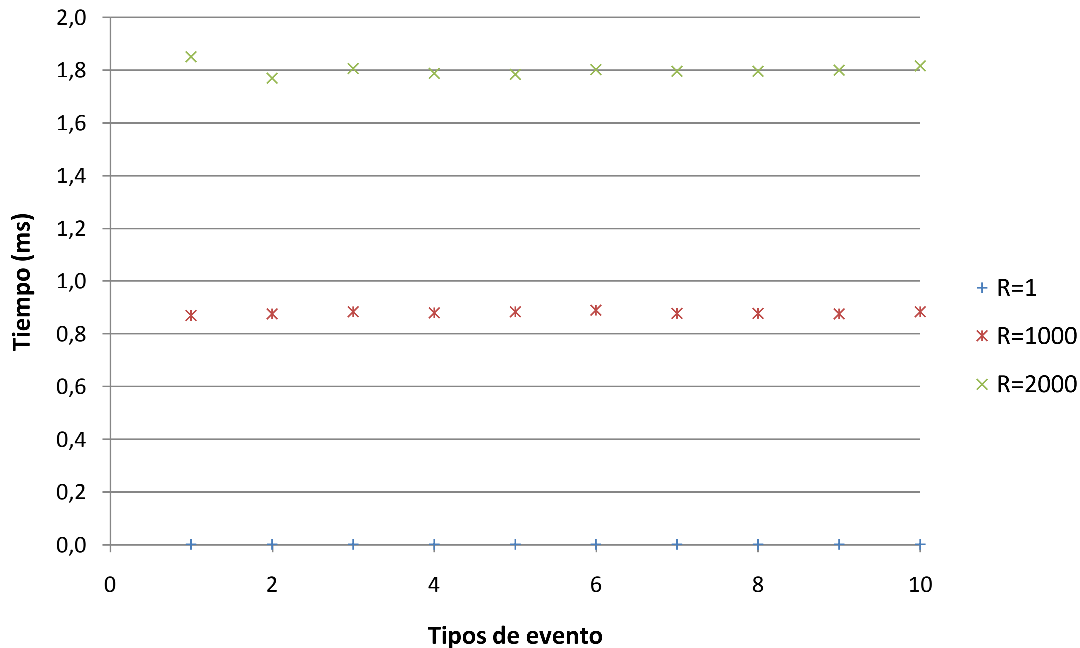


Figura 41. Tiempo medio del proceso de *matching* según el número de tipos de evento (con optimización, rangos alterados).

5.2. Evaluación del mecanismo de ejecución de reglas

A continuación se van a exponer dos experimentos relacionados con el mecanismo de ejecución de la operación de la regla, que consta por un lado del filtrado de la población destino, y por otro lado del cálculo de los procesos de datos que definen la operación.

5.2.1. Optimización sobre el filtrado de la población destino

Al igual que en una regla es posible definir la población fuente de manera genérica en lugar de indicar una entidad específica, se puede definir la población destino de la regla también de manera genérica. Según el algoritmo de ejecución de reglas propuesto, cuando una regla cuya población destino sea genérica se instancia, se debe llevar a cabo el filtrado de dicha población para obtener el conjunto de entidades a las que aplicar la operación. El filtrado de estas entidades conlleva evaluar la misma condición, la condición de filtrado, sobre cada una de las entidades, determinando qué entidades cumplen la condición y qué entidades no la cumplen. La condición de filtrado es un proceso de datos más dentro de la regla, cuyo resultado puede variar de una entidad a otra únicamente si en este proceso de datos intervienen propiedades de la población destino, ya que entonces los valores de estas propiedades son distintos para cada entidad. En cambio, en el caso en el que en la condición de filtrado no intervengan las propiedades de la población destino, el valor resultante del cómputo de la condición es siempre el mismo, independientemente de la entidad sobre la que se esté evaluando. Sin embargo, este cálculo seguiría ejecutándose tantas veces como entidades haya en la población destino. Así, resultaría interesante introducir una mejora en el

proceso de filtrado que considere si las propiedades de la entidad destino intervienen o no en la condición de filtrado, y en caso de que no lo hagan, tan sólo realice el cálculo de la condición una única vez. Si el resultado es positivo, todas las entidades de la población destino han pasado el filtrado y por tanto se debe ejecutar la operación sobre ellas. Si el resultado de la condición es negativo, ninguna de las entidades de la población destino cumple la condición, y por tanto el conjunto de entidades filtradas es vacío. En caso de que cualquier propiedad de la población destino intervenga en la condición de filtrado, se seguiría el proceso habitual, evaluando la condición para cada una de las entidades de manera independiente.

Resulta interesante plantear esta mejora debido a que es frecuente emplear reglas cuyas operaciones deben acometerse sobre un conjunto completo de entidades que conformen a un tipo (p. e.: levantar todas las persianas de la casa, o apagar todos los ordenadores por la noche), sin importar características particulares de cada entidad, y si se eleva el número de instancias dentro del tipo de la población destino, el tiempo de evaluación de la condición de filtrado aumentaría en consecuencia.

Dado este planteamiento, se han realizado experimentos para evaluar si esta optimización supondría obtener un tiempo de filtrado constante para el caso de reglas como las descritas. Las reglas empleadas en el experimento se han definido con la población fuente específica, la precondición trivial y la operación a realizar como una asignación simple a propiedad. La población destino de las reglas se ha definido de manera genérica, es decir, empleando un tipo de entidad en lugar de una entidad específica. Se ha considerado un único tipo de entidad para este experimento, ya que este parámetro es relevante para el proceso de *matching*, pero no afecta al proceso de filtrado. El único parámetro que se ha variado en cada iteración del experimento es el número de entidades que conformaban al tipo de la población destino de las reglas, de modo que se pueda estudiar el tiempo de evaluación de la condición de filtrado según la talla de la población a filtrar. Por un lado, se ha determinado el tiempo medio de evaluación de la condición de filtrado para el caso de reglas cuya condición de filtrado no involucre ninguna propiedad de la población destino, aplicando la optimización descrita. Por otro lado, se ha determinado en iguales condiciones el tiempo medio de evaluación de la condición de filtrado para el caso de reglas cuya condición de filtrado sí que implique propiedades de la población destino, de modo que la optimización no es aplicable, y se lleva a cabo la evaluación de la manera habitual. Se ha considerado el tiempo medio de evaluación de la condición de filtrado como el tiempo total que se tarda en recorrer todas las entidades de la población destino, y para cada una de ellas evaluar la condición de filtrado, dividido por el número de veces que se invoca este procedimiento de filtrado. Se han llevado a cabo ambos experimentos para un rango de entidades de la población destino entre 1 y 1.500.000 entidades.

Tras observar los resultados, tal y como cabía esperar, el tiempo medio de evaluación de la condición de filtrado, sin aplicar la optimización propuesta, aumenta linealmente conforme al número de entidades en la población destino, como se muestra en la Figura 42. En cambio, en la Figura 43 se observa que al aplicar la optimización el tiempo medio es prácticamente constante.

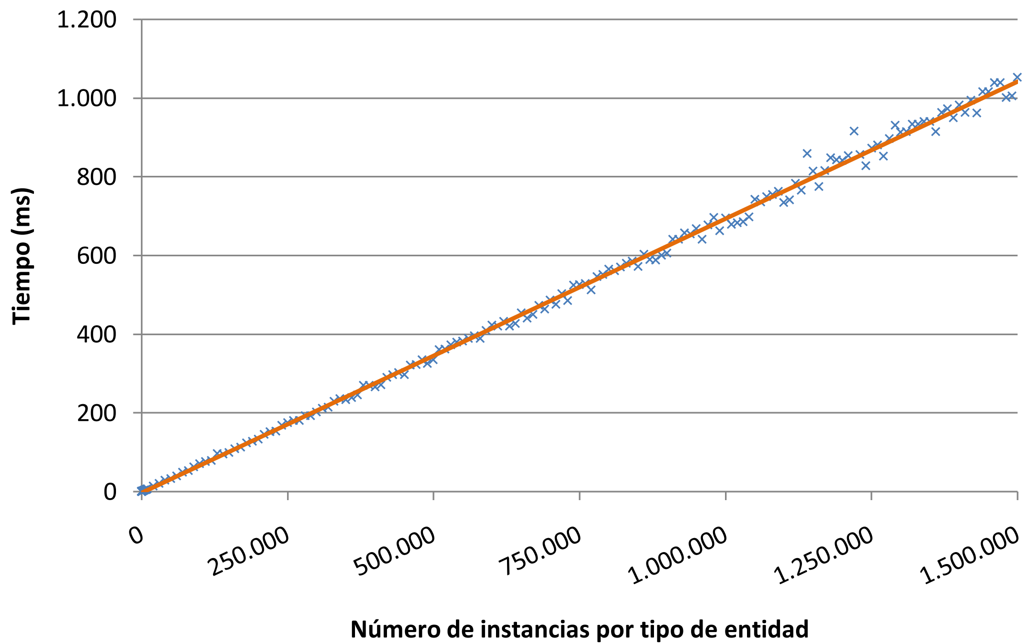


Figura 42. Tiempo medio de evaluación de la condición de filtrado (sin aplicar optimización)

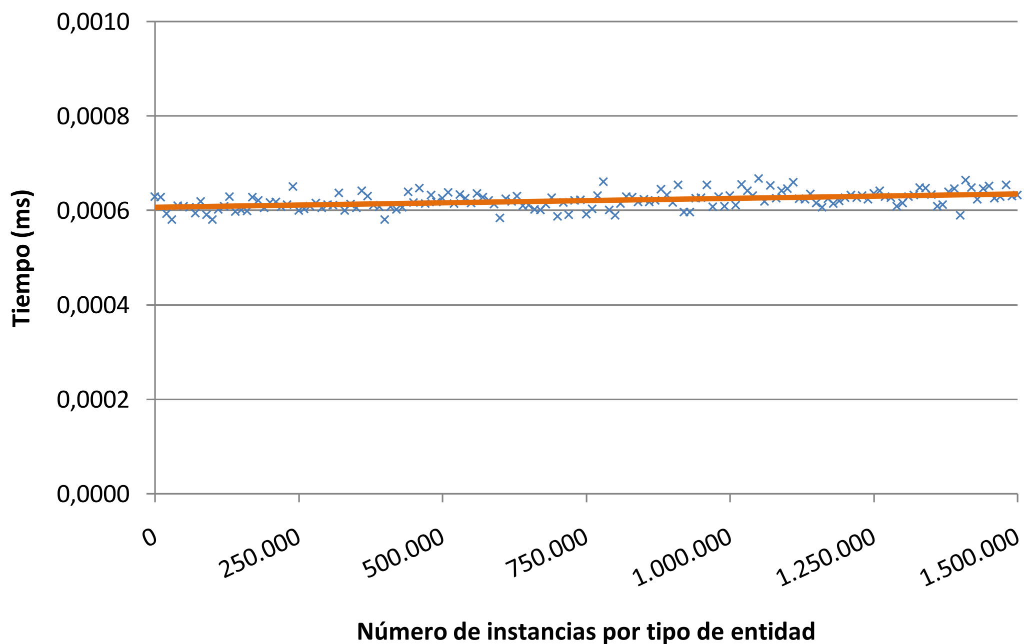


Figura 43. Tiempo medio de evaluación de la condición de filtrado (aplicando optimización)

A la vista de los resultados satisfactorios de este experimento, se ha considerado la posibilidad de aplicar esta misma optimización al caso de los procesos de datos asociados a la operación a realizar en la regla. Dado que la operación a aplicar es la misma para todas las entidades de la población destino, el resultado de los procesos de datos asociados a esta operación es el mismo también para todas las entidades de la población destino, si no intervienen propiedades de estas entidades en el proceso de datos considerado. Por tanto, se podría aplicar la optimización propuesta, calculando el valor resultante del proceso de datos una

única vez, y empleando este valor para todas las operaciones sobre las distintas entidades destino.

5.2.2. Tiempo de cálculo de los procesos de datos

Otro de los aspectos a evaluar es la complejidad que puede alcanzar un proceso de datos. Se ha demostrado que el procedimiento de cálculo del resultado de un proceso de datos es correcto, pero no se ha estudiado todavía qué complejidad en las operaciones permitiría alcanzar la aproximación mediante flujos de datos sin comprometer el rendimiento del sistema. Tal y como se ha comentado en otras ocasiones a lo largo del documento, los procesos de datos pueden representarse en forma de árbol, siendo la raíz el nodo objetivo del proceso de datos, los nodos del árbol los procesadores de datos u operadores, y las hojas del árbol los nodos fuente o proveedores de datos. Para ello, se ha realizado un experimento que consideraba reglas con poblaciones fuente y destino específicas, cuya precondición y condición de filtrado es trivial, y cuya operación consistía en una asignación a propiedad. El proceso de datos involucrados en la operación de las reglas se ha definido en forma de árbol binario perfecto[2] de nivel K, compuesto únicamente de operadores suma como nodos del árbol binario, y constantes cuyo valor era 1 como hojas del árbol. En el experimento se han realizado varias iteraciones de iguales características, en cada una de ellas variando el número de niveles del árbol entre K = 0 y K = 15, de manera que el número de operadores en el proceso de datos aumentaba en consecuencia. Un árbol binario perfecto con K niveles contiene 2^{K+1}– 1 nodos, incluyendo las hojas; por tanto, un proceso de datos de este estilo tendría, sin contar el nodo raíz, 2^{K+1} – 1 nodos, y según se ha definido, 2^{K} – 1 operadores.

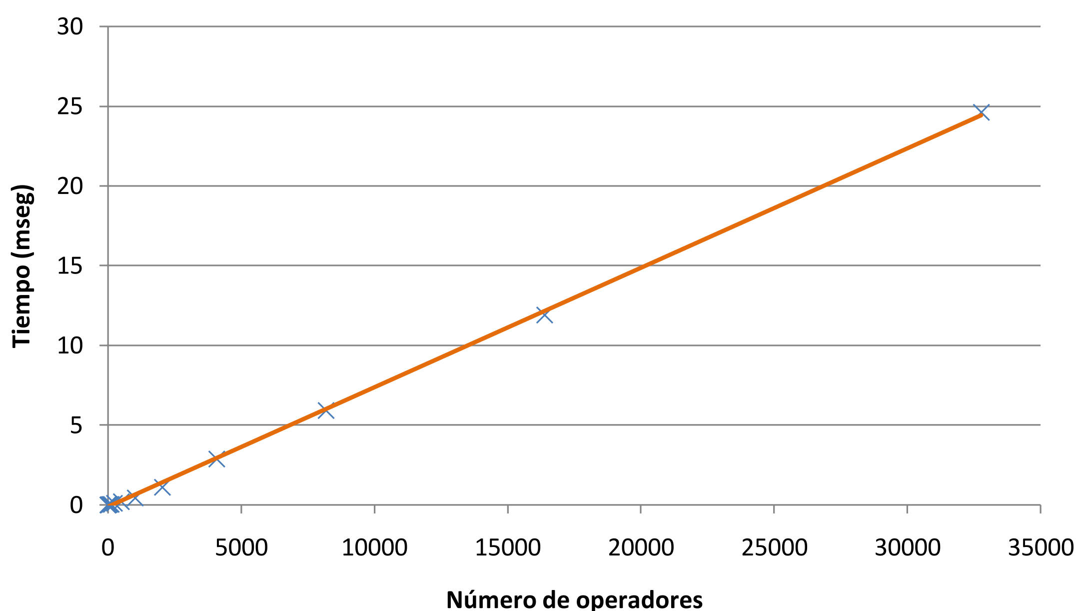


Figura 44. Tiempo medio de evaluación de un proceso de datos en base al número de operadores.

Se ha estudiado el tiempo medio que se ha tardado en realizar el cálculo de un proceso de datos para cada uno de los distintos niveles de operadores que se han

[2] Árbol binario cuyos nodos tienen siempre dos hijos, a excepción de las hojas, que no tienen hijos, y en el que además todas las hojas están en el mismo nivel.

considerado. El tiempo medio de cálculo se ha definido como el tiempo total empleado en calcular resultados de los procesos de datos, dividido por el número total de veces que se realiza el cálculo. Se puede observar en la Figura 44 que el tiempo medio de cálculo aumenta linealmente con el número de operadores involucrados en el proceso de datos. En la Figura 45 se muestra en detalle la misma gráfica, acotando el número de operadores del proceso de datos entre 0 y 31, lo que correspondería a procesos de datos con entre 0 y 5 niveles de nodos.

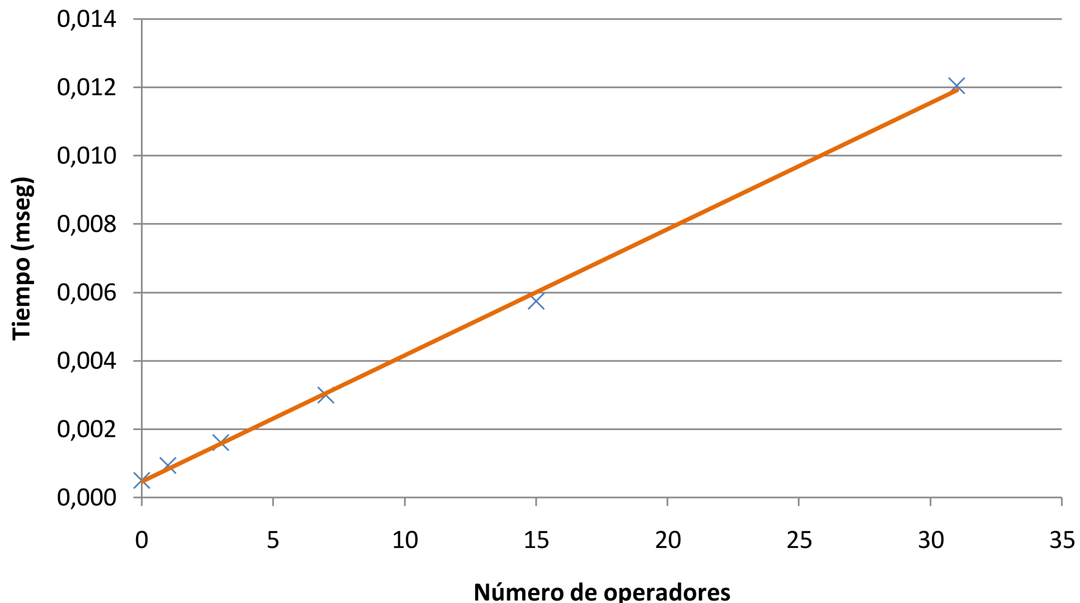


Figura 45. Detalle de la Figura 44.

5.3. Comparativa: procesamiento secuencial vs. procesamiento paralelo

Hasta el momento, el procesador de eventos ha llevado a cabo sus tareas de manera secuencial. Sin embargo, actualmente los microprocesadores proveen facilidades para la paralelización de tareas, que pueden ser aprovechadas por compiladores o *frameworks* para aumentar el rendimiento del sistema dividiendo la carga entre todos los procesadores de la máquina. Se han planteado una serie de experimentos para evaluar la ventaja que supondría la paralelización por medio de hilos de las tareas que lleva a cabo el procesador de eventos.

El procesador de eventos actual consta de dos fases diferenciadas, que se ejecutan en secuencia. Estas fases son, por un lado, el proceso de *matching*, y por otro, el proceso de evaluación del consecuente de la regla. Cada instante de tiempo en el que el procesador de eventos se pone en marcha, se realiza primero el proceso de *matching*, y la salida que produce dicho proceso es consumida por el proceso de evaluación, de modo que es factible llevar a cabo la paralelización de estas dos fases por separado.

Para llevar a cabo una comparación entre el procesamiento secuencial y paralelo de cada una de estas fases, se ha incorporado a la implementación una variante del algoritmo de procesamiento de reglas que contempla la paralelización de estas

tareas. A continuación, se muestran por separado los resultados de la paralelización de cada una de las fases.

5.3.1. Paralelización del proceso de *matching*

El proceso de *matching* implementado consume de manera secuencial un conjunto de ocurrencias de evento que se han producido en el sistema en un instante de tiempo determinado. Para cada ocurrencia de evento se realiza la misma tarea: consultar el conjunto de reglas disponibles en el ecosistema buscando correspondencias.

Para paralelizar el proceso de *matching*, se debe dividir el conjunto de ocurrencias de evento a procesar entre el número de hilos disponibles. Una primera aproximación sería un reparto estático y equitativo del conjunto de ocurrencias de evento. Para ello, por ejemplo, podría dividirse el conjunto de ocurrencias de evento en tantos subconjuntos como hilos se disponga, de modo que cada hilo recibe el mismo número de ocurrencias a procesar. Sin embargo, el problema de esta aproximación radica en que lo más probable es que algún hilo se quede bloqueado esperando información o realizando algún cálculo, o incluso que deba esperar para acceder a datos compartidos. De este modo, algunos hilos podrían quedar ociosos mientras otros todavía tienen trabajo pendiente, disminuyendo la aceleración que se podría obtener al paralelizar.

Otra aproximación es el reparto dinámico de las ocurrencias de evento, de modo que cada hilo tiene accesible el conjunto completo de ocurrencias de evento, e itera sobre dicho conjunto sincronizándose con el resto de hilos. En cada iteración, el primer hilo que queda ocioso toma la siguiente ocurrencia de evento del conjunto para procesarla. Esta aproximación, aunque mucho más efectiva que la aproximación estática, supone un mayor coste de sincronización.

En este caso concreto se ha optado por una aproximación intermedia. Todos los hilos disponibles tienen accesible de manera sincronizada el conjunto completo de ocurrencias de evento todavía por procesar, e iteran sobre dicho conjunto. Pero en este caso, en lugar de tomar una única ocurrencia de evento en cada iteración, toman N, siendo N mayor que uno. El primer hilo que quede ocioso podrá acceder al conjunto de ocurrencias de evento aún por procesar, para obtener las N siguientes. De este modo se reduce el coste de sincronización sobre el conjunto de ocurrencias, manteniéndose las ventajas del reparto dinámico.

En la versión paralela del algoritmo de *matching* que se ha implementado, se ha hecho uso de un *pool* de hilos de ejecución (*ThreadPool*[3]). Este mecanismo provisto por el *framework* .NET 3.5 de Microsoft[4] gestiona de manera automática la creación y destrucción de hilos de ejecución de manera eficiente, así como la asignación de tareas a dichos hilos. Por cada hilo creado dentro de un *ThreadPool* se debe indicar qué tarea debe realizar. En este caso, se ha establecido como tarea a realizar el acceso sincronizado a la lista de ocurrencias de evento pendientes, y la búsqueda de coincidencias para cada una de estas ocurrencias de evento. Cuando se han procesado todas las ocurrencias de evento pendientes, el hilo principal, que hasta entonces había quedado suspendido, retoma el control de la ejecución. Tras

[3]http://msdn.microsoft.com/es-es/library/system.threading.threadpool%28v=vs.90%29
[4]http://msdn.microsoft.com/es-es/library/w0x726c2%28v=vs.90%29

este proceso, se obtiene, como era de esperar, el conjunto de correspondencias entre ocurrencias de evento y reglas. Si este conjunto contiene elementos, se lleva a cabo el proceso de evaluación del consecuente de la regla.

Para este experimento se han realizado diversas iteraciones, en cada una de ellas variando el número de hilos a lanzar entre 1 (versión secuencial) y 10. Para cada iteración, se han realizado diversas pruebas variando el número de reglas en el ecosistema entre 1.000 y 25.000, y se han lanzado tantas ocurrencias de evento como reglas disponibles. Lo que interesa en este experimento es determinar qué sucede al aumentar el número de reglas sobre las que buscar *matchings*. En cada iteración se ha determinado el tiempo medio del proceso de *matching*. Este tiempo puede entenderse como el tiempo total que se tarda en procesar todas las ocurrencias de evento pendientes dividido por el número total de ocurrencias de evento procesadas. Esto indicaría el tiempo medio que se tardaría en realizar el *matching* de una ocurrencia de evento si el proceso fuera secuencial. De este modo, se han podido comparar las versiones paralelas con la versión secuencial, determinando la aceleración obtenida en cada caso.

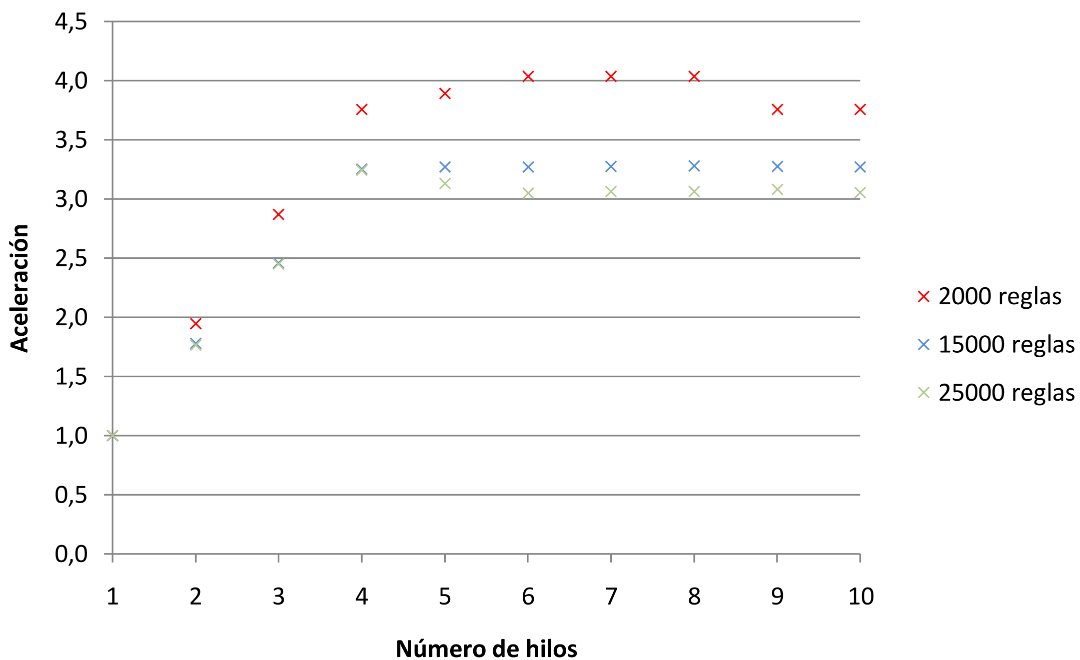


Figura 46. Aceleración en función del número de hilos empleados, para distinto número de reglas en el ecosistema.

En la Figura 46 se reportan los resultados obtenidos ejecutando estos experimentos con un procesador Intel Core2 Quad Q9550 que dispone de 4 núcleos físicos. Se observa que, para cualquier número de reglas en el ecosistema, a partir de 4 hilos la aceleración ya no mejora. Esto es normal, ya que si se dispone de 4 núcleos físicos, y se lanzan más de 4 hilos, varios hilos van a tener que compartir un mismo núcleo del procesador, y por tanto estos cambios de contexto entre hilos no permiten que la aceleración siga aumentando.

Para un número de hilos entre 1 y 4, se observa que la aceleración conseguida con 2000 reglas en el ecosistema, para un número de hilos menor al número de núcleos físicos disponibles, es prácticamente la aceleración teórica máxima posible. A medida que aumenta el número de reglas en el ecosistema, la pendiente

de crecimiento de la aceleración disminuye. Las diferencias observadas en la evolución de la aceleración entre ecosistemas con distinto número de reglas indican que, en general, para ecosistemas con menos de 10.000 reglas la aceleración máxima alcanzada es de 4, y para ecosistemas con más de 10.000 reglas la aceleración máxima alcanzada es aproximadamente de 3.

5.3.2. Paralelización del proceso de evaluación

Dado el conjunto de correspondencias entre ocurrencias de evento y reglas obtenido tras el proceso de *matching*, se debe realizar el procesamiento de cada una de las reglas que se han activado para evaluar su consecuente. Así, para cada regla que se indique en el conjunto de *matchings* encontrados, se debe realizar la misma tarea: filtrar la población destino y ejecutar la operación de la regla sobre cada una de las entidades de la población destino que pase el filtrado.

Al igual que se ha planteado paralelizar el proceso de matching, de igual modo se ha decidido paralelizar este proceso de evaluación de reglas. Para este caso, la carga a repartir entre los distintos hilos de ejecución son las reglas pendientes de evaluación, que hasta el momento se han consumido de manera secuencial.

El reparto de la carga entre los distintos hilos se realiza de nuevo empleando una aproximación mixta como la descrita en el apartado anterior. De este modo, todos los hilos tienen acceso al conjunto de reglas pendientes de evaluación, y el primer hilo que quede ocioso tomará las siguientes N reglas a evaluar.

Para este experimento se ha empleado de nuevo la estructura *ThreadPool*, descrita en el apartado anterior, para la gestión automática de la creación y destrucción de hilos de manera eficiente. La tarea asignada a cada hilo incluye el acceso sincronizado a la lista de reglas pendientes de evaluación, y la evaluación de dichas reglas.

En una primera versión de este experimento, se trató de paralelizar la evaluación de reglas cuya operación era una asignación a propiedad. Se ha encontrado que la paralelización de este tipo de reglas no obtiene resultados demasiado satisfactorios. Esto es debido a que la asignación del valor calculado a una propiedad conlleva el acceso a las estructuras de datos de las entidades para realizar una escritura, de modo que este acceso debe realizarse de manera exclusiva para evitar condiciones de carrera. Es por ello que se optó por realizar esta experimentación en reglas cuya operación es la ejecución de una acción. Este tipo de operación no requiere del acceso compartido para escritura a ninguna estructura de datos, ya que los parámetros de la acción a ejecutar son específicos de cada regla. Se han considerado, por simplicidad, acciones con un único parámetro de entrada.

Se han realizado diversas iteraciones variando el número de hilos disponibles entre 1 (versión secuencial) y 10. Para cada número de hilos, se han realizado diversas pruebas considerado la variación de la complejidad del proceso de datos de la operación a realizar. Retomando la definición de este proceso de datos como un árbol binario perfecto con K niveles, tal y como se describió en el apartado 5.2.2, para cada número de hilos se ha variado este número de niveles entre 0 y 5. Se ha fijado el número de tipos de entidad, el número de entidades por tipo y el número de tipos de evento en 4, 1.250 y 1 respectivamente. Se ha definido una

regla por cada entidad del ecosistema, para el único evento que existe. Estas reglas se han especificado con poblaciones fuente específicas y poblaciones destino genéricas. Al tener poblaciones destino genéricas, el filtrado y ejecución de la operación de la regla será más costoso, y se apreciará mejor la mejora obtenida con el procesamiento paralelo. Se han inyectado 5.000 ocurrencias de evento en el sistema. Cada una de estas ocurrencias activa una regla, de modo que al llegar al proceso de evaluación de reglas se dispone de un conjunto de reglas a evaluar de tamaño 5.000.

En cada iteración se ha determinado el tiempo medio del proceso de evaluación. Este tiempo puede entenderse como el tiempo total que se tarda en procesar todas las reglas a ejecutar obtenidas tras el proceso de *matching*, dividido por el número total de reglas que se han ejecutado. Esto da una idea del tiempo medio que se tardaría en realizar la evaluación del consecuente de una regla si el proceso fuera secuencial. De este modo, se han podido comparar las versiones paralelas con la versión secuencial, determinando la aceleración obtenida en cada caso.

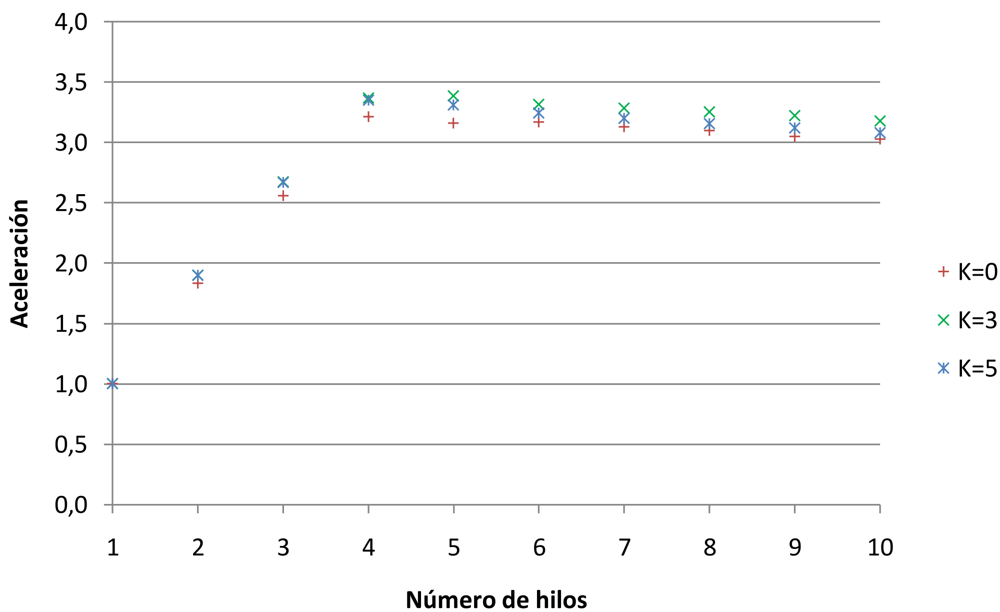


Figura 47. Aceleración en función del número de hilos empleados, para distinto número de niveles del proceso de datos.

Se ha ejecutado el algoritmo sobre un procesador Intel Core2 Quad Q9550 que dispone de 4 núcleos. En la Figura 47 se observa que la evolución de la aceleración es la misma para distintos niveles del proceso de datos, siendo aproximadamente 3,4 la aceleración máxima obtenida. En todos los casos considerados, a partir de 4 hilos de ejecución ya no se obtiene mejoría puesto que hay más hilos que núcleos, y las tareas de cambio de contexto introducen una sobrecarga que impide mejorar mediante la paralelización. En la Figura 48 sí que se observa variación en el tiempo medio del proceso de evaluación. Para cada nivel K considerado, a medida que aumenta el número de hilos disponibles, el proceso de evaluación se realiza más rápidamente, ya que la carga de procesamiento de reglas se divide entre los hilos disponibles. No obstante, cuanto mayor sea el número de niveles, mayor es el tiempo medio del proceso de evaluación como cabría esperar. Sin embargo, la mejora que introduce el algoritmo paralelo es la misma en todos los casos. Por tanto se puede concluir que el número de niveles del proceso de datos, aunque

afecte al tiempo medio de evaluación, no afecta a la aceleración obtenida con el procesamiento paralelo de las reglas para un número razonable de hilos como el que se ha considerado en las pruebas.

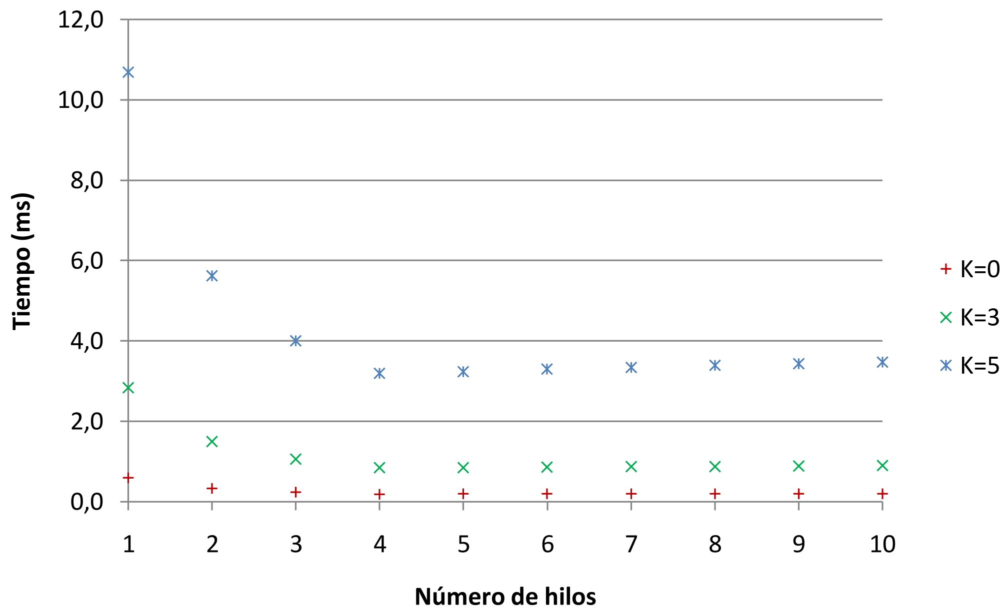


Figura 48. Tiempo medio del proceso de evaluación en base al número de hilos empleados, para distinto número de niveles del proceso de datos.

Se ha llevado a cabo otro experimento relacionado con el procesamiento paralelo de reglas para su ejecución. Dado que el número de niveles del proceso de datos no afecta a la aceleración, se ha buscado otro parámetro que pueda afectar. En este caso, se ha fijado la complejidad del proceso de datos en K = 3, el número de tipos de entidad en 4, y el número de tipos de evento en 1, y se ha optado por variar el número de entidades de cada tipo entre 125 y 2.500. Se ha definido una regla por cada entidad del ecosistema, cuyo evento es el único tipo de evento disponible en el ecosistema. La población fuente de estas reglas se define de manera específica, mientras que la población destino se describe como un tipo de entidad. Se han lanzado tantas ocurrencias de evento como reglas hay en el ecosistema, y cada ocurrencia de evento ha activado una única regla, de modo que el número de reglas activadas en cada iteración puede calcularse como E * T * I, siendo E el número de tipos de evento, T el número de tipos de entidad, e I el número de instancias por tipo de entidad. El propósito de variar el número de entidades por tipo es hacer más costoso el filtrado y la ejecución de la operación de una regla, ya que cuantas más entidades haya en la población destino más tiempo se empleará en ejecutar la regla.

La Figura 49 muestra la aceleración obtenida en base al número de hilos empleados, para diferente número de entidades por tipo. Como hasta ahora, a partir de 4 hilos, la aceleración ya no aumenta debido a que hay más hilos que núcleos en el procesador. En cuanto al rango entre 1 y 4 hilos de ejecución, se puede observar que para 125 entidades por tipo, lo que haría un total de 500 entidades en el ecosistema, y por tanto, 500 reglas a ejecutar, la aceleración máxima obtenida es de 2. A medida que aumenta el número de entidades por tipo en el ecosistema, la aceleración aumenta cada vez más rápido, hasta que llega un

punto en el que la pendiente de la recta de crecimiento de la aceleración ya no aumenta, por más que se incremente el número de entidades. En este caso, la aceleración máxima obtenida es de aproximadamente 3,4.

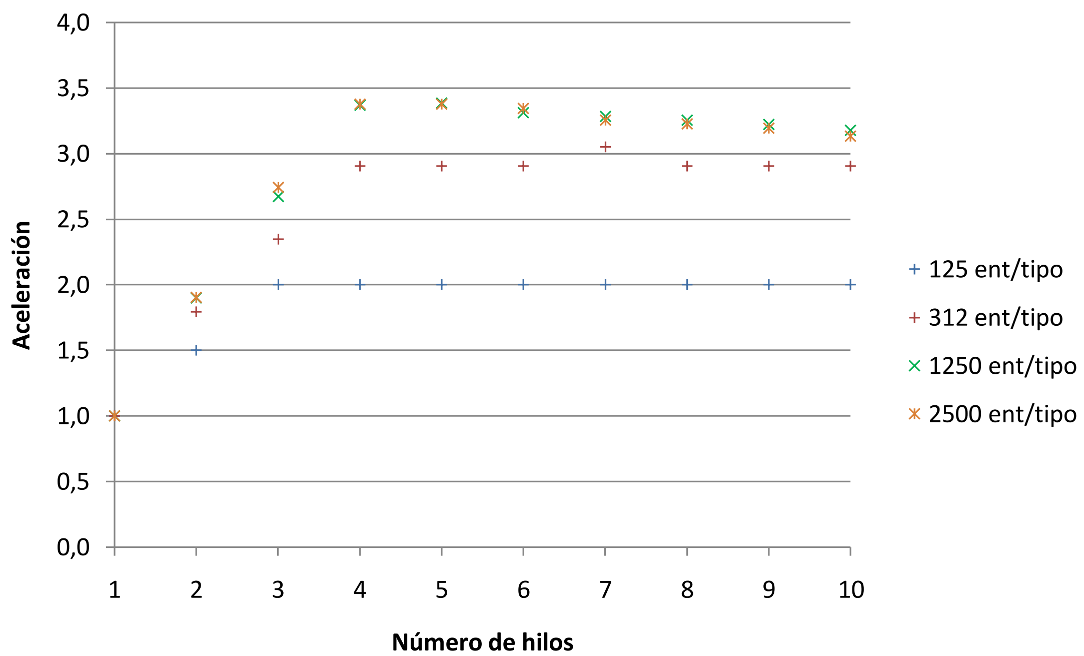


Figura 49. Aceleración según el número de hilos disponibles, para distinto número de entidades por tipo.

6. Conclusiones

En el presente documento se ha presentado un metamodelo que permite la edición de reglas de comportamiento de manera genérica, que puedan ser empleadas en varios dominios de aplicación que requiera la especificación de comportamiento. El empleo del metamodelado para la definición del lenguaje de especificación de reglas permite generalizar la construcción de una regla, abstrayendo el dominio de aplicación. Esto ha facilitado su aplicación a dos dominios inicialmente diferentes, como el de la personalización de entornos de Inteligencia Ambiental, y la especificación de comportamiento reactivo de entidades de juego. Además, ha permitido expresar reglas no solo para entidades específicas sino para poblaciones completas de entidades, lo cual supone mayor fluidez y rapidez en la edición si lo que se quiere es que todas las entidades de un tipo se comporten igual. Por otro lado, gracias al metamodelado es posible generalizar los conceptos y operadores que se involucran en los procesos de datos, de modo que se ofrece una alta expresividad dadas las numerosas combinaciones posibles de elementos que pueden emplearse.

Se ha mostrado el prototipo de un editor que permite la edición intuitiva de reglas de comportamiento basadas en este metamodelo sin requerir la escritura de código textual por parte del usuario. Mediante este prototipo se ha podido comprobar la validez del modelo de reglas, así como determinar que la expresividad permitida por el lenguaje de reglas propuesto alcanza los propósitos deseados: el mecanismo de definición de reglas de manera visual resulta sencillo de comprender para usuarios sin conocimientos previos de programación, pero no se pierde expresividad por ello, ya que la aproximación mediante flujos de datos hace posible combinar operadores entre sí, mientras que el metamodelado de los procesos de datos permite ampliar la cantidad de operadores disponibles en función del ámbito de aplicación.

Por otra parte, se ha implementado un motor de procesamiento de eventos capaz de instanciar las reglas definidas con el editor, y ejecutar las operaciones pertinentes, realizando cambios en el ecosistema y permitiendo la evolución del mismo. Mediante los mecanismos de activación de reglas y de cálculo de procesos de datos se ha podido comprobar que el metamodelo definido, además de ser útil y expresivo en diversos ámbitos, permite definir transformaciones en los datos correctamente. Para ello se ha implementado una herramienta de depuración. Ésta permite la carga de ecosistemas de prueba y trazas de eventos, para facilitar el seguimiento de la simulación a través de una interfaz gráfica capaz de mostrar el estado de cada entidad en el ecosistema paso a paso. También se ha determinado mediante diversos experimentos que tanto el lenguaje de reglas propuesto como el procesador de eventos implementado son escalables y permiten prestaciones muy superiores a las necesarias en ámbitos comunes de definición de comportamiento.

El lenguaje de reglas propuesto así como el editor están siendo empleados actualmente por miembros del grupo ISSI[5] del DSIC para la especificación de reglas de comportamiento para juegos sobre superficies interactivas, facilitando la tarea de programar el comportamiento de las entidades en juego. El mecanismo de activación de reglas implementando también se está empleando en la simulación

[5]Grupo de Ingeniería del Software y Sistemas de Información. http://issi.dsic.upv.es/

de dichos juegos. También a raíz del trabajo expuesto en este documento, se han derivado un total de 3 artículos de investigación [26][27][28]. El primero de ellos publicado en una conferencia de ámbito nacional, en el que se ha descrito el lenguaje visual de reglas que se ha propuesto. El segundo, ha sido publicado en una conferencia internacional en el ámbito de inteligencia ambiental y el tercero se trata de un artículo de revista internacional indexada en el índice JCR, que está pendiente de notificación tras una primera revisión cuyo resultado ha sido (Accept with minor changes). Dichos artículos se listan a continuación:

- Pons, P., Catalá, A., Jaén, J., Mocholí, J.A.:DaFRule: Un Modelo de Reglas Enriquecido mediante Flujos de Datos para la Definición Visual del Comportamiento Reactivo de Entidades Virtuales. Actas de las Jornadas de Ingeniería del Software y Bases de Datos, "JISBD 2011", pp. 989-1002 (2011).
- Catalá, A., Pons, P., Jaén, J., Mocholí, J.A.: Evaluating User Comprehension of DataFlows in Reactive Rules for Event-Driven AMI Environments. V International Symposium on Ubiquitous Computing and Ambient Intelligence, "UCAmI'11", pp. 337-344 (2011).
- Catalá, A., Pons, P., Jaén, J., Mocholí, J.A., Navarro, E.: A Meta-Model for DataFlow-Based Rules in Smart Environments: Evaluating User Comprehension and Performance. Invited submission to special issue "Software Engineering for Ambient Intelligence" in Science of Computer Programming Journal (IF: 1.306, COMPUTER SCIENCE, SOFTWARE ENGINEERING Q2 [32/99]).

6.1. Trabajo Futuro

Como trabajo futuro queda, por una parte, el refinamiento del metamodelo de reglas propuesto de manera que permita solventar algunas carencias expresivas que se han detectado tras el uso exhaustivo del editor.

Por otra parte, se pretende implementar una versión definitiva del editor empleando mecanismos de interacción más intuitivos y novedosos. Dado que el lenguaje de reglas propuesto se orienta principalmente a dotar de facilidades a usuarios no expertos en la edición de reglas de comportamiento, lo ideal sería que la herramienta o interfaz que acompañe a este lenguaje de reglas y que permita la edición de las mismas sea lo más usable e intuitivo que sea posible. En los últimos años se han realizado numerosos avances en el campo de la investigación de la interacción hombre-máquina (HCI), dando lugar a nuevas plataformas y técnicas de interacción más naturales. En concreto, las Interfaces Tangibles de Usuario (ITUs), como por ejemplo las superficies interactivas, permiten una interacción más familiar para el usuario ya que la interacción se realiza con las manos, empleando metáforas para la selección o el desplazamiento de elementos que resultan extremadamente intuitivas. Además, existen otros métodos de interacción, como es el empleo de elementos tangibles, que complementan a la interacción con las manos, permitiendo asociar elementos o acciones a los tangibles. Por tanto, la versión definitiva del editor debería desarrollarse sobre una superficie interactiva.

Gracias al prototipo del editor se han podido detectar algunos requisitos de diseño a tener en cuenta en la versión definitiva del mismo sobre una superficie interactiva. En primer lugar, para explotar todas las posibilidades de interacción que estas superficies ofrecen, se desea que la edición se realice totalmente con las manos, combinando el uso de elementos tangibles.

En segundo lugar, los elementos de la regla, a saber fuente, destino y evento, deben ser claramente visibles en cualquier momento de la edición de una regla. Sería ventajoso poder desplazar estos elementos por el área de edición según convenga por cuestiones de espacio. Por ejemplo, en el caso de tener que editar una regla compleja o muy extensa, quizás una reestructuración de los elementos a lo largo del área de edición facilite la comprensión de esta regla y evite cometer errores.

A la vista de la complejidad que pueden llegar a alcanzar las reglas, queda patente que la decisión de dividir la edición de una regla a lo largo de varias vistas supone una ventaja de usabilidad y debe mantenerse, ya que el usuario se centra en la edición de un único proceso de datos simultáneamente. De este modo el objetivo de cada proceso de datos queda bien explícito y no se pierde el enfoque sobre lo que se desea conseguir en esa vista en particular.

Otra cuestión que se ha detectado es que en el área de edición deberían aparecer tan sólo las propiedades y atributos que se vayan a involucrar en la regla, o más concretamente, en el proceso de datos que se encuentre en edición. Así, se evita tener todas las propiedades de la fuente y todos los atributos del evento visibles en el área de edición, ya que lo más frecuente es que no se empleen todos ellos. De modo que queda más espacio para la distribución de los elementos por el área de edición, a lo cual se debe unir que estas propiedades y atributos deben poder situarse en cualquier punto del área de edición, al igual que se plantea para el resto de elementos presentes en el área de edición No obstante, se debe idear alguna manera de relacionar cada propiedad o atributo con el elemento de la regla que le corresponda, por ejemplo, mediante códigos de colores asociados a la visualización de los elementos.

En cuanto a los mecanismos de comprobación de errores, para la versión definitiva del editor deberían considerarse únicamente mecanismos on-line, de modo que en lugar de emplear una consola y botones, a lo largo de la edición se vayan reportando mediante códigos de colores cualquiera de los errores que se detecten. Por tanto, dado que la comprobación de tipos en la edición de flujos de datos ya era un mecanismo on-line, sería necesario incluir el resto de mecanismos off-line del prototipo del editor como mecanismos on-line en la versión definitiva: asignación de flujo de datos al nodo objetivo, procesadores de datos correctamente conectados, y detección de ciclos.

Para la exploración de colecciones, sería interesante disponer de controles novedosos, cuya funcionalidad se mantenga, pero que sean capaces de adaptarse a interfaces 360º.

7. Bibliografía

[1] Wing, J.M.: Computational thinking. Commun. ACM vol.49, no.3, pp.33–35 (Marzo 2006).

[2] Pane, J.F., Ratanamahatana, C.A., Myers, B.A.: Studying the Language and Structure in Non-Programmers Solutions to Programming Problems. International Journal of Human-Computer Studies, vol. 54, no. 2, pp. 237–264 (Febrero 2001).

[3] Good, J., Howland, K., Nicholson, K.: Young People's Descriptions of Computational Rules in Role-Playing Games: An Empirical Study. Visual Languages and Human-Centric Computing (VL/HCC), 2010 IEEE Symposium, pp. 67–74 (21–25 Sept. 2010).

[4] Resnick, M., et al.: Scratch: programming for all. Commun. ACM vol. 52, no. 11, pp. 60–67 (Noviembre 2009).

[5] Resnick, M., Martin, F., Sargent, R., Silverman, B.: Programmable Bricks: Toys to Think With. IBM Systems Journal vol. 35, pp. 443–452 (1996).

[6] Begel, A.: LogoBlocks: A graphical programming language for interacting with the world. Electrical Engineering and Computer Science Department. MIT, Cambridge, MA (1996).

[7] Kelleher, C., Pausch, R.: Using storytelling to motivate programming. Commun. ACM vol. 50, no. 7, pp. 58–64 (Julio 2007).

[8] Repenning, A., Ioannidou, A., Zola, J.: AgentSheets: End-User Programmable Simulations. Journal of Artificial Societies and Social Simulation, vol. 3, no. 3 (Junio2000).

[9] Gindling, J., et al.: LEGOsheets: A Rule-Based Programming, Simulation and Manipulation Environment for the LEGO Programmable Brick. VL '95 Proceedings of the 11th International IEEE Symposium on Visual Languages (1995).

[10] Gallardo, D., Julia, C.F., Jorda, S.: TurTan: A tangible programming language for creative exploration, In proc. Tabletops, pp. 89–92 (2008).

[11] Suzuki, H., Kato, H.: Interaction-level support for collaborative learning: AlgoBlock an open programming language. Int. conf. on Computer support for collaborative learning, pp. 349–355 (CSCL '95).

[12] Horn, M.S., Jacob, R. J. K.: Designing tangible programming languages for classroom use. TEI '07. ACM, New York, NY, USA, pp. 159–162 (2007).

[13] Weiser, M.: The computer for the twenty-first century. Scientific American, vol. 265, no. 3, pp. 94–104, (1991).

[14] Cook, D. J., Augusto, J. C., Jakkula, V. R.: Ambient Intelligence: Technologies, Applications, and Opportunities. Pervasive and Mobile Computing, vol. 5, no. 4 (Agosto 2009).

[15] Weis, T., Handte, M., Knoll, M., Becker, C.: Customizable Pervasive Applications. In Proc. of PerCom'06, pp. 239–244 (2006).

[16] Sohn, T., Dey, A. K.: iCAP: An Informal Tool for Interactive Prototyping of Context-Aware Applications. Extended Abstracts of CHI 2003, pp. 974–975 (2003).

[17] Zhang, T., Bruegge, B.: Empowering the User to Build Smart Home Applications. Second International Conference On Smart homes and health Telematics, Singapore (15–17 Agosto 2004).

[18] Bonino, D., Corno, F., De Russis, L.: A User-Friendly Interface for Rules Composition in Intelligent Environments. In Proc. of ISAMI2011, Advances in Intelligent and Soft Computing Series, Springer Berlin / Heidelberg, vol. 92, pp. 213–217 (2011).
[19] Beckmann, C., Dey, A. K.: SiteView: Tangibly programming active environments with predictive visualization. Technical Report IRB-TR-03-019, Intel Research Berkeley (Julio 2003).
[20] Dey, A. K., Hamid, R., Beckmann, C., Li, I., Hsu, D.: a CAPpella: programming by demonstration of context-aware applications. In Proc. of CHI '04, pp. 33–40 (2004).
[21] García-Herranz, M., Haya, P., Alamán, X.: Towards a Ubiquitous End-User Programming System for Smart Spaces. Journal of Universal Computer Science (JUCS), vol. 16, no. 12, pp. 1633–1649 (2009).
[22] Kelleher, C., Pausch, R.: Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. ACM Computing Surveys,vol. 37, no. 2, pp. 83–137 (Junio 2005).
[23] Kurtev, I., Bézivin, J., Aksit, M.: Technological Spaces: An Initial Appraisal. Int. Federated Conf. (DOA, ODBASE, CoopIS), Industrial track, Irvine (2002).
[24] Parr, T.: The Definitive Antlr Reference: Building Domain-Specific Languages, The Pragmatic Programmers, May 24, 2007.
[25] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.: Introduction to Algorithms, MIT pressand McGraw-Hill, 1990.
[26] Pons, P., Catalá, A., Jaén, J., Mocholí, J.A.:DaFRule: Un Modelo de Reglas Enriquecido mediante Flujos de Datos para la Definición Visual del Comportamiento Reactivo de Entidades Virtuales. Actas de las Jornadas de Ingeniería del Software y Bases de Datos, "JISBD 2011", pp. 989-1002 (2011).
[27] Catalá, A., Pons, P., Jaén, J., Mocholí, J.A.: Evaluating User Comprehension of DataFlows in Reactive Rules for Event-Driven AMI Environments. V International Symposium on Ubiquitous Computing and Ambient Intelligence, "UCAmI'11", pp. 337-344 (2011).
[28] Catalá, A., Pons, P., Jaén, J., Mocholí, J.A., Navarro, E.: A Meta-Model for DataFlow-Based Rules in Smart Environments: Evaluating User Comprehension and Performance. Invited submission to special issue "Software Engineering for Ambient Intelligence" in Science of Computer Programming Journal (Pendiente notificación tras una minor review) (IF: 1.306, COMPUTER SCIENCE, SOFTWARE ENGINEERINGQ2 [32/99]).

8. Anexo A: Gramática ANTLR para la definición textual de reglas

```
rules_def[Stage s] : rule_def[s] rules_def[s] | ;

rule_def[Stage s] :
      EMPTY_REACTIVE_RULE NAME ruleName=STRING
      {
            ReactiveRule rr = s.CreateReactiveRule();
            rr.Name = $ruleName.text;
            rr.Precondition = new PreconditionDataProcess();
            rr.Precondition.SetTargetNode();
            rr.FilterPrecondition =
                  new FilterPreconditionDataProcess();
            rr.FilterPrecondition.SetTargetNode();
            PropertyBinding pb = new PropertyBinding(rr);
            pb.SetDataProcess(new BindingDataProcess());
      }
      END_EMPTY_REACTIVE_RULE |

      REACTIVE_RULE NAME ruleName=STRING
      {
            ReactiveRule rr = s.CreateReactiveRule();
            rr.Name=$ruleName.text;
      }
      'EVENT' eventName=STRING
      {
            rr.EvDefinition =
                  ecoDef.FindEventDefinitionByName($eventName.text);
      }
      source_def[rr]
      target_def[rr]
      OPERATION binding_type[rr]
      {
            rr.Precondition = new PreconditionDataProcess();
            rr.Precondition.SetTargetNode();
            rr.Precondition.SetSource(rr.Source);
            rr.Precondition.SetTarget(rr.Target);
            rr.Precondition.SetEventDefinition(rr.EvDefinition);
      }
      precondition[rr]
      {
            rr.FilterPrecondition =
                  new FilterPreconditionDataProcess();
            rr.FilterPrecondition.SetTargetNode();
            rr.FilterPrecondition.SetSource(rr.Source);
            rr.FilterPrecondition.SetTarget(rr.Target);
            rr.FilterPrecondition.SetEventDefinition(rr.EvDefinition);
      }
      filter_precondition[rr]
      operationdataprocess_header[rr]
      END_REACTIVE_RULE
;

source_def [ReactiveRule rr] :
      'SOURCE' ENTITY entityTypeName=STRING'::'entityName=STRING
      {
            Entity ent =
                  FindEntity($entityTypeName.text,$entityName.text);
            rr.Source = new EntitySourcePopulation(ent);
```

```
	}
	|'SOURCE' ENTITY_TYPE entityTypeName=STRING
	{
		EntityType et =
			ecoDef.FindEntityTypeByName($entityTypeName.text);
		rr.Source = new EntityTypeSourcePopulation(et);
	}
;

target_def[ReactiveRule rr] :
	'TARGET' ENTITY entityTypeName=STRING'::'entityName=STRING
	{
		Entity ent =
			FindEntity($entityTypeName.text,$entityName.text);
		rr.Target = new EntityTargetPopulation(ent);
	}
	|'TARGET' ENTITY_TYPE entityTypeName=STRING
	{
		EntityType et =
			ecoDef.FindEntityTypeByName($entityTypeName.text);
		rr.Target = new EntityTypeTargetPopulation(et);
	}
;

precondition[ReactiveRule rr] :
	PRECONDITION_DATA_PROCESS
	nodeconditions_header[rr.Precondition]
	dataprocess_def[rr.Precondition,rr.EvBinding]
	END_PRECONDITION_DATA_PROCESS
	|
	{
		rr.Precondition.SetDefaultCondition();
	}
;

filter_precondition[ReactiveRule rr] :
	FILTER_PRECONDITION_DATA_PROCESS
	nodeconditions_header[rr.FilterPrecondition]
	dataprocess_def[rr.FilterPrecondition,rr.EvBinding]
	END_FILTER_PRECONDITION_DATA_PROCESS
	|
	{
		rr.FilterPrecondition.SetDefaultCondition();
	}
;

binding_type[ReactiveRule rr] :
	{EntityType targetET = rr.Target.GetEntityTypeFromPopulation();}
	( PROPERTY propertyName=STRING
	{
		PropertyDefinition prop =
		targetET.FindPropertyDefinitionByName($propertyName.text);
		PropertyBinding pb = new PropertyBinding(rr);
		pb.SetPropertyTarget(prop);
	}
	| ACTION actionName=STRING
	{
		Action action =
			targetET.FindActionByName($actionName.text);
		ActionBinding ab = new ActionBinding(rr);
		ab.SetActionTarget(action);
```

```
      });

operationdataprocess_header[ReactiveRule rr] :
      OPERATION_DATA_PROCESS dataprocesses_def[rr]
      END_OPERATION_DATA_PROCESS
      |;

dataprocesses_def[ReactiveRule rr] :
      {
            BindingDataProcess dp = new BindingDataProcess();
            if(rr.EvBinding is PropertyBinding)
            {
                  PropertyBinding pb=rr.EvBinding as PropertyBinding;
                  pb.SetDataProcess(dp);
                  dp.SetTargetNode(pb.PropertyTarget);
                  dp.SetSource(rr.Source);
                  dp.SetTarget(rr.Target);
                  dp.SetEventDefinition(rr.EvDefinition);
            }
      }
      dataprocess_def[dp,rr.EvBinding] dataprocesses_def[rr]
      | ;

dataprocess_def[DataProcess dp, EventBinding eb] :
      DATA_PROCESS parameter[dp,eb]
      NODES nodes_def[dp] END_NODES
      DATA_FLOWS dataflows_def[dp] END_DATA_FLOWS
      END_DATA_PROCESS
;

nodes_def[DataProcess dp] :
      ( node_def[dp] nodes_def[dp] | );

node_def[DataProcess dp] :
      ( bounded_property_def[dp] | dpi_def[dp] | constant_def[dp] );

bounded_property_def[DataProcess dp] :
      BOUNDED_PROPERTY
      ENTITY ename=STRING
      TYPE tname=STRING
      {Entity ent=FindEntity($tname.text,$ename.text);}
      PROPERTY pname=STRING
      {
            PropertyDefinition pd = (ent.GetMetaType() as
            EntityType).FindPropertyDefinitionByName($pname.text);
            BoundedPropertyDefinitionSource bpds =
            dp.GetBoundedPropertyNodeByProperty(pd, ent);
      }
      position_def { bpds.Position = $position_def.pos; }
      END_BOUNDED_PROPERTY
      ;

dpi_def[DataProcess dp] :
      DATA_PROCESSOR_INSTANCE
      NAME name=STRING
      TYPE type=STRING
      { DataProcessor dProcessor =
            EcosystemDefinition.FindDataProcessorByName($type.text); }
      position_def
      { dp.AddDataProcessorInstance(dProcessor, $position_def.pos,
            $name.text); }
```

```
	END_DATA_PROCESSOR_INSTANCE
	;

position_def returns [float[\] pos] :
	POSITION v0=(FLOAT | INT) v1=(FLOAT|INT)
	{
		pos = new float[2];
		pos[0] = float.Parse($v0.text);
		pos[1] = float.Parse($v1.text);
	}
	|{ pos = null; }
	;

constant_def[DataProcess dp] :
	CONSTANT_PROVIDER
	NAME name=STRING
	TYPE type=STRING
	{MetaType mt = SelectPrimitiveType($type.text); }
	VALUE instance_value
	position_def
	{
		dp.AddConstantProviderNode
		($instance_value.i,$position_def.pos, $name.text);
	}
	END_CONSTANT_PROVIDER
;

dataflows_def[DataProcess dp] :
	( dataflow_def[dp] dataflows_def[dp] | );

dataflow_def[DataProcess dp] :
	DATA_FLOW FROM outputNode=STRING'::'outputPortName=STRING
	{
	Port outputPort = null;
	foreach(DataProcessNode dpn in dp.DataProcessNodeList)
	{
		if(dpn.Name == $outputNode.text)
		{
				outputPort=
					dpn.FindPortByName($outputPortName.text);
			if(outputPort!=null)
				break;
		}
	}
	}
	TO inputNode=STRING'::'inputPortName=STRING
	{
		Port inputPort = null;
		foreach(DataProcessNode dpn in dp.DataProcessNodeList)
	{
		if(dpn.Name == $inputNode.text)
		{
				inputPort=
					dpn.FindPortByName($inputPortName.text);
			if(inputPort!=null)
				break;
		}
	}
	dp.AddDataFlow(outputPort,inputPort);
	}
	END_DATA_FLOW
```

```
      ;

parameter[DataProcess dp,EventBinding eb] :
      PARAM STRING
      {
            BindingDataProcess bdp = dp as BindingDataProcess;
            ActionBinding ab = eb as ActionBinding;
            FormalParameter fp = ab.ActionTarget
                  .FindFormalParameterByName($STRING.text);
            ab.SetDataProcess(fp,bdp);
            bdp.SetTargetNode(fp);
            bdp.SetSource(eb.ReactiveRule.Source);
            bdp.SetTarget(eb.ReactiveRule.Target);
            bdp.SetEventDefinition(eb.ReactiveRule.EvDefinition);
      }
      |
      ;

nodeconditions_header[DataProcess dp] :
      NODE_CONDITIONS nodeconditions_def[dp] END_NODE_CONDITIONS | ;

nodeconditions_def[DataProcess dp] :
      ( nodecondition_def[dp] nodeconditions_def[dp] | );

nodecondition_def[DataProcess dp] :
      CONDITION nodecondition_type[dp] END_CONDITION ;

nodecondition_type[DataProcess dp] :
      PROPERTY STRING
      VALUE instance_value
      {
            char[] separator = new char[] {'.'};
            string[] split = $STRING.text.Split(separator);
            if(split[0]=="SOURCE")
            {
                  foreach (PropertyDefinitionSource pds in
                              dp.SourcePropertiesNodes)
                        if (pds.PropDefinition.Name == split[1])
                        {
                              dp.AddInstanceToSourceDataProcessNode
                                    (pds, $instance_value.i);
      break;
                        }
            }
            else if(split[0]=="TARGET")
            {
                  foreach (PropertyDefinitionSource pds in
                              dp.TargetPropertiesNodes)
                        if (pds.PropDefinition.Name == split[1])
                        {
                              dp.AddInstanceToSourceDataProcessNode
                                    (pds, $instance_value.i);
                              break;
                        }
            }
      }
      | ATTRIBUTE STRING
      {
            char[] separator = new char[] {'.'};
            string[] split = $STRING.text.Split(separator);
            EventAttributeDefinition ead = dp.GetEventDefinition()
```

```
                  .FindEventAttributeDefinitionByName(split[1]);
            EventAttributeDefinitionSource eads =
            dp.GetEventAttributeDefinitionSourceFromAttribute(ead);
      }
      VALUE instance_value
      {dp.AddInstanceToSourceDataProcessNode(eads,$instance_value.i);}
      ;

instance_value returns [Instance i]:
      ( str=QUOTED_STRING
            {i = new PrimitiveValue(RemoveQuotationMarks($str.text));}
      | INT { i = new PrimitiveValue(int.Parse($INT.text)); }
      | f=FLOAT { i = new PrimitiveValue(float.Parse($f.text)); }
      | v0=FLOAT v1=FLOAT { float[] vector = new float[2];
                        vector[0] = float.Parse($v0.text);
                        vector[1] = float.Parse($v1.text);
                        i = new PrimitiveValue(vector); }
      | bool { i = $bool.pv; }
      | entityTypeName=STRING'::'entityName=STRING
                { i=FindEntity($entityTypeName.text,$entityName.text);}
      );

bool returns [PrimitiveValue pv] :
      BOOLEAN
      {pv = newPrimitiveValue(String2Boolean($BOOLEAN.text));}
      ;
```