Two Case Studies on Generating Administrative Process Applications with AdminDSL

Antonio García-Domínguez[1], Ismael Jerez-Ibáñez[2], and Inmaculada Medina-Bulo[2]

[1] Department of Computer Science, University of York, York YO10 5GH, UK,
antonio.garcia-dominguez@york.ac.uk,
WWW home page: https://www-users.cs.york.ac.uk/~agd516/
[2] Department of Computer Science, University of Cadiz, Av. Universidad de Cádiz 10, Puerto Real 11519, Spain,
ismael.jerezibanez@alum.uca.es, inmaculada.medina@uca.es

Abstract. Some organizations end up reimplementing the same class of business process over and over: an "administrative process", which consists of managing a form through several states and involving various roles in the organization. This results in wasted time that could be dedicated to better understanding the process or dealing with the fine details that are specific to the process. Existing virtual office solutions require specific training and infrastructure and may result in vendor lock-in. In this paper, we propose using a high-level domain-specific language (AdminDSL) to describe the administrative process and a separate code generator targeting a standard web framework. We have implemented the approach using Xtext, EGL and the Django web framework, and we illustrate it through two case studies: a synthetic examination process which illustrates the architecture of the generated code, and a real-world workplace survey process that identified several future avenues for improvement.

Key words: model-driven engineering, domain-specific languages, business modeling, code generation

1 Introduction

In many organizations, there is a recurrent kind of business process which we call an "administrative process". These administrative processes involve managing a document with a certain structure and tracking it through different states. In each state, different parts of the document have to be viewable or editable by people with different roles in the organization. State transitions usually happen due to human decisions (possibly after a meeting, a review or some kind of negotiation), deadlines or a combination of both. The process usually concludes by reaching a "final" state (e.g. "accepted" or "rejected").

These processes are usually kept within a single organization and are not particularly complex by themselves, but their sheer number within some organizations can produce a high amount of repetitive work. Implementing each of

these processes from scratch wastes precious time on writing and debugging the same basic features (form handling, state tracking, internal directory integration and so on) which should have been invested in obtaining a better understanding of the process desired by the users and fine tuning the process-specific business logic. In some cases, the developer tasked with implementing the process is not familiar with some of the best practices of the target technology, needing more time and producing less than ideal solutions.

Another problem is that even after the process is correctly implemented, the framework the implementation is based upon may become obsolete to the point of requiring a complete rewrite. This is compounded by the fact that since the processes may have needed urgent changes and tweaks, they may not be well-documented anymore and may require careful reverse engineering, which is time consuming and prone to mistakes. It would have been much better if most of the code had been produced from a process description: changing the target framework of several obsolete applications would only require writing a different code generator and adding some customizations.

Our organization has evaluated various generic "virtual office" solutions for implementing these administrative processes and having them run in a high-level process engine. While these solutions were acceptable for the simplest cases, adding process-specific UI and business logic and integrating them with in-house systems would have required learning yet another technology that may become obsolete or lose support in the long run. It would be much better if the resulting implementations were based upon a standard web framework chosen by IT, so it could be maintained by regular staff.

This paper is an extended version of a BMSD 2015 conference paper [1]. In the original paper, we presented the first version of AdminDSL and provided a short description of the synthetic examination process and the generated Django [2] website. This version provides a better conceptual introduction to the language, a detailed description of the architecture of the generated code, and a discussion of which parts will usually need to be customized and how. It also provides a new real-world case study in which the authors collaborated with a contractor for the University of Cadiz to develop a workplace satisfaction survey web application. This case study covers not only the initial generation of the application, but also its later evolution until it was deployed to production and what was learned from it.

2 Related Work

There exist several business process management systems (BPMSs) that include some support for form-based steps within workflows, such as Bonita [3] or Intalio [4]. These engines are usually tightly integrated with a design tool which uses a graphical notation (usually BPMN-based) to describe the processes. This graphical notation is normally extended with engine-specific annotations to provide the full semantics of the process, and is persisted using XML-based formats.

While these systems can describe a much larger class of business processes than our DSL, the resulting process definitions are highly dependent on the underlying engine: migrating the same process to a new technology may require a rewrite. Version control is still possible with XML-based formats, but meaningful comparisons and merges require special-purpose tools. In addition, using a BPMS effectively requires a considerable amount of training, consulting and process analysis, which may not be feasible in smaller IT departments. Our approach focuses on a specific kind of business process (manage a complex document through multiple states, with different access rules in each state) and produces a standard web app that can be maintained as any other.

Scaling back from full-fledged BPMS engines, there have been many attempts to simplify the development of form-based applications. One recent initiative in this regard has been the EMF Forms [5] Eclipse project. Using its tools, users can define the domain model and abstract layout once and then render it in various technologies, such as SWT or JavaFX for desktop apps or Tabris for mobile apps. On the one hand, EMF Forms allows developers to specify the concrete layout for the resulting forms, unlike our DSL, which delegates on the generator for the presentation aspects. However, a DSL could cover aspects that EMF Forms does not, such as access control and state transitions.

Another related topic is domain-driven design (DDD): an approach on software development that asserts that the primary focus in most software projects should be the creation of an adequate model that abstracts the problem domain, and the implementation of the relevant business logic around it [6]. In this regard, several frameworks have been implemented to support DDD, enabling rapid iterations by generating a large portion of the application from a "pure" domain model (e.g. a set of Plain Old Java Objects or POJOs), such as Apache Isis [7] or OpenXava [8]. In a way, a DSL conforms to the same view of producing software from a description of the problem domain: the only difference is that the DSL is focused on a specific kind of problem domain, rather than the generic approach of a DDD tool, making it more productive in that particular case.

In the context of e-government, several works have identified some differences in the way e-government processes should be modeled, in comparison to the business processes in the private sector. Klischewski and Lenk argued that in the public sector, many processes involve unstructured decision making (whether by a single official or after a meeting) and negotiations, so officials would need to be able to alter the course of the process [9]. Later in the same work, Klischenski and Lenk proposed a set of "Admin Points" as reusable process patterns that could be used as a shared vocabulary for e-government business processes. In a DSL, many of these admin points which involve negotiation and unstructured decision making could be modeled using decision-based transitions.

3 Language Definition

This section will present AdminDSL, our domain-specific language for defining administrative processes. Before describing the abstract and concrete syntax in a more formal way, we will provide a conceptual introduction to the language:

- An administrative process is understood as a succession of states through which a structured document is viewed and edited by users with certain roles in the organization.
- A structured document contains a sequence of fields, which may be aggregated together into sections and then groups. Sections and groups are useful for presentation and for access control. In addition, groups can be repeated if desired, which may be useful for representing master/slave records such as order line items, student grades and so on.
- Roles are defined separately: users can belong to one or more of them, and the assignments are assumed to be changeable on the fly through the generated web applications. AdminDSL is decoupled of how these roles are actually implemented (e.g. through a database or an LDAP server), but it should allow generic key-value pairs that a generator might understand.
- Finally, a collection of states is defined for each process: the form is implicitly in the "initial" state when it is being created, and then immediately transitions to a different state. Later transitions can be done by user decision or by deadlines. In each transition, roles can be given permission to view or edit certain fields, groups or sections of the document. AdminDSL can limit permissions to the user with a certain role that participated in a previous transition: this allows for process isolation between users with the same role.

3.1 Abstract Syntax

Figure 1 shows a UML class diagram with the abstract syntax of our DSL (the underlying concepts). An Application is divided into Elements. There are five kinds of Elements. Site is the simplest one: it only declares the name of the application (e.g. "Billing"). Options elements contain key/name pairs (Property instances) that may be useful to external generators.

Next are Roles. These represent particular roles within the organization (such as "Accountant"). These elements provide a name and a set of Property instances which may be used by the generator to integrate the role with in-house user directories (such as an LDAP directory).

Entity elements represent data entities that must have been created before any documents can be filled in, such as "Country", "State" and so on. An Entity contains Field instances with the information to be stored about it. Every Field has a name and a domain-specific type (such as "currency" or "identity document") and zero or more Property instances providing additional information to generators.

Finally, the main and most complex kind of element is a Process. These represent entire administrative processes (e.g. "Request for Leave"). They can contain three kinds of ProcessElements:

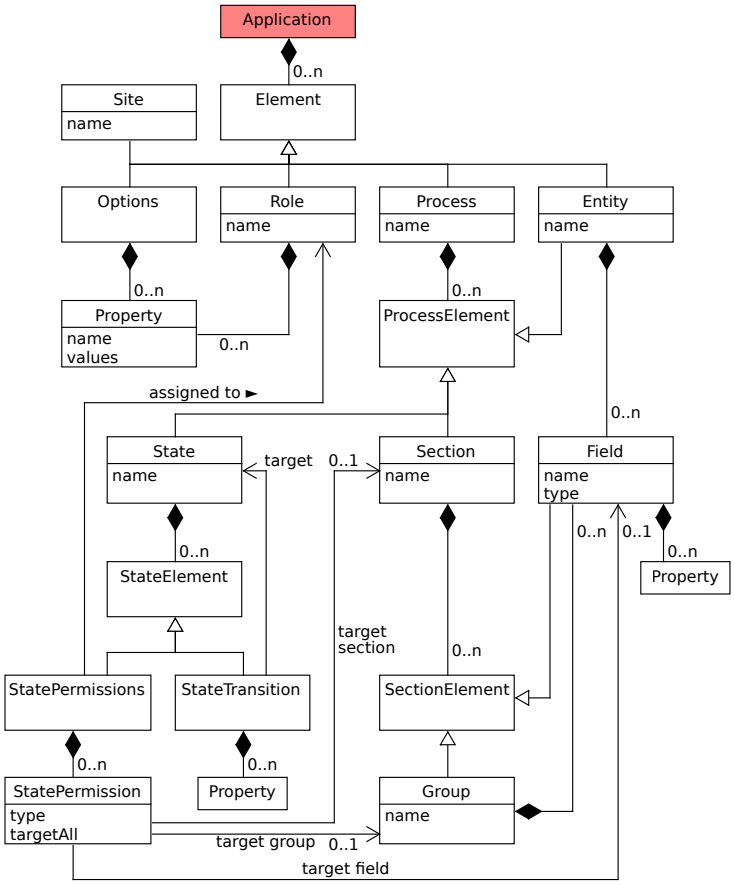

Fig. 1. UML class diagram of the DSL's abstract syntax.

– Entity instances, which will be specific to the process in this case. In the "Request for Leave" process, "LeaveReason" instances could be the various reasons for the leave.
– Section instances contain the Fields of the managed document, which can be optionally subdivided into Groups. A section could be "Billing information", and a group could be "bill item" (containing the "Price", "Quantity" and "Description" fields, for instance). This structure is useful for generating code and specifying access rules.
– State instances represent the states that the managed document can be in. It can contain a set of StatePermissions for each role of interest, which in turn contain StatePermission instances that describe the kinds of actions that are available to the role. A role can receive all permissions at once, or it can receive the ability to edit or view a single section, group or field.
A State can also contain StateTransitions to other states. A transition can activate when all the conditions (specified using Property instances) are

Listing 1. Simplified concrete syntax for our DSL

```
  site SiteName;
  options { (Property;)* }
  (role RoleName { (Property;)* } | role RoleName;)*
  (entity EntityName {
    (Type ((Property (, Property)*))? Name;)*
  })*

  (process ProcessName {
    (entity EntityName {
      (Type ((Property (, Property)*))? Name;)*
    })*
    (section SectionName {
      (Type ((Property (, Property)*))? Name;
      | group GroupName {
        (Type ((Property (, Property)*))? Name;)+
      } )*
    })+
    (state StateName {
      (permissions RoleName (from StateName->StateName)? {
        ((editable|viewable) all;
        | (editable|viewable) SectionName;
        | (editable|viewable) SectionName.GroupName;
        | (editable|viewable) SectionName.FieldName;
        | (editable|viewable) SectionName.GroupName.FieldName;
        )*
      })*
      (transition (Property (, Property)*) StateName;)*
    })+
})*
```

met: these conditions could combine explicit decisions by users ("Accepted" or "Rejected"), dates ("Past deadline") or custom business logic. Alternative paths to the same state can be modeled with several StateTransitions.

3.2 Concrete Syntax

Since editing and version control should not require any special-purpose tools, we have picked a textual notation for the DSL which is mostly a one-to-one mapping to the model entities.

Listing 1 shows a simplified version of our original EBNF grammar for the concrete syntax. As usual, `(x)+` means "one or more x", `(x)*` means "zero or more x", `(x)?` means "zero or one x" and `x|y` means "x or y". Whitespace is ignored. Literal `(` and `)` and keywords are shown in bold, to avoid confusion. A *Property* is of the form `Name = Value (, Value)*`.

As the grammar can fit into less than 40 lines after some simplifications, we can conclude that it is a rather simple DSL that should be easy to learn by the IT staff. However, it has a considerable number of cross-references in it, so it will require good tooling support to ensure that these references are not stale.

4 Case Study: Exam Process

In this section, we will illustrate AdminDSL by presenting a case study that describes a simple examination process and then generates a web application that implements it. After outlining the process itself, we will present the features, general architecture and design of the generated application, and evaluate the results obtained.

4.1 Description

The process in this case study is a test, involving two roles ("student" and "teacher") and these steps:

1. The student starts the test by introducing their personal information and answering the first part. Some of the questions are free-form, some have a predefined set of answers, and one of the questions pull answers from the database. Students cannot see each other's answers.
   The teacher can already see all the partially filled-in exams, but cannot enter any grades yet.
2. After a certain date, the second part of the test (with two numeric questions) becomes visible and the first part of the test is no longer editable by the student. Students can also fill in what they think about the test. Again, students cannot see each other's answers. Teachers can still see everything, but cannot enter any grades yet.
   The test can be turned in for examination before a certain deadline: after that deadline, it is turned in automatically and is no longer editable.
3. Once the test has been turned in, the teacher can grade it, but the student cannot see the grade yet: they only see their own answers.
4. After the teacher confirms the final grade, the examination is "closed". Every student can see their own answers and everyone's grades. All fields are now read-only.

A simplified version of the DSL-based description of the process is shown in Listing 2. Some of the dates and field names have been shortened to save space.

Line 1 declares that the application to be generated has the name "School". Lines 2–5 include several options for the code generator that targets the Django web framework: in particular, they suggest using a certain base template that follows the organizational image, which is included in a Django app available at a certain URL. Line 6 declares the previously mentioned "student" and "teacher" roles.

The rest of the listing from line 8 onwards is dedicated to the "exam" process. An entity "Answers3" is declared at line 9: its instances are used for the answers for question 3. In lines 10–27, the fields of the document are organized into three sections. The "test" section is divided into two groups and one additional field: using groups simplifies access control specifications later on. The optional fields have "blank" set to "True".

Listing 2. DSL-based examination process

```
site School;
options {
  django_base_template = "template/base.html";
  django_extra_apps = "template = https://.../";
  }
role student; role teacher;

process exam {
   entity Answers3 { string answer; }
   section personal {
    fullName studentname;
    identityDocument(label="National ID:") nid;
    email(label="Email") mail;
  }
  section test {
    group part1 {
      string(blank="True") q1;
      choice(values="A1,A2,A3",blank="True") q2;
      choice(table="Answers3",blank="True") q3;
    }
    group part2 {
      currency(label="Q4 (euros):",blank="True") q4;
      integer(label="Q5 (integer):",blank="True") q5;
    }
    choice(values="Good,OK,Bad",blank="True") opinion;
  }
   section evaluation { float grade; }

   state initial {
    transition(decision_by="student",
    after_date="2015/03/01-14:00:00",
    before_date="2015/03/07-14:00:00") part1;
  }
  state part1 {
    permissions teacher { viewable all; }
      permissions student from initial->part1 {
      editable personal, test.part1; }
    transition(after_date="...") part2;
  }
  state part2 {
    permissions teacher { viewable all; }
    permissions student from initial->part1 {
      viewable test.part1;
      editable personal, test.part2, test.opinion;
    }
    transition(decision_by="student",
    before_date="...") evaluation;
    transition(after_date="...") evaluation;
  }
  state evaluation {
    permissions teacher { viewable all;
      editable evaluation; }
    permissions student
    from initial->part1 { viewable personal, test; }
    transition(decision_by="teacher") closed;
  }
  state closed {
    permissions teacher { viewable all; }
    permissions student
    from initial->part1 { viewable all; }
      permissions student { viewable evaluation; } }
}
```

Lines 29–61 describe the 5 different states the process can be in. The "initial" state is a special case: it represents the state before the process starts, and its transitions describe who can start the process and when. The other 4 states match the four stages of the examination which were described before.

Line 36 illustrates the **`from`** `A->B` syntax that AdminDSL uses to limit the granted permissions to the user that triggered a certain state transition, providing process isolation between users with the same role. In this particular example, it ensures that students cannot see each other's answers. It is possible to have multiple **`permissions`** blocks for the same role, with different restrictions: for instance, the "closed" state allows students to see their own answers and each other's grades.

4.2 Implementation in AdminDSL

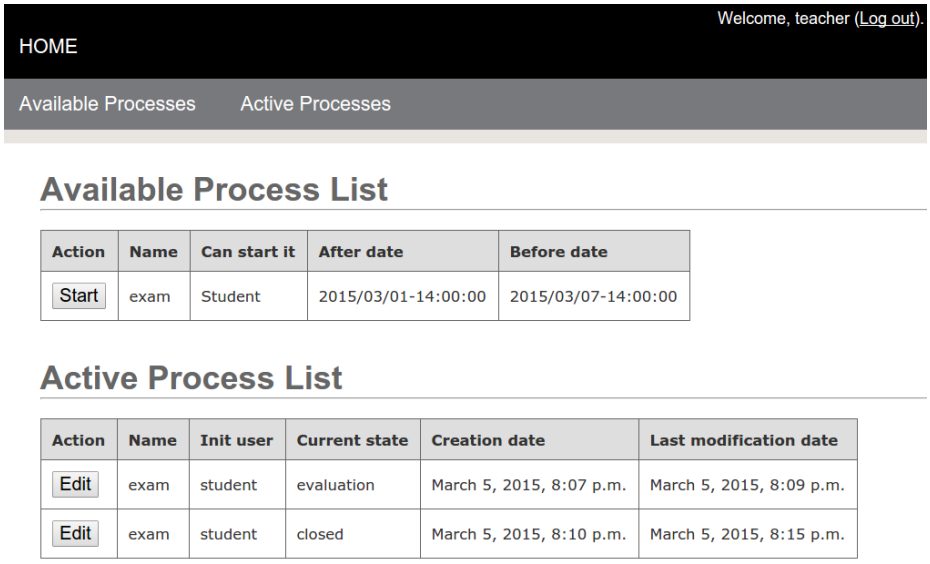

Fig. 2. Generated web app: process list

The parser and editor for the language in Section 3 have been implemented using Xtext [10]. From an EBNF grammar, Xtext generates a metamodel with the abstract syntax of the language and a set of Eclipse plugins which provide a parser and an advanced editor with live syntax checking and highlighting, autocompletion and an outline view.

We have also implemented a separate code generator that takes an Application described with our DSL and produces a web application in the Django framework. The generator is written in the Epsilon Generation Language [11], which provides modularity and the ability to have "protected regions" that are

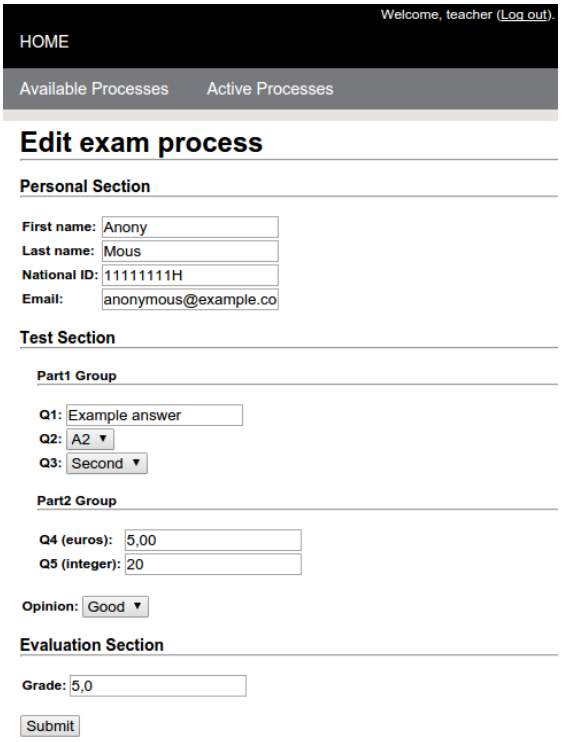

Fig. 3. Generated web app: process form

preserved when overwriting an existing file. Our current version of the EGL source code for the Django generator has 3045 LOC.

4.3 Generated application

The code generator produced from the example in Section 4.1 a ready-to-use Django site backed by a PostgreSQL relational database (shown in Figures 2 and 3). The generated site is largely divided into two parts:

- A support library that extends Django with the building blocks for implementing forms with fine-grained stateful permissions. This library was written manually in tandem with the code generator and is essentially copied into the project.
- A set of modules that are generated automatically, using the above support library to implement the processes defined in the AdminDSL description. Most elements in AdminDSL map directly to various combinations of the components in the support library.

This structure makes it possible to produce a high-quality web application based on a well-tested set of components, which can be rearranged by advanced

users that need a different user interface or additional functionality. The generated code can be more concise and thus be closer to what a developer would write manually.

This section will introduce the basic concepts behind the Django framework and present the key details of the support library and the generated modules. The section will then discuss the features included by default and the various ways in which the application can be extended.

Background: Django framework Django [1] is a web development framework that follows the Model-View Controller (MVC) pattern [12], providing a clear separation between the data managed by the application (the *models*), the ways in which users interact with this data (the *views*) and the logic that redirects incoming requests to the appropriate views (the *controllers*).

Django applications are modular, consisting of a set of *apps*: most applications combine a set of pre-built apps for common functionalities (e.g. forums or blogs) with a set of custom apps for application-specific logic (e.g. business logic). Every module can provide some of the following:

- *Models*: data entities that are persisted to the database. Most models are directly translated into database tables. Django provides its own object-to-relational mapping facilities and an incremental database migration framework.
- *Views*: Python functions that handle an incoming request for certain URLs in the web application. Most functions will then render a response by passing a *context* to a Django *template*.
  Django applications generally use a hierarchical approach for dispatching incoming requests to views: a global dispatcher forwards requests to an app-specific dispatcher, which then forwards the request to the appropriate view.
- *Forms*: in-memory representations of the data managed by a web form. These forms provide the required cleanup and validation logic. Developers can extend the Django form library with new fields and form types.
- *Administration pages*: apps can extend the built-in administration area with new pages for managing their models.
- *Signals and receivers*: some apps may want to react to events produced by other apps. In order to keep these apps decoupled, Django makes it possible to define *signals* that apps can send and *receivers* that can handle those signals.

The Django library also provides most of the basic functionality required by current web applications, such as user authentication and authorization, session management and internationalization, among others.

Support library The support library is a Django app (called "base_admindsl") that was manually written together with the Django code generator, making it possible to increase the abstraction level of the generated code and make it more readable and maintainable.

[1] https://www.djangoproject.com/

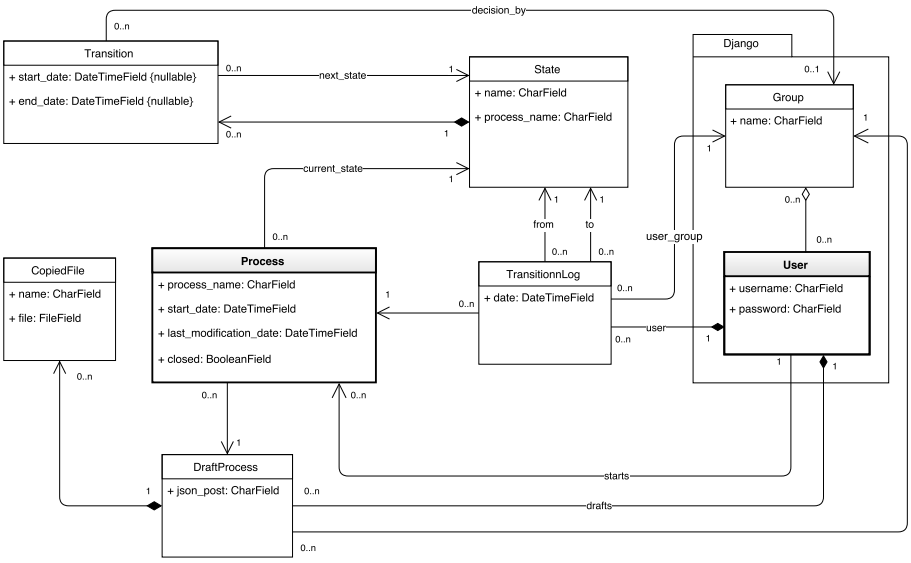

Fig. 4. Excerpt of the base Django models defined by the support library.

First, the support library contributes a set of Django models for persisting the processes and their instances. The most important ones are shown in the UML class diagram in Figure 4, and reuse some of the default models provided by the Django framework (shown in the "Django" package). The models are organized roughly into three groups:

- The Transition, State, and MaxProcessInstance models operate at the process description level. Every process has a collection of Transitions that link together various States. MaxProcessInstance models can limit the number of instances of a process that a certain group of users may create. These models allow administrators to configure parts of the processes without having to regenerate the code.
- The Process, BasePermSection, BasePermGroup, TransitionLog, DraftProcess, and CopiedFile models operate at the process instance level.
  Process is an abstract superclass for all the process models that represents a single instance of a process and stores the current state of the process, the start and last modification dates and whether it is closed (i.e. cannot trigger any more transitions).
  BasePermSection and BasePermGroup are abstract superclasses for all the generated models that implement the various sections and groups, respectively. These classes provide various utility methods that deal with permissions, such as `is\_editable`, `can\_view` and `can\_edit`.
  TransitionLog keeps track of every state transition triggered by a process instance, storing the date, source and target states, and the user and group

that triggered the transition. They are used to implement the process isolation restrictions that were shown in Listing 2 (see line 36).
DraftProcess and CopiedFile store draft process forms and uploaded files before they are persisted to the real models. Draft process forms are serialized as JSON data, so DraftProcess is kept decoupled from the structure of the real models.

The support library also contains various building blocks for the functionality required by the generated processes:

- Utility classes that extend the Django form handling components with new field types and the stateful permissions system required by AdminDSL.
- Generic views that can be reused from the app-specific dispatchers: these include user authentication, role switching and process management.
- Signals and default receivers for three kinds of events: process creation, manual process updates and automatic process updates.
- A command-line tool (`update_states`) that can be run periodically to perform automatic state transitions based on the current date.

Generated processes Every Process in the AdminDSL description is turned into a Django app that combines the building blocks provided by the support library to implement the described process.

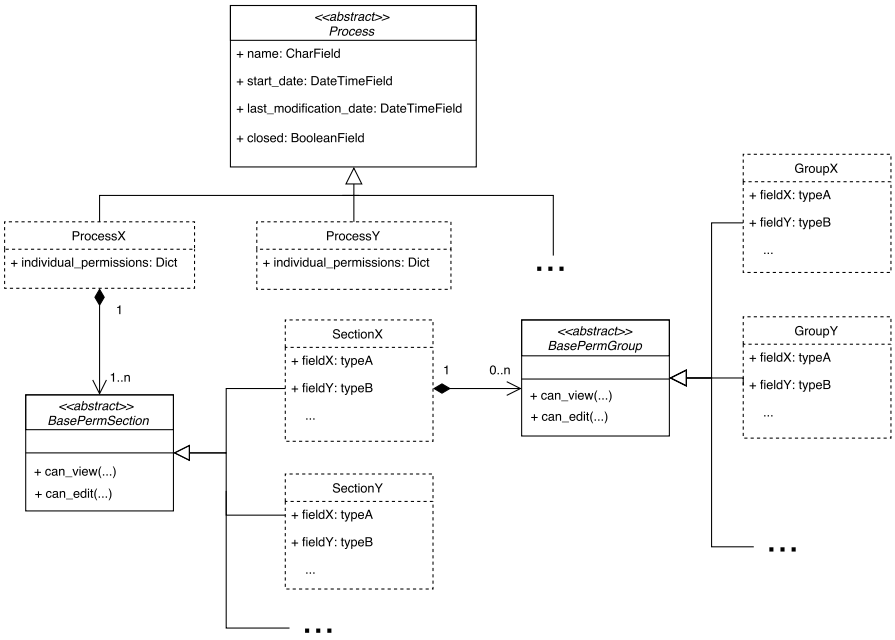

Fig. 5. Django models generated for each process

Every process-specific app defines its own set of Django models based on the abstract superclasses provided by the support library. As shown in Figure 5, the mapping is relatively straightforward: AdminDSL processes are turned into subclasses of the Process model, AdminDSL sections are turned into subclasses of BasePermSection and AdminDSL groups are turned into subclasses of BasePermGroup.

The generator also produces default HTML templates for rendering the forms and extends the Django administration area with pages for managing the various process instances. The generator does not need to produce any Django views: instead, it generates an app-specific dispatcher that reuses the views in the support library.

The rest of the AdminDSL elements are mapped as follows:

- AdminDSL roles are mapped to Django roles, created through database migrations that ensure these roles exist in the database.
- Global AdminDSL entities are added as extra models to the support library.
- Process-specific AdminDSL entities are added as extra models to the process-specific Django app.
- AdminDSL states are mapped to instances of the State model of the support library.
- AdminDSL transitions are mapped to instances of the Transition model of the support library.
- AdminDSL permissions are mapped to Django permissions, which are assigned to the appropriate Django roles. This makes it possible to change the permissions of the various states without regenerating the code.

4.4 Observed limitations

The DSL, its tooling and the evaluated generator currently present several limitations. One self-imposed limitation is that they do not aim to produce 100% of the required code in all cases: the generated code will usually need to be customized in some way, due to additional requirements on the user interface, custom business logic that has to be added, or unexpected integrations with legacy systems. Following the accepted approach in the existing literature [13], we have chosen to keep the DSL small and focused on describing administrative processes.

As shown in Section 4.3, the resulting web applications are divided into a manually written support library with all the required building blocks and a set of automatically generated components that arrange these blocks according to the AdminDSL description. These generated components are further divided into code (for the parts that do not need to change as often) and data (for the parts that must be editable by regular users). This separation allows users with special requirements to rearrange or further extend the generated applications, while helping produce quickly a first version that can be validated with the end users.

The DSL is focused on describing the current process, and does not have any provisions for migrating running processes to a new version with different states or very different information. Our current approach is to delegate on the target framework of the generator: for instance, since version 1.7 the Django framework has built-in data and schema migrations.

Since we transition to a new state as soon as its conditions are met and ignore all other transitions in the old state, we implicitly only allow one state to be active at a time. While this makes the DSL less general than a full-fledged business process modeling language such as BPMN [14], largely based on Petri nets, it is on par with some of the "virtual office" platforms we evaluated and the legacy applications it is intended to replace.

States do not enforce preconditions, invariants or postconditions yet, beyond simply checking that the mandatory editable fields have been filled in. We intend to add support for the most common conditions to the DSL in the short term: the most advanced cases will be delegated to a protected region, which will have to be filled in by the developer.

5 Case Study: Workplace Satisfaction Survey

This section presents a different case study in which a web application for performing workplace satisfaction polls was developed. Unlike the previous synthetic example, this application has been successfully deployed within the University of Cadiz and is currently under active use.

Similarly to the previous section, this section starts by describing the first version of the process and showing how it was translated to AdminDSL. The section will then discuss how the generated application evolved through several iterations, adding new variants of the original processes and acquiring custom features coded manually. Finally, the overall experience of using AdminDSL will be evaluated and future avenues of improvement will be identified.

5.1 Description

The original intent of the system was collecting the impressions of the staff at the University of Cadiz about their work environment, in order to evaluate the current situation and make plans for improvement. The system would collect information initially from each employee, which would then be revised by their line manager and their area supervisor, and if necessary go through several iterations until there was mutual agreement between all parties. The system would be monitored by external observers, which would make sure that the necessary data quality concerns were being addressed.

The form itself consisted of 16 sections, some of which would be filled in through external systems and some of which would be filled in manually. Forms would go through five states:

1. On the "initial" state, the employee would fill in most of the form and transition it into the "revision" state.

2. On the "revision" state, the line manager would review the form and pass it on to the area supervisor (state "areareviewion"). In this state, the employee and the observer could still read the form, but not edit.
3. On the "areareviewion" state, the area supervisor would review the form and add some special comments if necessary, and then either close the form (state "closed") or return it to the employee for further improvement (state "emprevision"). The employee, line manager and observer would still have read-only access to the form.
4. On the "emprevision" state, the employee would correct some of the issues identified by the area supervisor and return it to the "revision" state.
5. On the "closed" state, the form would be read-only for all the parties involved, and no further transitions are allowed.

An excerpt of the resulting AdminDSL description is shown in Listing 3. One minor detail is the requirement for a "Month" entity, as some of the sections require indicating the specific months in the year in which certain work sites are used.

5.2 Evolution

The first prototype of the application was developed and validated in record time, thanks to the AdminDSL code generator. However, as anticipated in Section 4.4, it required various iterations and manual adjustments before a final version could be deployed to production.

The first set of manual changes integrated the system with third party systems. Some of the sections in Listing 3 are never editable manually, such as the employee identification section. Instead, the BaseProcessForm superclass of all forms was modified to implicitly fill in this section based on the logged in employee. The Django templating system was extended with new tags that pulled in from these third party systems as well.

Another set of manual changes dealt with the user interface: help texts were added, and the layout was changed to use the Bootstrap library for a more attractive appearance. The UI was simplified from a generic "virtual office" arrangement to a closer match with the underlying process, hiding some of the internal complexity in the stateful forms implemented in AdminDSL. A screenshot of the final UI is shown in Figure 6.

After a pilot run, it was found that the sheer number of responses to the survey required reorganizing the approach into a hierarchy of forms rather than a simple collection. Employees would fill in a form, and the forms would be first grouped by work area and campus, then by campus and then finally for the university. The original process in the AdminDSL was turned into four processes (one per level):

– The "employee form" process: the employee fills in the form, and in the other the form is read-only to all parties.
– The "campus-area form" process: the system aggregates all responses from each work area in each campus, and the line manager reviews the form.

Listing 3. First version of the UCA Workplace Satisfaction Survey process

```
site WorkplaceSurveys;
...
role admin; role manager;
role employee; role observer;

process survey {
  entity Month { string name; }
  section(label="1. Workplace ID") id { ... }
  section(label="2. Work schedule") schedule { ... }
  section(label="3. Work description") desc { ... }
  ...
  section(label="6. Work conditions") cond { ... }
  ...
  section(label="14. Workload metrics") load { ... }
  section(label="15. Other aspects") other { ... }
  section(label="16. Manager comments") obs { ... }

  state initial {
    permissions employee {
      editable desc, ..., cond, load, other;
      viewable idworkplace, schedule, ..., obs;
    }
    transition(decision_by="employee",
    max="1") revision;
  }
  state revision {
    permissions manager {
      editable desc, ..., cond, load, other;
      viewable idworkplace, schedule, ..., obs;
    }
    permissions employee from initial->revision {
      viewable all;
    }
    permissions observer { viewable all; }
    transition(decision_by="manager") areарevision;
  }
  state areарevision {
    permissions admin {
      editable desc, ..., cond, load, other, obs;
      viewable idworkplace, schedule, ...;
    }
    ... rest can only view ...
    transition(decision_by="admin") closed;
    transition(decision_by="admin") emprevision;
  }
  state emprevision {
    permissions employee from initial->revision {
      editable desc, ..., cond, load, other;
      viewable idworkplace, schedule, ..., obs;
    }
    ... other roles can view all ...
    transition(decision_by="employee") revision;
  }
  state closed {
    ... all above can only view ...
  }
}
```

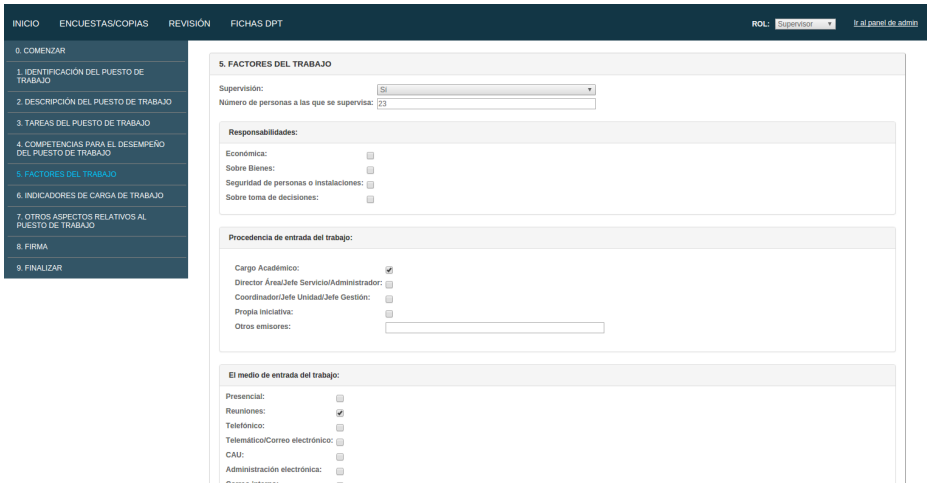

Fig. 6. Screenshot of the final user interface for the workplace satisfaction survey web application

– The "area form": the system aggregates all responses from each campus, and the area supervisor reviews the form.
– The "general form": the system aggregates all responses from the area forms, and the central administrators review the form.

AdminDSL does not directly support this concept of "aggregate" forms, so the developers had to manually implement Django views that would produce them. In addition, since AdminDSL could not capture the relationships between line managers and employees and between area supervisors and line managers, it was necessary to extend the data model to include this information and use it to assign the appropriate aggregate forms to the relevant line managers and area supervisors.

5.3 Evaluation

In this case study, AdminDSL helped produce the first prototype in record time. In addition to the process form itself, the code generator took care of the routine setup work that consists of defining the base models, configuring Django and bringing in dependencies. With the prototype in hand, the project managers were able to quickly ascertain that a different approach based on aggregate forms would scale better with the available personnel for the survey.

As expected, the code generated by AdminDSL had to be customized to cover for the aspects that are not covered by the language: a simplified user interface, integration with third party systems and starting new processes by aggregating results from others. Perhaps some of these aspects could be covered by other small DSLs in the future.

As iterations went on, many protected regions were added to the code generated by AdminDSL, so the customizations could be preserved across regenerations. This approach did not scale as well as intended, as in many cases it was necessary to add code outside the protected regions. Rather than a limitation of AdminDSL, this would be a limitation of the code generator itself, which is purely text-based: perhaps a language-aware generator could detect most manual changes and merge newly generated code with them.

In fact, having done most of the work, the final refinement iterations stopped using the AdminDSL code generator and instead manually recombined the building blocks in the support library from Section 4.3. This backs our assertion that using a support library is a best practice for generating code, as the inevitable manual iterations at the end of a project will still take advantage of it.

One issue that we identified during the manual iterations is that the generated per-section and per-group Django models (shown in Figure 5) introduced an unnecessary level of complexity when writing manual code or SQL queries. We are currently working into simplifying our approach, using a single model per process and leaving sections/groups to the user interface.

The case study also highlighted several limitations in the current version of the AdminDSL language, which could be revised in later revisions:

- The **`permissions`** block did not allow for specifying multiple roles at once, which results in repetition in some states in which several roles had the same permissions.
- The "initial" state could not be reused in Listing 3, as it is a special case: instead, a new copy named "emprevision" had to be defined.
- The aggregate form processes mentioned in 5.2 were mostly the same: it may be useful to allow for process inheritance or reuse, to avoid these repetitions.
- The application had to be manually extended to support employee-manager relations, in order to limit the employee forms that each manager could see. It would be useful to be able to limit a **`permissions`** block not only by the transition log, but also by relations between users.

6 Conclusions

A simpler class of business processes (administrative processes) are very common in many organizations today: these processes basically consist of managing a form through many states, involving various roles in the organization. Implementing the basic logic and infrastructure for them again and again wastes precious time that could be used on understanding better the process and implementing the fine details correctly.

This paper has presented an approach to improve the efficiency of implementing these solutions, while avoiding lock-in into a particular technology: using a high-level domain-specific language (DSL) for describing the process and writing a separate code generator for each target technology. The approach has been illustrated by describing an examination process, and has been implemented

with Xtext [10] on the DSL side and EGL [11] on the code generation side. The code generator produces a ready-to-use site that follows the best practices of the Django web framework [2], accelerating the implementation of the process. The generated code is based on a manually-written support library: when later iterations hit the limits of our code generator, developers are still able to extend and reuse its building blocks.

After the synthetic case study, a real-world case study for a workplace satisfaction survey developed within the University of Cádiz was discussed. The first version of the original survey was quickly developed with AdminDSL, and the developers were able to ascertain in the early stages of development that an alternative approach based on aggregating multiple levels of forms was needed. After rapid iteration of prototypes of AdminDSL, developers then manually extended the application to cover for the aspects that AdminDSL did not cover (custom user interface, external system integration, form aggregation and so on). While the experience with AdminDSL has been positive, it has highlighted the need for a smarter code generator that is friendlier to manual modifications, and has shown several ways in which AdminDSL could be improved or assisted by other small DSLs.

We are currently running two more case studies within the University of Cádiz and studying their results. After taking into account the feedback from our first real-world case study, we intend to use these two case studies to obtain more detailed metrics of the usability and productivity of AdminDSL. Once the Django support in AdminDSL is mature, we will open the tool to the general public and start working on generators for other web frameworks that are common in our organization (e.g. Symfony 2).

Acknowledgments

This work was funded by the research project "Mejora de la calidad de los datos y sistema de inteligencia empresarial para la toma de decisiones" (2013-031/PV/UCA-G/PR) of the University of Cádiz.

References

1. García-Domínguez, A., Jerez-Ibáñez, I., Medina Bulo, I.: Domain-Specific Language for Generating Administrative Process Applications. In: Proceedings of the 5th International Symposium of Business Modeling and Software Design, Milan, Italy, SciTePress (July 2015) 178–183
2. Django Software Foundation: Home page of the Django web framework. `https://djangoproject.com` (March 2015) Last checked: March 6th, 2015.
3. Bonitasoft: Homepage of the Bonita BPM project. `http://www.bonitasoft.com/` (March 2015) Last checked: March 3rd, 2015.
4. Intalio, Inc.: Homepage of the Intalio|BPMS project. `http://www.intalio.com/products/bpms/overview/` (March 2015) Last checked: March 3rd, 2015.

5. Eclipse Foundation: Homepage of the EMF Forms project. https://www.eclipse.org/ecp/emfforms/ (March 2015) Last checked: March 3rd, 2015.
6. Evans, E.J.: Domain-Driven Design: Tackling Complexity in the Heart of Software. first edn. Addison Wesley, Boston (August 2003)
7. Apache Software Foundation: Apache Isis. http://isis.apache.org/ (March 2015) Last checked: March 3rd, 2015.
8. OpenXava.org: OpenXava homepage. http://www.openxava.org/web/guest/home (March 2015) Last checked: March 3rd, 2015.
9. Klischewski, R., Lenk, K.: Understanding and modelling flexibility in administrative processes. In: Electronic Government. Volume 2456 of Lecture Notes in Computer Science. Springer (2002) 129–136
10. Eclipse Foundation: Xtext project homepage. http://www.eclipse.org/Xtext/ (September 2014) Last checked: March 2nd, 2015.
11. Eclipse Foundation: Epsilon project homepage. https://eclipse.org/epsilon/ (2015) Last checked: March 5th, 2015.
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. 1 edn. Addison Wesley, Reading, Mass (October 1994)
13. Fowler, M.: Domain Specific Languages. first edn. Addison-Wesley Professional (September 2010)
14. Object Management Group: Business Process Model and Notation 2.0.2. http://www.omg.org/spec/BPMN/2.0.2/ (January 2014) Last checked: March 2nd, 2015.