Lenguaje específico del dominio para generación de aplicaciones de procesos administrativos

Antonio García Domínguez, Ismael Jerez Ibañez e Inmaculada Medina Bulo

Departamento de Ingeniería Informática, Universidad de Cádiz
Av. de la Universidad 10, CP 11519, Puerto Real, Cádiz
antonio.garciadominguez@uca.es, ismael.jerezibanez@alum.uca.es,
inmaculada.medina@uca.es,
http://neptuno.uca.es/{~agarcia,~imedina}

Resumen Muchas organizaciones reimplementan una y otra vez el mismo tipo de proceso de negocio «administrativo», en el que un formulario es manipulado por múltiples roles a lo largo de varios estados. Esta reimplementación hace perder un tiempo que se podría haber usado en entender mejor el proceso o cubrir los detalles que sí son específicos del proceso. Por otro lado, las soluciones existentes basadas en motores de procesos de negocio requieren formación e infraestructura específicas y pueden encerrar al usuario en una tecnología concreta. En este trabajo se propone usar un lenguaje de alto nivel para describir el proceso administrativo y producir a partir de él un sitio web en un marco estándar de desarrollo web que sea fácil de mantener por los técnicos de la organización. Se ha implementado el enfoque mediante tecnologías de código abierto, y se ilustra a través de un caso de estudio.

1. Introducción

En muchas organizaciones, uno de los tipos de procesos de negocio más comunes es lo que llamaríamos un proceso «administrativo». Estos procesos administrativos consisten en gestionar un documento estructurado (generalmente un formulario) a lo largo de varios estados. En cada estado, distintas secciones del documento pueden ser visibles o editables por distintos roles en la organización, y los estados se suceden a través de plazos o decisiones tomadas a mano (posiblemente tras la reunión de un comité o algún tipo de negociación). El proceso suele concluir llegando a algún estado final («aceptado» o «rechazado», por ejemplo).

Estos procesos normalmente se realizan por completo dentro de una organización y no son especialmente complejos de por sí, pero su abundancia puede generar mucho trabajo repetitivo en su implementación. Rehacer cada uno de estos procesos desde cero requiere invertir tiempo en implementar las mismas características básicas de siempre (gestión de formularios y estados, integración con sistemas de autenticación corporativos y demás), que se podría haber usado en conocer mejor el proceso a través de un prototipado más rápido y atender los detalles puntuales del proceso que no se pueden estandarizar. En algunos casos, el desarrollador encargado de implementar el proceso no conoce todas las

capacidades del marco empleado, tardando más tiempo en obtener una versión inicial y obteniendo implementaciones poco satisfactorias.

En otros casos, el marco de desarrollo web escogido para implementar el proceso puede haberse quedado obsoleto hasta el punto de exigir una reescritura de la aplicación. Este fue el caso del marco Symfony, cuyo diseño cambió totalmente en las versiones 2.x y dejó a muchas aplicaciones sin correcciones de errores para el propio marco. A esto se le añade el hecho de que los procesos pueden haber sufrido cambios y retoques urgentes, con lo que no estarán bien documentados y su reimplementación requeriría un complicado proceso de ingeniería inversa. Habría sido mejor si la mayor parte del código se hubiera producido a partir de una descripción de alto nivel del proceso: cambiar la tecnología detrás de varias aplicaciones obsoletas entonces sólo exigiría escribir un generador distinto y traducir las personalizaciones puntuales.

Existe una variedad de motores de procesos de negocio generales y soluciones de oficinas virtuales, pero en cuanto el proceso requiere algún tipo de lógica personalizada sus soluciones crecen rápidamente en complejidad. Por otro lado, para una organización con un número limitado de desarrolladores como una universidad, exigirían una formación específica en una tecnología más que podría quedarse obsoleta o perder soporte a largo plazo. Sería mejor que el proceso resultante estuviera en un marco corriente de desarrollo web que estuviera estandarizado dentro de la organización, de forma que en todo momento hubiera varios técnicos disponibles para su mantenimiento.

En este trabajo se presenta un enfoque basado en un lenguaje específico de dominio («Domain Specific Language» o DSL) para describir esta clase particular de procesos de negocio y generar rápidamente una primera versión de una aplicación web que pueda ser mantenida sin conocimientos especiales. El lenguaje ha sido implementado mediante Eclipse Xtext [5], y se ha elaborado un primer generador a partir de sus descripciones en el Epsilon Generation Language [7], produciendo código para el conocido marco de desarrollo web Django[4]. El sitio web producido está listo para ser usado y puede ser mantenido por cualquier desarrollador familiarizado con Django.

El resto del trabajo se estructura de la siguiente forma: la sección 2 describe algunas de las herramientas existentes de modelado de procesos de propósito general y algunas soluciones más específicas para procesos administrativos o aplicaciones basadas en formularios, y las compara con el enfoque propuesto. La sección 3 presenta el lenguaje específico de dominio, empezando con los conceptos subyacentes y terminando con la notación textual adoptada. La sección 4 ilustra el enfoque aplicando el lenguaje y el generador implementados a un caso de estudio basado en un proceso de examinación, y evalúa sus capacidades y limitaciones actuales. Por último, la sección 5 proporciona una serie de conclusiones y lista las líneas abiertas de trabajo futuro.

2. Trabajos relacionados

Existe una variedad de motores de gestión de procesos de negocio («Business Process Management Systems» o BPMS) que incluyen soporte para pasos basados en formularios, como Bonita [3] o Intalio [11]. Estos motores suelen estar emparejados con una compleja herramienta de diseño que utiliza una notación gráfica (en muchos casos basada en BPMN) para describir el proceso. La notación gráfica se suele extender con anotaciones específicas del motor para completar la semántica del proceso, y las definiciones de los procesos se guardan generalmente en formatos basados en XML.

Aunque estos sistemas pueden describir un amplio conjunto de procesos de negocio, las definiciones resultantes dependen fuertemente del motor emparejado a la herramienta de diseño: migrar el mismo proceso a otra tecnología requerirá una reescritura completa. Se puede realizar control de versiones con formatos XML, pero hacer comparaciones intuitivas y reunir ramas que hayan manipulado distintas partes del proceso requerirán herramientas especiales. Por otro lado, usar uno de estos motores requiere un esfuerzo considerable de formación y consultoría, que puede no ser factible en organizaciones de menor tamaño. El enfoque propuesto en este trabajo trata de evitar esta complejidad en la notación y en la infraestructura centrándose en un tipo concreto de proceso de negocio (un proceso administrativo), y produciendo una aplicación web que usa un marco estándar ya conocido por el personal de la organización. Otro beneficio de emplear un DSL específico del tipo de proceso es poder incluir detalles que no entrarían en un lenguaje de procesos más general, como los controles de acceso de cada parte del formulario en función del estado y rol del usuario.

Ha habido más iniciativas aparte de los BPMS para simplificar el desarrollo de aplicaciones orientadas a formularios. El proyecto EMF Forms [6] de la fundación Eclipse es un ejemplo reciente. EMF Forms permite definir el modelo de datos y la estructura de alto nivel del formulario y generar una implementación en varias tecnologías, como SWT o JavaFX para el escritorio o Tabris para aplicaciones móviles. Mientras que EMF Forms cubre el aspecto de la presentación, el DSL propuesto en este trabajo se centra más en la lógica de estados y el control de acceso de las distintas partes del formulario. Se podría barajar la posibilidad de escribir un generador del DSL propuesto a EMF Forms, para combinar las ventajas de ambos enfoques.

Otra iniciativa relacionada es la del diseño dirigido por dominio («Domain Driven Design» o DDD). Este enfoque prioriza la implementación de un modelo del dominio representativo como núcleo del sistema y punto de unión del resto de componentes [9]. Apache Isis [1] u OpenXava [15] son dos marcos DDD que pueden producir una parte considerable de la aplicación a partir de un modelo del dominio «puro» (por ejemplo, un conjunto de «beans» Java, que se limitan a exponer su estado a través de pares de métodos setX/getX). Estas herramientas de DDD se parecen al DSL propuesto en tanto que tratan de producir una aplicación a partir de una descripción del modelo del dominio, pero el DSL está centrado en un problema específico y puede producir una mayor parte de la lógica requerida.

Centrándose en el contexto del gobierno electrónico, en la literatura actual se recogen varias diferencias en la forma en que los procesos de negocio de gobierno electrónico deberían diseñarse de forma distinta a los procesos de negocio del sector privado. Klischewski y Lenk afirmaron que en el sector público, muchos procesos requieren tomas de decisiones poco estructuradas (ya sea por una persona o por un comité) y negociaciones, por lo que es necesario que los gestores puedan alterar el flujo del proceso con mayor flexibilidad [12]. En el DSL propuesto, algunas de las transiciones de estados pueden basarse en decisiones tomadas por roles concretos. Los generadores podrían añadir la capacidad de que los administradores de las aplicaciones pudieran forzar ciertos cambios de estado en situaciones inesperadas.

Becker et al. propusieron un enfoque distinto para modelar procesos de gobierno electrónico, denominado PICTURE [2]. En vez de una notación gráfica de propósito general como BPMN, PICTURE define un proceso como un conjunto de secuencias alternativas de bloques reutilizables específicos del dominio. La secuencia concreta es escogida a partir de las circunstancias concretas de la instancia del proceso. Mientras que Becker et al. evitaron una notación clásica de diagramas de flujos para simplificar el análisis de procesos existentes, en el DSL propuesto se usa una notación textual para simplificar los generadores de código y reducir la cantidad de formación necesaria en su uso.

3. Definición del lenguaje

Resumiendo las anteriores secciones, los requisitos del lenguaje son:

- La descripción del proceso debería incluir la información a guardar, los roles que participan y los estados por los que pasa el documento.
- Cada estado debería indicar quién puede acceder a qué partes del documento, y de qué forma.
- El lenguaje debería permitir definir directamente transiciones basadas en plazos y/o decisiones, y debería ser posible integrar en el código generado lógica personalizada para ciertas transiciones.
- El lenguaje debería aportar suficiente información para generar una implementación inicial, y a la vez ser lo bastante sencillo para que crear un nuevo generador no sea muy costoso.
- El lenguaje debería permitir añadir anotaciones específicas del generador sin exigir cambios en su gramática.
- La edición y el control de versiones debería ser posible con herramientas estándar.

3.1. Sintaxis abstracta

La figura 1 muestra un diagrama de clases UML con la sintaxis abstracta del DSL propuesto. Una Application se divide en Elements, de los que hay cinco tipos. Site es el más sencillo: declara el nombre de la aplicación (por ejemplo, «Facturas»). Los elementos Options tienen pares clave/valor (instancias de

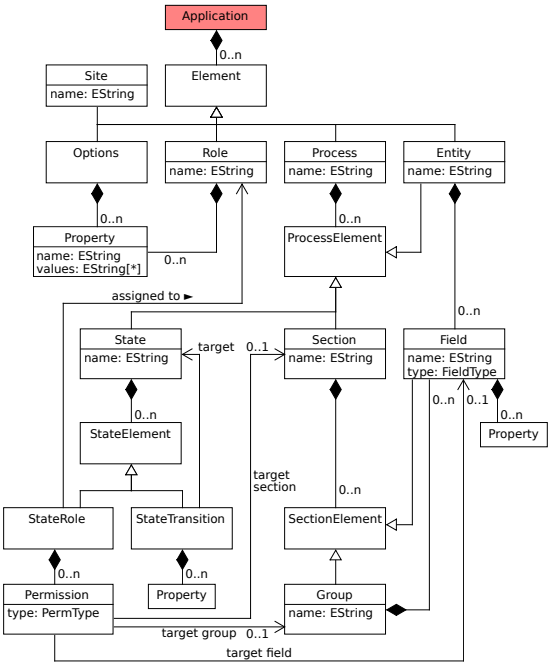

Figura 1. Diagrama de clases UML con la sintaxis abstracta del DSL propuesto.

Property) que pueden ser útiles para generadores externos. Estos pares/clave valor son utilizados también en otras clases del metamodelo, como Field, StateTransition: para ahorrar espacio, en esos casos se han omitido sus atributos.

Los Roles definen los distintos roles que intervienen de la organización, como «Contable» o «Gestor del Departamento». Estos elementos aportan un nombre y una serie de objetos Property que podrían ser usados por el generador para integrar el rol con sistemas internos de autenticación (como un directorio LDAP). Si no se indicase nada, los generadores podrían implementarlo como un rol interno de la aplicación, a gestionar a través de su interfaz de administración interna («back-office»).

Las instancias de Entity son entidades del modelo de datos que deben haberse creado antes de poder rellenar cualquier documento, como «País» o «Estado». Una Entity contiene Fields con la información a guardar acerca de sus instancias. Cada Field tiene un nombre y un tipo específico del dominio administrativo (como «moneda» o «documento de identidad») y cero o más instancias de Property que aportan información adicional a los generadores.

Finalmente, el tipo de elemento más importante y complejo es un Process. Cada instancia describe a un proceso completo (por ejemplo, «Petición de Baja»). Contienen a su vez varios tipos de ProcessElements:

- Instancias de ENTITY específicas del proceso, como sería el caso de RAZÓN-BAJA para el proceso de «Petición de Baja» si se quisiera limitar el conjunto de razones disponibles para la baja.
- Instancias de SECTION que organizan los FIELD del documento gestionado y se pueden subdividr opcionalmente en GROUPS.
  Por ejemplo, en el caso del proceso de facturas se podría tener la sección «Conceptos», y un grupo sería «Concepto» (con los campos «Precio», «Cantidad» y «Descripción», por ejemplo). Esta estructura es útil para generar código y para especificar reglas de control de acceso.
- Instancias de STATE, con los estados en los que puede estar el proceso. Un STATE tiene asociado un conjunto de STATEROLEs para cada rol de interés, que a su vez tienen instancias de PERMISSION con cada una de las acciones que un rol puede realizar sobre las distintas partes del documento en el estado en cuestión. Un rol puede recibir todos los permisos de una vez, o puede sólo editar o ver unas secciones, grupos o campos determinados.
  Un STATE también puede contener STATETRANSITIONS con transiciones a otros estados. Una transición puede activarse cuando todas las condiciones (especificadas mediante instancias de PROPERTY) se cumplen: estas condiciones podrían combinar decisiones explícitas por usuario («Aceptado» o «Rechazado»), fechas («Fuera de plazo») o lógica de negocio personalizada. Se podrían disponer varias reglas alternativas para ir al mismo estado usando múltiples STATETRANSITIONS.

Se han implementado varias reglas de validación sobre los modelos. Actualmente están en Xtend [8] (un dialecto de Java) y se están migrando a OCL:

- Debe haber exactamente un SITE.
- No se permiten dos ROLE o dos PROCESS con el mismo nombre.
- El nombre de un PROCESS sólo debe user caracteres alfabéticos.
- En un PROCESS, no puede haber dos SECTION con el mismo nombre.
- En un STATE, no puede haber dos STATEROLE hacia el mismo rol.
- En una SECTION, no puede haber dos elementos con el mismo nombre.
- En un GROUP, no puede haber dos FIELD con el mismo nombre.
- En un contenedor de PROPERTY, no puede haber dos con el mismo nombre.
- Debe haber al menos una SECTION definida por cada PROCESS.
- Debe haber al menos un estado «initial» por cada PROCESS.

3.2. Sintaxis concreta

Dado que la edición y el control de versiones no deberían exigir herramientas especializadas, se decidió utilizar una notación textual para el DSL. La notación textual es muy cercana a la sintaxis abstracta: por lo general, existe una correspondencia directa entre el concepto abstracto y su sintaxis.

El listado 1 muestra una gramática simplificada para la sintaxis concreta. De la forma habitual, `(x)+` significa «uno o más `x`», `(x)*` significa «cero o más `x`», `(x)?` quiere decir «cero o un `x`» y `x|y` es «`x` o `y`». El espaciado es ignorado, y

Listado 1. Sintaxis concreta del DSL propuesto.

```
site NombreSitio;
options { (Property;)* }

(role NombreRol { (Property;)* } | role NombreRol;)*
(entity NombreEntidad { (Tipo ((Property (, Property)*))? Nombre;)* })*

(process NombreProceso {
  (entity NombreEntidad { (Tipo ((Property (, Property)*))? Nombre;)* })*
  (section NombreSección {
     ( (email|choice|...otros FieldType...) ((Property (, Property)*))? Nombre;
     | group NombreGrupo { (Tipo ((Property (, Property)*))? Nombre;)+ }
     )*
  })+
  (state NombreEstado {
    (permissions NombreRol {
      ( (editable|viewable) all;
      | (editable|viewable) NombreSección;
      | (editable|viewable) NombreSección.NombreGrupo;
      | (editable|viewable) NombreSección.NombreCampo;
      | (editable|viewable) NombreSección.NombreGrupo.NombreCampo;
      )*
    })*
    (transition (Property (, Property)*) NombreEstado;)*
  })+
})*
```

las palabras clave y paréntesis literales se muestran en negrita. Una PROPERTY es de la forma `Nombre = Valor (, Valor)*`.

Dado que la gramática ocupa menos de 30 líneas en este formato simplificado, puede verse que es un lenguaje bastante sencillo que debería ser fácil de aprender. Sin embargo, tiene muchas referencias cruzadas, por lo que sería útil tener un editor que pudiese comprobarlas de forma continua. El editor debería disponer de autocompletado sensible al contexto, como en el momento de rellenar las referencias a un campo concreto para el control de acceso.

El analizador sintáctico se podría implementar de forma que no limitase los nombres de las PROPERTY asociadas a los distintos conceptos, de manera que se pueda aportar información al generador sin tener que cambiar la gramática. Cada generador documentará las opciones adicionales que aporte más allá de las definidas por el DSL en sí.

4. Caso de estudio

En esta sección se presentará un caso de estudio que utiliza el DSL para describir un proceso sencillo de examinación y generar una aplicación que lo implementa. Tras una descripción general del proceso, se detallarán los puntos más importantes de la implementación y se evaluarán los resultados obtenidos

4.1. Descripción

El proceso de este caso estudio modela un examen, con dos roles («estudiante» y «profesor») y estos pasos:

1. El estudiante inicia el proceso introduciendo su información personal y respondiendo a la primera parte del examen. Algunas de las preguntas son de respuesta libre, otras tienen respuestas predefinidas, y una de las preguntas saca las posibles respuestas de la base de datos.
   El profesor ya puede ver los exámenes parcialmente realizados, pero no puede introducir todavía notas.
2. Tras una cierta fecha, la segunda parte del examen (con dos preguntas numéricas) se hace visible y la primera parte ya no es editable por el estudiante. Los estudiantes pueden rellenar lo que piensan sobre el examen, y los profesores pueden seguir viéndolo todo, pero no pueden introducir notas todavía.
3. Tras una cierta fecha o si el estudiante lo indica explícitamente, el examen es «entregado» y ya deja de ser editable por el estudiante. El profesor ya puede poner la nota, pero no es visible todavía por el estudiante.
4. Una vez el profesor cierre la evaluación, el examen entra en estado «cerrado» y el estudiante ya puede ver la nota. Todos los campos quedan de sólo lectura.

El listado 2 muestra la descripción del proceso mediante el DSL propuesto. Algunas fechas y campos se han acortado por motivos de espacio. La línea 1 indica que la aplicación a generar se llama «Examen». Las líneas 2–5 incluyen varias opciones para el generador dirigido a Django. En concreto, sugieren usar una determinada plantilla base con la imagen corporativa, alojada en una «app» Django bajo una cierta dirección. La línea 7 declara los roles «estudiante» y «profesor» antes mencionados.

El resto del listado de la línea 9 en adelante está dedicado al proceso «examen». Una entidad llamada «Respuestas3» es declarada en la línea 11: sus instancias son las opciones a elegir por los estudiantes en la pregunta 3. Entre las líneas 13–32, los campos del documento se organizan en tres secciones. La sección «preguntas» se divide en dos grupos y un campo: mediante los grupos se organiza mejor la interfaz y se simplifican las especificaciones de control de acceso. Los campos opcionales tienen «blank» a `True`, ya que algunas preguntas pueden quedarse en blanco.

Los 5 estados están descritos entre las líneas 34 y 58. El estado «initial» es un caso especial: representa el estado previo al inicio del proceso como tal, y sus transiciones describen quién puede iniciar el proceso y cuándo. Los otros 4 estados coinciden con los pasos que se describieron anteriormente.

4.2. Implementación

El analizador sintáctico y editor para el lenguaje en la sección 3 han sido implementados mediante Xtext [5]. A partir de una gramática EBNF, Xtext produce un metamodelo Ecore[16] con la sintaxis abstracta del lenguaje y un editor avanzado con comprobación automática y coloreado de sintaxis, autocompletado, vistas de navegación y refactorizaciones sencillas. La figura 2 muestra una toma de pantalla del editor generado. El editor ha sido extendido manualmente para mejorar el autocompletado en la asignación de permisos.

Listado 2. Proceso de examinación descrito mediante el DSL propuesto.

```
site Examen;
options {
  django_base_template = "template/base.html";
  django_extra_apps = "template = https://.../";
}

role estudiante; role profesor;

process examen {

 entity Respuestas3 { string respuesta; }

 section personal {
  fullName nombreEstudiante;
  identityDocument(label="Documento identidad:") docid;
  email(label="Email") mail;
 }

 section preguntas {
  group parte1 {
   string(blank="True") p1;
   choice(values="R1,R2,R3", blank="True") p2;
   choice(table="Respuestas3", blank="True") p3;
  }
  group parte2 {
   currency(label="P4 (euros):", blank="True") p4;
   integer(label="P5 (entero):", blank="True") p5;
  }
  choice(values="Bien,Normal,Mal", blank="True") opinion;
 }

 section evaluacion { float nota; }

 state initial {
  transition(decision_by="estudiante", after_date="...", before_date="...") parte1;
 }
 state parte1 {
  permissions profesor   { viewable all;                          }
  permissions estudiante { editable personal, preguntas.parte1; }
  transition(after_date="...") parte2;
 }
 state parte2 {
  permissions profesor { viewable all; }
  permissions estudiante {
   viewable test.parte1;
   editable personal, test.parte2, test.opinion;
  }
  transition(decision_by="estudiante", before_date="...") evaluacion;
  transition(after_date="...") evaluacion;
 }
 state evaluacion {
  permissions profesor { viewable all; editable evaluacion; }
  permissions estudiante { viewable personal, preguntas; }
  transition(decision_by="profesor") cerrado;
 }
 state cerrado {
  permissions profesor { viewable all; }
  permissions estudiante { viewable all; }
 }
}
```

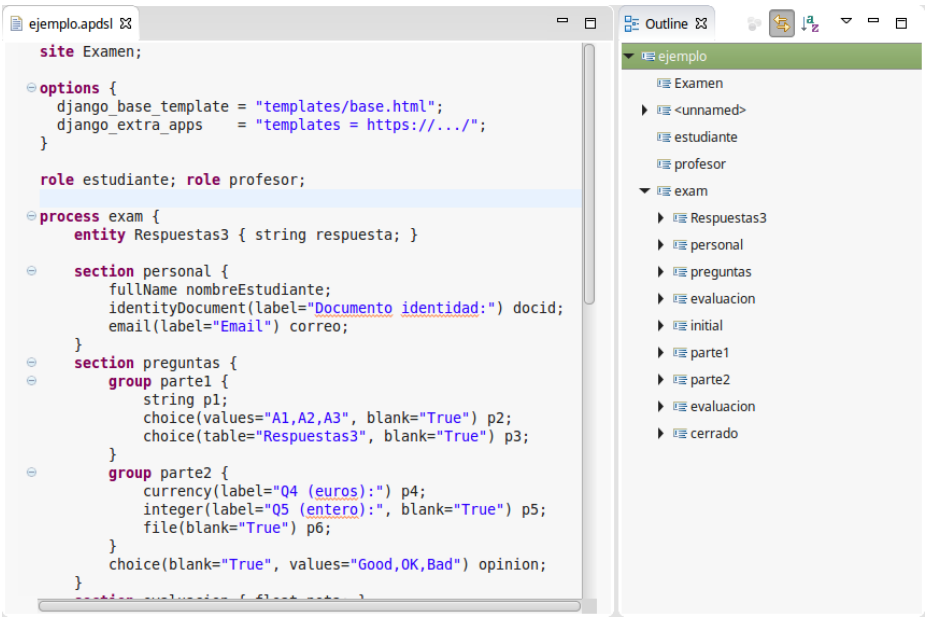

Figura 2. Toma de pantalla del editor basado en Xtext

Figura 3. Toma de pantalla de la aplicación web generada: lista de procesos

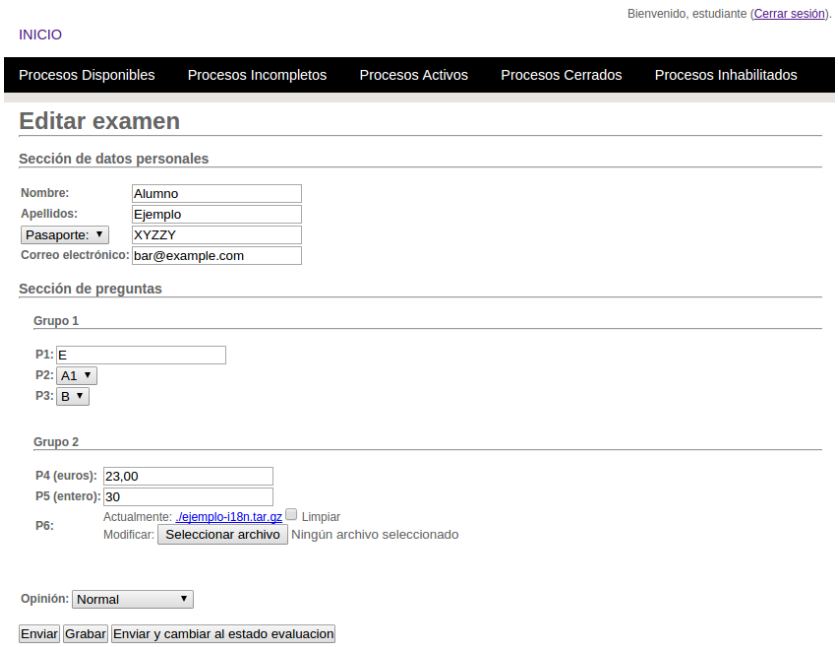

Figura 4. Toma de pantalla de la aplicación web generada: formulario del proceso

Se ha implementado un generador por separado que toma una descripción en el DSL propuesto y produce una aplicación web para el marco Django, usando una base de datos PostgreSQL. El generador está escrito en EGL («Epsilon Generation Language» [7]), que proporciona modularidad y la capacidad de tener «regiones protegidas» que se conservan al reescribir un archivo existente. Actualmente, el generador tiene 4847 líneas de código.

En el caso de la descripción de la sección 4.1, la aplicación generada ocupó 1344 líneas de código Python, 1527 líneas de plantillas HTML y unas 266 líneas de guiones de soporte y documentación. Estas líneas cubren todo lo necesario para guardar los formularios, validarlos, implementar los estados y los controles de acceso, poder traducir la interfaz a varios idiomas y guardar formularios parcialmente rellenados, entre otros aspectos. Aunque es muy probable que el código generado deba ser personalizado, los desarrolladores no tendrán que empezar desde cero y tendrán una base fundamentada en las prácticas recomendadas del marco, pudiendo iterar más rápido en la definición del proceso a implementar.

En las figuras 3 y 4 se puede ver el aspecto de la aplicación generada. El enfoque de la aplicación es el de una «oficina virtual»: tal y como se ve en la figura 3 el usuario dispone de una variedad de procesos «disponibles» (listos para iniciar), «activos» (iniciados y no en un estado final), «cerrados» (en un estado sin transiciones salientes) e «inhabilitados» (han dejado de ser iniciables).

Opcionalmente, los usuarios pueden dejar un formulario a medio rellenar y volver a él posteriormente: estos procesos son «incompletos».

Por otro lado, la figura 4 es el formulario que ve el estudiante en el estado «parte2». Dicho formulario dispone de tres opciones: «Enviar» normalmente el formulario, «Grabar» su estado actual sin enviarlo (dejándolo como un borrador) o «Enviar y cambiar al estado evaluación», por el que el alumno entregaría su examen antes del final del plazo. Este tercer botón está disponible gracias a la transición basada en decisiones que se especificó en la línea 48 del listado 2.

4.3. Limitaciones actuales

El DSL, el editor y el generador presentan actualmente varias limitaciones. Una limitación natural es que el código generado normalmente requerirá algún tipo de retoque manual, dada la necesidad de incluir alguna lógica particular de presentación o de negocio, o integrar la aplicación con algún sistema heredado imprevisto. Se ha seguido el enfoque aceptado en la literatura existente [10], en el que se prefiere tener un DSL más pequeño y centrado en algo particular. El generador para Django y otros futuros generadores tendrán que establecer los mecanismos necesarios para separar los retoques del código generado (p.ej. mediante las regiones protegidas antes mencionadas).

Hasta la fecha, sólo se ha evaluado la herramienta a través de casos de estudio basados en nuestras experiencias manteniendo varias aplicaciones internas de la Universidad de Cádiz. Una vez la herramienta esté lo suficientemente madura, se planea realizar experiencias con estudiantes o técnicos de la Universidad de Cádiz u otras organizaciones, comparando su productividad con y sin el DSL y analizando sus impresiones y sugerencias de mejora.

El DSL se centra en describir el proceso actual, y de por sí no tiene indicaciones para migrar procesos actualmente en ejecución a una nueva versión con estados o campos muy distintos. En esta versión inicial del DSL, se ha decidido delegar la miración de datos a la herramienta de migraciones que tenga el marco web: por ejemplo, Django implementa la infraestructura necesaria para realizar migraciones de la estructura de la base de datos y sus datos desde la versión 1.7. La migración de procesos y su monitorización quedan también delegados al entorno que produciría el generador.

La semántica de los procesos implementados sólo permite que esté activo un estado a la vez. Aunque esto hace que el DSL sea menos general que un lenguaje de modelado de procesos de negocio de propósito general como BPMN [14], basado en redes de Petri, o WS-BPEL [13], es equivalente a algunas de las plataformas de «oficina virtual» y aplicaciones heredadas que se encuentran en la Universidad de Cádiz.

Los estados no incluyen soporte actualmente para precondiciones, invariantes o postcondiciones que no sea la comprobación de los campos obligatorios oportunos. Se planea añadir soporte para las condiciones más comunes al DSL en futuras versiones: en la práctica, los casos más complejos se delegarán a una región protegida.

5. Conclusiones y trabajo futuro

Existe un tipo más sencillo de procesos de negocios (procesos «administrativos») que son muy comunes. Estos procesos consisten en gestionar un formulario a lo largo de varios estados y roles dentro de una organización. Implementar la lógica básica y la infraestructura para estos procesos una y otra vez malgasta un tiempo precioso que se podría aprovechar para entender mejor el proceso e implementar los detalles que son realmente específicos del proceso.

Este trabajo ha propuesto un enfoque para acelerar la implementación de estos procesos, a la vez que se evita quedar encerrado en una tecnología concreta. El enfoque consiste en describir el proceso con un lenguaje específico de dominio y usar un generador definido por separado para producir una aplicación web en el marco preferido por la organización. El enfoque se ha ilustrado describiendo un proceso de examinación, y ha sido implementado mediante Xtext y EGL. El generador produce una aplicación web que puede ser usada inmediatamente, siguiendo las prácticas recomendadas del marco web Django y dando rápidamente una base que retocar según las necesidades del usuario.

Los resultados mostrados tienen varias limitaciones. Es de esperar que el código generado tenga que ser personalizado, ya que un DSL no puede cubrir todas las posibilidades concebibles. Igualmente, el DSL no trata de reemplazar a un lenguaje completo de modelado de procesos como BPMN: está diseñado específicamente para procesos administrativos. Por último, el DSL sólo se ha evaluado hasta la fecha con casos de estudio internos.

Existen varias líneas de trabajo futuro. A medida que se desarrollen casos de estudio más avanzados, se expandirá el DSL con conceptos como límites en el número de procesos por usuario, el control de la visibilidad de procesos entre usuarios o las precondiciones y postcondiciones de los estados. Además de mejorar el editor y los generadores, se planea incluir validaciones continuas más avanzadas del proceso y visualizaciones gráficas. A medio plazo, se desea desarrollar generadores de código para otras tecnologías (como Symfony 2 o un motor BPMN) y realizar estudios empíricos de la utilidad del DSL con estudiantes y técnicos de dentro y fuera de la Universidad de Cádiz, evaluando su usabilidad y su productividad. A largo plazo, se está considerando desarrollar una notación gráfica para el lenguaje, procurando separar los detalles de presentación del diagrama del modelo en sí (que debería guardarse en el mismo formato textual actual).

Agradecimientos

Este trabajo ha sido financiado por el proyecto de investigación «Mejora de la calidad de los datos y sistema de inteligencia empresarial para la toma de decisiones» (2013-031/PV/UCA-G/PR) del Plan Propio de Investigación de la Universidad de Cádiz.

Referencias

1. Apache Software Foundation: Apache Isis. http://isis.apache.org/ (marzo 2015), fecha de última comprobación: 23 abril 2015.
2. Becker, J., Pfeiffer, D., Räckers, M.: Domain specific process modelling in public administrations–the PICTURE-approach. In: Electronic Government, Lecture Notes in Computer Science, vol. 4656, pp. 68–79. Springer (2007)
3. Bonitasoft: Página principal del proyecto Bonita BPM. http://www.bonitasoft.com/ (marzo 2015), fecha de última comprobación: 23 abril 2015.
4. Django Software Foundation: Página principal del marco web Django. https://djangoproject.com (marzo 2015), fecha de última comprobación: 23 abril 2015.
5. Eclipse Foundation: Página principal del proyecto Xtext. http://www.eclipse.org/Xtext/ (septiembre 2014), fecha de última comprobación: 23 abril 2015.
6. Eclipse Foundation: Página principal del proyecto EMF Forms. https://www.eclipse.org/ecp/emfforms/ (marzo 2015), fecha de última comprobación: 23 abril 2015.
7. Eclipse Foundation: Página principal del proyecto Epsilon. https://eclipse.org/epsilon/ (2015), fecha de última comprobación: 23 abril 2015.
8. Eclipse Foundation: Página principal del proyecto Xtend. http://www.eclipse.org/xtend/ (marzo 2015), fecha de última comprobación: 28 junio 2015.
9. Evans, E.J.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison Wesley, Boston, primera edn. (Aug 2003)
10. Fowler, M.: Domain Specific Languages. Addison-Wesley Professional, primera edn. (septiembre 2010)
11. Intalio, Inc.: Página principal del proyecto Intalio|BPMS. http://www.intalio.com/products/bpms/overview/ (marzo 2015), fecha de última comprobación: 23 abril 2015.
12. Klischewski, R., Lenk, K.: Understanding and modelling flexibility in administrative processes. In: Electronic Government, Lecture Notes in Computer Science, vol. 2456, pp. 129–136. Springer (2002)
13. OASIS: Web Service Business Process Execution Language (WS-BPEL) 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html (abril 2007), fecha de última comprobación: 24 abril 2015.
14. Object Management Group: Business Process Model and Notation 2.0.2. http://www.omg.org/spec/BPMN/2.0.2/ (enero 2014), fecha de última comprobación: 24 abril 2015.
15. OpenXava.org: Página principal del proyecto. http://www.openxava.org/web/guest/home (marzo 2015), fecha de última comprobación: 23 abril 2015.
16. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework. Addison-Wesley Professional, segunda edn. (diciembre 2008)