AUTHORING TOOLS FOR WORKFLOWS ON HIERARCHIES OF BUSINESS DOCUMENTS

by

ALEXANDROS KATOPODIS

MASTER'S THESIS

Thesis submitted in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science
University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes, Heraklion, GR-70013, Greece

Thesis Advisor: Prof. *Anthony Savidis*

This work has been supported by the Institute of Computer Science of the Foundation for Research and Technology - Hellas (FORTH).

University of Crete
Computer Science Department

AUTHORING TOOLS FOR WORKFLOWS ON HIERARCHIES OF BUSINESS DOCUMENTS

by
ALEXANDROS KATOPODIS

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science

Heraklion, October 2014

AUTHORING TOOLS FOR WORKFLOWS ON HIERARCHIES OF BUSINESS DOCUMENTS

ALEXANDROS KATOPODIS

Master's Thesis

University of Crete

Computer Science Department

Abstract

A Workflow Management (WFM) system is a software system that defines, creates, and manages the execution of workflows by using process definitions, interacting with workflow participants, and invoking the use of applications and tools. A workflow is an automated version of a business process that involves the flow of documents, information, and tasks between participants using a set of procedural rules. These automated versions are specified in process definitions, which are created by authoring tools that interface with one or more workflow runtime systems. WFM systems and their superset Business Process Management (BPM) systems are used by organizations to reduce costs, increase efficiency, and minimize errors.

While existing WFM/BPM systems can offer many advantages and are able to cover a wide range of use cases, they can be costly, difficult to use, or both. To achieve a cost effective and less complex solution the *"Development of innovative multi-channeled digital services"* project was initiated. It is intended to provide a WFM system that features a web interface for FORTH employees to execute and manage internal business processes. This thesis covers the development of configurable and flexible authoring tools used to design data models for this system.

In this vain, three major authoring tools are presented. First, a process editor that is used to define processes as directed graphs of activities with user roles and actions attached to each activity. Secondly, a document model editor that defines hierarchies of documents used in processes and their actions. And thirdly, a form editor that defines form structures that can be attached to document models and process actions. Finally, we will present case studies of business processes created for the WFM system.

ΕΡΓΑΛΕΙΑ ΔΙΑΧΕΙΡΙΣΗΣ ΡΟΩΝ ΕΡΓΑΣΙΩΝ ΣΕ ΙΕΡΑΡΧΙΕΣ ΕΠΙΧΕΙΡΗΜΑΤΙΚΩΝ ΕΓΓΡΑΦΩΝ

ΑΛΕΞΑΝΔΡΟΣ ΚΑΤΩΠΟΔΗΣ

Μεταπτυχιακή Εργασία

Πανεπιστήμιο Κρήτης
Τμήμα Επιστήμης Υπολογιστών

Περίληψη

Ένα σύστημα διαχείρισης ροών (σύστημα WFM) ορίζει, δημιουργεί, και διαχειρίζεται την εκτέλεση ροών με τη χρήση ορισμών, την αλληλεπίδραση με τους συμμετέχοντες της ροής, και την κλήση εφαρμογών και εργαλείων. Ως ροή ορίζεται η αυτοματοποιημένη εκδοχή μιας επιχειρηματικής διεργασίας που εμπεριέχει τη ροή εγγράφων, πληροφορίας, και εργασιών ανάμεσα στους συμμετέχοντες μέσα στα πλαίσια ενός συνόλου διαδικαστικών κανόνων. Αυτές οι αυτοματοποιημένες εκδοχές περιγράφονται με ορισμούς διαδικασιών, οι οποίοι δημιουργούνται από εργαλεία διαχείρισης που αλληλοεπιδρούν με ένα ή περισσότερα συστήματα εκτέλεσης ροών. Τα συστήματα WFM και γενικότερα τα συστήματα διαχείρισης επιχειρηματικών διεργασιών (συστήματα BPM), χρησιμοποιούνται από οργανισμούς για την μείωση εξόδων, την αύξηση της αποδοτικότητας, και την ελαχιστοποίηση των λαθών.

Μολονότι τα υπάρχοντα WFM/BPM συστήματα προσφέρουν πολλά πλεονεκτήματα και μπορούν να καλύψουν πολλές περιπτώσεις χρήσης, έχουν υψηλό κόστος, δυσκολία στη χρήση, ή και τα δύο. Το έργο "Σχεδιασμός και ανάπτυξη πλατφόρμας διάθεσης φιλικών προς τον τελικό χρήστη ψηφιακών υπηρεσιών" ξεκίνησε με σκοπό τη δημιουργία μιας χαμηλού κόστους και λιγότερο σύνθετης λύσης. Στόχος του έργου είναι η κατασκευή ενός συστήματος WFM που θα παρέχει μια web διεπαφή για να μπορούν οι υπάλληλοι του ΙΤΕ να διαχειρίζονται και να εκτελούν εσωτερικές επιχειρηματικές διεργασίες. Η εργασία αυτή αφορά την ανάπτυξη παραμετροποιήσιμων και ευέλικτων εργαλείων διαχείρισης για την σχεδίαση μοντέλων δεδομένων για το σύστημα αυτό.

Στο κείμενο που ακολουθεί παρουσιάζονται τρία κύρια εργαλεία διαχείρισης. Το πρώτο είναι ένα πρόγραμμα επεξεργασίας διεργασιών που ορίζει διεργασίες ως κατευθυνόμενους γράφους δραστηριοτήτων με ρόλους χρηστών και δράσεις συνδεδεμένες με κάθε δραστηριότητα. Το δεύτερο είναι ένα πρόγραμμα επεξεργασίας μοντέλων εγγράφων το οποίο ορίζει ιεραρχίες εγγράφων που χρησιμοποιούνται στις δράσεις διεργασιών. Τέλος, το τρίτο είναι ένα πρόγραμμα επεξεργασίας φορμών που ορίζει δομές φορμών που μπορούν να προσαρτηθούν σε μοντέλα εγγράφων και δράσεις διεργασιών. Κατόπιν, θα παρουσιάσουμε παραδείγματα εφαρμογής ολοκληρωμένων επιχειρηματικών διεργασιών για το σύστημα WFM.

Acknowledgements

I'd like thank my supervisor prof. Savidis for his guidance and support in all the years we worked together, both as a supervisor and a professor. I'd also like to express my gratitude to the HCI lab at FORTH for providing an ideal environment to work. Furthermore, I'd like to thank my colleagues at the office for making daily routine fun. Last but definitely not least, I'd like to thank my family for (still) putting up with me, especially my sister for being there whenever I needed her.

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Context

The object of this thesis is the design and development of authoring tools for the *"Development of innovative multi-channeled digital services"* project, which itself is subproject number two of the *"Digital services for the optimization of operational, financial and administrative processes of FORTH"* act. Subproject two is internally named *Monk* and hereby will be referred as such in this thesis.

Monk's mission is the implementation of a platform that provides user-friendly digital services via a web browser to the end-user. Specifically, Monk aims to provide administrative support for FORTH's institutes in their day to day processes, subsidized actions they undertake, the financial / administrative work they produce, as well as processes executed mainly by staff that is part of FORTH and of other collaborating educational and research institutes in Greece and abroad.

To accomplish this mission the automation of the most frequently executed internal workflows and processes is required. The main goals are:

- To reduce the cost of human resources used
- To reduce the execution time and optimize processes

These goals will be reached with the study, analysis, reorganization, adaption, automation, and computerization of administrative and other processes, their dependencies, and their points of interaction.

There is a set functional requirements that project Monk must support and includes:

- The use of digital signatures for document signing
- Interoperability with the ERP system
- The providing of digital services, and additionally to support the digital management of processes

- The providing of an accessible version that conforms to the Web Content Accessibility Guidelines (WCAG) 2.0 standard [1] with AA conformance level
- The providing of a mobile interface for phones and tablets—where needed—that conforms to the Mobile Web Best Practices 1.0 guidelines [2]

1.2 Role

The Monk project consists of two main architectural components: the runtime system and the authoring system. The runtime system is responsible for process management, user management, document management, session management, data storage, and the generation of the end-user facing web interface. The authoring system includes the tools that author models and data that can be later imported into the runtime system via an API provided for business data access. Figure 1 details Monk's macro-architecture. In the context of Monk's macro-architecture the authoring system is independent of the runtime system save for its connection with the business data access component.

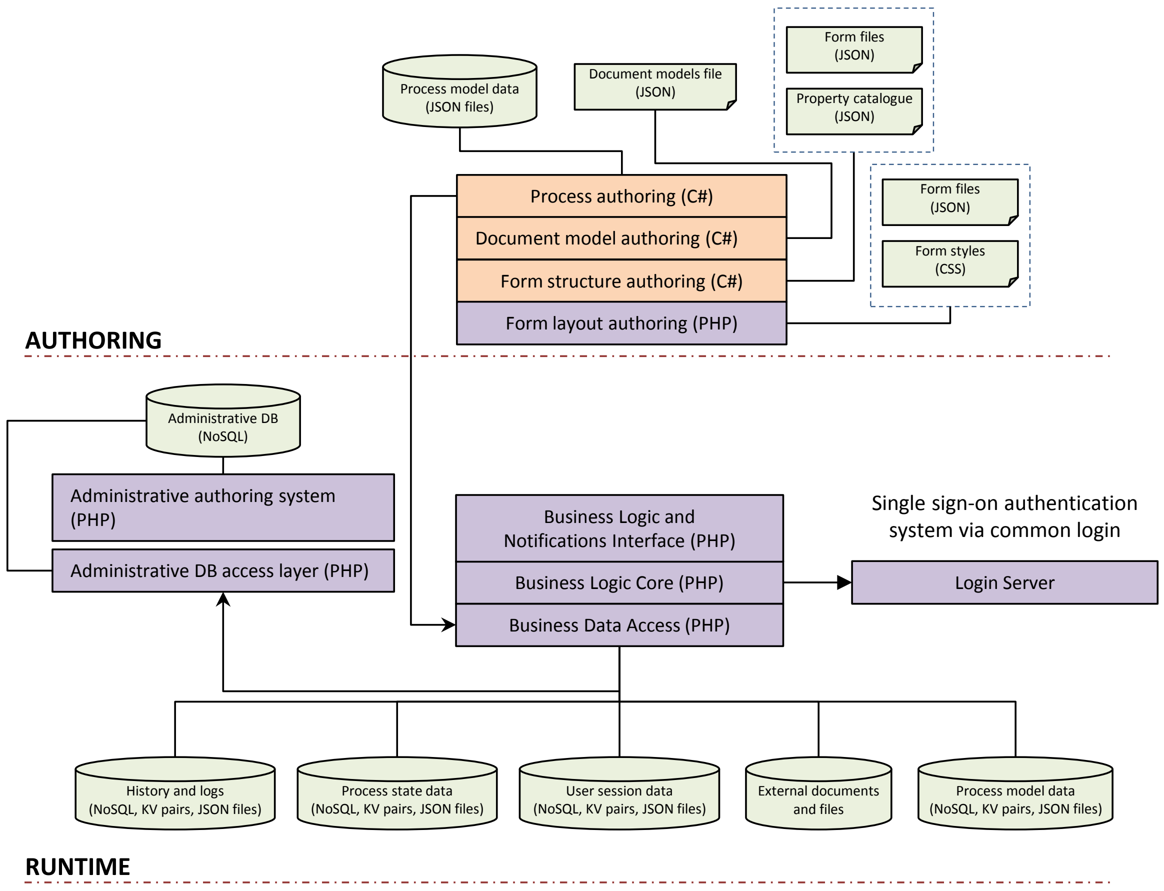

Process model data (JSON files)
Document models file (JSON)
Form files (JSON)
Property catalogue (JSON)
Form files (JSON)
Form styles (CSS)
Process authoring (C#)
Document model authoring (C#)
Form structure authoring (C#)
Form layout authoring (PHP)
AUTHORING
Administrative DB (NoSQL)
Administrative authoring system (PHP)
Administrative DB access layer (PHP)
Business Logic and Notifications Interface (PHP)
Business Logic Core (PHP)
Business Data Access (PHP)
Single sign-on authentication system via common login
Login Server
History and logs (NoSQL, KV pairs, JSON files)
Process state data (NoSQL, KV pairs, JSON files)
User session data (NoSQL, KV pairs, JSON files)
External documents and files
Process model data (NoSQL, KV pairs, JSON files)
RUNTIME

Figure 1: Monk's macro-architecture

The authoring system's technical mission is to provide easy-to-use tools for authoring process models, document models, and form structure and layout. Further, they are created for internal use by employees that need not be advanced users and require minimal training. Since the tools are internally developed, they are tailor-made for the design of the system and are flexible enough to accommodate changes and additions to the system's design and requirements.

1.3 Requirements

The authoring system defines these functional requirements for its tools:

- The creation of a graphical tool that can create and edit descriptions of business process models. The business processes are represented as a directed graph of activities that include actions and are bound to user roles. The tool must be able to refer to actions and user roles provided by the runtime system.

- The creation of a graphical tool that can create and edit document models. A document model is of a tree-like structure with leaf nodes being documents of a certain type and composite nodes being document groups. Each node in this tree-like structure has a unique path that refers to itself.
- The creation of a graphical tool that can create and edit form structures with the option of creating a custom layout separately. A form structure is a tree-like construct where each node is a form element that may or may not have children depending on its type.
- All process, document model, and form files must be uniquely identified by a string id.
- The file format for each of the above editors must be JSON [3] encoded with schemas that define its structure. Also, files generated by the editors must be suitable for importing into the runtime system.

1.4 Architecture

Figure 2 shows a macro-architecture view of all the components that constitute the Monk authoring system. At the bottom of the stack lies the .NET Common Language Runtime (CLR) that all the components depend on to run, as they are all written in the C# language. Above the CLR is the Windows Presentation Foundation (WPF) [4] library, which is a graphics subsystem for Windows. WPF is used by the mahapps.metro, ConfigUIGenerator, and Common libraries as well as by all the editors. The Json.NET library that is on top of the CLR has no dependency to WPF.

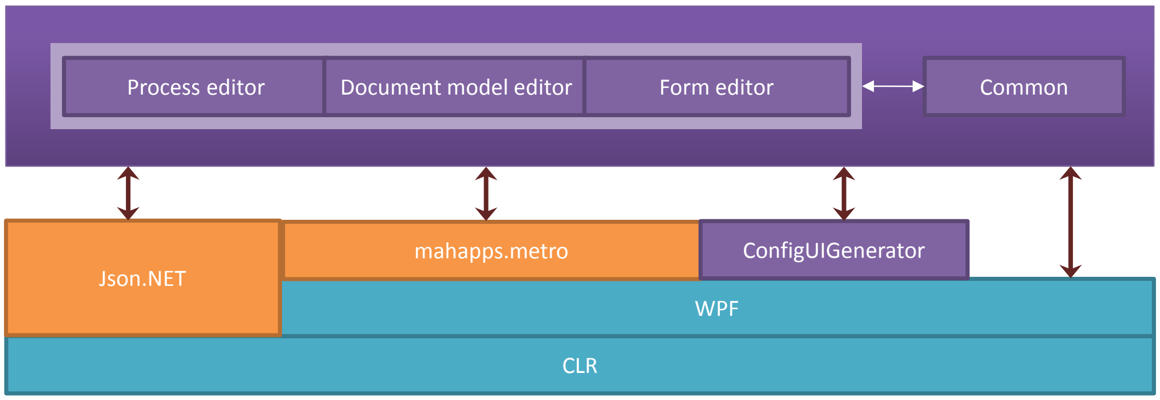


Figure 2: The Monk authoring system macro-architecture

Json.NET [5] is a third party-library used for parsing, validating, serializing to and deserializing from JSON. It supports a subset of JSON schema version 3 [6] for validation. All of the authoring editors as well as the common library depend on it.

Mahapps.metro [7] is another third-party library that provides styles and themes for controls. Also, it offers its own custom control and window classes. All of the authoring editors as well as the common library depend on it.

ConfigUIGenerator (section 6.4) is a library created for generating editor UIs for object properties via user-defined property specifications. All of the authoring editors as well as the common library depend on it.

The Common library includes common classes and interfaces used by all the editors in the authoring system. More specifically it provides:

- JSON utility methods on top of Json.NET
- An interface and classes for implementing undo / redo functionality
- Common undoable commands
- Extension methods for WPF dependency objects and the `List` class
- Common icon paths, control styles, and theme resources
- Common validation rules
- An interface for tree view items

1.5 Thesis structure

The structure of the following chapters will be as such:

Chapter 2 will present related work in three areas: form authoring, business process authoring, and automatic configuration interfaces. Chapters 3 to 5 provide detailed analysis of the three Monk authoring tools: Chapter 3 features the form editor, chapter 4 the document model editor, and chapter 5 the process editor. Each chapter includes a description of the authoring tool discussed, an architectural overview, descriptions for each user-interface component used, and a walkthrough of the file format(s) used. Chapter 6 lists the general features that are present in all authoring tools. These features include localization, validation, undo/redo, automatic configuration user-interfaces, and other. Chapter 7 presents case studies created for both the authoring system and the runtime system. Finally, chapter 8 discusses future feature implementations that would be desirable additions to the authoring system.

2. Related Work

2.1 Form Authoring

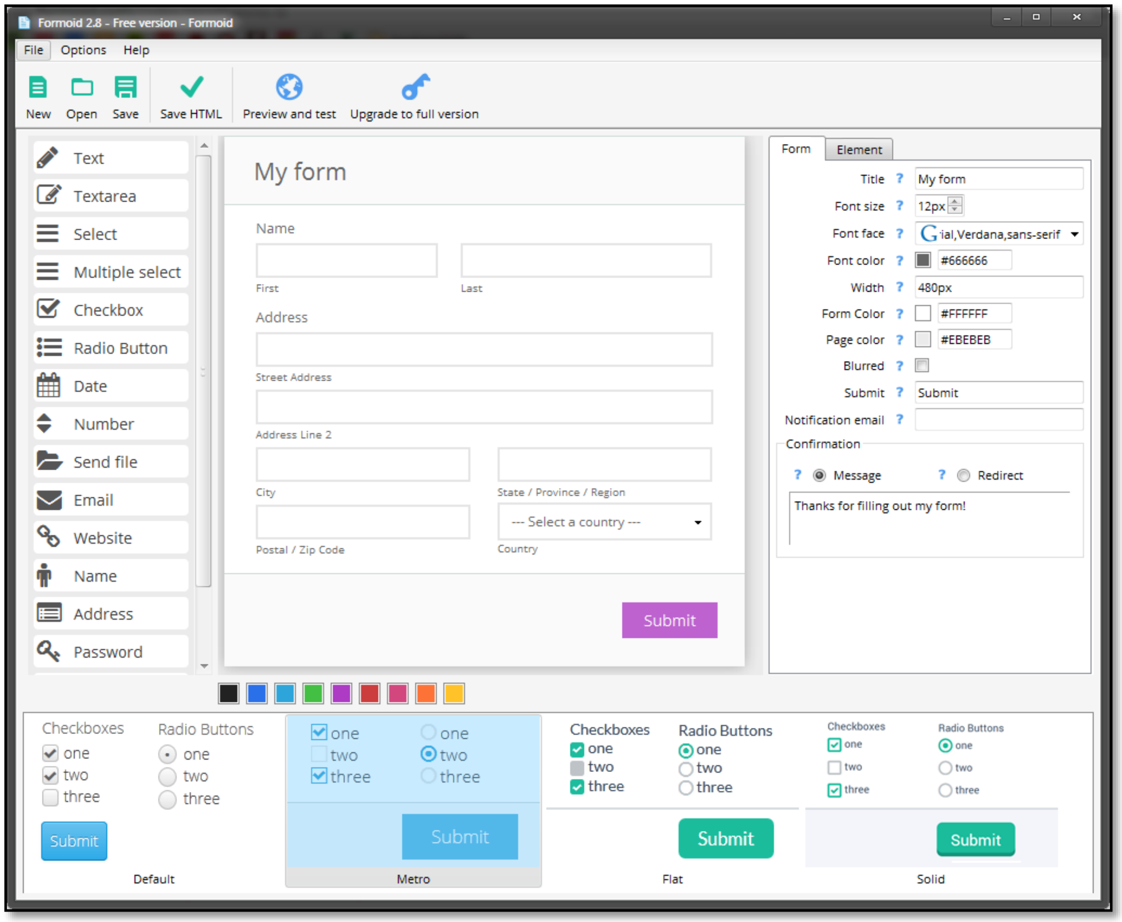
2.1.1 Formoid

Formoid [8] is a graphical tool for generating web forms. It uses a drag-n-drop live preview area and a form element property editor to create the form structure and user interface (Figure 3). The live preview area allows for moving form elements up or down as well as removing them. Forms can be saved in Formoid's intermediate JSON format, or can be exported as HTML, JavaScript, PHP and CSS files. Generated forms can also be hosted online on the software developer's server.

Standard HTML5 form element types [9] such as text, textarea, and date are provided. Also provided are composite types i.e. form elements with predetermined appearance and multiple form elements contained within them. Composite types include "Name", "Address" etc. The complete list of element types included is:

- Text
- Textarea
- Select
- Multiple Select
- Checkbox
- Radio Button
- Date
- Number
- Send File
- Email
- Website
- Name

- Address
- Password
- Phone
- Captcha



Title: My form
Font size: 12px
Font face: Arial,Verdana,sans-serif
Font color: #666666
Width: 480px
Form Color: #FFFFFF
Page color: #EBEBEB
Blurred: ☐
Submit: Submit
Notification email:
Confirmation: [✓ Message] Redirect
Thanks for filling out my form!

Figure 3: The Formoid main window. Includes an element selector, a live preview area, and form element property editor

Formoid does not separate between structure and presentation. Each form element has properties that affect both. For example, the "Date" form element has a "Label" property that affects structure, but also includes a "Field Size" property that affects the width of the text box holding the date. Properties for the whole form are in the same vein including "Font Size", "Form Color", etc.

When generating forms for web usage, Formoid includes CSS styles, JavaScript files, and PHP code to complement the generated form HTML file. The Bootstrap CSS library [10] is used as a base for the four themes included in the tool. For validation and general scripting purposes jQuery [11], and jQuery plugins are used. Also for validation purposes, the "Captcha" form element uses the recaptcha [12] PHP library. Additional PHP code is generated to handle page rendering and POST data from the form.

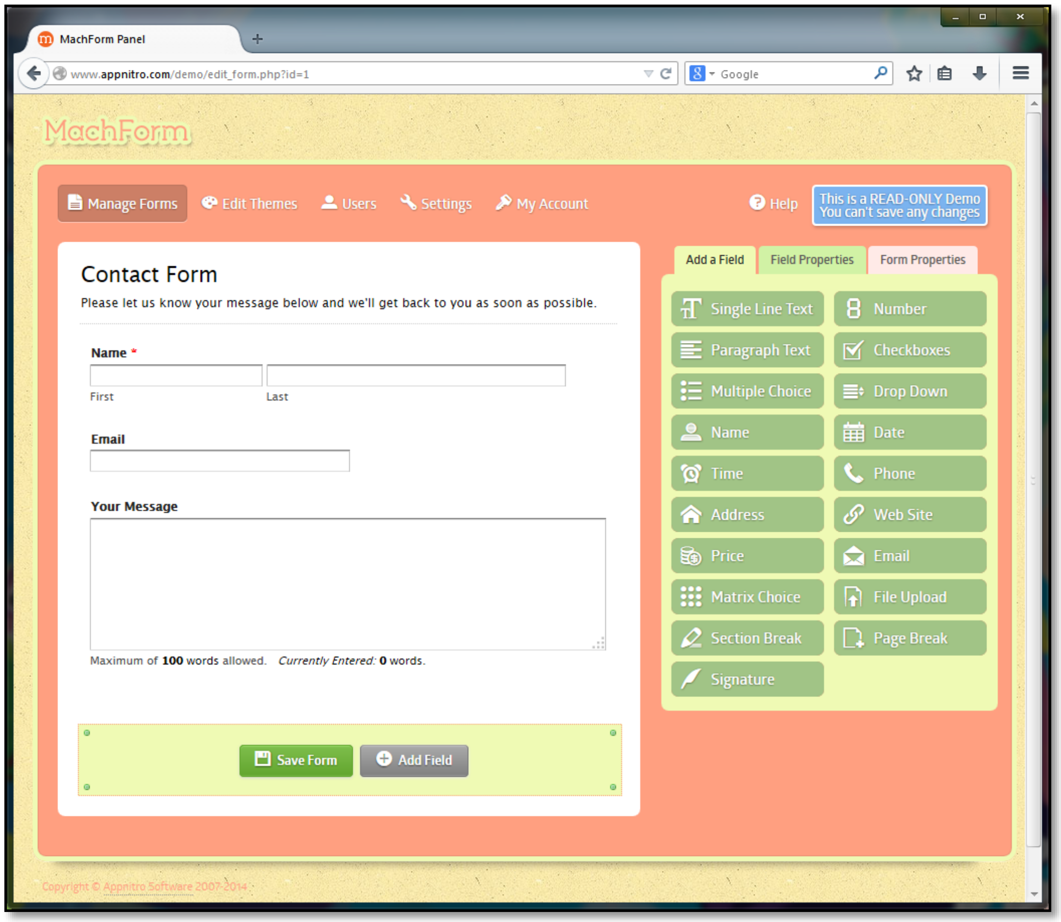
2.1.2 MachForm

MachForm [13] is a PHP application that creates and manages web forms. It provides a web-based user-interface that includes form creation and management, user management, form theme creation, and submitted form entry management. Forms can be created using a drag-n-drop live preview area and a form element property editor. Created forms and submitted entries are stored in a MySQL database. Generated forms are then served from the application. The application supports embedding the form in web pages via JavaScript code that loads the form into the page, an iframe, and direct linking.

Apart from the standard HTML5 form element types [9], MachForm provides composite types and an element for form pagination. Composite types include "Price" and "Matrix Choice", while the "Page Break" element type is used for pagination. The complete list of types included is:

- SingleLineText
- Number
- ParagraphText
- Checkboxes
- MultipleChoice
- DropDown
- Name
- Date
- Time
- Phone

- Address
- WebSite
- Price
- Email
- MatrixChoice
- FileUpload
- SectionBreak
- PageBreak
- Signature



Contact Form

Please let us know your message below and we'll get back to you as soon as possible.

Name *

First Last

Email

Your Message

Maximum of **100** words allowed. *Currently Entered:* **0** words.

Figure 4: The MachForm form creator. Includes a live preview area and a tabbed control for adding and editing form elements

MachForm does not expose an intermediate form structure format, but it does internally store form structures in database tables. There is also no full separation between structure and presentation in form element properties. There exists though support for general form theming including fonts, backgrounds, and borders as well as support for inserting custom classes into form elements for further CSS theming.

Additional to the standard validation options such as min / max for numbers or character limits for text areas MachForm provides rule-based logic for controlling form element visibility, skipping form pages, and sending notification emails. A rule contains triplets of the form ("element", "condition", "value") that are called conditions. A rule can be set to be evaluated successfully if all of its conditions are true or if any one condition is true.

Finally, submitted entries are stored in the database and can be viewed from the web interface. There is support for filtering using the same rule scheme as mentioned above, and for selecting which fields will be displayed. Entries can be exported as Excel files (*.xls), comma separated files (*.csv), and simple tab separated text files.

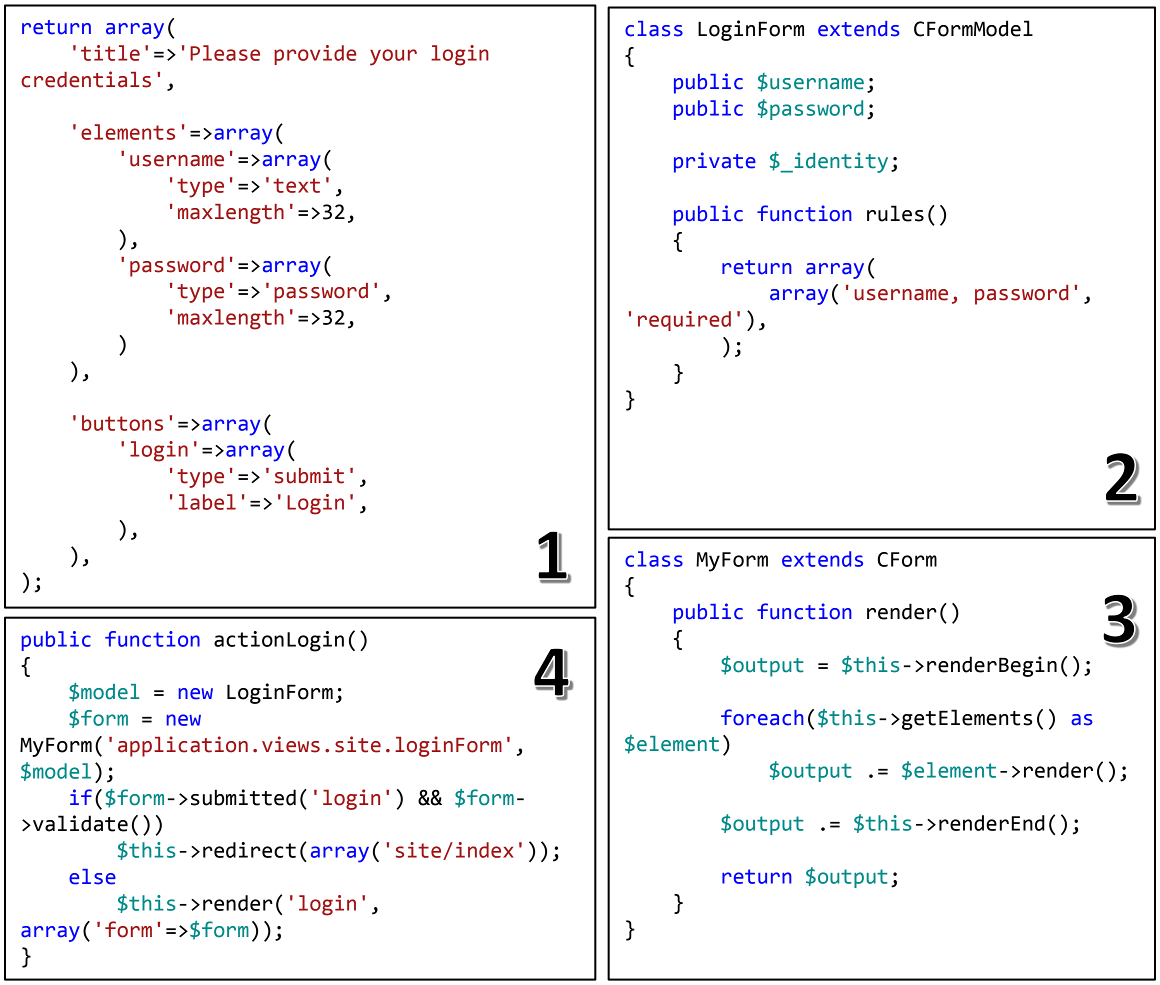
2.1.3 Yii Framework Form Builder

The form builder for the Yii PHP framework [14] is a set of classes that allow for the creation of form structures and the controlling of form rendering within an application that uses the framework. Form specifications are defined in a PHP file containing a specially formatted array. These specifications along with a model can be used to instantiate a form object. This object can then be rendered into HTML. Figure 5 details the creation of simple login form including the form specification, the form model, a form object with an overridden render method, and action code that instantiates and renders the form.

A form specification file has three main properties: "title", "elements", and "buttons". The "elements" property is an array of form element specifications (input elements, static text, and sub-forms), while the "buttons" property is an array of button elements. Input form element specifications have standard properties such as "type", "hint", and "label" and can include additional properties that are directly translated into HTML attributes. Form input element types include all the standard HTML5 input types [9]. Also defined are types for input elements with multiple items such as "dropdownlist", and "checkboxlist". The complete list of input form element types is:

- text
- hidden
- password
- textarea
- file
- radio
- checkbox
- listbox
- dropdownlist
- checkboxlist
- radiolist
- url
- email
- number
- range
- date

By separating a form definition into specification files, models, and form objects the form builder achieves full separation between structure and presentation. Specification files define the structure of the form, models define the backing store and validation rules, and the form object is used for rendering the form in HTML. The render method of the form object can be overridden, thus enabling custom presentation for the whole form or select elements.



1

```
return array(
    'title'=>'Please provide your login
credentials',

    'elements'=>array(
        'username'=>array(
            'type'=>'text',
            'maxlength'=>32,
        ),
        'password'=>array(
            'type'=>'password',
            'maxlength'=>32,
        )
    ),

    'buttons'=>array(
        'login'=>array(
            'type'=>'submit',
            'label'=>'Login',
        ),
    ),
);
```

2

```
class LoginForm extends CFormModel
{
    public $username;
    public $password;

    private $_identity;

    public function rules()
    {
        return array(
            array('username, password',
'required'),
        );
    }
}
```

3

```
class MyForm extends CForm
{
    public function render()
    {
        $output = $this->renderBegin();

        foreach($this->getElements() as
$element)
            $output .= $element->render();

        $output .= $this->renderEnd();

        return $output;
    }
}
```

4

```
public function actionLogin()
{
    $model = new LoginForm;
    $form = new
MyForm('application.views.site.loginForm',
$model);
    if($form->submitted('login') && $form-
>validate())
        $this->redirect(array('site/index'));
    else
        $this->render('login',
array('form'=>$form));
}
```

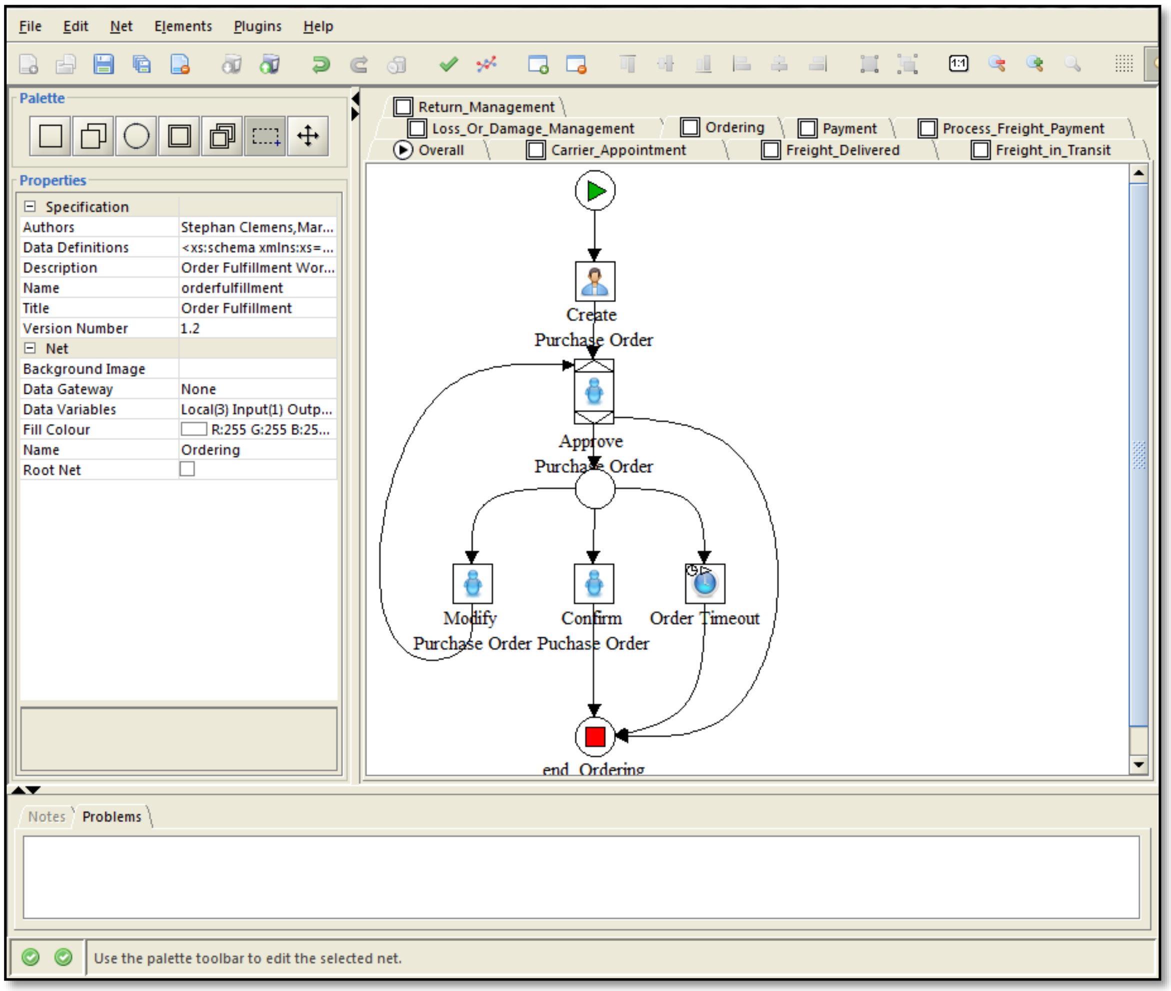
Figure 5: A simple login form example for the Yii from builder. Code snippet 1 shows a specification, code snippet 2 shows a model, code snippet 3 shows a form object, and code snippet 4 shows action code

2.2 Business Process Authoring

2.2.1 YAWL System

The YAWL System [15] (also referred to as YAWL Environment) is a complete suite of applications for business process authoring and execution. It comes in two flavors: YAWL4Study which is optimized for testing in single user environments, and YAWL4Enterprise that is the version most suitable for production purposes on a server. Both flavors have complete feature parity. The two major components of the system are the runtime environment and the process editor. Both these components utilize the YAWL (Yet Another Workflow Language) language.

The YAWL language [16] is based on Petri nets [17], an abstract formal model of information flow, and on research of existing workflow patterns [18]. However, Petri nets cannot support all of these patterns, namely multiple instance patterns, cancellation patterns, and generalized OR-join. As a result, the language extends Petri nets with constructs such as composite tasks and direct transitions. YAWL is based on formal semantics which makes it verifiable with techniques like static analysis.



⊟ Specification	
Authors	Stephan Clemens,Mar...
Data Definitions	<xs:schema xmlns:xs=...
Description	Order Fulfillment Wor...
Name	orderfulfillment
Title	Order Fulfillment
Version Number	1.2
⊟ Net	
Background Image	
Data Gateway	None
Data Variables	Local(3) Input(1) Outp...
Fill Colour	☐ R:255 G:255 B:25...
Name	Ordering
Root Net	☐

Use the palette toolbar to edit the selected net.

Figure 6: The YAWL Process Editor

A workflow defined in YAWL contains at least one workflow net that is the top-level net. The other potential nets are hierarchically below the top-level net forming a tree-like structure. Each net contains *conditions* and *tasks*. Tasks can be atomic or composite. Tasks can also define any number of instances. At least two conditions are present in a net, a unique input condition and a unique output condition. Tasks and conditions in a net can be connected using edges called *flows*. By default, tasks cannot have more than one outgoing and incoming flow. To add more incoming flows tasks are decorated with a *join*. For more outgoing flows they are decorated with *splits*. Both these decorations can define how flows are to be handled.

The YAWL Process Editor (Figure 6) allows for the creation and editing of workflow specifications in an XML-based format. It also provides facilities for workflow analysis and verification. Its main layout consists of an element editor that can modify properties for conditions, tasks, and flows, and a tabbed view that includes graphical representations of all the nets in a specification.

The runtime environment consists of servlets. In order for the environment to run, a servlet container like Apache Tomcat is needed to host the servlets. Likewise, for storage purposes a database backend like PostgreSQL can be used. The YAWL engine within the environment is responsible for controlling the control-flow and data perspectives defined in a workflow specification, while a resource service handles the allocation of resources. The administrator interface can upload workflow specification, execute and manage running cases, register services and client applications, and add, edit, and remove user roles, participants, positions, assets, and organizational groups.

2.2.2 Bonita BPM

Bonita BPM [19] is a business process management system. It comprises three main components:

- Bonita Studio: An eclipse based editor for the authoring of process workflow and forms

- BPM engine: A Java API that allows the creation, instantiation, execution, and deletion of processes. Also responsible for process definition and process instance persistence, and the execution of flow. Uses the Hibernate ORM
- Bonita BPM Portal: A user-interface for managing and administering tasks. Uses GWT

Bonita Studio uses the BPMN 2.0 standard [20] to author process workflows. Its main graphical layout (Figure 7) consists of a BPMN element pallet on the upper left side. These elements can be added to the main process graph view. On the bottom right side is an element editor that can alter appearance, general, and other BPMN element properties. The element editor also offers the functionality to add forms.

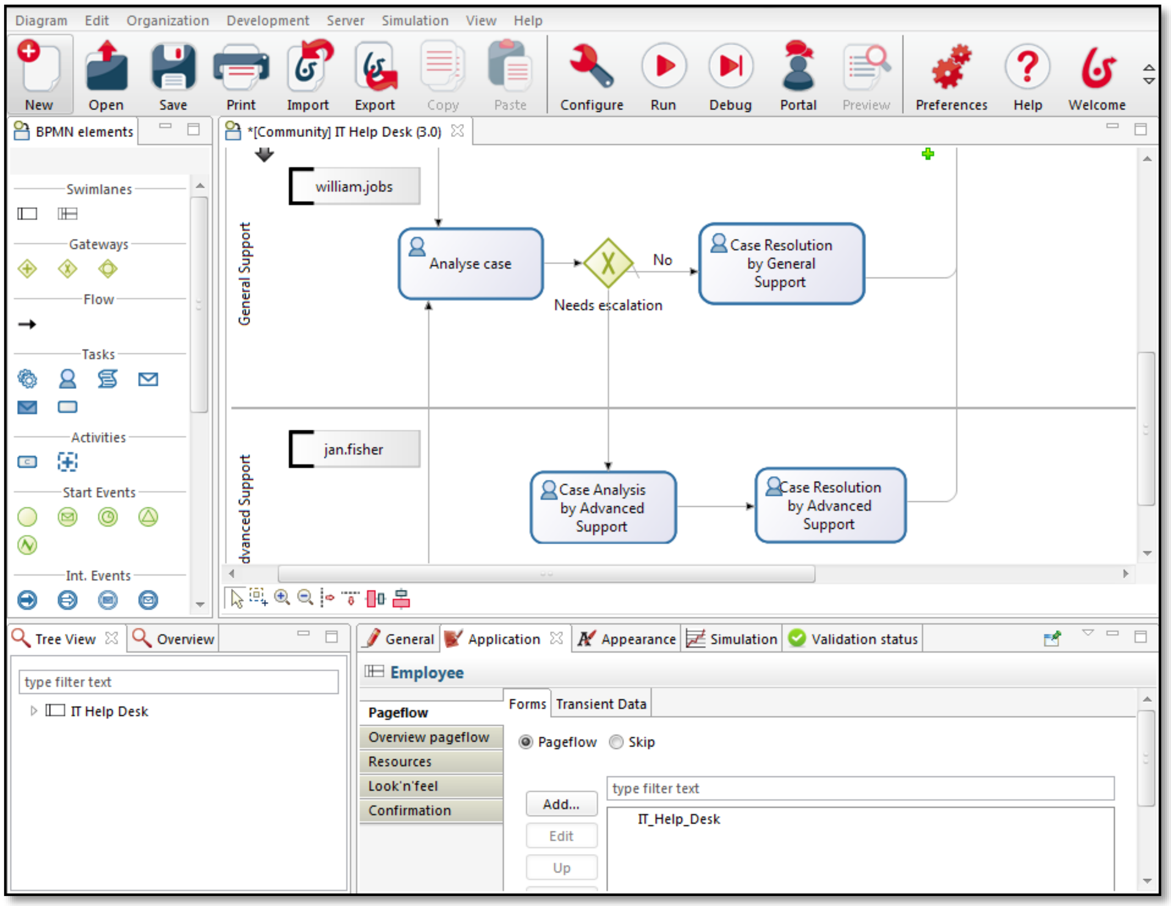


Figure 7: The Bonita Studio main view

Bonita Studio offers a form authoring graphical editor that uses a grid layout to place form elements. All the standard input types are provided along with some non-standard ones like "Editable grid" and "Table". Custom validators for each element can be added. Form layout and structure are not separated but forms can be exported to XML files.

Additional features of Bonita studio include the ability to run process simulations, an editor to manage organizational structures, control over starting and stopping the BPM engine, and process execution and debugging.

The BPM portal provides a user-interface to the BPM engine. It has two main interfaces, one for users, and one for administrators. The user side of the portal has facilities to initiate tasks and to handle incoming tasks. Administrator features include the configuration of profiles, management of organization members, groups and roles, addition of new processes, and management of running processes.

2.3 Automatic Configuration User Interfaces

2.3.1 Windows Forms PropertyGrid Class

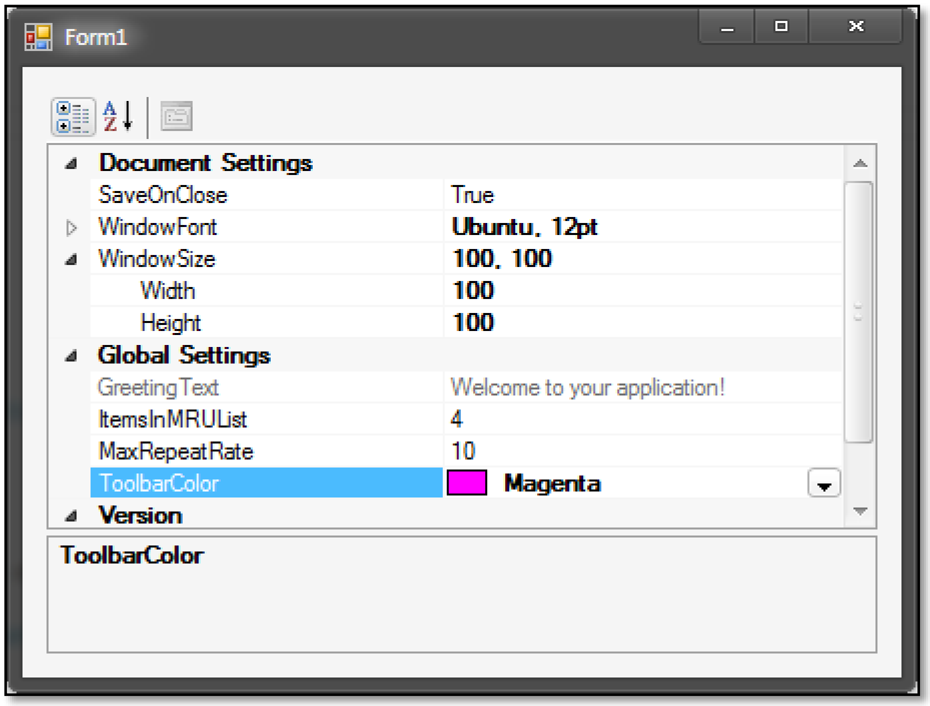
The `PropertyGrid` control [21] is an object property editor generator for the Windows Forms graphical API [22]. It accepts an object as an input and via reflection generates editors for the object's public properties.

The default graphical layout of the control (Figure 8) consists of a toolbar at the top that provides property sorting by category and by name, a grid area in the middle that includes tuples of property labels and editors, and a help pane at the bottom that displays help text if available.

Attributes can alter the appearance and even the inclusion of a property in the grid. Some of these attributes are:

- DisplayName: Changes the label of the property in the grid
- Description: Sets the text displayed in the help pane

- Category: Changes the category that the property belongs to. Properties belong to the "Misc" category by default
- DefaultValue: Sets the default value for the property
- ReadOnly: Sets if the property can be modified. its editor appears muted if true
- BrowsableAttribute: Is used to make properties not appear in the grid



Document Settings	
SaveOnClose	True
WindowFont	**Ubuntu, 12pt**
WindowSize	**100, 100**
Width	**100**
Height	**100**
Global Settings	
GreetingText	Welcome to your application!
ItemsInMRUList	4
MaxRepeatRate	10
ToolbarColor	🟪 **Magenta**
Version	

ToolbarColor

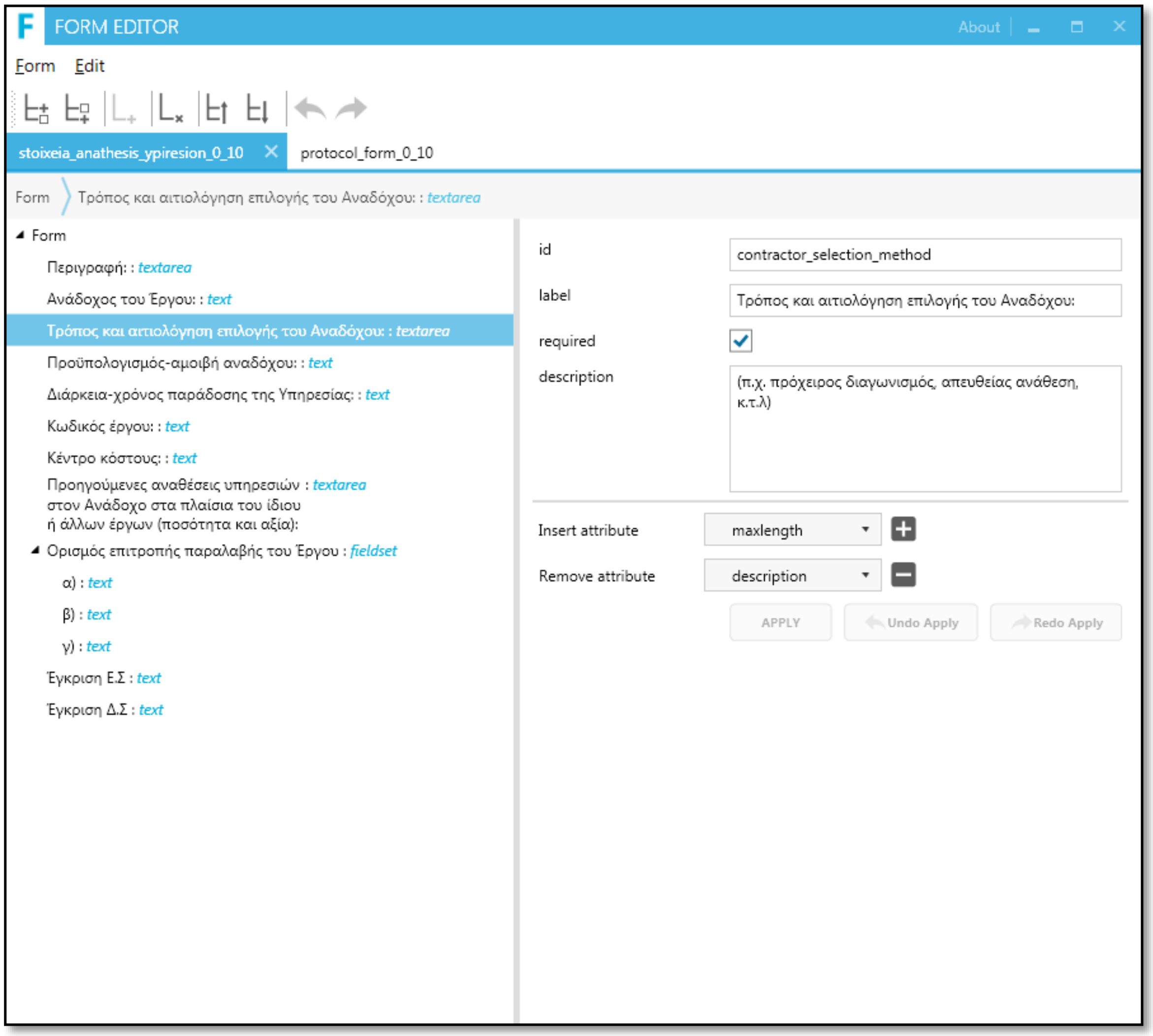
Figure 8: Generated editors from a PropertyGrid control

`PropertyGrid` provides support for some built-in complex types such as `Font`, `Size`, and `Color`, while providing the `TypeConverter` and `UITypeEditor` for adding support for custom complex types. The user-interface customization options are fairly limited. Said options include font and background color, help pane and toolbar visibility, grid line and border color, and initial property sorting mode. Finally, while the input object can be changed dynamically with the grid updating to new property values, properties cannot be added or removed dynamically.

3. Form Editor

3.1 Overview

The form editor (Figure 9) is the authoring tool used for the specification of form structures used in business processes. Specifying form structure means that any information pertaining to layout is not included. Form layout can optionally be defined at a later stage using a form layout editor that takes a form structure file as input.



Form 〉 Τρόπος και αιτιολόγηση επιλογής του Αναδόχου: : *textarea*

- Form
  - Περιγραφή: : *textarea*
  - Ανάδοχος του Έργου: : *text*
  - Τρόπος και αιτιολόγηση επιλογής του Αναδόχου: : *textarea*
  - Προϋπολογισμός-αμοιβή αναδόχου: : *text*
  - Διάρκεια-χρόνος παράδοσης της Υπηρεσίας: : *text*
  - Κωδικός έργου: : *text*
  - Κέντρο κόστους: : *text*
  - Προηγούμενες αναθέσεις υπηρεσιών στον Ανάδοχο στα πλαίσια του ίδιου ή άλλων έργων (ποσότητα και αξία): : *textarea*
  - Ορισμός επιτροπής παραλαβής του Έργου : *fieldset*
    - α) : *text*
    - β) : *text*
    - γ) : *text*
  - Έγκριση Ε.Σ : *text*
  - Έγκριση Δ.Σ : *text*

id	contractor_selection_method
label	Τρόπος και αιτιολόγηση επιλογής του Αναδόχου:
required	✓
description	(π.χ. πρόχειρος διαγωνισμός, απευθείας ανάθεση, κ.τ.λ)
Insert attribute	maxlength
Remove attribute	description

Figure 9: The form editor's main view

The form editor allows for the creation of a tree-like structure of form elements such as text, fieldset, number etc. Form elements in the tree-like structure can also be rearranged. Each of these elements can have its individual attributes modified by a form element editor that changes depending on which form element is selected. Optional attributes can be added or removed at will in the form element editor, with each form element having a different set of available attributes to select.

3.2 Architecture

Form editor's general architecture is presented in Figure 10. Some secondary classes have been omitted for visual brevity. The architecture is divided into two layers. The business layer and the user-interface layer. The business layer includes classes and data that form the backbone of the application while the user-interface layer includes all the controls and classes that are relevant to the user-interface.

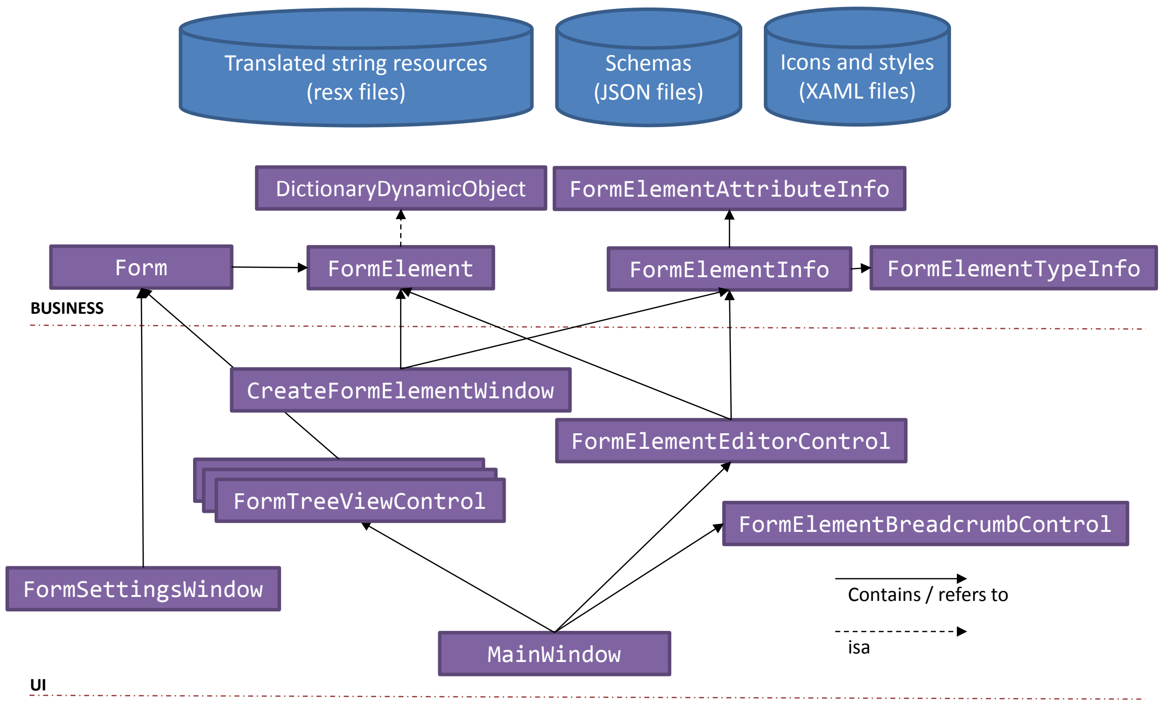


Figure 10: Form editor architecture

In the business layer, the class that describes a form's structure is called `Form`. `Form` provides methods for form element addition, removal, and reordering. Also provided are methods for initializing an instance from a JSON file along with schema validation. `Form` has string properties for general form identification such as `Name`, `FriendlyName`, and `Description`. The `Fields` property is a collection that includes all of the form's elements of type `FormElement`.

Since all form elements can have a variable number of attributes, form elements are represented as dynamic objects that can create and remove properties at runtime. To accomplish this the `DictionaryDynamicObject<T>` superclass was created. `DictionaryDynamicObject<T>` is derived from the non-instantiable `DynamicObject` class that is part of the .NET framework class library. `DynamicObject` enables the definition of dynamic object behavior on operations like trying to get or set object properties and calling methods.

Naturally, the `FormElement` class derives from `DictionaryDynamicObject<FormElement>`. `FormElement` provides the same functionality for form element addition, removal, and reordering as `Form` if its "type" attribute is a composite type (i.e. can have child form elements). It also defines mandatory attributes depending on type on object construction. In the case of creating a `FormElement` object with the default constructor a form element of type "text" is created.

For form elements to function correctly, type information is required to be available at runtime. The `FormElementInfo` static class provides just that. The class provides information on which attributes and child types (if any) a form element type can have by returning `FormElementTypeInfo` objects. For use by the form element editor it also returns `FormElementAttributeInfo` objects that define the micro-editor type to be used to edit an attribute (see 6.4.4). Appendix B (form element specifications) contains detailed specifications for form element types and attributes.

Apart from the classes mentioned above, the business layer includes data in the form of .resx files for translated string resources, a JSON schema file for form JSON files, toolbar vector icons in XAML resource dictionaries, and a XAML resource dictionary for various styles.

The user-interface layer consists of three main windows. These are:

- `FormSettingsWindow`, which is the form setting dialog
- `CreateFormElementWindow`, which is the dialog used to create form elements
- and `MainWindow` which is the main application window

`FormSettingsWindow` binds to `Form` objects to edit their properties. It uses the ConfigUIGenerator facilities for editor user-interface generation. `CreateFormElementWindow` creates `FormElement` objects and refers to `FormElementInfo` for runtime type information. It also uses the ConfigUIGenerator facilities for editor user-interface generation.

`MainWindow` is the main application window that contains the menu bar, toolbar, and the central tab control. Each tab maps to a form structure to be edited. The tab control binds itself to a collection of `FormTreeViewControl`s and contains a single `FormElementEditorControl` as well as a single `FormElementBreadcrumbControl`. Both `FormElementEditorControl` and `FormElementBreadcrumbControl` bind to the selected form element of the current form tree view. `FormTreeViewControl` binds to `Form` objects.

3.3 User Interface

3.3.1 Breadcrumbs

A breadcrumb control (Figure 11) for form elements is used to show the selected form element's position in the form tree hierarchy. It is also possible to navigate to any previous form element in this hierarchy by clicking on the element's crumb. The form element breadcrumb control is defined in the `FormElementBreadcrumbControl` class.

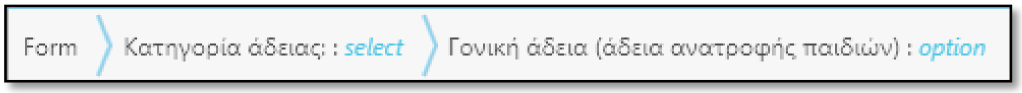

Form 〉 Κατηγορία άδειας: : select 〉 Γονική άδεια (άδεια ανατροφής παιδιών) : option

Figure 11: Form editor's breadcrumb control

3.3.2 Tree View

A tree view control (Figure 12) is used to display the hierarchical structure of a form. Each form element is displayed as tuple of label and type attributes (which are mandatory). Composite elements can be collapsed or expanded while all elements trigger a form element editor change when clicked. The form tree view is defined in the `FormTreeViewControl` class.

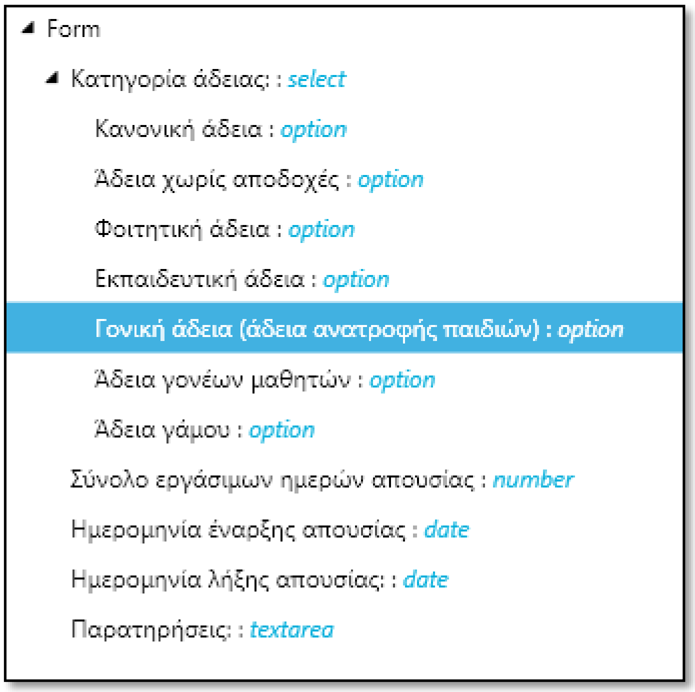
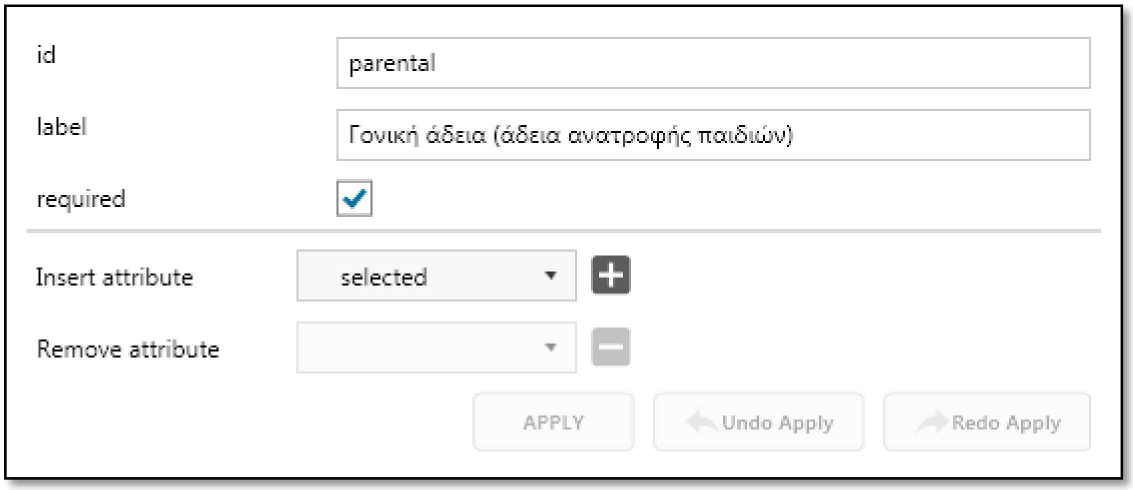

◢ Form
◢ Κατηγορία άδειας: : select
Κανονική άδεια : option
Άδεια χωρίς αποδοχές : option
Φοιτητική άδεια : option
Εκπαιδευτική άδεια : option
Γονική άδεια (άδεια ανατροφής παιδιών) : option
Άδεια γονέων μαθητών : option
Άδεια γάμου : option
Σύνολο εργάσιμων ημερών απουσίας : number
Ημερομηνία έναρξης απουσίας : date
Ημερομηνία λήξης απουσίας: : date
Παρατηρήσεις: : textarea

Figure 12: Form editor's tree view for form elements

3.3.3 Fields

To edit form element attributes as a well as add / remove them, the form element editor control is used (Figure 13). The control binds to a form element and creates attribute editors via ConfigUIGenerator's facilities, also creating relevant validation rules for each editor.

Additionally, collections of attribute names available for addition / removal are created. Form element editor also exposes a history of apply commands for undo / redo support via a cache of undo histories that is kept internally. Finally, in case of validation errors form element editor exposes a property to indicate if changes can be applied or not. The form element editor is defined in the `FormElementEditorControl` class.



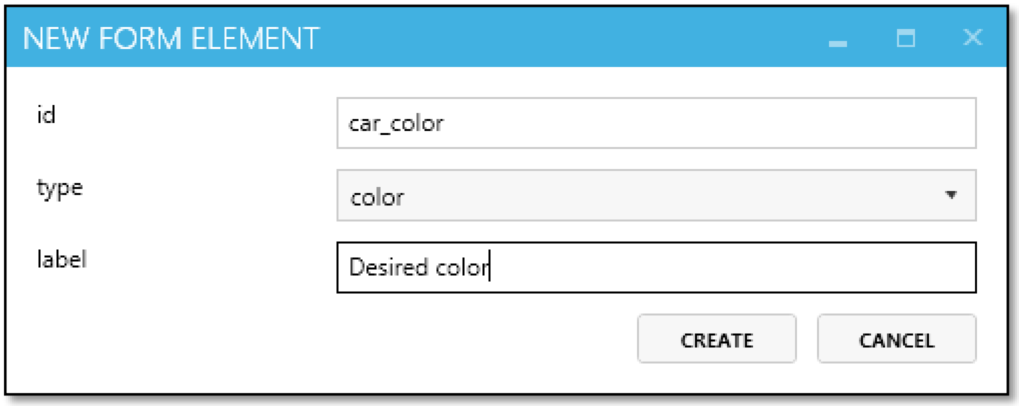
id	parental
label	Γονική άδεια (άδεια ανατροφής παιδιών)
required	✓
Insert attribute	selected
Remove attribute	

Figure 13: Form editor's form element editor

3.3.4 Other Features

Element Creation

A modal window (Figure 14) is used for the creation of form elements and their addition to the form tree hierarchy. The attributes that can be edited are "id", "type", and "label". The "id" attribute is checked for validation errors, namely if it does not solely contain letters, numbers, and underscores and if an "id" attribute of the same value has not been defined elsewhere. If there are validation errors, the "CREATE" button will be deactivated so a new form element cannot be added.

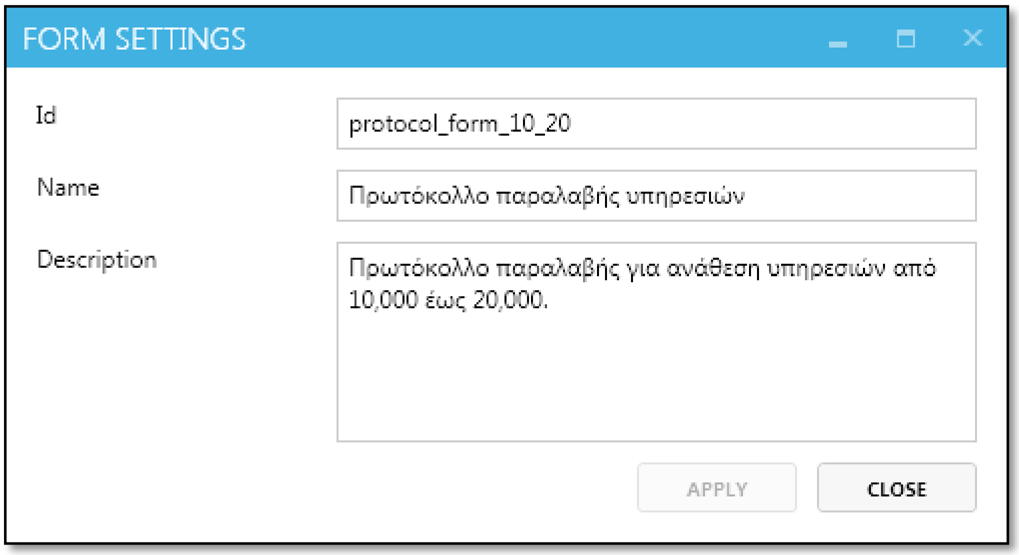


id	car_color
type	color
label	Desired color

Figure 14: Form element creation window

Form Settings

A modal window (Figure 15) is used for the modification of the form structure that is currently being edited. The properties that can be edited are the form's "Id", "Name", and "Description". The "Id" property is verified to only contain letters, numbers, and underscores. The "APPLY" button is activated if there are any changes to the properties and there no validation errors.



Id	protocol_form_10_20
Name	Πρωτόκολλο παραλαβής υπηρεσιών
Description	Πρωτόκολλο παραλαβής για ανάθεση υπηρεσιών από 10,000 έως 20,000.

Figure 15: Form settings window

3.4 File format

The form editor's file format is a human readable JSON encoded format. It defines a top-level object (the form) with informational data properties and a collection of form elements (fields). Each element can contain children if allowed by its type. A minimal example can be seen in Figure 16.

A form object is required to have a "name" string property and a "fields" array property. It can also have a "friendlyName" string property and a "description" string property. The "name" string property acts as an identifier and can only contain letters, numbers, and underscores. The "fields" array property can only contain objects that represent form elements.

```json
{
  "fields": [
    {
      "children": [],
      "type": "text",
      "id": "name",
      "label": "Name",
      "required": true
    },
    {
      "children": [],
      "type": "text",
      "id": "surname",
      "label": "Surname",
      "required": true
    },
    {
      "children": [],
      "type": "number",
      "id": "age",
      "label": "Age",
      "required": true,
      "min": "18",
      "max": "120",
      "step": "1"
    }
  ],
  "name": "sample_form",
  "friendlyName": "Personal details"
}
```

Figure 16: A minimal example for a personal details form

Form element objects are required to have an "id" string property, a "type" string enumerated property, and a "label" string property. They can also have a "description" string property, and a "children" array property if their type allows for child form elements. The "id" property follows the same identifier rules as the "name" property for the form object. Depending on the form element "type" property, other properties can be defined. A table of these properties (also referred to as attributes) can be seen in appendix B.1 Form Element Attribute Specifications. The possible values for the "type" string enumerated property are:

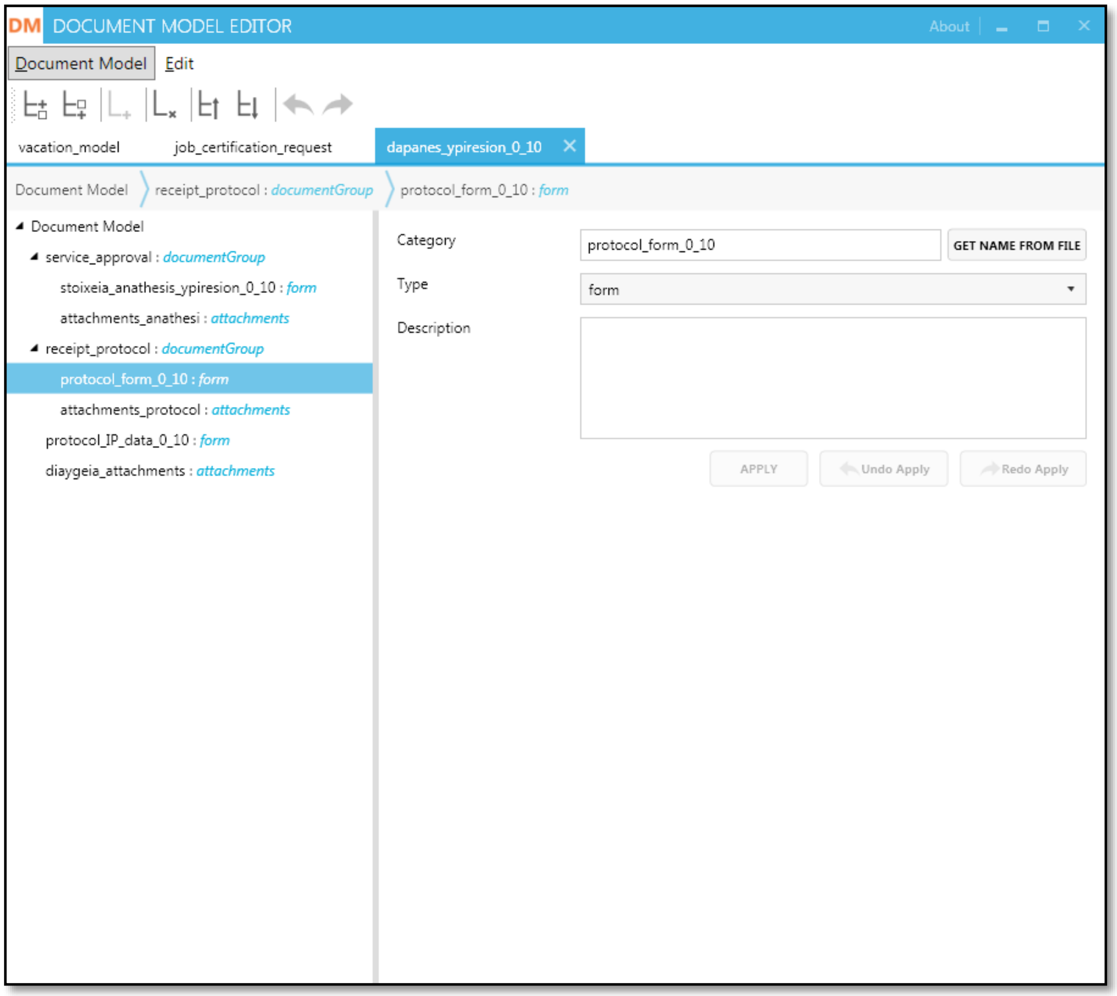
- text
- textarea
- email
- url
- tel
- range
- number
- date
- datetime
- month
- week
- time
- color
- toggle
- password
- file
- select
- togglegroup
- fieldset
- optgroup
- option

These type names follow closely the input type names defined in the HTML5 W3C candidate recommendation [9]. Attribute names also follow this rule but with a few more notable exceptions. For example, the "description" and "label" attributes for input elements do not exist in the HTML5 living standard.

4. Document Model Editor

4.1 Overview

The document model editor (Figure 17) is the authoring tool used to define document trees for use in business processes. A document tree is a tree-like structure containing two types of nodes, documents (leaf nodes) and document groups (composite nodes). A fully defined document tree is a business processes document model, meaning that all the documents that are to be part of a business process exist inside the document model.



Document Model 〉 receipt_protocol : *documentGroup* 〉 protocol_form_0_10 : *form*

- Document Model
  - service_approval : *documentGroup*
    - stoixeia_anathesis_ypiresion_0_10 : *form*
    - attachments_anathesi : *attachments*
  - receipt_protocol : *documentGroup*
    - protocol_form_0_10 : *form*
    - attachments_protocol : *attachments*
  - protocol_IP_data_0_10 : *form*
  - diaygeia_attachments : *attachments*

Category	protocol_form_0_10
Type	form
Description	

Figure 17: The document model editor's main view

A document can have various types, including text files, spreadsheets, forms defined in the form editor, and even other document models. Document groups exist only to hold other documents. Document tree nodes can be rearranged and have their properties edited by a document model node editor. Types can be edited even after creation time for leaf nodes, while composite nodes can only have their name property changed.

4.2 Architecture

Document model editor's general architecture is presented in Figure 18. Some secondary classes have been omitted for visual brevity. The architecture is divided into two layers. The business layer and the user interface layer. The business layer includes classes and data that form the backbone of the application while the user interface layer includes all the controls and classes that are relevant to the user-interface.

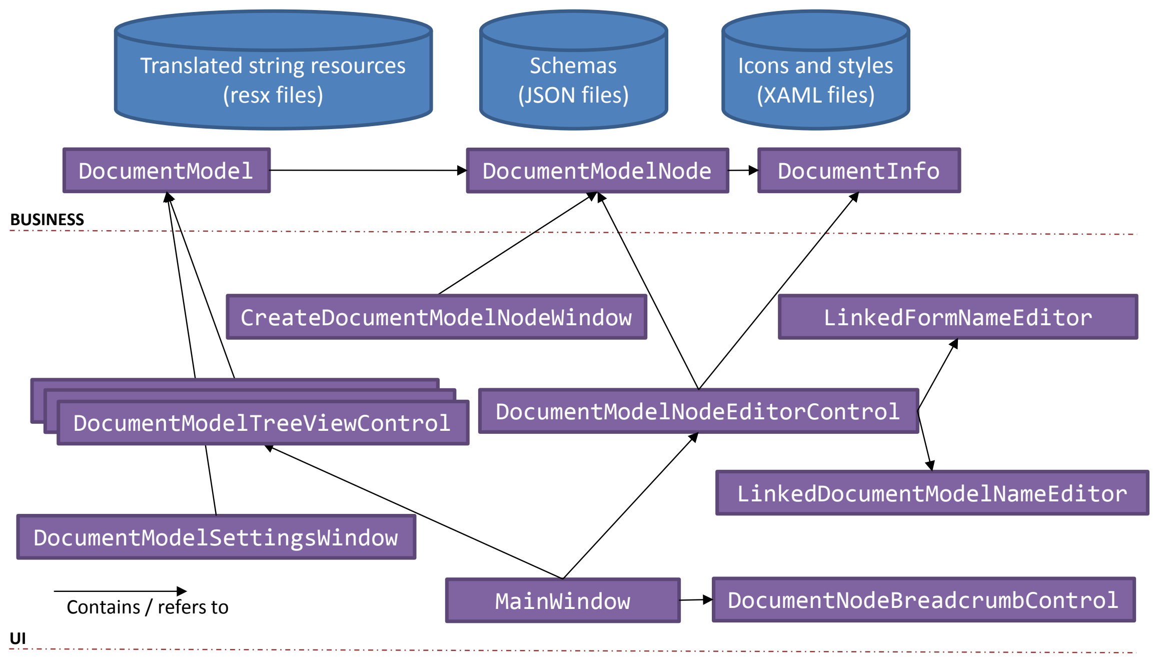

Translated string resources (resx files)
Schemas (JSON files)
Icons and styles (XAML files)
DocumentModel
DocumentModelNode
DocumentInfo
BUSINESS
CreateDocumentModelNodeWindow
LinkedFormNameEditor
DocumentModelTreeViewControl
DocumentModelNodeEditorControl
LinkedDocumentModelNameEditor
DocumentModelSettingsWindow
Contains / refers to
MainWindow
DocumentNodeBreadcrumbControl
UI

Figure 18: Document model editor architecture

Into the business layer, the class that describes a document model is `DocumentModel`. `DocumentModel` contains the properties that describe a document model, such as `Name`, `FriendlyName`, and `Description`. It contains methods for node insertion and removal as well as node reordering. Methods for object instantiation from JSON files are provided. The collection that contains all the document model nodes is located in the `Nodes` property.

The `Nodes` collection includes elements of type `DocumentModelNode`. `DocumentModelNode` objects can either be document groups (composite nodes) or documents (leaf nodes). For a `DocumentModelNode` object to be a document group its `Name` property must be defined and the `DocumentInfo` property must be null. To be a document, the `Name` property must be null and the `DocumentInfo` property must contain a valid `DocumentInfo` object. `DocumentModelNode` objects also contain a `Children` property that is only used if the document model node is a document group.

The `DocumentInfo` class contains properties that define a leaf document model node. These properties are `Type`, `Category`, `Description`, and `LinkedDocumentModelName`. `Type` describes the document, `Category` acts as the document identifier, and `Description` contains extra details about the document. `LinkedDocumentModelName` is used only if the document's type is "linkedDocumentModel" and contains the target document model name.

Apart from the classes mentioned above, the business layer includes data in the form of .resx files for translated string resources, JSON schema files for document model JSON files and form JSON files (used in the case of form linking to document model nodes), toolbar vector icons in XAML resource dictionaries, and a XAML resource dictionary for various styles.

The user interface layer consists of three main windows. These are:

- `DocumentModelSettingsWindow`, which is the document model setting dialog
- `CreateDocumentModelNodeWindow`, which is the dialog used to create document model nodes
- and `MainWindow` which is the main application window

`DocumentModelSettingsWindow` is used as a modal dialog for editing the basic descriptive document model properties and binds to `DocumentModel` objects. It uses `ConfigUIGenerator` to generate its editor user-interface. `CreateDocumentModelNodeWindow` is the modal dialog used to create and add document model nodes into the document model. It can create both document groups and documents by returning `DocumentModelNode` objects. It uses `ConfigUIGenerator` for its editor user-interface as well.

`MainWindow` is the main application window that contains the menu bar, toolbar, and the central tab control. Each tab maps to a document model to be edited. The tab control binds itself to a collection of `DocumentModelTreeViewControl`s and contains a single node editor control as well as a single node breadcrumb control. Both these controls bind to the selected document model node of the current document tree view. `DocumentModelTreeViewControl` binds to `DocumentModel` objects.

Depending on which document type is selected, the document model node editor might have to substitute the default editor for the `Category` property with a `LinkedFormNameEditor` in case of a "form" document type, or it might have to add a `LinkedDocumentModelNameEditor` editor in case of a "linkedDocumentModel" document type.

4.3 User Interface

4.3.1 Breadcrumbs

A breadcrumb control (Figure 19) for document model nodes is used to show the selected node's position in the document tree hierarchy. It is also possible to navigate to any previous node in this hierarchy by clicking on the node's crumb. The document model node breadcrumb control is defined in the `DocumentNodeBreadcrumbControl` class.

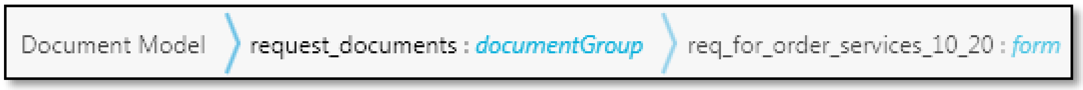

Document Model 〉 request_documents : *documentGroup* 〉 req_for_order_services_10_20 : *form*

Figure 19: Document model editor’s breadcrumb control

4.3.2 Tree View

A tree view control (Figure 20) is used to display the hierarchical structure of a document model. Each document model node is displayed as tuple of `DocumentInfo.Category` and `DocumentInfo.Type` properties if it is a document, or as tuple of the `Name` property and "documentGroup" type if it is a document group. Document groups can be collapsed or expanded while all nodes trigger a document model node editor change when clicked. The document model tree view is defined in the `DocumentModelTreeViewControl` class.

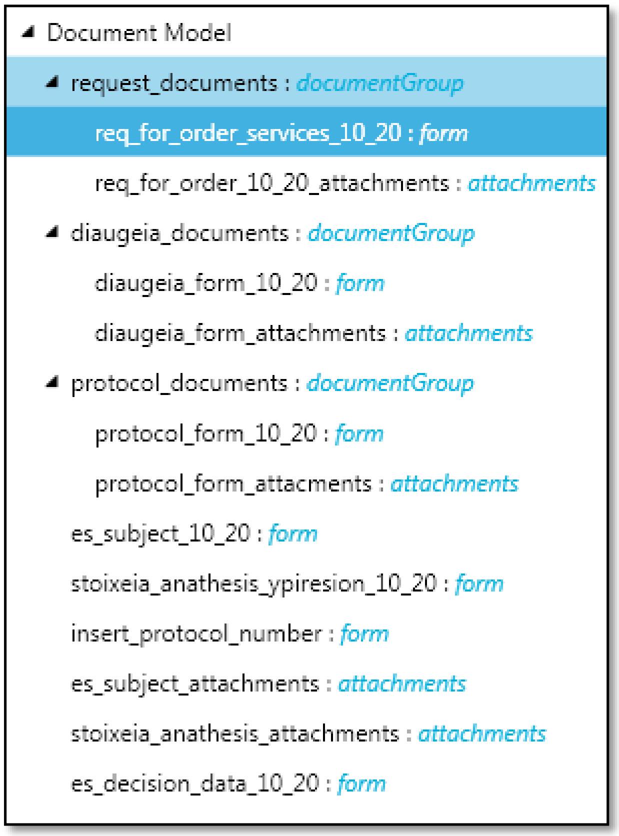

◢ Document Model
- ◢ request_documents : *documentGroup*
  - req_for_order_services_10_20 : *form*
  - req_for_order_10_20_attachments : *attachments*
- ◢ diaugeia_documents : *documentGroup*
  - diaugeia_form_10_20 : *form*
  - diaugeia_form_attachments : *attachments*
- ◢ protocol_documents : *documentGroup*
  - protocol_form_10_20 : *form*
  - protocol_form_attacments : *attachments*
- es_subject_10_20 : *form*
- stoixeia_anathesis_ypiresion_10_20 : *form*
- insert_protocol_number : *form*
- es_subject_attachments : *attachments*
- stoixeia_anathesis_attachments : *attachments*
- es_decision_data_10_20 : *form*

Figure 20: Document model editor's tree view for document model nodes

4.3.3 Node Editor

The document model node editor (Figure 21) is used for editing document groups and documents. In each case ConfigUIGenerator is used to generate the relevant editor user-interfaces. Relevant validation rules are added when the target node is changed. In the case of editing documents, editors for the `Category` property are interchanged when the document type is changed to / from "form", and a new editor for the `linkedDocumentModel` property is added / removed when the document type is changed to / from "linkedDocumentModel". An internal command history cache exists to handle undo / redo operations for each document model node in the document model. The existence of validation errors is exposed via a property to signify if changes can be applied or not. The document model node editor is defined in the `DocumentModelNodeEditorControl` class

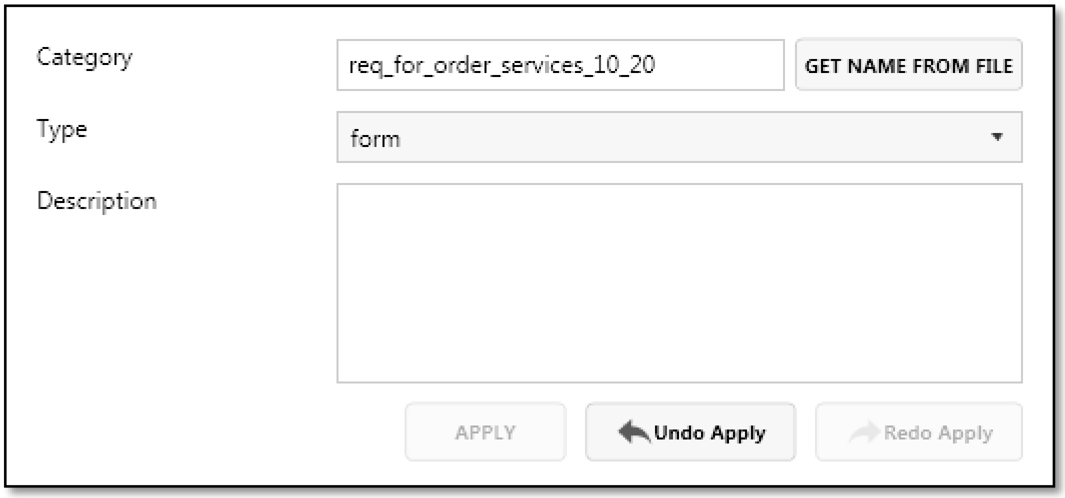
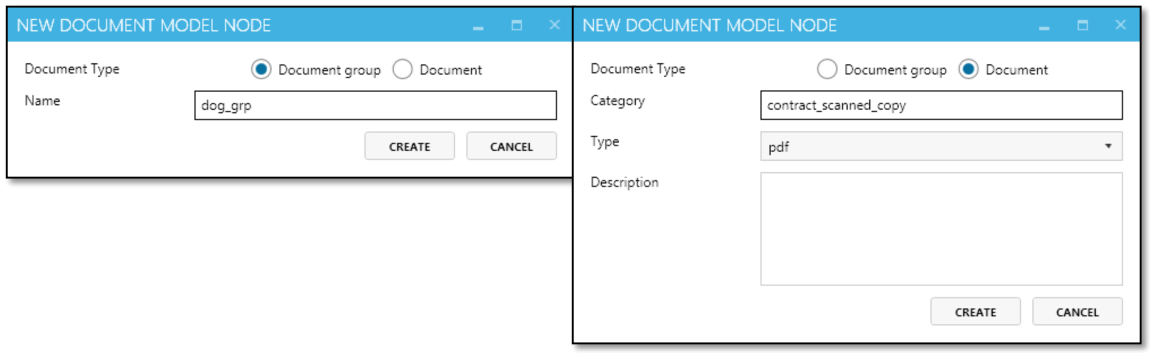

Category: req_for_order_services_10_20
Type: form
Description:

Figure 21: Document model editor's document model node editor

4.3.4 Other Features

Document Model Node Creation

A modal window (Figure 22) is used for the creation of either documents or document groups. In the case of the latter an editor is provided for `DocumentModel`'s `Name` property. For the former, editors are provided for `DocumentInfo`'s `Category`, `Type`, and `Description` properties. The `Name` and `Category` properties are validated for their uniqueness and for their values only containing letters, numbers, and underscores. In case of validation errors the "CREATE" button is deactivated, blocking the addition of a new node into the document model.


NEW DOCUMENT MODEL NODE

Document Type: [✓ Document group] Document

Name: dog_grp

NEW DOCUMENT MODEL NODE

Document Type: Document group [✓ Document]

Category: contract_scanned_copy

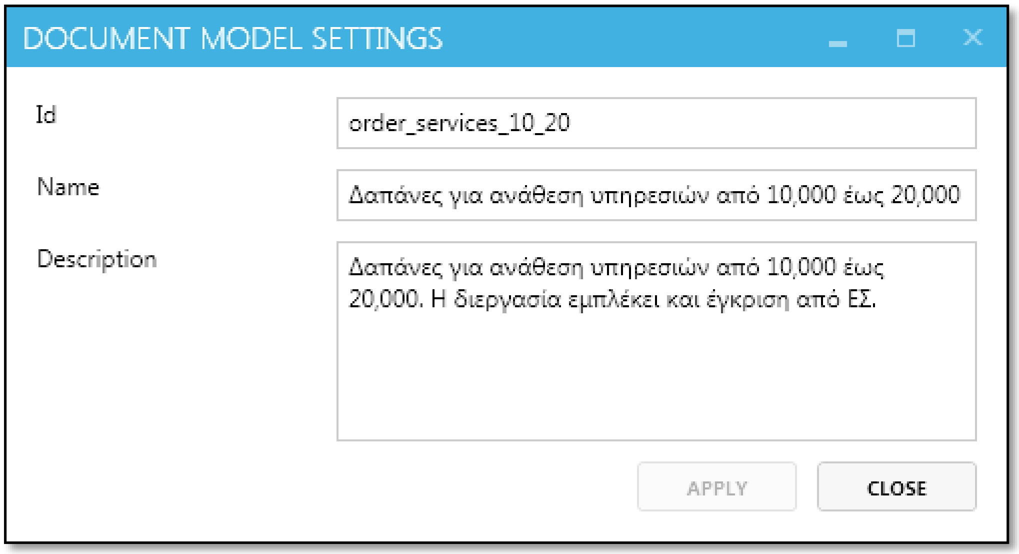
Type: pdf

Description:

Figure 22: Document model node creation window

Document Model Settings

A modal window (Figure 23) is used for the modification of the document model that is currently being edited. The properties that can be edited are the document model's "Id", "Name", and "Description". The "Id" property is verified to only contain letters, numbers, and underscores. The "APPLY" button is activated if there are any changes to the properties and there no validation errors.


DOCUMENT MODEL SETTINGS

Id: order_services_10_20

Name: Δαπάνες για ανάθεση υπηρεσιών από 10,000 έως 20,000

Description: Δαπάνες για ανάθεση υπηρεσιών από 10,000 έως 20,000. Η διεργασία εμπλέκει και έγκριση από ΕΣ.

Figure 23: Document model settings window

4.4 File Format

The document model editor's file format is a human readable JSON encoded format. It defines a top-level object that represents a document model for a business process. This object contains informational data properties and a collection of document model nodes (nodes). Each node can contain children if it's a document group. A minimal example can be seen in Figure 24.

A document model object must have a "name" string property that can only contain letters, numbers, and underscores. It can also contain the "friendlyName" and "description" optional string properties. A required array property called "nodes" is responsible for containing all the document tree nodes.

```
{
  "name": "job_application_documents",
  "nodes": [
    {
      "name": "job_application",
      "children": [
        {
          "documentInfo": {
            "category": "cv",
            "type": "pdf",
            "description": ""
          },
          "children": []
        },
        {
          "documentInfo": {
            "category": "cover_letter",
            "type": "form",
            "description": ""
          },
          "children": []
        }
      ]
    }
  ],
  "friendlyName": "Job application document model"
}
```

Figure 24: A minimal example for a job application document model

The "nodes" array can have items of type object. These objects can have a "name" string property, a "children" array property, and a "documentInfo" object property. The "name" string property acts a document group identifier. It is unique in the context of document group names and can only contain letters, numbers, and underscores. The "children" array property contains similar objects to the "nodes" array and is used if a document group is described.

The "documentInfo" object property defines objects that have required "category" string properties and "type" enumerated properties, and optional "description" and "linkedDocumentModelName" string properties. The possible values for "type" are:

- doc
- text

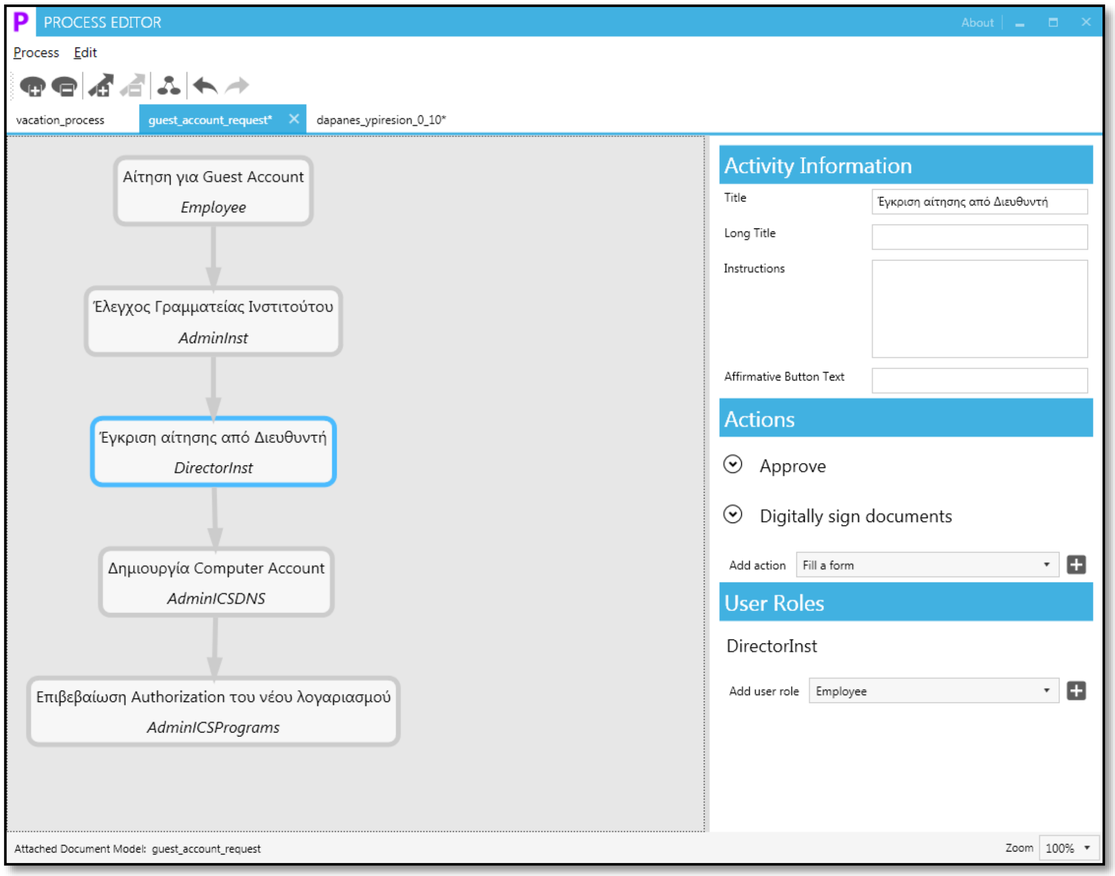
- pdf
- xls
- bin
- image
- form
- linkedDocumentModel
- attachments

Most of these types are self-explanatory but two warrant further explaining, "attachments" and "linkedDocumentModel". The "attachments" type defines a document model node that can contain an arbitrary number of file attachments of any file type. The "linkedDocumentModel" type defines a special document model node that refers to another document model.

5. Process Editor

5.1 Overview

The process editor (Figure 25) is the tool used to author business processes for use in the process runtime system. Processes are designed as directed graphs with exactly one node with no incoming edges and exactly one node with no outgoing edges. The node with no incoming edges is the starting point of the process, and the node with no outgoing edges is the final step of the process. Each node represents a process step that is called an *activity*. Each activity can include multiple *actions* and can be associated with one or more *user roles*.



Αίτηση για Guest Account
Employee

Έλεγχος Γραμματείας Ινστιτούτου
AdminInst

Έγκριση αίτησης από Διευθυντή
DirectorInst

Δημιουργία Computer Account
AdminICSDNS

Επιβεβαίωση Authorization του νέου λογαριασμού
AdminICSPrograms

Title: Έγκριση αίτησης από Διευθυντή
Long Title:
Instructions:
Affirmative Button Text:

Approve
Digitally sign documents
Add action: Fill a form

DirectorInst
Add user role: Employee

Attached Document Model: guest_account_request
Zoom: 100%

Figure 25: The process editor's main view

Activities generally describe a set of actions needed to progress to the next activity in the sequence of the process. Actions are essentially all the functionality that the business process runtime system can provide to the user. For example, digitally signing a document is considered an action. The same applies for filling a form or reviewing a document. Actions have types with the most major type being "DOCUMENT". Actions of this type act upon a set of documents defined in the document model that is attached to the business process. Other type include: "NSA" for non-system actions, "SYSTEM" for system actions, and "SAP" for actions that interface with SAP systems.

User roles define what a user can do in an organization. A single user can be associated with multiple user roles, and a user role can be a superset of user roles. This means that the user role set has a tree-like hierarchy. Example roles include "Employee", "President" etc. The user whose role is associated with the initial activity in a process has the ability to initiate the business process and is called *initiator*.

Business processes generally work the same for all the units in an organization. For the business processes that work differently in certain units, affiliated units can be defined for a process. For example, if a process is defined for unit A, an employee of unit B cannot initiate the process even if the initiating roles for the process include "Employee". Units are organized in tree-like structures with each unit possibly containing sub-units.

5.2 Architecture

Process editor's general architecture is presented in Figure 26. Some secondary classes have been omitted for visual brevity. The architecture is divided into two layers. The business layer and the user interface layer. The business layer includes classes and data that form the backbone of the application while the user interface layer includes all the controls and classes that are relevant to the user-interface.

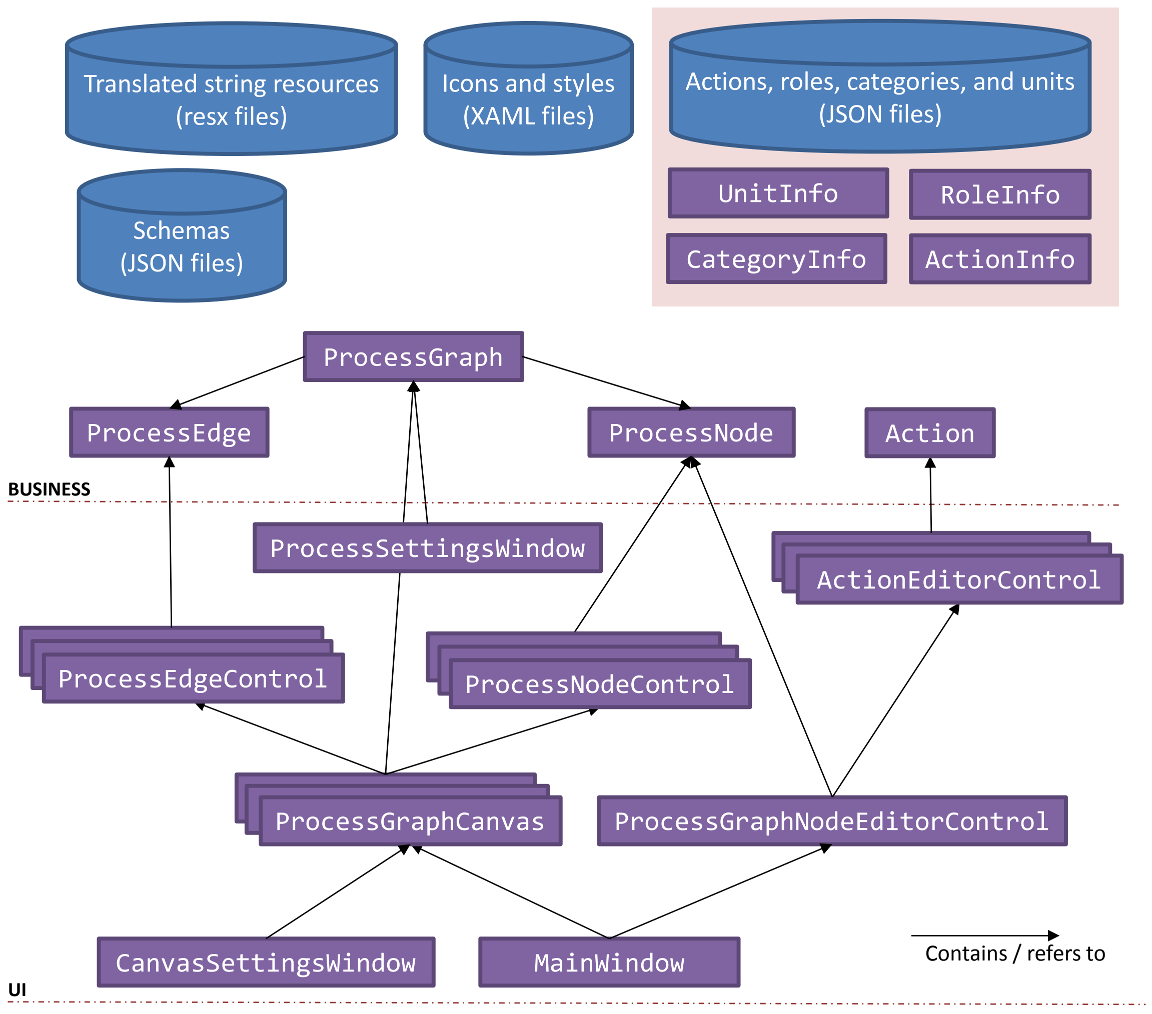


Figure 26: Process editor architecture

The business layer includes core classes that define models for the process graph and its components: `ProcessGraph`, `ProcessEdge`, `ProcessNode` and `Action`. Also included are classes that provide runtime information on process categories, organization units, user roles and actions. These classes are `CatgoryInfo`, `UnitInfo`, `RoleInfo` and `ActionInfo` respectively.

The `ProcessGraph` class contains properties with general information about the business process such as `AttachedDocumentModelName` and `AttachedDocumentModelPath` that define the name and physical file path of the attached document model, `UnitPaths` which is a set of organizational unit path strings, and `Name` which is the processes' identifier. Also contained are the `Edges` and `Activities` dictionaries for process graph edges and nodes respectively. These dictionaries are indexed by unsigned integer keys. Methods are provided for the addition and removal of nodes and edges, as well as for the relocation of edge targets.

`ProcessNode` is the class that represents a process activity. It has informational string properties such as `Title`, `LongTitle`, and `Instructions`. The `Id` unsigned integer property is used as a unique identifier for the node. `Actions` is a collection of `Action` objects. Target process node ids are held in the `Targets` set, while user role strings are held in the `UserRoles` set. Methods are provided for the addition and removal of user roles and targets, and for the reordering of actions. `ProcessEdge` is a simple class that includes the keys for the source and target process nodes.

Apart from the classes mentioned above, the business layer includes data in the form of .resx files for translated string resources, JSON schema files for process JSON files, JSON catalog files for organizational units, user roles, process categories, and activity actions, toolbar vector icons in XAML resource dictionaries, and a XAML resource dictionary for various styles.

The UI layer has three main window classes: `CanvasSettingsWindow`, `ProcessSettingsWindow`, and `MainWindow`. The rest are editor controls (`ProcessGraphNodeEditorControl` and `ActionEditorControl`) and controls for drawing the process graph (`ProcessGraphCanvas`, `ProcessNodeControl` and `ProcessEdgeControl`).

`CanvasSettingsWindow` is used as a modal window for setting the visual settings for the process graph. These settings include the background, connector color, activity background color, etc. The window binds to the currently edited `ProcessGraphCanvas` object. `ConfigUIGenerator` is used for the automatic generation of the editors.

`ProcessSettingsWindow` is used as a modal window for editing the processes' informational properties such as `Id` and `Name`. It is also used to select the organizational units assigned to the process, and the category where the process belongs to. The window binds to the process graph of the currently edited `ProcessGraphCanvas` object. `ConfigUIGenerator` is used for the automatic generation of the informational property editors.

`MainWindow` is the main application window that contains the menu bar, toolbar, and the central tab control. Each tab maps to a process graph to be edited. The tab control binds itself to a collection of `ProcessGraphCanvas`es and contains a single `ProcessGraphNodeEditorControl`. `ProcessGraphNodeEditorControl` binds to the selected process graph node (or activity) of the current process graph canvas. `ProcessGraphCanvas` binds to `ProcessGraph` objects. `ProcessGraphNodeEditorControl` also can contain multiple editors for activity actions of type `ActionEditorControl`. `ActionEditorControl`s bind to `Action` objects.

A process graph canvas includes multiple `ProcessNodeControl` and `ProcessEdgeControl` controls to visualize the process graph nodes and edges respectively. Both these control map to their business layer counterparts: `ProcessEdge` and `ProcessNode`.

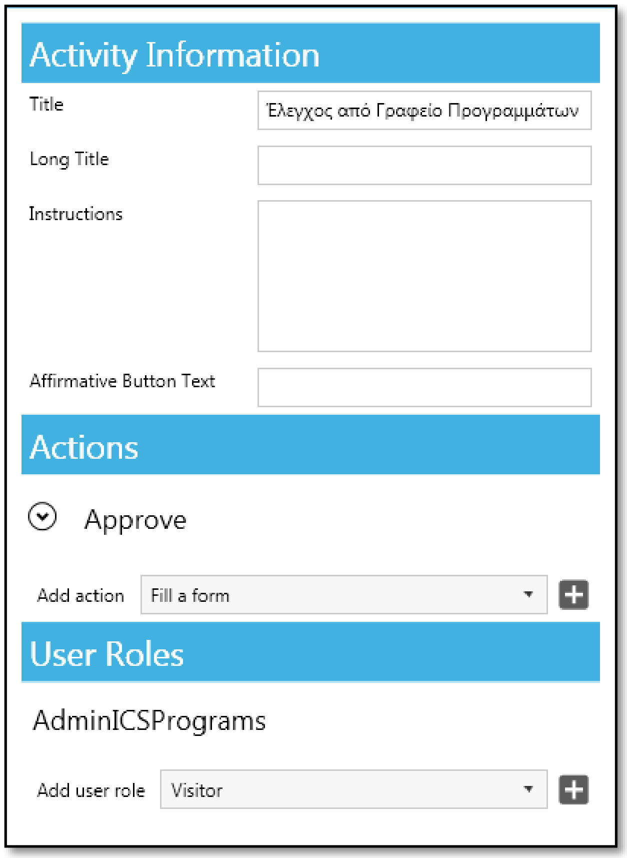
5.3 User Interface

5.3.1 Process Graph Node Editor

The process graph node editor (Figure 27) is used for editing the properties of the selected process graph node (or activity). The editors are split into three sections; *activity information*, *actions*, and *user roles*. The *activity information* section consists of these settings:

- Title: The activity's title, short form

- Long Title: The activity's title in long form. Optional
- Description: Activity description. Optional
- Affirmative Button Text: Text to override the default affirmative button text. Optional


Activity Information

Title	Έλεγχος από Γραφείο Προγραμμάτων
Long Title	
Instructions	
Affirmative Button Text	

Actions

Approve

Add action: Fill a form

User Roles

AdminICSPrograms

Add user role: Visitor

Figure 27: Process editor's graph node editor

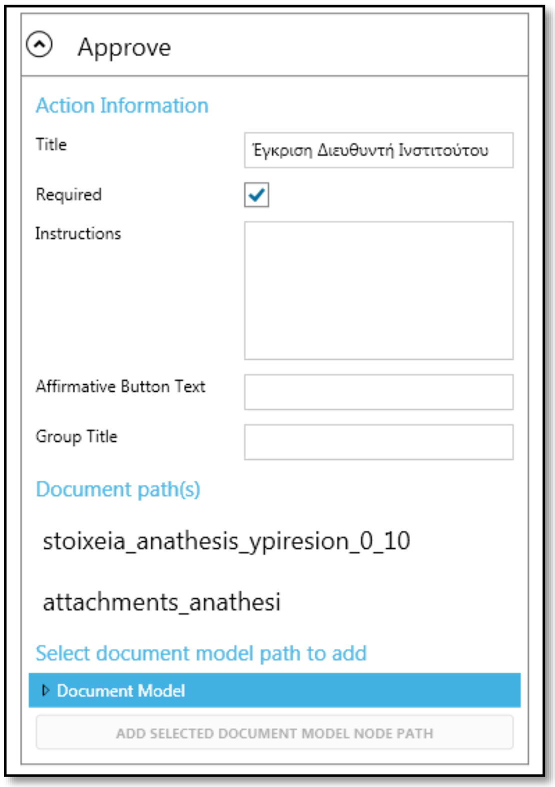
The *actions* section allows the user to add, remove and reorder actions for the selected node. Each action is represented by an action editor that is detailed below. Available actions are populated by the action catalog.

Finally, the *user roles* section allows for the addition and removal of user roles into the selected node.

The process graph node editor is defined in the `ProcessGraphNodeEditorControl` class.

5.3.2 Action Editor

The action editor (Figure 28) is used for editing activity actions. There are editors for action information, and—depending on the activity type—editors for the addition / removal of document paths, and editors for linking another business process to the activity.


Approve

Action Information

Title: Έγκριση Διευθυντή Ινστιτούτου

Required: ✓

Instructions:

Affirmative Button Text:

Group Title:

Document path(s)

stoixeia_anathesis_ypiresion_0_10

attachments_anathesi

Select document model path to add

Figure 28: Process editor's action editor

Action information settings include:

- Title: The action’s title. Optional
- Required: Indicates if the action is required. Default value is true
- Instructions: Action instructions. Optional

- Affirmative Button Text: Text to override the default affirmative button text. Optional
- Group Title: An action group title. Actions with the same group title get visually grouped in business process system runtime. Optional

The editor that adds / removes document paths to the action is used for actions of type "DOCUMENT" and allows for the selection of multiple document paths from the document model attached to the process via a tree view identical to the one used in the document model editor.

The editor that links another business process to the activity is used for actions of the type "LAUNCHPROCESS". It can get the linked business process name via a file selector.

The action editor is defined in the `ActionEditorControl` class.

5.3.3 Process Graph Canvas

The process graph canvas (Figure 29) is used for the visualization and the manipulation of the process graph. It derives from WPF's `Panel` class and overrides the `MeasureOverride` and `ArrangeOverride` methods for custom layout behavior. Part of this custom behavior is the support for zoom in / zoom out. The canvas provides methods for adding / removing edges and nodes from the graph, and for populating the canvas from a process JSON file and optionally a visual information JSON file. A command history is kept for undo / redo support. The graph canvas intercepts mouse events to clear node / edge selection, to drag inserted edges, and to raise an event for getting focus.

Process graph canvas exposes properties for the process graph it keeps, the attached document model, the node control that is currently selected, and the current scale factor (zoom). Properties for visual elements are also exposed, such as the node border brush, the edge foreground brush, and the node text brush. The canvas also defines the `X` and `Y` attached properties for all the child object coordinates within the canvas.

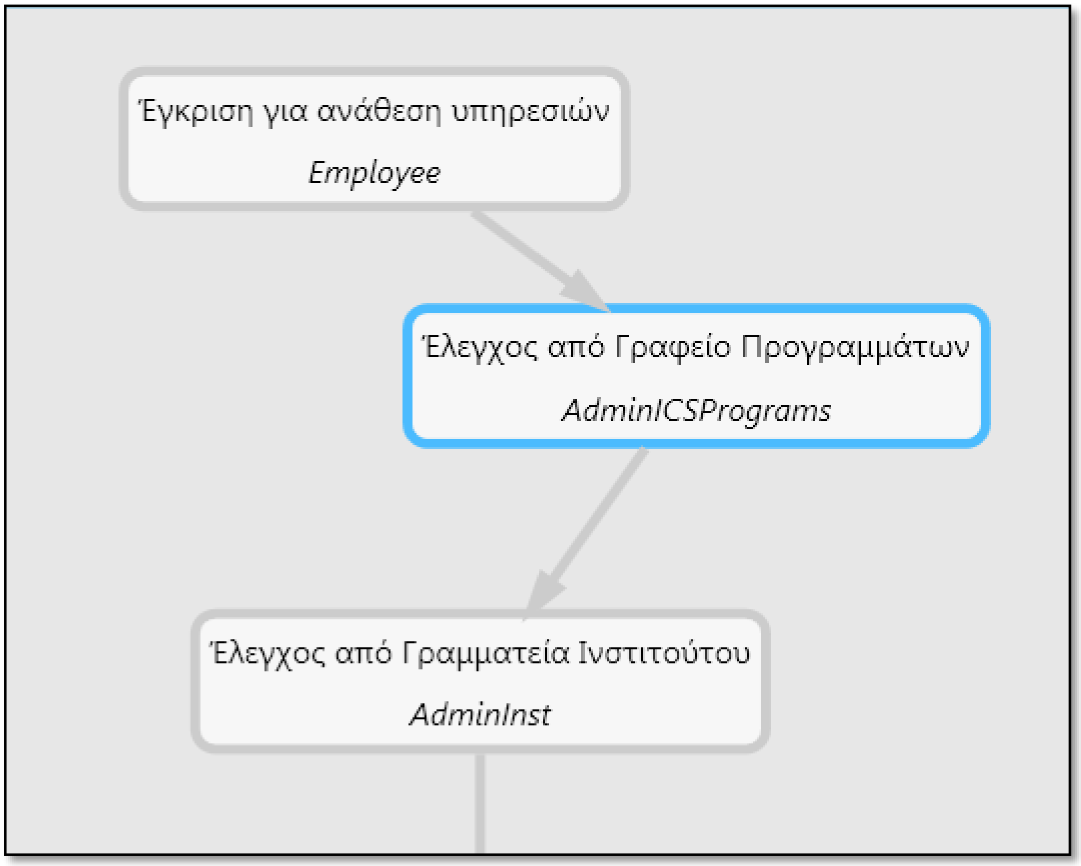

Έγκριση για ανάθεση υπηρεσιών
Employee
Έλεγχος από Γραφείο Προγραμμάτων
AdminICSPrograms
Έλεγχος από Γραμματεία Ινστιτούτου
AdminInst

Figure 29: Process editor's graph canvas

The process graph canvas is defined in the `ProcessGraphCanvas` class.

5.3.4 Other Features

Pan Scroll Viewer

Pan scroll viewer is a control that extends WPF's `ScrollViewer` class to add panning behavior via pressed middle mouse button dragging. It is used to contain the process graph canvas. It is defined in the `PanScrollViewer` class.

Process Node

The process node control visualizes an activity on the graph canvas. It is presented as a square with a thick border and rounded edges. Inside is the activity's title on the top, and the participation user role(s) in the bottom. The control intercepts mouse events for node dragging within the parent canvas. It also raises an event when its drag is complete.

Process Edge

The process edge control is a path in the form of an arrow that connects to process graph nodes together. Its `Source` and `Target` properties include the source and target nodes respectively. Using the coordinates and size of these nodes the process edge can calculate its initial and final position. The control also intercepts mouse events for edge dragging within the parent canvas. It also raises an event when the edge drag is completed.

5.4 Graph Layout

While graph nodes can be arranged manually with their coordinates being saved in a separate JSON file, there also exists the option to layout graph nodes automatically. This is achieved by using a simplified version of Sugiyama's scheme [23].

Graph node position calculation is done by the static template class `GraphLayoutCalculator<T>`, where T refers to the node key type. `GraphLayoutCalculator` provides a single public method named `Calculate`. `Calculate` accepts a collection of `GraphNodeData` objects (explained below), two double values for padding and margin, and returns a dictionary of `Point` objects mapped to the node key type.

`GraphNodeData<T>` is a template class that describes a graph node. The properties that is exposes are:

- `Key`: the node key of type T
- `NeighborNodes`: A set of neighboring node keys
- `Width`: The node width, of type double
- `Height`: The node height, of type double

Internally when calculating the graph layout, the graph layout calculator makes sure there are no circles. If there are, an exception is thrown. Then, the first layer is populated by nodes that have no incoming edges. With the first layer populated, the nodes in the next layers will be populated by their neighboring nodes. With each layer pass, neighbor nodes that were in lower layers will go into the top layer and have dummy nodes replace their previous positions. When there are no nodes left in the top layer, the layers are returned for the final layout calculation. Nodes are positioned in grid cells that are calculated by the max node width and height. With the node positions calculated, the `Calculate` method returns a node position dictionary of type `Dictionary<T, Point>`.

5.5 File Formats

5.5.1 Action Catalog

The action catalog is a JSON file that includes all the actions that are available to the process editor. The action catalog is defined by a top-level object that contains an "actions" array property. Each element in this array is an object. This object includes a "name" string property that is required and can only contain letters, numbers, and underscores. Also required is a "friendlyName" string property. The "description" string property is optional, as well as the "terminatesProcess" Boolean property. The "type" enumerated string property is required and can have one of these values:

- DOCUMENT: For actions that use document tree nodes
- NSA: For actions that are executed outside of the process runtime system
- SAP: For actions that interface with SAP systems
- SYSTEM: For actions that are provided by the process runtime system
- LAUNCHPROCESS: For special actions that can launch other processes

5.5.2 Unit Catalog

The unit catalog is a JSON file that contains a tree structure holding the units and subunits of an organization. The unit catalog is defined as a top-level object that includes a "units" array property of objects. Each of these objects contains the required "name" and "category" properties, and can require a "subUnits" property. The "name" property is a string, "category" is an enumerated string. Current possible values for "category" are:

- Organization
- Institute
- Laboratory
- Administration

The "subUnits" property is an array of objects that are similar to the objects included the "units" array property

5.5.3 Category and Role Catalogs

The category catalog contains all the available categories that a business process can belong to. The role catalog contains all the available roles in an organization that can participate in business process activities. Both these catalogs are JSON files that contain an array of strings.

5.5.4 Process Files

The process editor's file format for process files is a human readable JSON encoded format. It defines a top-level object that represents a model for a business process. This object contains informational properties and a map of activity objects.

In more detail, the top-level object must contain a "name" string property. This property acts as an identifier and can only contain letters, numbers, and underscores. The "friendlyName" and "description" string properties are optional. The "categories" string property is also optional and can define in which category the business process belongs. The "attachedDocumentModelName" and "attachedDocumentModelPath" string properties exist to define the identifier and the fully qualified path of a document model to be attached to the business process. Additionally "attachedDocumentModelName" has the same value pattern as the "name" identifier property. The amp of activity objects is suitably called "activities" and includes activity objects indexed by their unsigned integer ids.

Activity objects may contain the "title", "longTitle", "instructions", "affirmativeText" informational string properties. These objects can also contain sets of user roles (strings) and node targets (unsigned integers) appropriately named "userRoles" and "targets" respectively. Finally, activity objects can contain an "actions" array property containing action objects.

Action objects are required to define a "name" property that identifies the action. Action objects may define the "title", "instructions", "affirmativeText", and "groupId" informational string properties. Actions with the same "groupId" value get visually grouped in the business process runtime. Action objects may define a "documentPaths" set of dot-delimited document model paths. This set is relevant only when the action that is referred to in the "name" property is specified to be of type "DOCUMENT". The final property that an action object may have is the "linkedProcessName" string property. This property is an identifier for another process model name and is relevant only to actions of type "LAUNCHPROCESS".

6. General Features

6.1 Localization

As localization was an important concern, the authoring tools were designed with localizability in mind. There are three methods used for localization: key-value pairs from BAML files, localized resource files (*.resx), and localized attributes for property display names and descriptions.

The authoring tools are also globalized. This essentially means that a default UI culture had to be set (en-US in our case) along with a neutral resource language as a fallback. After these additions, building a project will result in a satellite resource assembly being generated along with it.

6.1.1 Key-Value Pairs from BAML Files

To translate strings defined in XAML files, key-value pairs must be extracted from the BAML form of XAML files. The keys in these key-value pairs are Uid properties that must be set for each translatable element. This allows to track and merge changes that happen in the localization process during development time. The Uid values have to be unique and are best added automatically by a tool to avoid key collisions. The msbuild tool provides this functionality with the `/t:updateuid` parameter to add Uids to XAML files, and the `/t:checkuid` parameter to check Uids in XAML files.

Resource Key	Localization Category	Value
TextBlock_1:System.Windows.Controls.TextBlock.Text	Text	Insert Form Element Before
TextBlock_2:System.Windows.Controls.TextBlock.Text	Text	Insert Form Element After
TextBlock_3:System.Windows.Controls.TextBlock.Text	Text	Insert Child Form Element
TextBlock_4:System.Windows.Controls.TextBlock.Text	Text	Remove Form Element
TextBlock_5:System.Windows.Controls.TextBlock.Text	Text	Move Form Element Up
Button_3:System.Windows.Controls.ContentControl.Content	Button	Apply

Table 1: Sample translatable key-value pairs

Once a project is built, key-value pairs from BAML files can be extracted by parsing the also generated satellite assembly. Tools like LocBaml [24] can get this job done. LocBaml outputs CSV files with the translatable key-value pairs (Table 1). These CSV files can be later used to generate satellite assemblies that include translated resources.

6.1.2 Resource Files

Localized resource files are used for translatable string resources that are not present in XAML files. Name-value pairs are defined in XML formatted (*.resx) files which can then be converted into binary .resources files. These binary resource files can then be embedded within satellite assemblies. Resource data can be accessed in code via automatically generated classes.

To build a satellite assembly with the localized resources a .resx file that includes the desired culture as an extension before the .resx extension needs to be created. For example, if the neutral culture resource file is named "Messages.resx", the Greek resource file would be named "Messages.el.resx" and the French Canadian resource file would be named "Messages.fr-CA.resx".

6.1.3 Localized Attributes

ConfigUIGenerator allows for the displaying of property display names and descriptions defined in property attributes. These attributes are not localized by default. To be able to make localized property display names and descriptions two new attributes had to be created: `LocalizedDisplayNameAttribute` which is derived from `DisplayNameAttribute` and `LocalizedDescriptionAttribute` which is derived from `DescriptionAttribute` (Figure 30). Both these attributes look up resource data for translated strings of property display names and descriptions.

```
class LocalizedDisplayNameAttribute : DisplayNameAttribute
{
    public LocalizedDisplayNameAttribute(string displayName)
    {
        this.DisplayNameValue=Messages.ResourceManager.GetString(displayName);
    }
}
```

Figure 30: The LocalizedDisplayNameAttribute class

6.2 Validation

6.2.1 JSON Schema

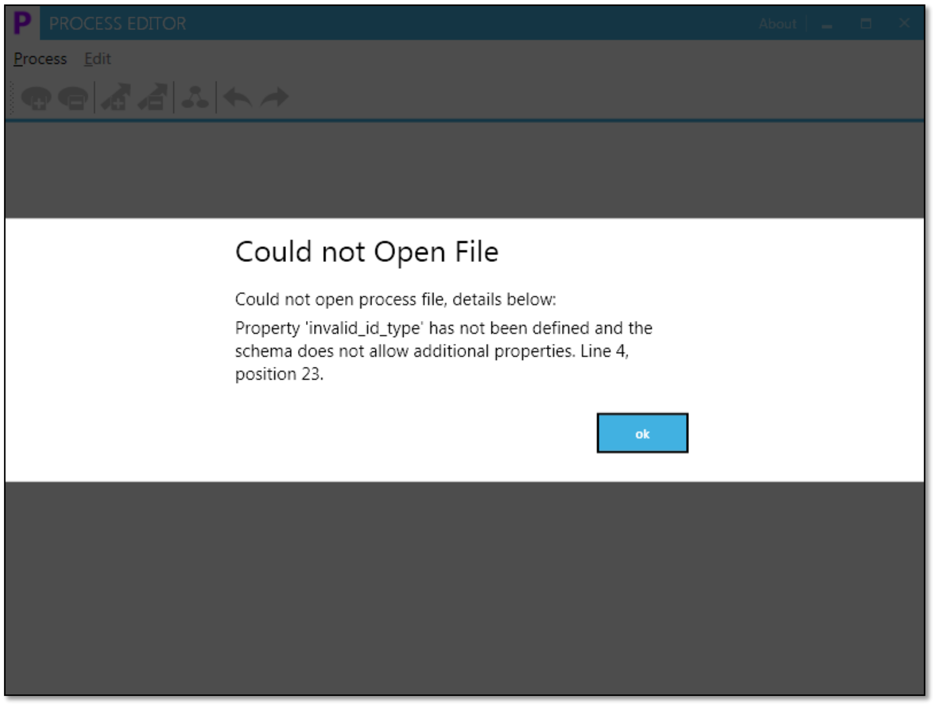
To validate JSON files version 3 of JSON schema is used. JSON schema is *"a JSON based format for defining the structure of JSON data"* [6]. JSON schema defines a JSON format that describes the structure of JSON objects and their properties (including value type, required definition, and description among others). Apart from validation purposes JSON schema can be used for documentation, hyperlink navigation, and interaction control of JSON data.

An example JSON schema can be seen in Figure 31, it describes a catalog of business process activity action specifications. This JSON schema defines an object that can include an array property of name "actions". The "actions" array includes objects that must define a "name" string property and must satisfy a certain regular expression pattern, can define a "friendlyName" string property, can define a "description" string property, can define a "type" property that only be set as one of the enumerated values given, and can define a "terminatesProcess" Boolean property.

```
{
  "$schema": "http://json-schema.org/draft-03/schema#",
  "title": "Actions",
  "description": "A catalogue of business process activity action
specifications",
  "type": "object",
  "properties": {
    "actions": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "name": {
            "type": "string",
            "pattern": "^\\w+$",
            "description": "Action identifier, can only contain letters,
numbers, and underscores",
            "required": true
          },
          "friendlyName": { "type": "string" },
          "description": { "type": "string" },
          "type": {
            "enum": ["DOCUMENT", "NSA", "SAP", "SYSTEM", "LAUNCHPROCESS"]
          },
          "terminatesProcess": { "type": "boolean" }
        }
      }
    }
  }
}
```

Figure 31: A JSON schema for a catalog of process actions

In this work JSON schema is used to define JSON formats for user role catalogs, process activity action catalogs, document models, form structures, and process models. Full schemas are given in Appendix A (data schemas). All JSON files are validated based on their schema when they are opened in their respective authoring tool. In case of validation errors, error messages are displayed along with descriptive messages of the errors (Figure 32).



Could not Open File

Could not open process file, details below:

Property 'invalid_id_type' has not been defined and the schema does not allow additional properties. Line 4, position 23.

Figure 32: JSON schema validation error handling

6.2.2 Programmatic JSON Validation

Due to limitations in the Json.NET library and JSON schema not all JSON file validation can be achieved via JSON schema files. In these cases, validation is achieved via programmatic means. Json.NET exposes the necessary classes—in our case `JsonTextReader`—for parsing JSON files. This means that JSON files can be parsed and validated before being serialized if needed.

An example case of programmatic JSON evaluation can be seen in the DocumentModelEditor authoring tool. The additional validation constraints are that within any given document model node object only one of the "name" and "documentInfo" properties can be defined. Further, "documentInfo.category" and "name" values must be unique within the context of a document model. The first constraint can be expressed in JSON schema by defining multiple value types for a given property. These value types can be seen as sub-schemas. Since, as of this writing, Json.NET does not support multiple value type definitions for properties, validation for this constraint had to be implemented programmatically. The second constraint cannot be expressed in JSON schema altogether and had to be implemented programmatically as well.

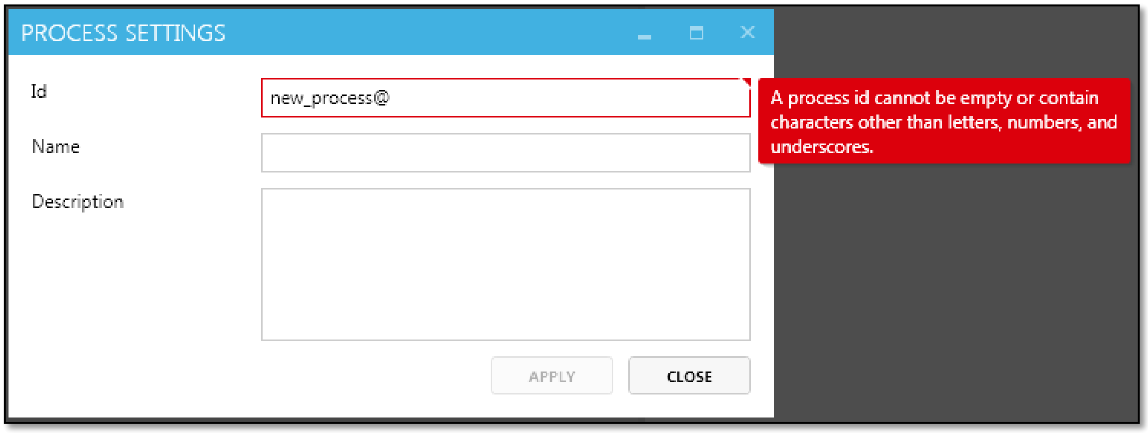
6.2.3 Runtime Validation

In many editor fields in the authoring tools not all values can be accepted at all times. User input needs to be validated and visual feedback needs to be shown. The WPF binding mechanism is used to bind all the model property values to their respective editor values. As such, binding validation is used. To insert validation logic into a binding one must add `ValidationRule`-derived objects to the `ValidationRules` property of a `Binding` object. The `ValidationRule`-derived objects must override the `Validate` method with their own validation logic by returning a `ValidationResult` object indicating successful or unsuccessful validation results. An example validation rule used in the Common library that checks if a string matches a regular expression pattern can be seen in Figure 33.

```
public class MatchRegexValidationRule : ValidationRule
{
    private string _pattern = "";
    public string Pattern
    {
        get { return _pattern; }
        set { _pattern = value; }
    }

    private string _validationErrorMessage = "Pattern not matched";
    public string ValidationErrorMessage
    {
        get { return _validationErrorMessage; }
        set { _validationErrorMessage = value; }
    }

    public override ValidationResult Validate(object value, CultureInfo
cultureInfo)
    {
        string strval = value.ToString();

        if (!Regex.IsMatch(strval, Pattern))
        {
            return new ValidationResult(false, ValidationErrorMessage);
        }
        else
        {
            return new ValidationResult(true, null);
        }
    }
}
```

Figure 33: MatchRegexValidationRule checks if a string matches a regular expression pattern

When data is marked as invalid, an error adorner is rendered on top of the editor providing visual feedback (Figure 34). This adorner has a customizable look-and-feel. Also, the `Validation.HasError` attached property is set to true and a `Validation.Error` event is triggered. Validation error messages are exposed through the `Validation.Errors` attached property.



A process id cannot be empty or contain characters other than letters, numbers, and underscores.

Figure 34: A rendered validation error adorner

In some cases, validation rules need to be added and removed from a binding dynamically. This can be achieved my manipulating the `ValidationRules` property of a `Binding` object. For GridConfigUI editors, the same effect can be achieved by adding new property specifications with new validation rules.

6.3 Undo / Redo

The Common library provides undo / redo functionality via the `CommandHistory` class and the `IUndoableCommand` interface. `CommandHistory` contains functionality for undo / redo operations, for marking history as dirty, and for adding new undoable commands to the stack. `IUndoableCommand` is a simple interface that defines properties and methods for an undoable command.

6.3.1 CommandHistory Class Overview

Figure 35 shows `CommandHistory`'s public members. First is the `PropertyChanged` event that fires when of its public properties changes. `CanUndo` and `CanRedo` are read-only properties that indicate if undo and redo operations are possible respectively. `HasUnsavedChanges` is a read-only property that exposes the history state. It is true when either the history contains unsaved commands, or when the history has been explicitly set as dirty. `CurrentUndoCommand` and `CurrentRedoCommand` are again read-only properties that return the top command from the undo and redo stacks respectively. There are two methods for undo and redo that are named as such, and a method called Edit that adds undoable commands to the command history (thus "editing" the command history). The MarkAsSaved and MarkAsUnsaved methods control the explicit marking of the command history as not dirty and dirty.

```
public class CommandHistory : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    public bool CanUndo { get; }

    public bool CanRedo { get; }

    public bool HasUnsavedChanges { get; }

    public IUndoableCommand CurrentUndoCommand { get; }
    public IUndoableCommand CurrentRedoCommand { get; }

    public void Undo();
    public void Redo();

    public void Edit(IUndoableCommand command);

    public void MarkAsSaved();
    public void MarkAsUnsaved();
}
```

Figure 35: CommandHistory's public members

6.3.2 IUndoableCommand Interface Overview

Figure 36 shows the IUndoableCommand interface in its entirety. Apart from the expected Undo and Redo methods, IUndoableCommand provides a CanRedo read-only property that indicates if an undo is possible, and a Name read-only property that contains a displayable name for the command. The Name property can be localized by using resource files and returning localized strings (see 6.1.2 Resource Files).

```
public interface IUndoableCommand
{
    string Name { get; }
    bool CanRedo { get; }

    void Undo();
    void Redo();
}
```

Figure 36: The IUndoableCommand interface

6.3.3 Per-editor Command History in DocumentModelEditor and FormEditor

DocumentModelEditor and FormEditor apart from including a global command history for each form or document model that is open, also include a command history for each form or document model element. As a result there are two levels of undo / redo functionality: Global, that deals with tree view commands and saved / unsaved marking, and per-editor, that can undo or redo applied editor changes. To accomplish per-editor undo / redo functionality, an editor command history cache is saved in the editor control. This cache maps models to command histories. Each time the underlying model changes the active command history changes as well, reflecting its state on the editor UI (Figure 37).

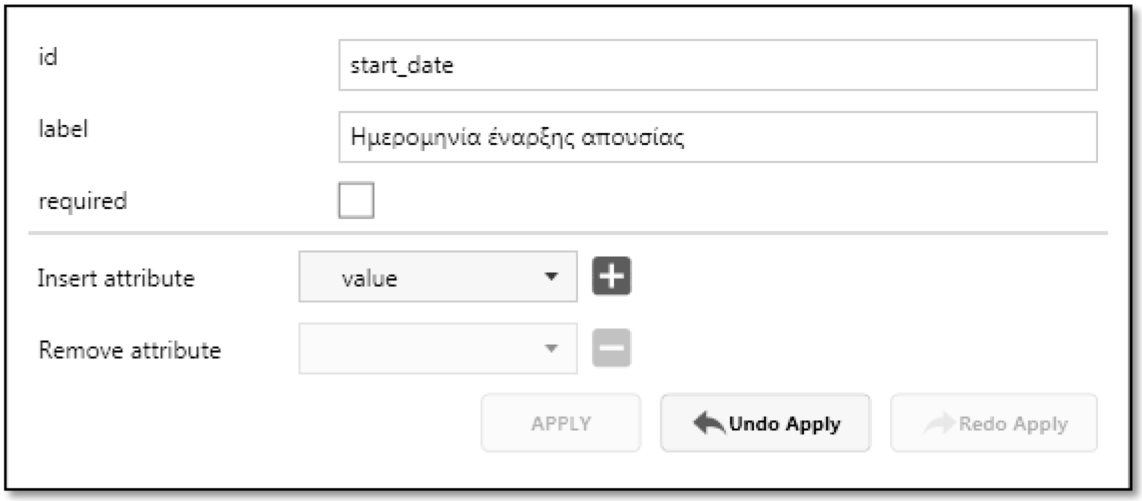

id	start_date
label	Ημερομηνία έναρξης απουσίας
required	☐
Insert attribute	value
Remove attribute	

Figure 37: Per-editor undo / redo functionality

6.4 Automatic Configuration User Interfaces

6.4.1 Overview

ConfigUIGenerator is a library created to quickly generate editor UIs for object properties via user-defined property specifications. It currently provides micro-editors for built-in data types and some WPF-specific classes. Also provided is a grid-like UI (`GridConfigUI`) to hold and present all the generated micro-editors. An example generated editor UI is provided in Figure 38.

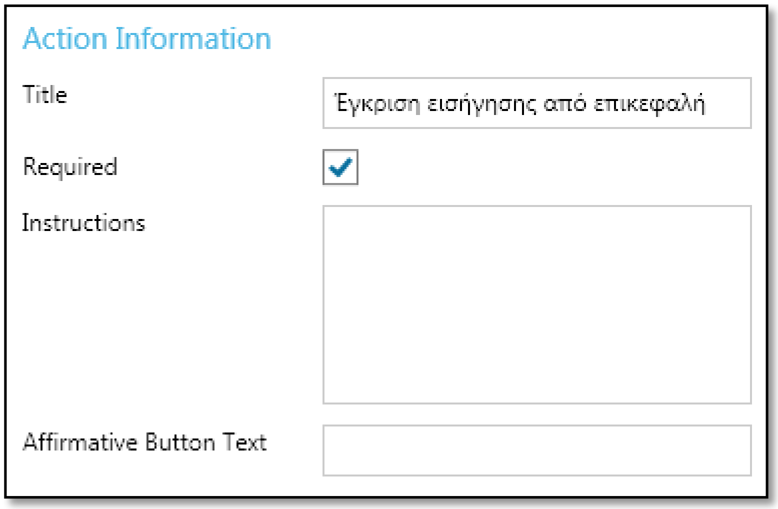


Figure 38: GridConfigUI in action

To generate an editor UI, one must first define an `ObservableCollection` of `PropertySpecifications` either in XAML or programmatically and set `GridConfigUI`'s `PropertySpecifications` property to that collection. To be able to generate the editor UI the `ObjectSource` property must be set to the object whose properties will be edited. Then, when the `GenerateUI` method is executed an editor UI matching the property specifications defined will be generated.

6.4.2 Architecture

You can see ConfigUIGenerator's micro-architecture in Figure 39. A detailed description of the basic architectural parts follows.

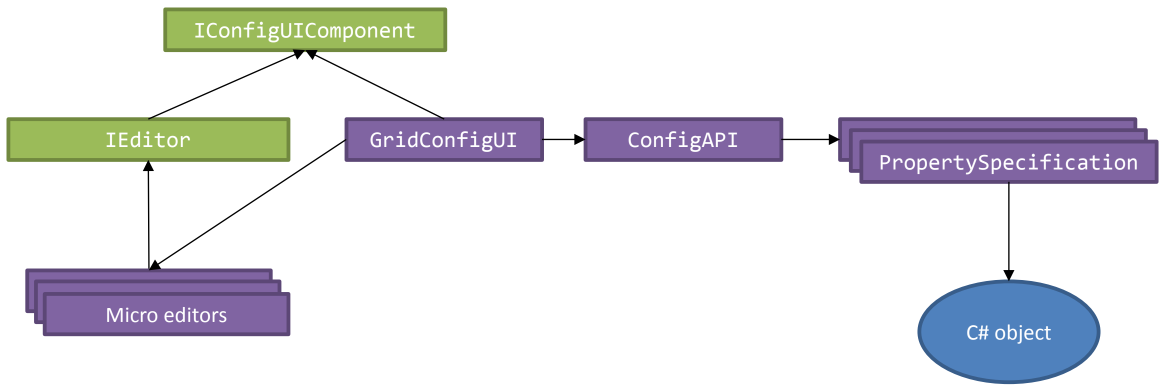


Figure 39: ConfigUIGenerator micro-architecture

The base interface for all the major ConfigUIGenerator components is `IConfigUIComponent` (Figure 40). `IEditor` extends it and `GridConfigUI` implements it. The `Apply` method applies changes to object properties and stores original property values. `Preview` also applies changes but does not store original property values. `Revert` reverts property values back to their original state. `Refresh` updates the micro-editor UI values. `GenerateUI` generates the editor UI and returns a reference to it.

```
public interface IConfigUIComponent
{
    void Apply();
    void Revert();
    void Preview();

    void Refresh();
    FrameworkElement GenerateUI();
}
```

Figure 40: The IConfigUIComponent interface

`IEditor` is the interface that all micro-editors implement. It extends `IConfigUIComponent` by exposing the `AttachedPropertyHolder` and `AttachedObject` properties and the `SetValue` method (Figure 41). `AttachedPropertyHolder` holds property information, validation rules and the intermediate value of the property. `AttachedObject` holds the object reference to be edited. `SetValue` sets property values by bypassing UI editor values.

```
public interface IEditor : IConfigUIComponent
{
    PropertyHolder AttachedPropertyHolder { get; set; }

    object AttachedObject { get; set; }

    void SetValue(object value);
}
```

Figure 41: The IEditor interface

`GridConfigUI` is a top-level editor UI that can also be seen as a composite editor. It expects a collection of property specifications and an object source to generate an editor UI. Internally, it generates a `ConfigAPI` object that translates property specifications to property holders and generates micro-editors for each property holder.

6.4.3 Property Specifications

The `PropertySpecification` class is a container that holds relevant information that describes a property (Figure 42). The `Name` property is the target property name. `PropertyType` is the target property type. `EditorType` is the type of micro-editor to use in the generated editor UI. If `EditorType` is not defined ConfigUIGenerator will try to find a matching micro-editor based on the matched property's type. `PossibleValues` is a collection of possible values for the property, for example a brush property specifications could have a list of brushes as possible values. `ValidationRules` is a collection of `ValidationRule` objects that are used for validation. This will be further discussed in the micro-editors section below.

```
public class PropertySpecification
{
    public string Name { get; set; }

    public Type PropertyType { get; set; }

    public Type EditorType { get; set; }

    public IEnumerable<object> PossibleValues { get; set; }

    private ObservableCollection<ValidationRule> _validationRules
        = new ObservableCollection<ValidationRule>();
    public ObservableCollection<ValidationRule> ValidationRules
    {
        get { return _validationRules; }
        set { _validationRules = value; }
    }
}
```

Figure 42: The PropertySpecification container class

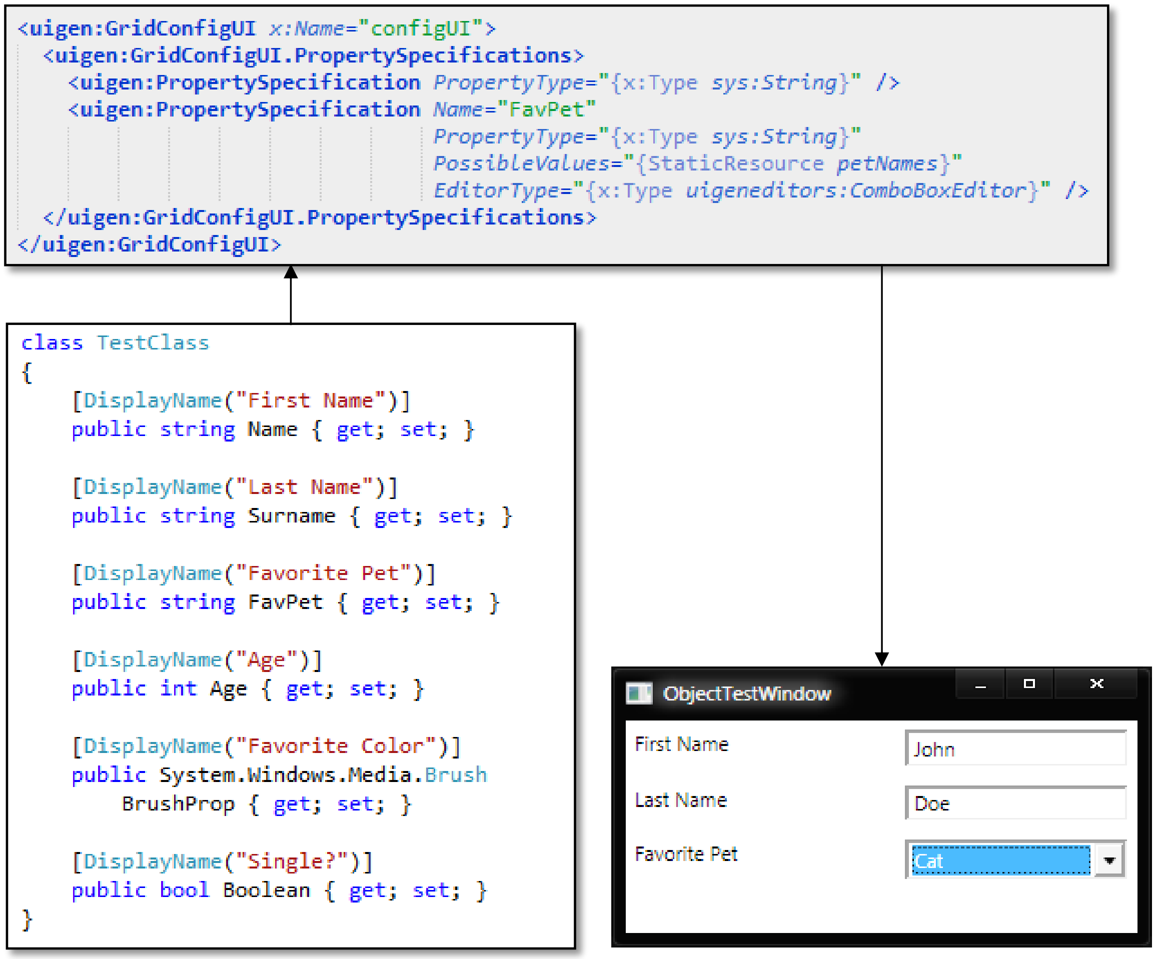
A property specification does not necessarily map to one property. ConfigUIGenerator has a matching policy that, depending on what properties are defined in a property specification, can match any number of properties. This is done when property specifications are added to `ConfigAPI`. The policy is as such:

- If both `PropertyType` and `Name` are specified, match a property with the name and the specific type.
- If only `PropertyType` is specified, match all properties of the specific type
- If only `Name` is specified, match a property with that name

If a property is matched multiple times, the latest specification overrules all the previous ones. For example, given two property specifications:

- The first one having specified `PropertyType`
- The second one having specified `Name`, `EditorType`, and the same `PropertyType`

The property specified in the second specification will have a micro-editor of type `EditorType`, while all the other properties of type `PropertyType` will have a default micro-editor (Figure 43).



```
<uigen:GridConfigUI x:Name="configUI">
  <uigen:GridConfigUI.PropertySpecifications>
    <uigen:PropertySpecification PropertyType="{x:Type sys:String}" />
    <uigen:PropertySpecification Name="FavPet"
                                 PropertyType="{x:Type sys:String}"
                                 PossibleValues="{StaticResource petNames}"
                                 EditorType="{x:Type uigeneditors:ComboBoxEditor}" />
  </uigen:GridConfigUI.PropertySpecifications>
</uigen:GridConfigUI>
```

```
class TestClass
{
    [DisplayName("First Name")]
    public string Name { get; set; }

    [DisplayName("Last Name")]
    public string Surname { get; set; }

    [DisplayName("Favorite Pet")]
    public string FavPet { get; set; }

    [DisplayName("Age")]
    public int Age { get; set; }

    [DisplayName("Favorite Color")]
    public System.Windows.Media.Brush
        BrushProp { get; set; }

    [DisplayName("Single?")]
    public bool Boolean { get; set; }
}
```

First Name	John
Last Name	Doe
Favorite Pet	Cat

Figure 43: An example of property matching using property specifications

6.4.4 Micro-editors

Micro-editors are the basic building block for composite editors like `GridConfigUI`. They provide value editing, events, and validation. Micro-editors included in ConfigUIGenerator are:

- `BrushTextBoxEditor`
- `CheckBoxEditor`

- ComboBoxEditor
- MultilineTextBoxEditor
- TextBoxEditor

Micro-editors for other types can easily be added by creating classes that implement the IEditor interface. Newly created micro-editors can also extend existing micro-editors to have a different behavior rather than support a new data type. Both these techniques are demonstrated in DocumentModelEditor and ProcessEditor. For instance in ProcessEditor, UpdateOnLostFocusMultilineTextBoxEditor and UpdateOnLostFocusTextBoxEditor classes extend base ConfigUIGenerator micro-editors to change the value binding update source trigger behavior to fire on lost focus. In DocumentModelEditor, LinkedDocumentModelNameEditor and LinkedFormNameEditor classes extend the TypeEditor<T> abstract class to create micro-editors with different UI but similar behavior to other micro-editors.

Micro-editors that extend TypeEditor<T> also provide support for adding value validation rules. Since WPF data binding is used internally for value updating, the WPF ValidationRule class is used for validation. To create a validation rule one must simply create an object that extends the WPF ValidationRule class and override the Validate function with validation logic. Default validation rules can be defined in the micro-editor class and additional validation rules can be added from defined rules in property specifications. An example of default validation rule usage is in the ConfigUIGenerator BrushTextBoxEditor micro-editor, where the input is checked for valid conversion into a brush.

6.4.5 GridConfigUI Class Overview

As said before GridConfigUI is a top-level editor UI. It needs a collection of property specifications and an object source to generate the UI (Figure 44). Once generated, property specifications and object sources can be changed dynamically simply by changing / modifying their property values.

```
public partial class GridConfigUI : UserControl, IConfigUIComponent
{
// ...
    public ObservableCollection<PropertySpecification> PropertySpecifications;

    public object ObjectSource;

    private static void OnPropertySpecificationsPropertyChanged(
        DependencyObject d, DependencyPropertyChangedEventArgs e);

    private static void OnObjectSourcePropertyChanged(DependencyObject d,
DependencyPropertyChangedEventArgs e);

    private void PropertySpecifications_CollectionChanged(object sender,
NotifyCollectionChangedEventArgs e);

// ...
```

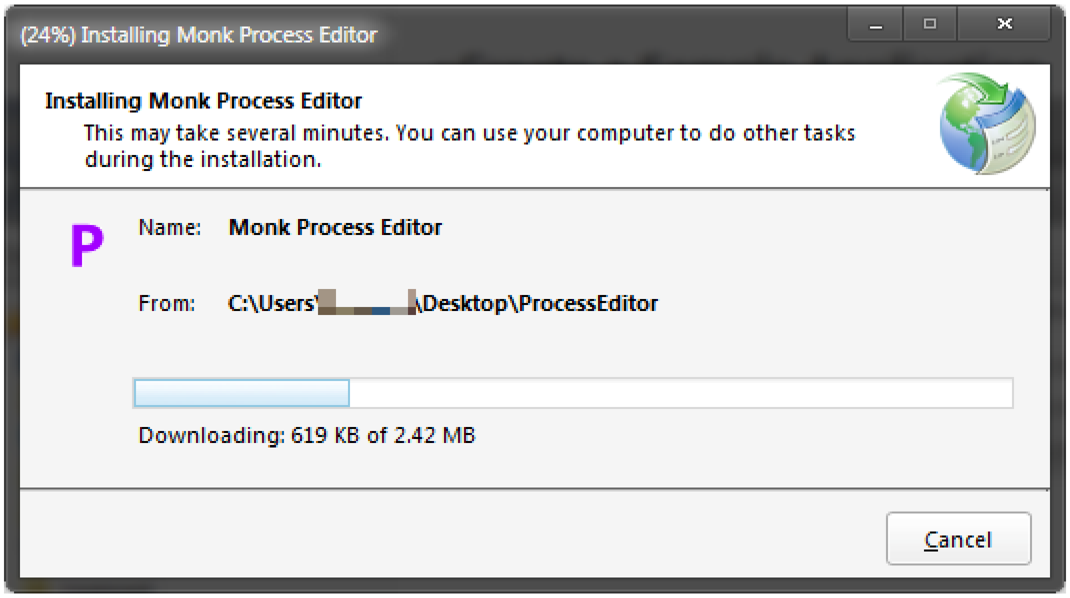
Figure 44: GridConfigUI members responsible for dynamic addition and removal of property specifications and object sources

Adding and removing property specifications is especially useful when dealing with dynamic objects. When adding property specifications either new micro-editors are added to the editor UI or existing micro-editors are replaced by other matched micro-editor types. Removing property specifications results in either removing micro-editors or replacing existing micro-editors. Changing the object source will result in the whole editor UI regenerating itself.

6.5 Miscellaneous

6.5.1 Installers

All three authoring tools use ClickOnce [25] deployment technology (Figure 45). This allows for easy installation and updating of the tools, as well as installing the correct .NET framework version that the tools depend on. Further, application icons can be defined, additional required files can be added to the installation and file associations with file icons can be created.



(24%) Installing Monk Process Editor

Installing Monk Process Editor

This may take several minutes. You can use your computer to do other tasks during the installation.

Name: Monk Process Editor

From: C:\Users\[redacted]\Desktop\ProcessEditor

Downloading: 619 KB of 2.42 MB

Figure 45: The ClickOnce installer for the process editor

ClickOnce supports both the traditional windows application paradigm (start menu entries and desktop shortcuts) and browser applications that are run, not installed. In our case the first method is used. ClickOnce applications are isolated and can be installed without administrator privileges. Application deployment is governed by two XML manifest files. An application manifest (*.exe.manifest) specifies the application assemblies, dependent libraries, and permissions. The deployment manifest (*.application) includes the current version number, update behavior, and a publisher certificate. ClickOnce applications can be launched with the deployment manifest file.

6.5.2 Tabbed Views

In order to provide the ability to edit multiple forms, document models, and processes, tabbed views are used throughout the MONK authoring tools. The Common library includes a XAML resource dictionary that defines styles for the `TabControl` and `TabItem` WPF classes. This resource dictionary gives the tabbed view its own look and feel, the ability to close the tab via a button on the tab header, and the display of an appended asterisk on the header title if the included tab content is not saved (Figure 46).

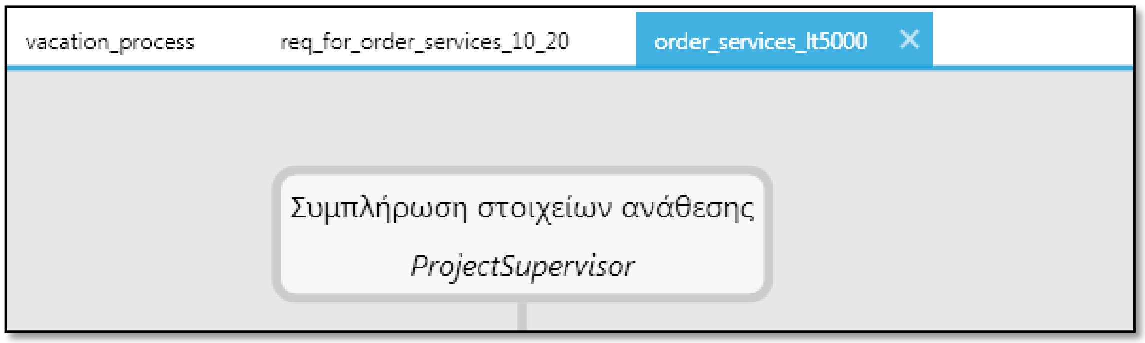


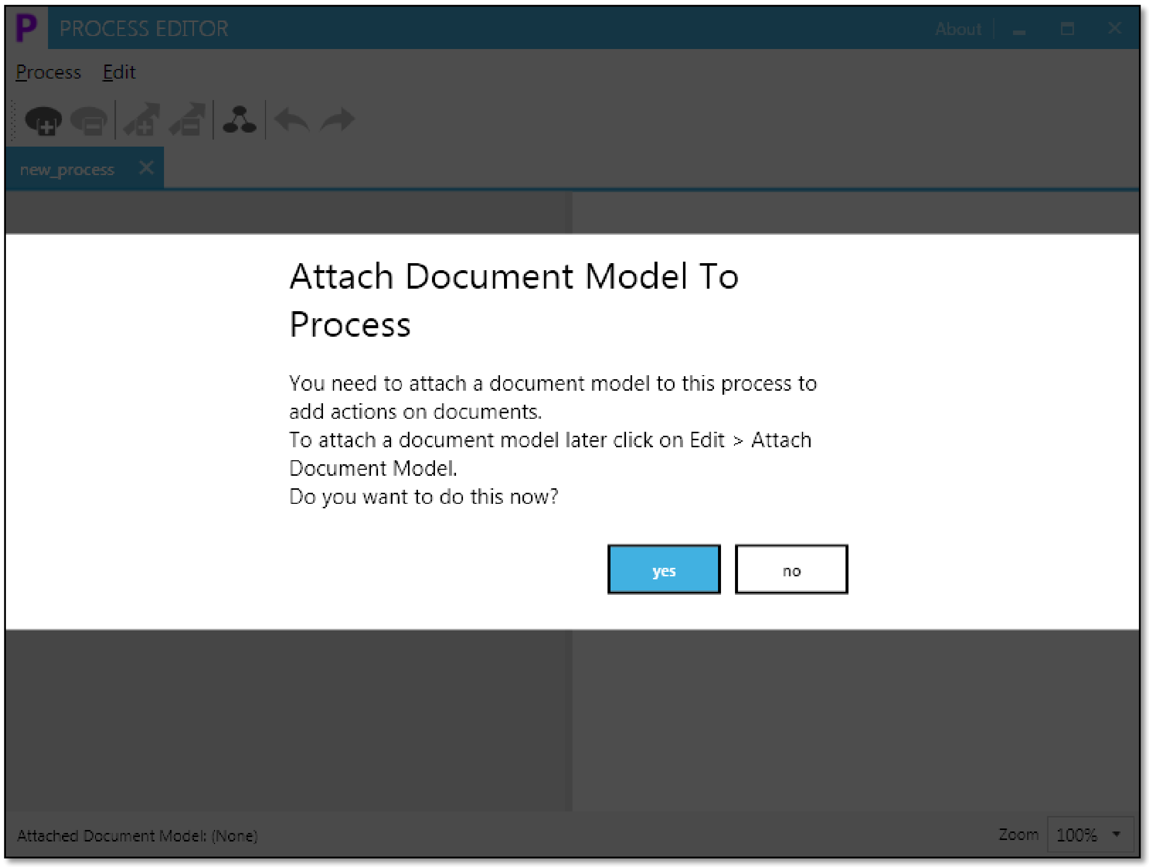
Figure 46: A tabbed view in the process editor

6.5.3 Modal Message Dialogs

There are cases where messages must be displayed to either request immediate user action or to provide urgent information (Figure 47). Modal message dialogs in the style of the windows 8 message dialog [26] are used to accommodate these needs. The mahapps.metro library provides basic message dialog functionality via the `DialogManager.ShowMessageAsync` extension method. The Common library also provides an extra message dialog control, `ValidationErrorMessageDialog`, which extends `BaseMetroDialog` and is used to display validation errors in a scrollable view inside the message dialog.

Message dialogs are used in cases where:

- A document model can be attached to a process (process editor)
- The document attachment has succeeded or failed (process editor)
- The process graph cannot be arranged because of circles (process editor)
- There are unsaved changes in an editor tab (all editors)
- A save is attempted in a path of an already opened file (all editors)



Attach Document Model To Process

You need to attach a document model to this process to add actions on documents.
To attach a document model later click on Edit > Attach Document Model.
Do you want to do this now?

Attached Document Model: (None)

Figure 47: A message dialog presented when creating a new process in the process editor

7. Case Studies

To seed the creation of process models using the authoring tools existing business processes within FORTH's institute of computer science were identified and analyzed. Interviews were conducted to get a first oral impression of how specific business process work and flow between participants. Subsequently, based on the interviews, processes were analyzed and expressed in natural language. Later, steps in the natural language were transformed into a formal language that was defined by a simple grammar that combined user roles and actions. The next step was to produce preliminary graphs that used the formal specifications as input. With the graphs in hand, process flow could be validated for correctness. As a last step, feedback was requested from other institutes on potential differences in their equivalent processes.

All the work mentioned above was also used to create a list of user roles, in conjunction with FORTH's organization chart. With each new business process analysis potentially new user roles were revealed. When a draft list was ready, further interviews were conducted to narrow down user roles and create a finalized version. Deconstructing how processes work also helped in creating action definitions for use in process activities. The table of actions currently supported by the runtime system can be seen in Table 2 and the table of user roles defined can be seen in Table 3.

Action Id	Description
fill	Fills a form
approve	Documents submitted for approval. Can view and then accept, reject, or return the documents
forward	Indicates which documents will be forwarded to the next activity. Usually used in tandem with the review action
check	Documents submitted for checking. Can view document and verify viewing
upload	Uploads a document
edit	Edits a document
review	Views and (potentially) edits a document
view	Displays a document for viewing (read only)
digitalSign	Digitally sign documents

Table 2: Actions currently supported by the runtime system

Role Id	Description	Units
President	President of FORTH	FORTH
AdminPresident	President's secretariat	FORTH
SupervisorKD	Central administration supervisor	KD
AdminKD	Central administration secretariat	KD
SupervisorHR	Human resources supervisor	KD
AdminHR	Human resources	KD
SupervisorBudgets	Budget supervisor	KD
AdminBudgets	Budget department	KD
SupervisorAccounting	Accounting supervisor	KD
AdminAssets	Assets department	KD
AdminPurchasing	Purchasing-client department	KD
AdminCashier	Cashier	KD
AdminAssetsAndProcurment	Assets and procurement department	KD
AdminITSupportSAP	IT support department	KD
LegalOffice	Legal office	KD
AdminContractsAndProjects	Project department	KD
SupervisorTechnicalServices	Technical service supervisor	KD
AdminTechnicalServices	Technical service secretariat	KD
Employee		FORTH, KD, IESL, ICS, IMBB, IACM, IMS, ICE-HT
DirectorInst	Institute director	IESL, ICS, IMBB, IACM, IMS, ICE-HT
AdminInst	Institute secretariat	IESL, ICS, IMBB, IACM, IMS, ICE-HT
AdminLab	Lab secretariat	StrongFieldPhysics, AtomsMoleculesClusters
LabSupervisor	Lab supervisor	StrongFieldPhysics, AtomsMoleculesClusters
ProjectSupervisor	Project supervisor	StrongFieldPhysics, AtomsMoleculesClusters
AdminICSPrograms	ICS project office	ICS
AdminVacationsICS	Leave registration secretariat	ICS
AdminICSDNS	Secretariat of networks	ICS

AdminInstStockRoom	Biology Storage room secretariat	IMBB
AdminInstAccounting	Biology accounting	IMBB
AdminInstPurchasingOrder	Biology purchasing secretariat	IMBB

Table 3: User roles defined for use in the runtime system

By using existing forms and various documents, user roles, actions, preliminary graphs, and process steps in formal language one can use the authoring tools to define processes used in the runtime system. A typical workflow for creating a process would be to:

1. Create form structures in the form editor (if any forms are needed by the process)
2. (Optional) Define form layouts using the form layout utility
3. Create a document model using the document model editor, attaching forms to document tree nodes if form structures were created in step 1
4. Create a process graph by using the process editor and attaching the created document model to the created process

Changed or new form files affect the document model, and a changed document model file affects the process graph. Additionally, form layout files depend on form files. These dependencies and the general workflow for process creation using the authoring tools can be seen in Figure 48.

With the methodology mentioned above we constructed processes that correspond to existing processes used at FORTH, as case studies for both the authoring tools and the runtime system. In the following sections we will present overviews of how these processes work along with their form structures, document models, and process graphs.

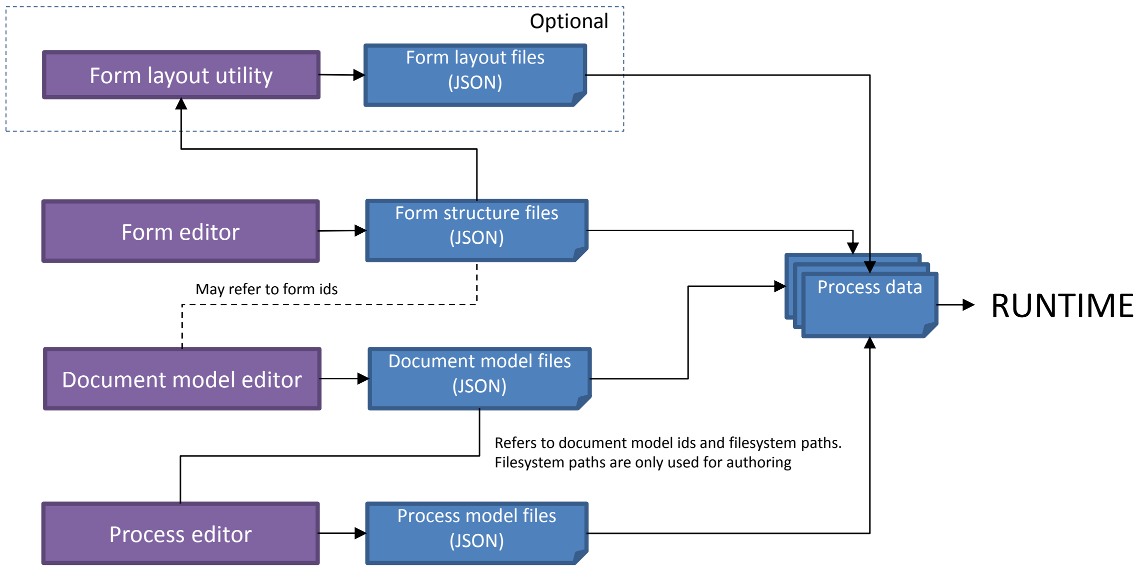

Optional
Form layout utility
Form layout files (JSON)
Form editor
Form structure files (JSON)
May refer to form ids
Document model editor
Document model files (JSON)
Refers to document model ids and filesystem paths. Filesystem paths are only used for authoring
Process editor
Process model files (JSON)
Process data
RUNTIME

Figure 48: Dependencies and general workflow for process creation using the authoring tools

7.1 Leave Management

This process pertains to the application for leave by an employee, its approval by various levels of administration depending on the type of leave, and its final filing into the payroll system. The following types of leave exist:

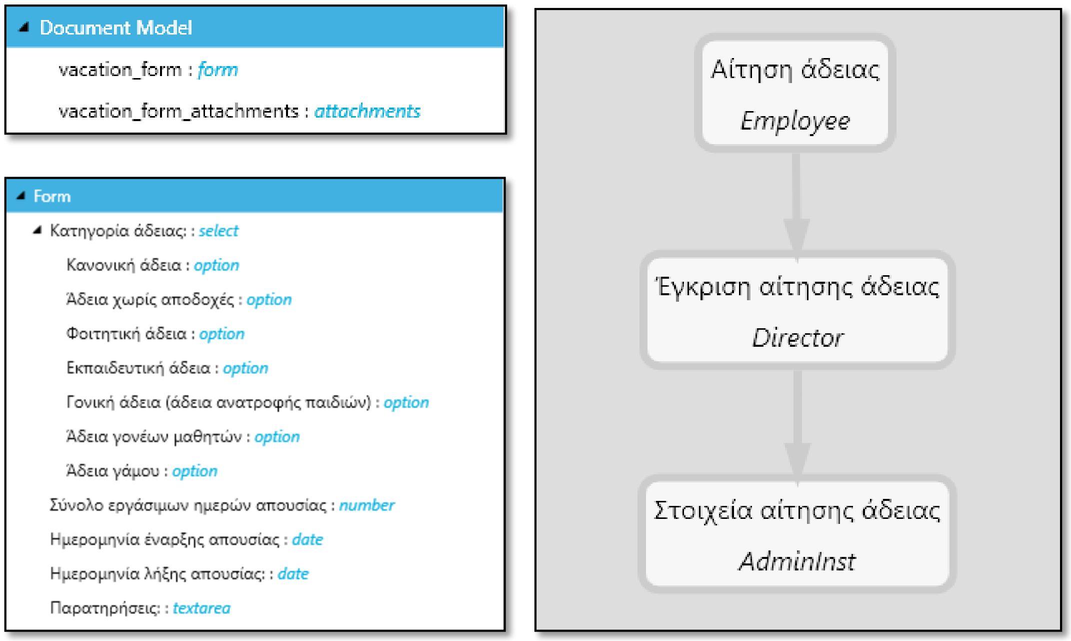
- Normal
- Unpaid leave
- Study leave
- Educational leave
- Parenting leave
- Student parent leave
- Marriage leave
- Maternity leave
- Sick leave
- Unjustified leave

The current procedure for staff leaves is:

1. The employee fills an application form
2. The supervisor signs the application
3. The forms are collected by the institute secretariat
4. The director signs the application
5. The application is registered in the leave book and in SAP by the secretariat

This process is quite simple as it contains a single form (Figure 49): the leave application, a document tree with two nodes (Figure 49): the leave application and file attachments, and three participating roles in the process graph: the employee, the director, and the institute secretariat. The process graph consists of three activities (Figure 49):

1. Leave application: Includes a *fill* action for filling the leave application and an *upload* action for uploading attachments to the application
2. Leave application approval: Includes an *approve* action for approving all documents sent by the employee in the previous activity
3. Leave application check: Includes a *check* action for the secretariat to verify that the application is handled


Document Model
vacation_form : *form*
vacation_form_attachments : *attachments*

Form
Κατηγορία άδειας: : *select*
Κανονική άδεια : *option*
Άδεια χωρίς αποδοχές : *option*
Φοιτητική άδεια : *option*
Εκπαιδευτική άδεια : *option*
Γονική άδεια (άδεια ανατροφής παιδιών) : *option*
Άδεια γονέων μαθητών : *option*
Άδεια γάμου : *option*
Σύνολο εργάσιμων ημερών απουσίας : *number*
Ημερομηνία έναρξης απουσίας : *date*
Ημερομηνία λήξης απουσίας: : *date*
Παρατηρήσεις: : *textarea*

Figure 49: The document model (upper left), the leave application form (lower left), and the process graph (right) for the "Leave Management" process

7.2 Purchase Management - Expenditure for Services from 0 to 10,000 Euros

This process deals with the purchases of services by the various units of FORTH. With the completion of the process, the administrative system of accounting in the asset management unit is updated.

This is a more elaborate process that contains more participating user roles and documents. The document model and forms can be seen in Figure 51. The process graph (Figure 50) has ten activities:

1. Approval for service assignment: Includes a *view* action providing feedback for the future addition of a protocol number, a *fill* action for filling a service approval form, an *upload* action for adding form attachments, and a *digitalSign* action for signing the form
2. Approval by lab supervisor: Includes an *approve* action for approving the service approval form and attachments
3. Audit from program office: Includes a *view* action providing feedback for the future addition of a protocol number, a *review* action for reviewing the service approval form and attachments, and a *forward* action to indicate which documents will be forwarded to the next activity
4. Approval by institute director: Includes the *approve* and *digitalSign* actions for approving and signing the service approval form and for approving the attachments
5. Filling of protocol number: Includes a *fill* action for filling the protocol number form, and a *view* action for viewing the service approval form and attachments
6. Receipt protocol: Includes a *view* action for viewing the protocol number form, a *fill* action for filling the receipt protocol form, an *upload* action for attachments to the previous form, and a *digitalSign* action for signing the receipt protocol form and the protocol number form

7. Audit from institute secretariat: Includes a *review* action for reviewing the receipt protocol form and attachments, and a *forward* action to indicate that the receipt protocol form will be forwarded to the director.
8. Signing of receipt protocol: Includes the *approve* and *digitalSign* action for approving and signing the receipt protocol form and attachments, and the protocol number form.
9. Diavgeia attaching: Includes an *upload* action for uploading attachments related to the Diavgeia transparency Program initiative [27], and a *forward* action to indicate that all documents will be forwarded to the cashier
10. Cashier: Includes a *check* action for the cashier to verify that all documents were received and the service supplier payment will be handled

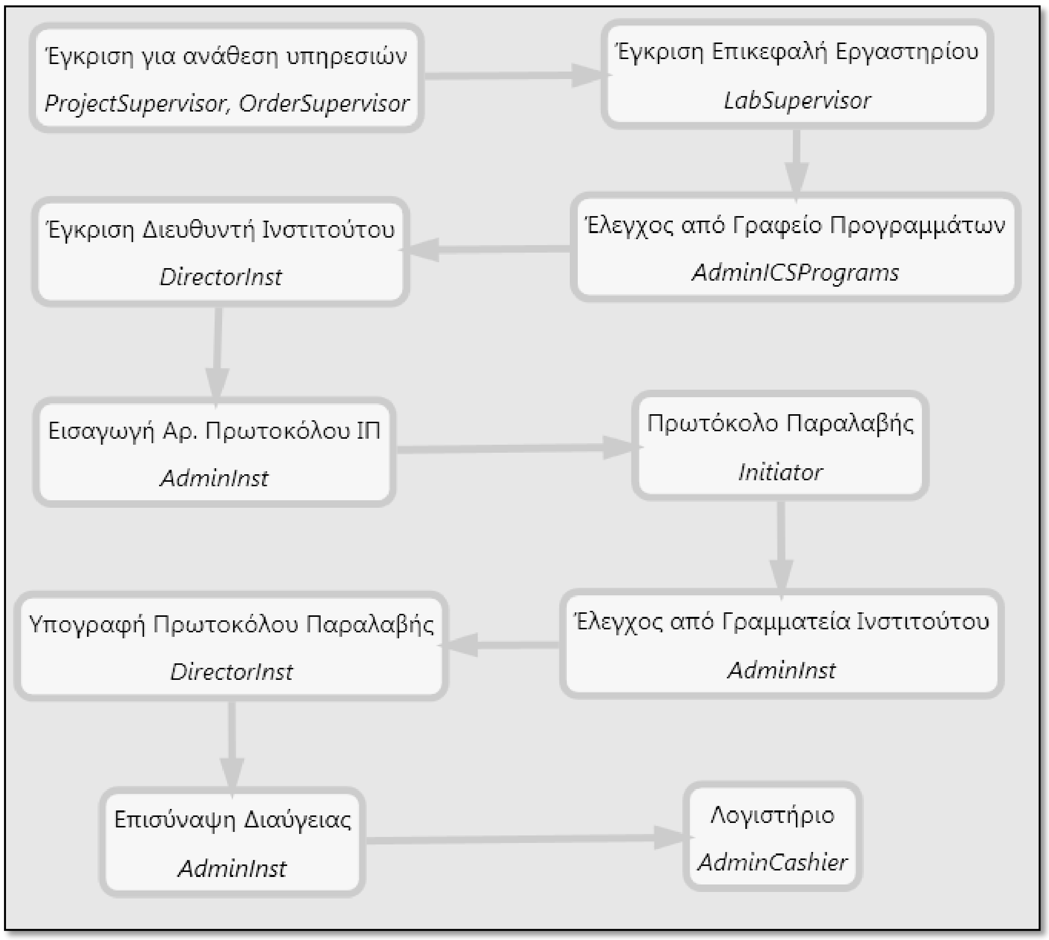
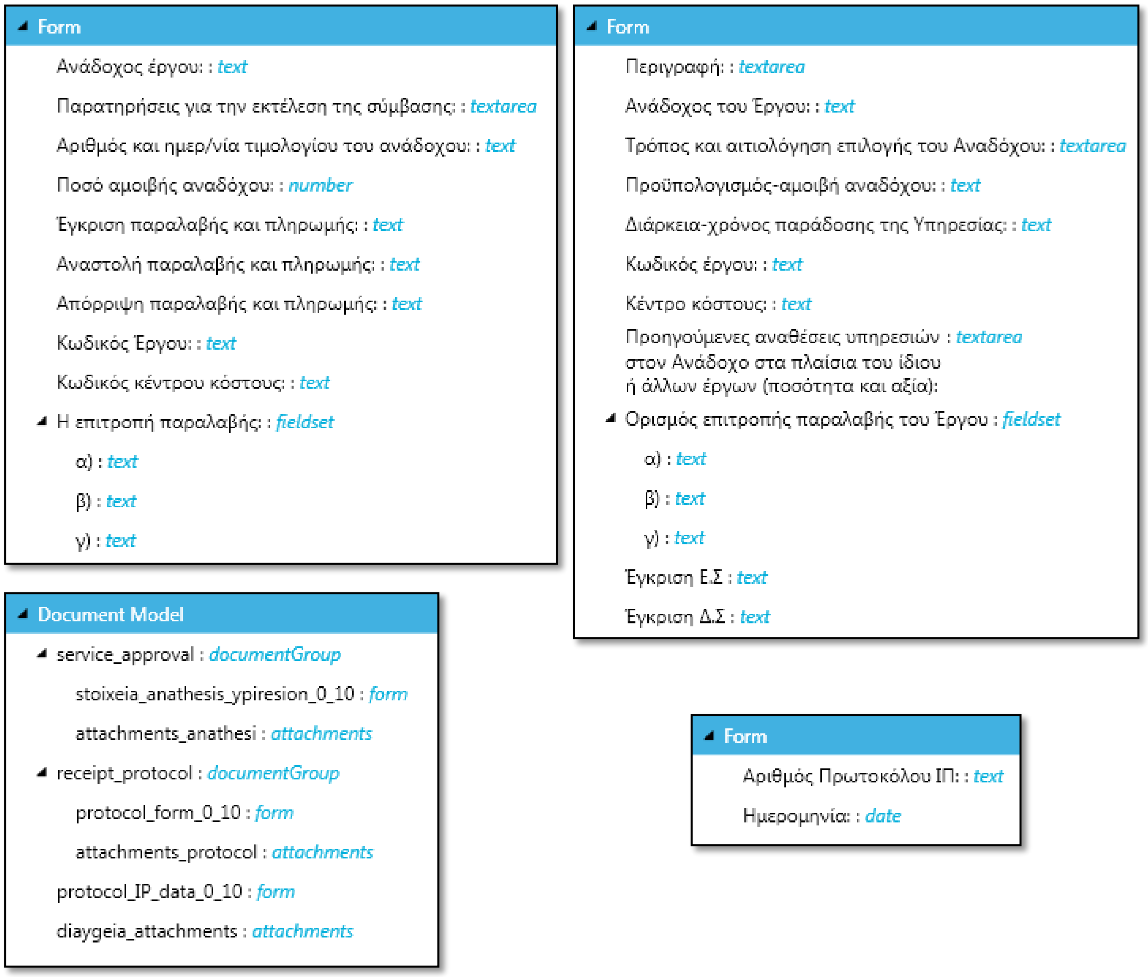

Έγκριση για ανάθεση υπηρεσιών
ProjectSupervisor, OrderSupervisor
Έγκριση Επικεφαλή Εργαστηρίου
LabSupervisor
Έλεγχος από Γραφείο Προγραμμάτων
AdminICSPrograms
Έγκριση Διευθυντή Ινστιτούτου
DirectorInst
Εισαγωγή Αρ. Πρωτοκόλου ΙΠ
AdminInst
Πρωτόκολο Παραλαβής
Initiator
Έλεγχος από Γραμματεία Ινστιτούτου
AdminInst
Υπογραφή Πρωτοκόλου Παραλαβής
DirectorInst
Επισύναψη Διαύγειας
AdminInst
Λογιστήριο
AdminCashier

Figure 50: The graph of the “Expenditure for Services from 0 to 10,000 Euros” process



- Form
  - Ανάδοχος έργου: : *text*
  - Παρατηρήσεις για την εκτέλεση της σύμβασης: : *textarea*
  - Αριθμός και ημερ/νία τιμολογίου του ανάδοχου: : *text*
  - Ποσό αμοιβής αναδόχου: : *number*
  - Έγκριση παραλαβής και πληρωμής: : *text*
  - Αναστολή παραλαβής και πληρωμής: : *text*
  - Απόρριψη παραλαβής και πληρωμής: : *text*
  - Κωδικός Έργου: : *text*
  - Κωδικός κέντρου κόστους: : *text*
  - Η επιτροπή παραλαβής: : *fieldset*
    - α) : *text*
    - β) : *text*
    - γ) : *text*

- Form
  - Περιγραφή: : *textarea*
  - Ανάδοχος του Έργου: : *text*
  - Τρόπος και αιτιολόγηση επιλογής του Αναδόχου: : *textarea*
  - Προϋπολογισμός-αμοιβή αναδόχου: : *text*
  - Διάρκεια-χρόνος παράδοσης της Υπηρεσίας: : *text*
  - Κωδικός έργου: : *text*
  - Κέντρο κόστους: : *text*
  - Προηγούμενες αναθέσεις υπηρεσιών στον Ανάδοχο στα πλαίσια του ίδιου ή άλλων έργων (ποσότητα και αξία): : *textarea*
  - Ορισμός επιτροπής παραλαβής του Έργου : *fieldset*
    - α) : *text*
    - β) : *text*
    - γ) : *text*
  - Έγκριση Ε.Σ : *text*
  - Έγκριση Δ.Σ : *text*

- Document Model
  - service_approval : *documentGroup*
    - stoixeia_anathesis_ypiresion_0_10 : *form*
    - attachments_anathesi : *attachments*
  - receipt_protocol : *documentGroup*
    - protocol_form_0_10 : *form*
    - attachments_protocol : *attachments*
  - protocol_IP_data_0_10 : *form*
  - diaygeia_attachments : *attachments*

- Form
  - Αριθμός Πρωτοκόλου ΙΠ: : *text*
  - Ημερομηνία: : *date*

Figure 51: Document model (lower left) and forms for the "Expenditure for Services from 0 to 10,000 Euros" process. The forms presented are: the service approval form (upper left), the service approval form (upper right), and the protocol number form (lower right)

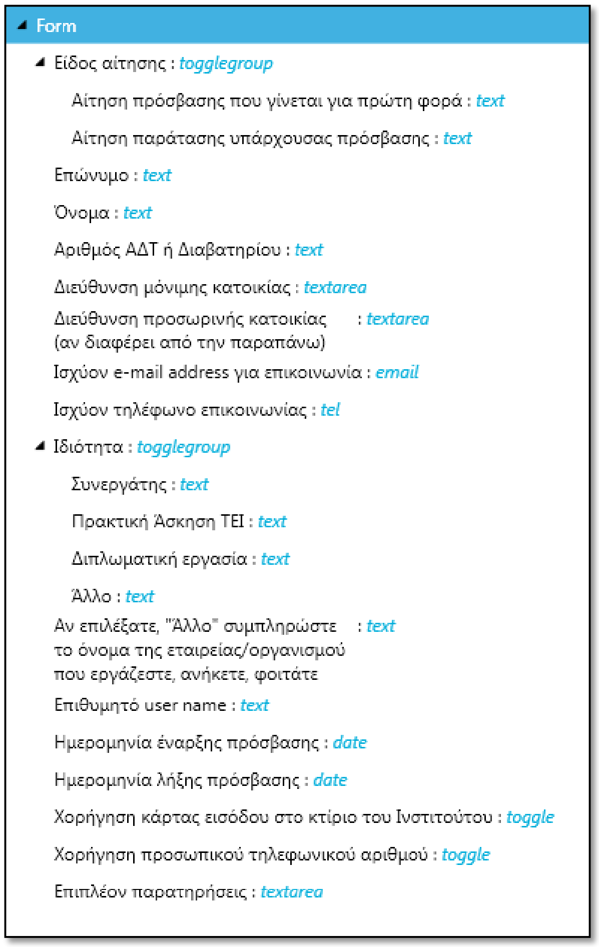
7.3 Guest Account Request

This process is used to allow employees to apply for access to the computer systems of FORTH. The final approved application is checked by the project office, and the user is notified.

This process features actions that are not intrinsic to the runtime system and must be conducted manually by the user. For example, the system administrators must manually create a user account using existing infrastructure.

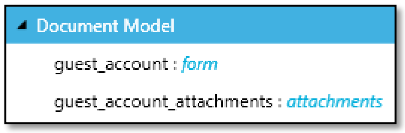
Apart from that, the process has two document model nodes (Figure 52): a guest account application, and its attachments. The process graph (Figure 52) includes these activities:

1. Application for guest account: Contains *fill* and *digitalSign* actions for filling and signing the guest account application form, and an *upload* action for uploading the application's attachments
2. Audit from institute secretariat: Contains a *view* action for viewing the guest account application form and its attachments, and a *forward* action that indicates that the application must be forwarded when a physical intellectual property rights application has been submitted
3. Approval from institute director: Contains an *approve* action for approving the guest account application form and its attachments, and a *digitalSign* action for signing the guest account application
4. Account Creation: Contains a *view* action for viewing the guest account application form and its attachments, and a *forward* action indicating that a user account must be created before forwarding the document to the project office
5. Confirm authorization of new account: Includes a *check* action for the project office to authorize the account and complete the process



Form

- Είδος αίτησης : *togglegroup*
  - Αίτηση πρόσβασης που γίνεται για πρώτη φορά : *text*
  - Αίτηση παράτασης υπάρχουσας πρόσβασης : *text*
- Επώνυμο : *text*
- Όνομα : *text*
- Αριθμός ΑΔΤ ή Διαβατηρίου : *text*
- Διεύθυνση μόνιμης κατοικίας : *textarea*
- Διεύθυνση προσωρινής κατοικίας (αν διαφέρει από την παραπάνω) : *textarea*
- Ισχύον e-mail address για επικοινωνία : *email*
- Ισχύον τηλέφωνο επικοινωνίας : *tel*
- Ιδιότητα : *togglegroup*
  - Συνεργάτης : *text*
  - Πρακτική Άσκηση ΤΕΙ : *text*
  - Διπλωματική εργασία : *text*
  - Άλλο : *text*
- Αν επιλέξατε, "Άλλο" συμπληρώστε το όνομα της εταιρείας/οργανισμού που εργάζεστε, ανήκετε, φοιτάτε : *text*
- Επιθυμητό user name : *text*
- Ημερομηνία έναρξης πρόσβασης : *date*
- Ημερομηνία λήξης πρόσβασης : *date*
- Χορήγηση κάρτας εισόδου στο κτίριο του Ινστιτούτου : *toggle*
- Χορήγηση προσωπικού τηλεφωνικού αριθμού : *toggle*
- Επιπλέον παρατηρήσεις : *textarea*



Document Model

- guest_account : *form*
- guest_account_attachments : *attachments*

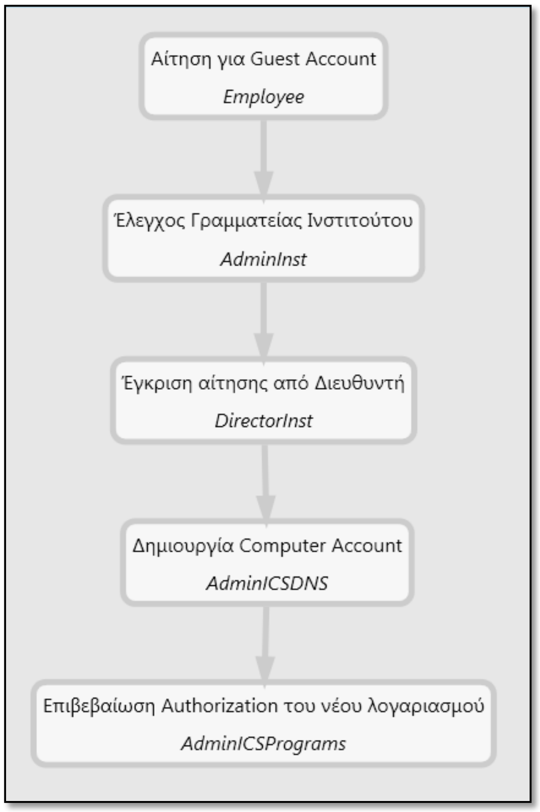

Αίτηση για Guest Account
Employee
Έλεγχος Γραμματείας Ινστιτούτου
AdminInst
Έγκριση αίτησης από Διευθυντή
DirectorInst
Δημιουργία Computer Account
AdminICSDNS
Επιβεβαίωση Authorization του νέου λογαριασμού
AdminICSPrograms

Figure 52: The guest account application form (left), the document model (upper right), and the process graph (lower left) for the "Guest Account Request" process

8. Future Work

8.1 Localization

As discussed earlier in chapter 6.1, localization facilities are present in all authoring tools. At the moment though, only US English is supported. There is translation work needed to be done to add extra languages to the tool user-interfaces. There are two main targets for translating the tools: resource files and extracted key-value pairs from BAML files.

Resource files can be created and edited using the tools shipping with the Visual Studio IDE, or by using other third-party tools like ResXManager [28] or Resx Editor [29]. For the translation of key-value pairs from BAML files the LocBaml tool [24] is fairly limited in features. As a result the creation of a tool that supports syncing and merging operations for translatable strings is a possibility. An alternative would the use of a third-party tool like Visual Locbaml [30].

Tool user-interface localization is not the only possible localization work. Since the runtime system is localizable, it is only natural that at some point form and process files could be translated. There are some ways one could accomplish this. One could be to create a localization tool for authored form and process JSON files. Another way would be to add localization features directly into the Process Editor and the Form Editor. In any case, some basic requirements would be to support string extraction from JSON files, synchronization of new translations, and merging of new or removed translatable strings.

8.2 Additional Editor Features

Both the Process Editor and the Form Editor could benefit from certain enhancements and feature additions.

For the Form Editor, a feature to generate a form preview could offer users a general idea of the final product even with default layout and form element presentation. Additionally, support for complex validation rules can give more power to form structure author. Support for complex form element types is another enhancement that offers more powerful form authoring. Candidate types could be a data grid, an address type, and maybe other organization specific types.

A feature that could benefit the Process Editor immensely is the rehearsal and validation of authored processes. This would be reduce the number of uploads and manual testing needed to import process into the runtime system. Visually the process graph could be enhanced in several ways, examples being support for resizing of graph nodes, and additional edge types like curved and orthogonal.

8.3 Extensions to ConfigUIGenerator

ConfigUIGenerator's type support is currently fairly limited. A default editor that provides support for complex types would rectify the situation. Further, the library could provide more built-in micro-editors for commonly used WPF classes. ConfigUIGenerator currently offers one top-level micro-editor container user-interface, `GridConfigUI`. Additional top-level user-interfaces could be added along with composite containers, for example a tabbed container. Finally, an API could be provided for the creation of custom top-level user-interfaces.

9. Bibliography

[1] World Wide Web Consortium, Web content accessibility guidelines (WCAG) 2.0, 2008.

[2] World Wide Web Consortium, Mobile Web Best Practices 1.0, 2008.

[3] ECMA International, ECMA-404 The JSON Data Interchange Format, 2013.

[4] Microsoft, "Windows Presentation Foundation," [Online]. Available: http://msdn.microsoft.com/en-us/library/ms754130(v=vs.110).aspx. [Accessed 24 August 2014].

[5] J. Newton-King, "James Newton-King - Json.NET," [Online]. Available: http://james.newtonking.com/json. [Accessed 24 August 2014].

[6] G. Court, K. Zyp and F. Galiegue, "draft-zyp-json-schema-03 - JSON Schema: core definitions and terminology," [Online]. Available: http://tools.ietf.org/html/draft-zyp-json-schema-03. [Accessed 2 July 2014].

[7] P. Jenkins, J. Ginnivan, B. Forster, A. Mitchell, D. Daume and J. Karger, "MahApps.Metro Documentation," [Online]. Available: http://mahapps.com/. [Accessed 24 August 2014].

[8] Formoid, "Formoid - Beautiful CSS Form Generator," [Online]. Available: http://formoid.com/. [Accessed 26 August 2014].

[9] R. Berjon, S. Faulkner, T. Leithead, E. D. Navara, E. O'Connor, S. Pfeiffer and I. Hickson, "A vocabulary and associated APIs for HTML and XHTML," [Online]. Available: http://www.w3.org/TR/html5/forms.html. [Accessed 15 July 2014].

[10] M. Otto and J. Thornton, "Bootstrap," [Online]. Available: http://getbootstrap.com/. [Accessed 26 August 2014].

[11] J. Resig, "jQuery," [Online]. Available: http://jquery.com/. [Accessed 26 August 2014].

[12] L. Von Ahn, B. Maurer, C. McMillen, D. Abraham and M. Blum, "recaptcha: Human-based character recognition via web security measures," *Science*, vol. 321, no. 5895, pp. 1465-1468, 2008.

[13] Appnitro Software, "MachForm – PHP HTML Form Builder," [Online]. Available: http://www.appnitro.com/. [Accessed 10 August 2014].

[14] Yii Software LLC, "Yii PHP Framework: Best for Web 2.0 Development," [Online]. Available: http://www.yiiframework.com/. [Accessed 15 August 2014].

[15] The YAWL Foundation, "YAWL," [Online]. Available: http://www.yawlfoundation.org/. [Accessed 17 August 2014].

[16] W. v. d. Aalst and A. t. Hofstede, "YAWL: Yet Another Workflow Language (Revised Version)," Queensland University of Technology, Brisbane, 2003.

[17] T. Murata, "Petri nets: Properties, analysis and applications.," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541-580, 1989.

[18] Workflow Patterns Initiative, "Workflow Patterns Home Page," [Online]. Available: http://www.workflowpatterns.com/. [Accessed 17 August 2014].

[19] Bonitasoft, Inc., "Bonitasoft - Open Source Workflow & BPM software," [Online]. Available: http://www.bonitasoft.com/. [Accessed 17 August 2014].

[20] OMG, "BPMN 2.0," 2011. [Online]. Available: http://www.omg.org/spec/BPMN/2.0/. [Accessed 17 August 2014].

[21] Microsoft, "PropertyGrid Class (System.Windows.Forms)," [Online]. Available: http://msdn.microsoft.com/en-us/library/system.windows.forms.propertygrid(v=vs.110).aspx. [Accessed 20 August 2014].

[22] Microsoft, "Windows Forms," [Online]. Available: http://msdn.microsoft.com/en-us/library/dd30h2yb(v=vs.110).aspx. [Accessed 20 August 2014].

[23] K. Sugiyama, S. Tagawa and M. Toda, "Methods for Visual Understanding of Hierarchical System Structures," *Systems, Man and Cybernetics, IEEE Transactions on,* vol. 11, no. 2, pp. 109-125, February 1981.

[24] Microsoft, "LocBaml Tool Sample," [Online]. Available: http://msdn.microsoft.com/en-us/library/ms771568(v=vs.85).aspx. [Accessed 2 August 2014].

[25] Microsoft, "ClickOnce Deployment," [Online]. Available: http://msdn.microsoft.com/en-us/library/t71a733d(v=vs.80).ASPX. [Accessed 5 August 2014].

[26] Microsoft, "Guidelines for message dialogs," [Online]. Available: http://msdn.microsoft.com/en-US/library/windows/apps/hh738363. [Accessed 1 August 2014].

[27] ΔΙΑΥΓΕΙΑ, "Ανάρτηση Πράξεων στο Διαδίκτυο | Πρόγραμμα Δι@ύγεια," [Online]. Available: https://diavgeia.gov.gr. [Accessed 30 September 2014].

[28] T. Englert, "ResX Resource Manager," [Online]. Available: http://resxresourcemanager.codeplex.com/. [Accessed 8 September 2014].

[29] J. Vermorel, "RESX Editor," [Online]. Available: http://resx.sourceforge.net/. [Accessed 8 September 2014].

[30] Seventh Software OÜ, "WPF Application Localization - Visual LocBaml," [Online]. Available: http://visuallocbaml.com/. [Accessed 8 September 2014].

Appendix A (data schemas)

A.1 Process Model JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-03/schema#",
  "title": "Process",
  "description": "A business process specification",
  "type": "object",
  "properties": {
    "name": {
      "type": "string",
      "pattern": "^\\w+$",
      "description": "process identifier, can only contain letters, numbers,
and underscores",
      "required": true
    },
    "friendlyName": { "type": "string" },
    "description": { "type": "string" },
    "attachedDocumentModelName": {
      "type": "string",
      "pattern": "^\\w+$",
      "description": "The identifier of the attached document model"
    },
    "attachedDocumentModelPath": {
      "type": "string",
      "description": "A fully qualified path to the attached document model
json file"
    },
    "activities": {
      "type": "object",
      "patternProperties": {
        "^([1-9][0-9]*)$": {
          "type": "object",
          "properties": {
            "title": { "type": "string" },
            "longTitle": { "type": "string" },
            "instructions": { "type": "string" },
            "affirmativeText": {
              "type": "string"
            },
            "userRoles": {
              "type": "array",
              "items": { "type": "string" },
              "uniqueItems": true
            },
```

Figure 53: Process model JSON schema

```
            "targets": {
              "type": "array",
              "items": {
                "type": "integer",
                "minimum": 1
              },
              "uniqueItems": true
            },
            "actions": {
              "type": "array",
              "items": {
                "type": "object",
                "properties": {
                  "name": { "type": "string", "required": true },
                  "linkedProcessName": {
                    "type": "string",
                    "pattern": "^\\w+$"
                  },
                  "title": { "type": "string" },
                  "required": { "type": "boolean" },
                  "instructions": { "type": "string" },
                  "affirmativeText": { "type": "string" },
                  "groupId": {
                    "type": "string",
                    "description": "actions with the same groupId get grouped"
                  },
                  "documentPaths": {
                    "type": "array",
                    "items": {
                      "type": "string",
                      "description": "a document path inside of the attached
document model, ex. docNode1.docNode11.docNode112",
                      "pattern": "^\\w+(\\.\\w+)*$"
                    },
                    "uniqueItems": true
                  }
                }
              }
            },
            "targetSelectionStrategy": {
              "description": "Not used, may be removed in the future",
              "enum":[ "OR", "XOR"],
              "default":"XOR"
            }
          }
        }
      },
      "required": true,
      "additionalProperties":false
    }
  }
}
```

Figure 54: Process model JSON schema, continued

A.2 Form Structure JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-03/schema#",
  "title": "Form",
  "type": "object",
  "properties": {
    "name": {
      "type": "string",
      "pattern": "^\\w+$",
      "description": "form identifier, can only contain letters, numbers, and
underscores",
      "required": true
    },
    "friendlyName": { "type": "string" },
    "description": { "type": "string" },
    "fields": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "id": {
            "type": "string",
            "pattern": "^\\w+$",
            "description": "form identifier, can only contain letters, numbers,
and underscores",
            "required": true
          },
          "type": {
            "enum": [
              "text",
              "textarea",
              "email",
              "url",
              "tel",
              "range",
              "number",
              "date",
              "datetime",
              "month",
              "week",
              "time",
              "color",
              "toggle",
              "password",
              "file",
              "select",
              "togglegroup",
              "fieldset",
              "optgroup",
              "option"
            ],
```

Figure 55: Form structure JSON schema

```
          "label": { "type": "string", "required": true },
          "required": { "type": "boolean", "default": true },
          "children": { "type": "array", "items": {"$ref":
"#/properties/fields/items"}
          },
          "description": { "type": "string" },
          "placeholder": { "type": "string", "description": "text, search, tel,
url, email"},
          "value": {
            "type": "string",
            "description": "all"
          },
          "accept": {
            "type": "string",
            "description": "file **comma separated list of mime types**"
          },
          "max": {
            "type": "string",
            "description": "range, number, date **float or date str**"
          },
          "min": {
            "type": "string",
            "description": "range, number, date **float or date str**"
          },
          "pattern": { "type": "string", "description": "text, email, url" },
          "step": {
            "type": "string",
            "description": "range, number **positive float**"
          },
          "maxlength": {
            "type": "string",
            "description": "text, email, url, password, tel **positive int,
infinite if negative**"
          },
          "spellcheck": {
            "type": "boolean",
            "description": "text"
          },
          "multiple": {
            "type": "boolean",
            "description": "togglegroup, select"
          },
          "selected": {
            "type": "boolean",
            "description": "option **boolean, first option with selected is
accepted if multiple is false**"
          }
        },
        "additionalProperties": false
      }
    }
  }
}
```

Figure 56: Form structure JSON schema, continued

A.3 Document Model JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-03/schema#",
  "title": "Document Model",
  "description": "A process document model representation",
  "type": "object",
  "properties": {
    "nodes": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "name": {
            "type": ["string", "null"],
            "pattern": "^\\w+$",
            "description": "Document model group identifier. Unique in the
context of document model names, can only contain letters,numbers, and
underscores"
          },
          "documentInfo": {
            "type": ["object", "null"],
            "properties": {
              "category": {
                "type": "string",
                "pattern": "^\\w+$",
                "required": true,
                "description": "Document identifier. Unique in the context of
document model categories. Can only contain letters, numbers, and underscores.
Can also refer to a form name if type is form"
              },
              "type": {
                "enum": ["doc", "text", "txt", "pdf", "xls", "bin", "image",
"form", "linkedDocumentModel", "attachments"],
                "required": true,
                "description": "Document type. linkedDocumentModel refers to a
sub document model that's attached to the document model (presumably to be used
by processes called by other processes). attachments defines an arbitrary
number and arbitrary types of documents"
              },
              "description": { "type": "string" },
              "linkedDocumentModelName": { "type": "string", "pattern":
"^\\w+$" }
            }
          },
          "children": { "type": "array", "items": {"$ref":
"#/properties/nodes/items"} }
        }
      }
    }
  }
}
```

Figure 57: Document model JSON schema

A.4 Role Catalog JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-03/schema#",
  "title": "Roles",
  "description": "A catalogue of process role specifications",
  "type": "array",
  "items": {
    "type": "string"
  }
}
```

Figure 58: Role catalog JSON schema

A.5 Action Catalog JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-03/schema#",
  "title": "Actions",
  "description": "A repository of business process activity action
specifications",
  "type": "object",
  "properties": {
    "actions": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "name": {
            "type": "string",
            "pattern": "^\\w+$",
            "description": "Action identifier, can only contain letters,
numbers, and underscores",
            "required": true
          },
          "friendlyName": {
            "type": "string",
            "required": true
          },
          "description": { "type": "string" },
          "type": {
            "enum": ["DOCUMENT", "NSA", "SAP", "SYSTEM", "LAUNCHPROCESS"],
            "required": true
          },
          "terminatesProcess": { "type": "boolean" }
        }
      }
    }
  }
}
```

Figure 59: Action catalog JSON schema

A.6 Unit Catalog JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-03/schema#",
  "title": "Organizational Units Catalogue",
  "description": "A tree structure holding the units and subunits of an
organization",
  "type": "object",
  "properties": {
    "units": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "name": {
            "type": "string",
            "required": true
          },
          "category": {
            "enum": ["Organization", "Institute", "Laboratory",
"Administration"],
            "required": true
          },
          "subUnits": {
            "type": "array",
            "items": {"$ref": "#/properties/units/items"}
          }
        }
      }
    }
  }
}
```

Figure 60: Unit catalog JSON schema

A.7 Category Catalog JSON schema

```
{
  "$schema": "http://json-schema.org/draft-03/schema#",
  "title": "Roles",
  "description": "A catalog of business process categories",
  "type": "array",
  "items": {
    "type": "string"
  }
}
```

Figure 61: Category catalog JSON schema

Appendix B (form element specifications)

B.1 Form Element Attribute Specifications

Attribute	Description	Can be used in	Required
label	Label for form elements, is used as a legend for fieldsets. Mandatory.	all elements	Yes
type	Form element type. Mandatory.	all elements	Yes
placeholder	Placeholder value for some text input elements.	text, textarea, email, url, tel	No
value	Initial value for form elements.	text, textarea, email, url, tel, range, number, date, datetime, month, week, time, color, toggle, password	No
accept	A comma separated list of mime types.	file	No
max	A float or date string, indicates upper bound.	range, number, date	No
min	A float or date string, indicates lower bound.	range, number, date	No
pattern	A regular expression that the form element's value is checked against.	text, email, url	No
step	A positive float, limits the increments at which a numeric value can be set.	range, number	No
maxlength	Positive integer, sets maximum value length. Infinite if negative.	text, email, url, password, tel	No

required	Boolean value, denotes required fields.	text, textarea, email, url, tel, range, number, date, datetime, month, week, time, color, toggle, password, file, select	Yes
spellcheck	Boolean value, Indicates if a text field should be checked for grammar and spelling.	text, textarea	No
description	A text description of a form field.	all elements	No
id	Form element identifier, can only contain letters, numbers, and underscores. Mandatory.	all elements	Yes
multiple	Boolean value, indicates if a select menu can have multiple items selected.	togglegroup, select	No
selected	Boolean value, indicates if an option is selected. The first option with selected is accepted if multiple is not set to true.	option	No

Table 4: Form element attribute specifications

B.2 Form Element Specifications by Type

Element type	Available attributes	Child element type(s)	Description
text	label, type, placeholder, value, pattern, maxlength, required, spellcheck, description, id	None	A single line text input field.
textarea	label, type, placeholder, value, pattern, maxlength,	None	A text input field, used for bigger amounts of text.

	required, spellcheck, description, id		
email	label, type, placeholder, value, pattern, maxlength, required, description, id	None	An email input field.
url	label, type, placeholder, value, pattern, maxlength, required, description, id	None	A URL input field.
tel	label, type, placeholder, value, maxlength, required, description, id	None	A telephone number input field.
range	label, type, value, min, max, step, required, description, id	None	A numeric range input field.
number	label, type, value, min, max, step, required, description, id	None	A number input field.
date	label, type, value, required, description, id	None	A date input field (year, month, and day).
datetime	label, type, value, required, description, id	None	A date input field (year, month, day and time).
month	label, type, value, required, description, id	None	A date input field (month and year).
week	label, type, value, required, description, id	None	A date input field (week and year).

time	label, type, value, required, description, id	None	A date input field (time).
color	label, type, value, required, description, id	None	A color input field.
toggle	label, type, value, required, description, id	None	A togglable element.
password	label, type, required, description, id	None	A password input field.
file	label, type, accept, required, description, id	None	An element for file uploads.
fieldset	label, type, description, id	text, textarea, email, url, tel, range, number, date, datetime, month, week, time, color, toggle, password, file	An element for logical grouping of form elements.
select	label, type, required, multiple, description, id	option, optgroup	A menu of options.
option	label, type, selected, description, id	None	Child element for the select element.
optgroup	label, type, description, id	option	Child element for the select element, groups option elements.
togglegroup	label, type, description, id	toggle	A logical group for toggle elements.

Table 5: Form element specifications by type

Appendix C (icons)

C.1 Application Icons

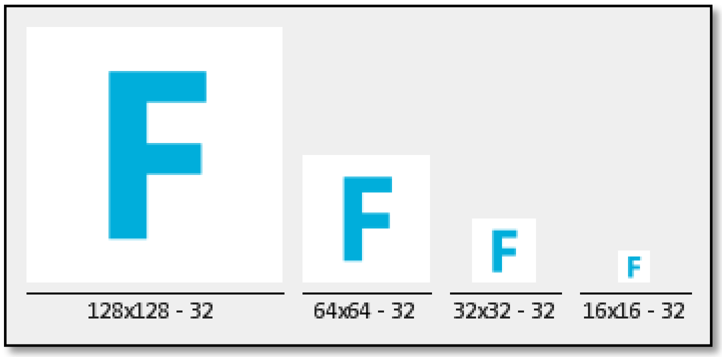


Figure 62: Form editor application icon

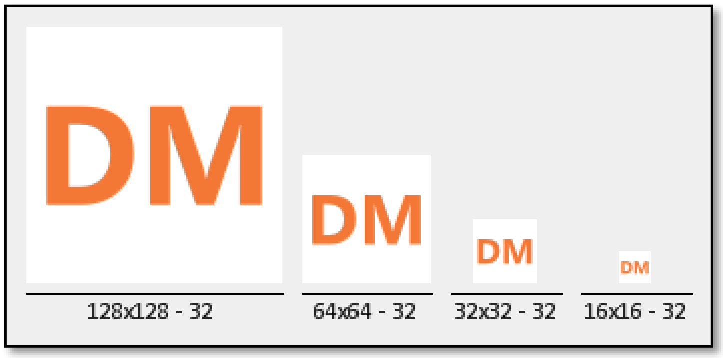


Figure 63: Document model editor application icon

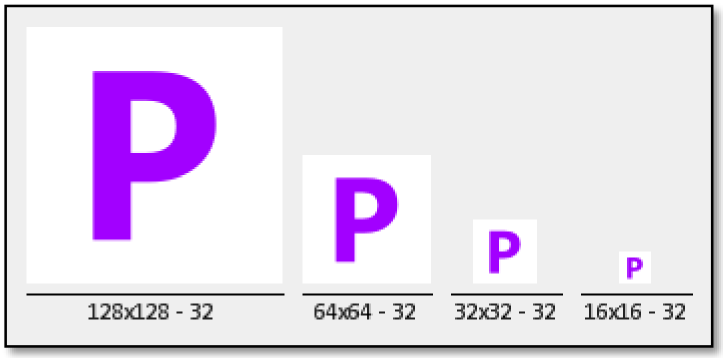


Figure 64: Process editor application icon

C.2 File Icons

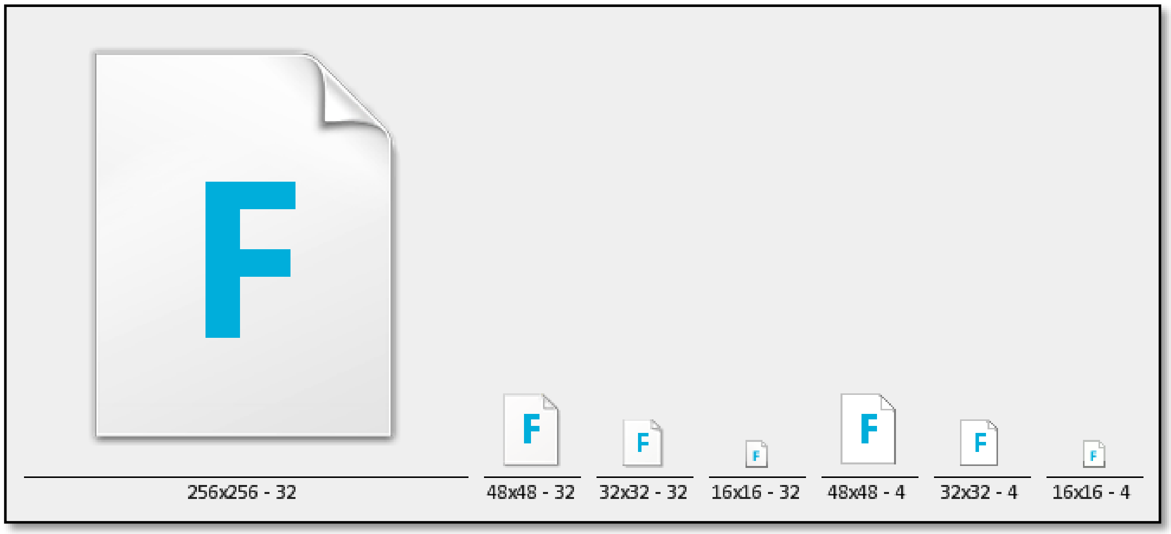


Figure 65: Form editor file icon

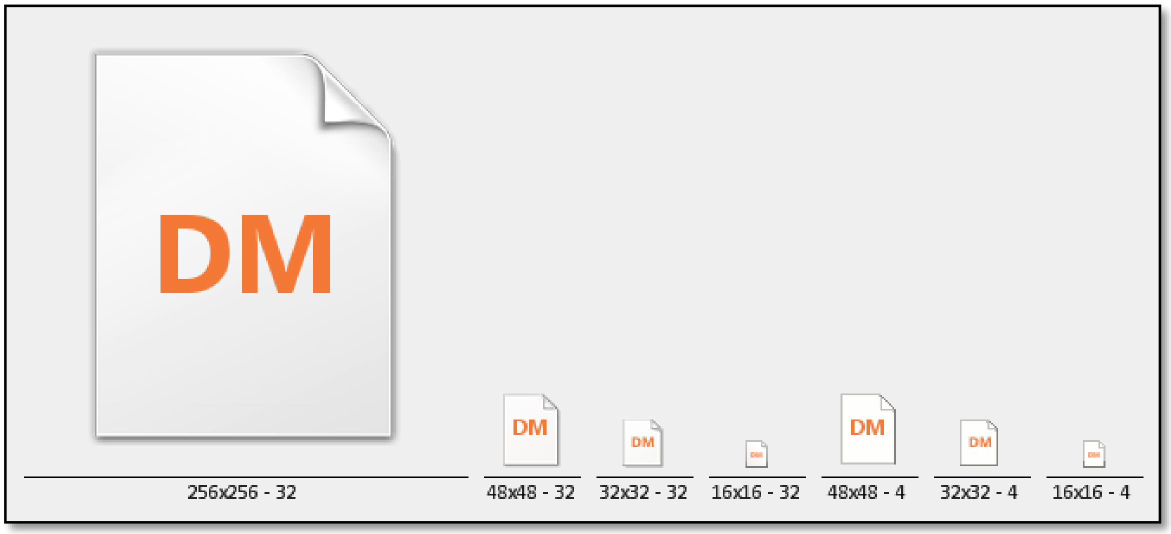


Figure 66: Document model editor file icon

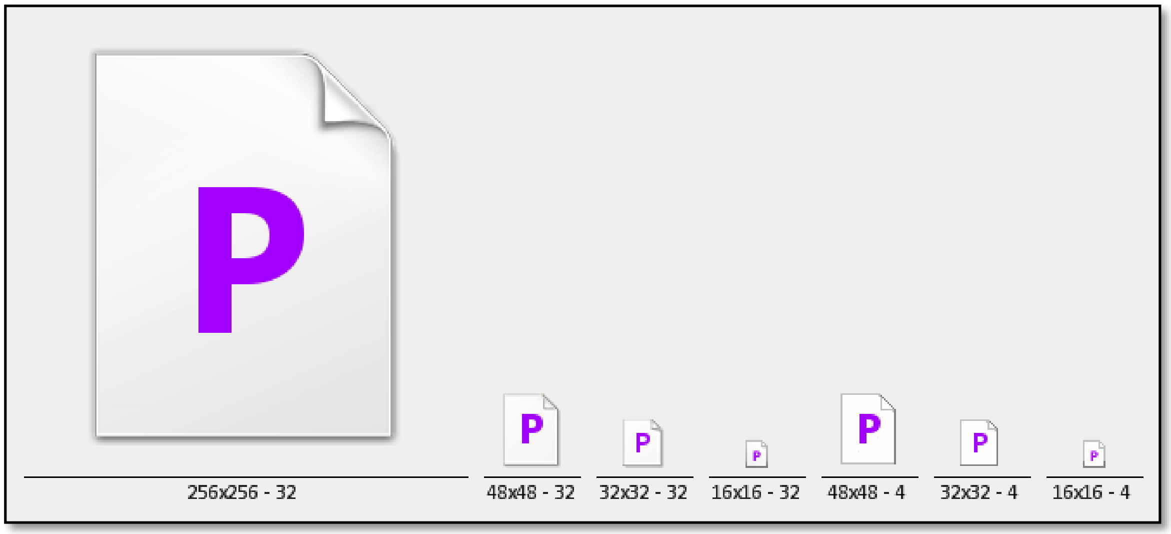


Figure 67: Process editor file icon

C.3 Monochrome Vector Icons

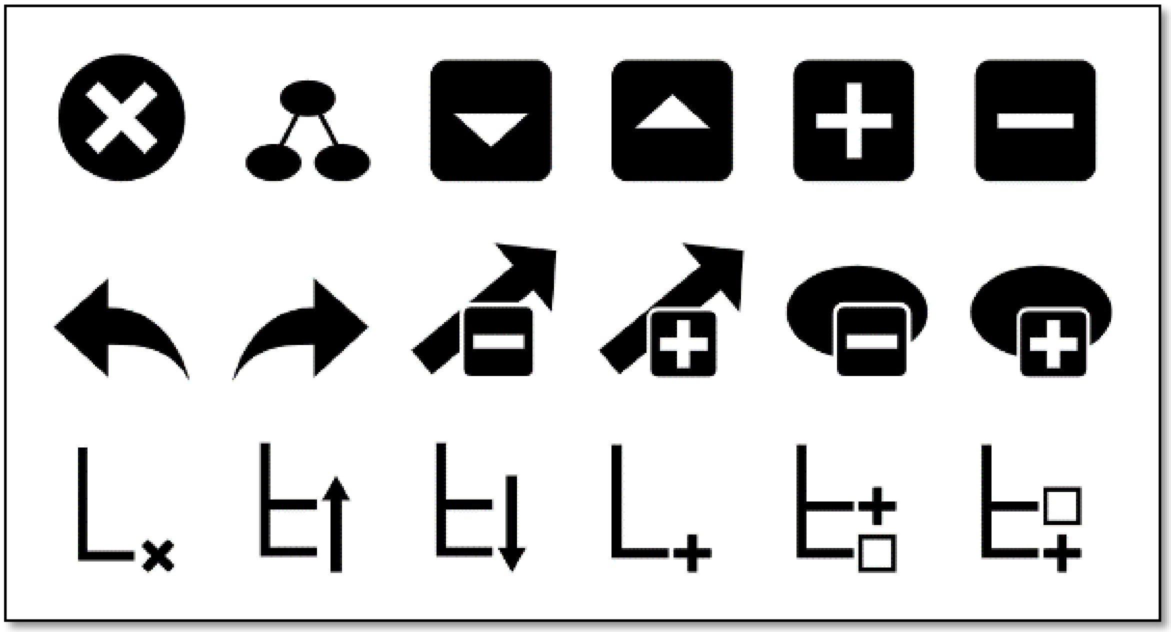


Figure 68: Vector icons used by all authoring tools