APROMORE Implementation

Marie-Christine Fauvet[*], Marcello La Rosa[†]
marie-christine.fauvet@imag.fr

Author	**Date**	**Update**
Marie	19/04/2010	Rename modules, update architecture. Remove relation file.
Marie	26/04/2010	Fix messages and orchestrations.
Marie	07/05/2010	Add interactions with editor.
Marie	17/05/2010	Integrate doc: Apromore Formats.docx.
Marie	28/05/2010	Add Delete and Edit operations (interfaces & interactions).
Marie	24/06/2010	Fix Edit choreography.
Marie	12/07/2010	Update architecture description.
		Versions may have 0 or * derived version(s).
Marie	13/07/2010	Update annotation specifications.
Marie	23/07/2010	Import, edit, create, and export GUIs are described.
Marie	28/07/2010	Service interactions revisited.
Marie	03/08/2010	About data synchronisation in Section 2.
Marie	03/08/2010	Add Toolbox, similarity search & merge.
Marcello	12/08/2010	Modify cpf data model.
Marie	13/08/2010	Modify meta-data model.
Marie	01/10/2010	Modification of edit model interface.
		Description of Write(New)Annotation operations.

[*]University of Grenoble, LIG, France.
[†]Queensland University of Technology - BPM research group, Brisbane, Australia.

Contents

List of Figures

Preamble

This document is a draft under continuous modifications.

1 Overall architecture

AProMoRe is implemented via a three-layered service oriented architecture and deployed over the internet (see Fig. 1). The Enterprise layer hosts the *Manager*—a public enterprise service which exposes all the repository features via Web service operations for integration with third-party applications, e.g. a BPM System. Moreover, these operations can be accessed via a Web *Portal*, which in turn delegates model editing functionality to Oryx (`http://bpt.hpi.uni-potsdam.de/Oryx/WebHome`). Other editors may be integrated, such as YAWL[1], etc.

The Intermediary layer hosts the *Canonizer* which is an adapter responsible for (de-)canonizing process models as they are imported/exported into/from the repository. The *Toolbox* is a façade over the advanced operations that can be performed on the stored process model collections. Access to these models is achieved via the *Data access* service in the Basic layer, which encapsulates data-centric operations for reading/writing data upon requests made by the other services. Finally, the *Access Control* service controls security aspects such as user authentication and authorization.

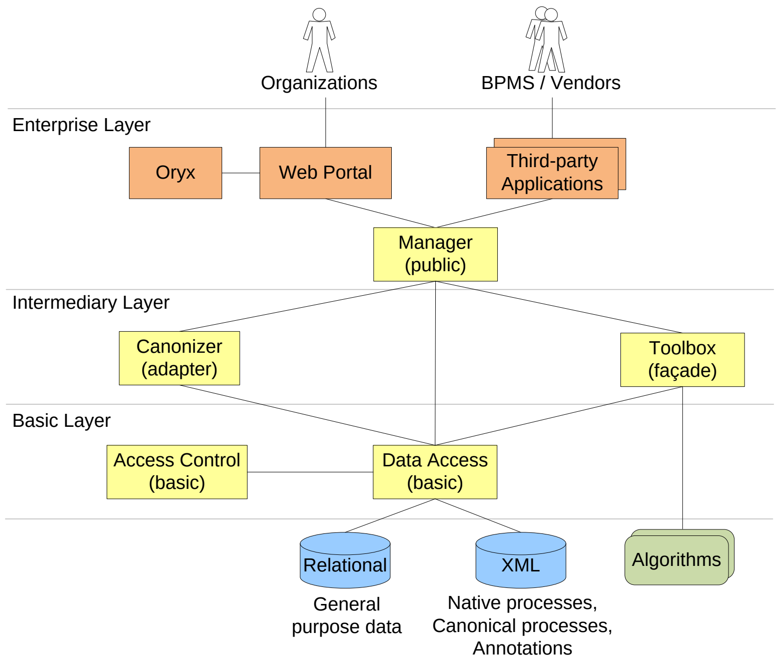


Figure 1: Apromore architecture

[1]`http://www.yawlfoundation.org`

2 Conceptual data model

Date	Comment
12/07/2010	Group concept is missing.

This section presents the conceptual model capturing information stored in Apromore databases. The class diagram depicted in Figure 2 models general purpose data while the one given in Figure 4 details canonical model.

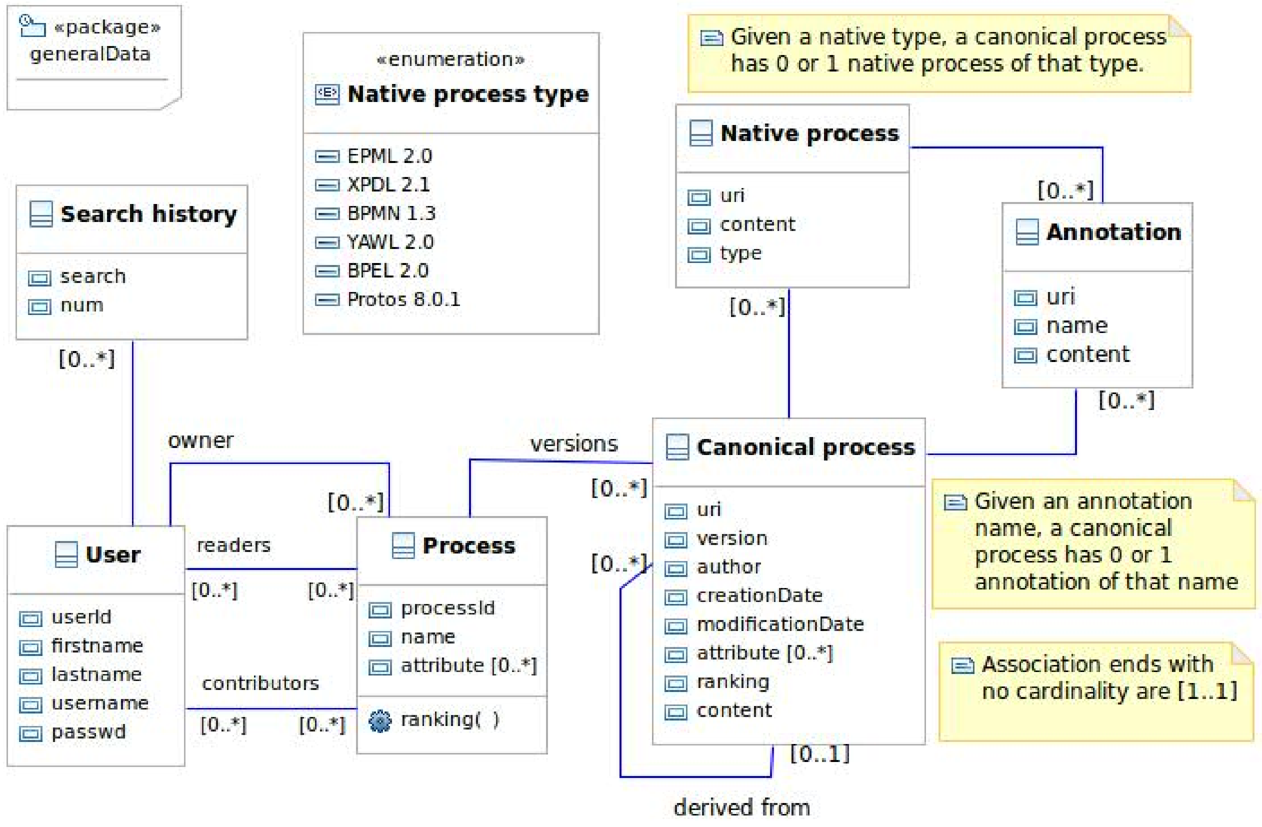


Figure 2: Users, processes, native and canonical formats

2.1 Users

Apromore portal offers public a limited number of operations. Authorised users must sign in; when connected each user is assigned a role which defines operations she/he is authorised to perform.

Roles are defined below:

- Public role gives permissions to ask for an account and signin (as well as read access to process summaries). Once identified, the anonymous user is granted the role *registered user*.
- Registered user role gives permissions to:

  - Import or create processes and information related to them (such as process versions, canonical representation, and native process, etc..).
  - Modify processes she/he owns (e.g. she/he created).
  - Nominate other users as readers or contributors to her/his own processes.
  - Modify processes for which she/he has been nominated as a contributor.
  - Read access to detailed information of processes for which she/he has been nominated as a reader.
  - Rank any processes.
- Administrator role gives read/write permissions on all information stored in the repository.

A user, identified by an id, has a first-name, last-name, username and password. Each user has a role.

Users are also associated with the history of their last searches performed with Apromore web portal. This information is used by the portal to allow users to choose a search among their past searches. Given a user, a search item is identified by the search expression; it is also identified by a timestamp (num attribute).

2.2 Processes

A process is identified by an id (attribute processId), it has a name and an owner. It may have contributors and readers (see class Process in Figure 2), and a list (possibly empty) of attributes. An attribute is meant to capture specific meta-data users may want to associate with process models, it is a pair <name, value>.

A process model has also a canonical description which captures its abstract syntax. The Canonical Process Format (CPF) has been derived from the analysis of commonalities among the native languages: EPC, BPMN 1.2, WF-nets, YAWL, BPEL and Protos. A CPF description is identified by its uri and has a content. As a process is likely to change over time, thus multiple versions are kept for it (each process version has exactly one canonical description). In the class diagram depicted in Figure 2, this is captured by the association versions. Each canonical process version has a name, a creation date, the date of its last update and an author (authors are not stored as users in the database), and a list (possibly empty) of attributes (pairs of <name, value>). Each version is identified by its name and the id of the process it belongs to. A version might be derived from another one (association derived from) while a version may have many derived versions (process versions are structured as a tree).

A process has a ranking derived by those given by users to each of its versions. Processes which were first imported into the repository with a given native format have an original native type.

Processes can be created by applying operations on other processes (a merge for instance). This is captured by the class association Operation type.

Annotations and native descriptions: Annotations (APF) captures meta-data associated with a CPF (e.g. documentation, layout, costs, etc.). An APF description cannot be empty, it contains at least one annotation element. Each annotation element may refer to a specific canonical element within the CPF or to the overall CPF. Annotations are highly extensible by specifying extra attributes and child elements beyond the basic ones. Also, there are two standard annotation types that extend the basic annotation type:

- Documentation: containing a list of Documentation elements, each of anyType
- Graphics: capturing the concrete syntax of a process model. This has been derived from the analysis of the concrete syntax of all NPFs above. Different from the CPF, this format includes the superset of all the NPF concrete syntaxes.

A canonical process may have many native descriptions, each of which of a different type (see enumeration Native process type). Apromore supports six Native Process Formats (NPFs):

- EPML 2.0: to serialize EPCs, eEPCs, iEPCs, C-EPCs and C-iEPCs models
- XPDL 2.1: to serialize BPMN 1.2 models
- PNML 1.3: to serialize WF-Nets
- YAWL 2.0: to serialize YAWL models
- BPEL 2.0: to serialize BPEL models
- Protos 8.0.1: to serialize Protos models

An APF is identified by its uri, it has a name (e.g. 'original', 'Top-down', etc) and a content. Similarly, an NPF is identified by its uri, associated with a native type and has a content. An NPF can be assigned multiple APFs, e.g. each capturing a different concrete syntax but an APF has only one original NPF (see association between Native process and Annotation).

Moreover, although a CPF may have been derived from one NPF, it can be exported to multiple NPFs (see the association between Canonical process and Native process). Given a native type, a CPF can only have 0 or 1 NPF of that type. Eventually, an CPF may be associated to many APFs. However, given an APF name, a CPF cannot have more then one APF of that name.

CPF, APF and NPF descriptions: CPF, NPF, and APF descriptions are currently stored as raw data (XML description) abstracted by the attributes content in their respective classes. Figure 3 depicts the generation of the description in I2 for a given process version defined in I1. The process version is first imported into the repository. From its native description, Apromore generates its CPF and APFs descriptions as well as meta-data to feed the database whose conceptual schema is given in Figure 2. During its life span, the process version may be associated with many APF. When an export is required, the CPF is de-canonised (with or without annotations), and among the meta-data stored in the database, those supported by I2 are synchronised with the resulting description.

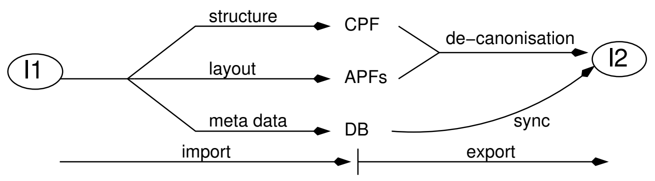


Figure 3: From a language to another: data involved

The UML class diagram shown in Figure 4 captures canonical processes, for details see [1]. Some informations captured in Figure 2 are redondant as they might be present in native languages as well (for instance, process name, version name, owner, documentation, etc.). Apromore keeps them synchronised.

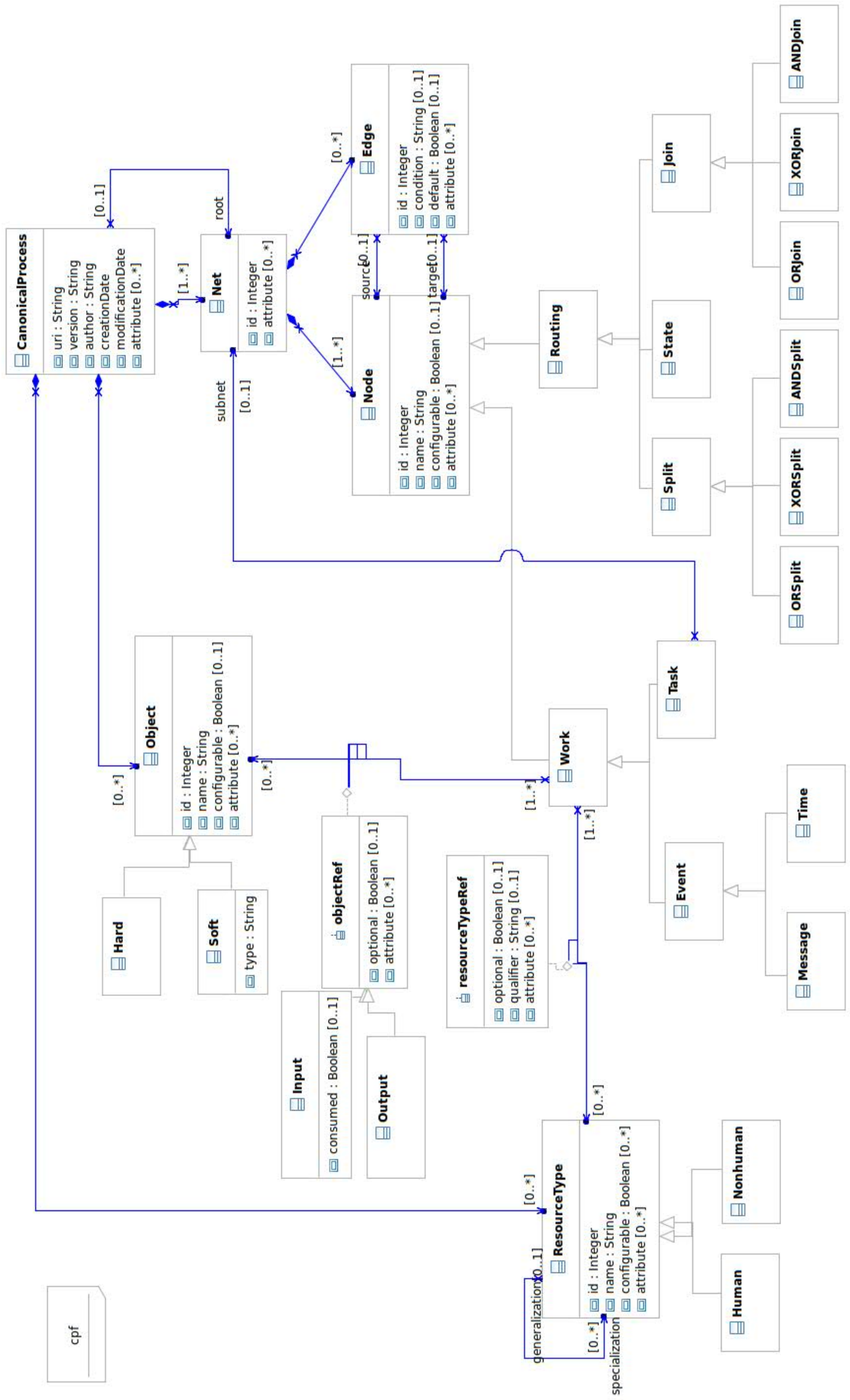


Figure 4: UML class diagram for canonical processes

3 User interface

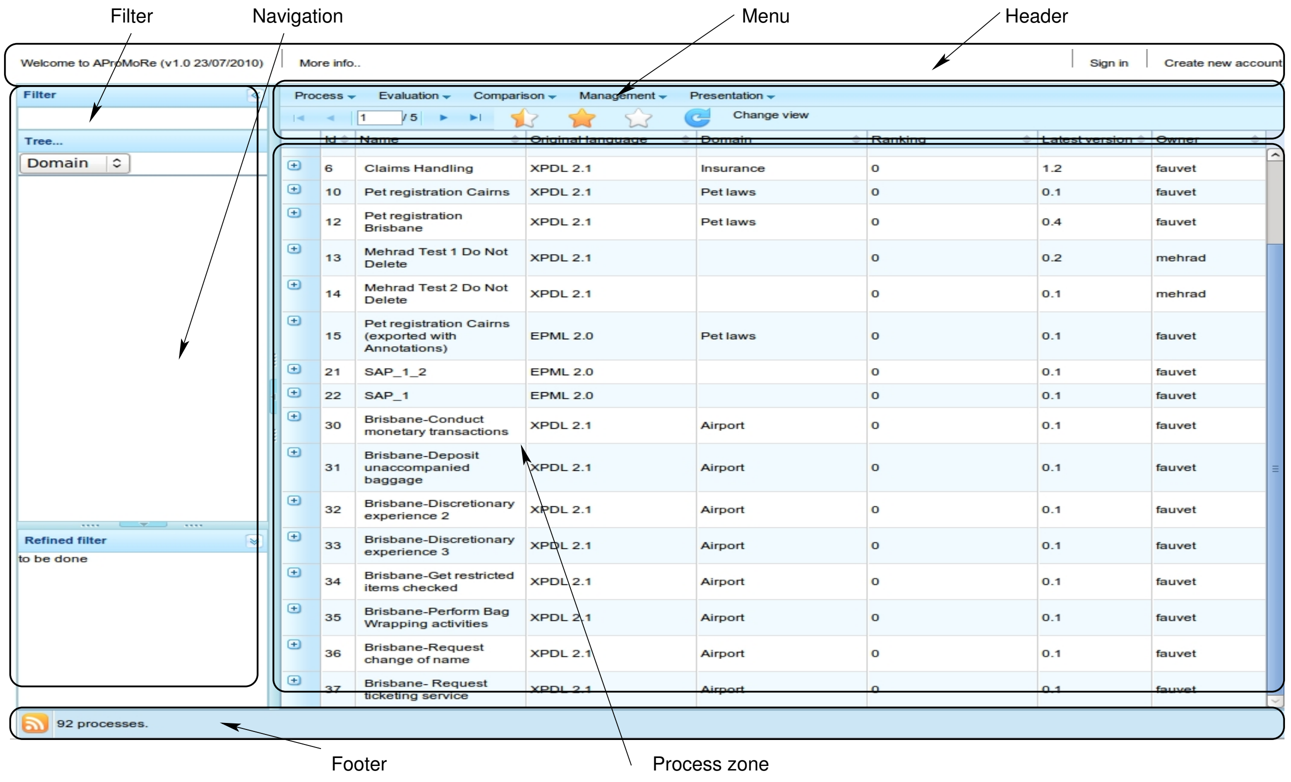
Date	Comment
29/03/2010	- header interactions not detailed. - not operator must be defined in search expression language. - search language implementation not detailed. - menu not described. - screenshots illustrating interfaces are missing.

Apromore portal is a web application which offers users graphical interfaces so they access the repository manager (eclipse project is Apromore-portal.)

3.1 Home page

	Components
View	WebContent/index.zul
Controller	src/apromore.portal.dialogController.MainController.java

Figure 5 shows Apromore home page organisation in the *public mode* (i.e. it is accessible by anonymous users and does not require authentification). When a user successfully signs in, the mode is switched to the *connected mode* (see Section 3.2).



to be done

Id	Name	Original language	Domain	Ranking	Latest version	Owner
6	Claims Handling	XPDL 2.1	Insurance	0	1.2	fauvet
10	Pet registration Cairns	XPDL 2.1	Pet laws	0	0.1	fauvet
12	Pet registration Brisbane	XPDL 2.1	Pet laws	0	0.4	fauvet
13	Mehrad Test 1 Do Not Delete	XPDL 2.1		0	0.2	mehrad
14	Mehrad Test 2 Do Not Delete	XPDL 2.1		0	0.1	mehrad
15	Pet registration Cairns (exported with Annotations)	EPML 2.0	Pet laws	0	0.1	fauvet
21	SAP_1_2	EPML 2.0		0	0.1	fauvet
22	SAP_1	EPML 2.0		0	0.1	fauvet
30	Brisbane-Conduct monetary transactions	XPDL 2.1	Airport	0	0.1	fauvet
31	Brisbane-Deposit unaccompanied baggage	XPDL 2.1	Airport	0	0.1	fauvet
32	Brisbane-Discretionary experience 2	XPDL 2.1	Airport	0	0.1	fauvet
33	Brisbane-Discretionary experience 3	XPDL 2.1	Airport	0	0.1	fauvet
34	Brisbane-Get restricted items checked	XPDL 2.1	Airport	0	0.1	fauvet
35	Brisbane-Perform Bag Wrapping activities	XPDL 2.1	Airport	0	0.1	fauvet
36	Brisbane-Request change of name	XPDL 2.1	Airport	0	0.1	fauvet
37	Brisbane- Request ticketing service	XPDL 2.1	Airport	0	0.1	fauvet

92 processes.

Figure 5: Apromore home page

The page is divided into zones:

- Header with four buttons (see details Section 3.2).
- Process zone: in the home page this zone supplies process summaries organised as a table (see section 3.3).
- Filter allows connected users to apply search queries on processes (see Section 3.4).
- Navigation provides a tree navigation through available processes (see Section 3.6).
- Footer is a zone where short messages are displayed.

3.2 Header

	Components
View	WebContents/macros/header.zul
Controller	apromore.portal.dialogController.HeaderController.java

The header contains four buttons:

- "Welcome to AProMoRe (v0.5 22/06/2010)": link to Apromore release notes[2].
- "More info...": link to Apromore web site.
- "Sign in": opens a dialog box so registered users can log in.
- "Create new account": opens a dialog a box so users can register to Apromore.

3.3 Process zone

	Components
View	WebContents/macros/processtable.zul
Controller	apromore.portal.dialogController.ProcessTableController.java

When first loaded, this window shows summaries of available processes, organised as a table. Possible interactions are (they are all enabled in public mode):

- Click on Change view switches from table view to thumbnails view (and *vice-versa*).
- Depending on the number of items to display, paging facility is provided.

On table view mode:

- Click on an column name orders rows according to this column.
- Click on +: displays summary of process versions.
- Click on -: hides process versions.
- Click on a process name: selects/unselects the latest version of the process.

[2]http://code.google.com/p/apromore

- Ctrl+click performs selection of all processes between the one where the click occurs and the closest already selected (if any).
- Click on a version name: selects/unselects the version
- Click on mid-orange star: reverts selection
- Click on orange star: selects all process latest versions
- Click on white star: unselects all selected processes.
- Click on circular blue arrow: refreshes the table.

On thumbnails view mode:

- *to be defined*

3.4 Search

	Components
View	WebContents/macros/simplesearch.zul
Controller	apromore.portal.dialogController.SimpleSearchController.java

This feature is enabled in connected mode only.

By entering an expression containing keywords, connected users can search for processes that match the keywords.

For each registered user, a search history is memorised during sessions and then stored in the database at logout. At login, the session starts with the search history retrieved from the database (if any). The maximum number of items kept in a search history is defined in Apromore/apromore.portal.common.Constants.java (see constant maxSearches).

Given a keyword k, a process satisfies k if either its name, or its domain, or its native type, or its owner's first-name, or its owner's last-name, or the name of one of its versions, contains k (case insensitive). This list can be modified by altering the view keywords defined in Apromore database (see Apromore-database/Scripts/apromore.sql).

Expressions must conform the grammar:

```
E  ⟶ T | E ; T /* ; semantics is or. */
T  ⟶ F | T , F /* , semantics is and. */
F  ⟶ K | ( E ) /* K may contain any character except ',' and ';'. Spaces placed at the beginning or at
                  the end of K are ignored.  */
```

If the search does not contain any keywords, all processes are retrieved.

Examples:

```
yawl;protos
   /* returns processes which satisfies yawl or protos. */
(yawl; protos), pets law
   /* returns processes which satisfies 'pets law' and either yawl or protos. */
```

Implementation details are given in appendix A.

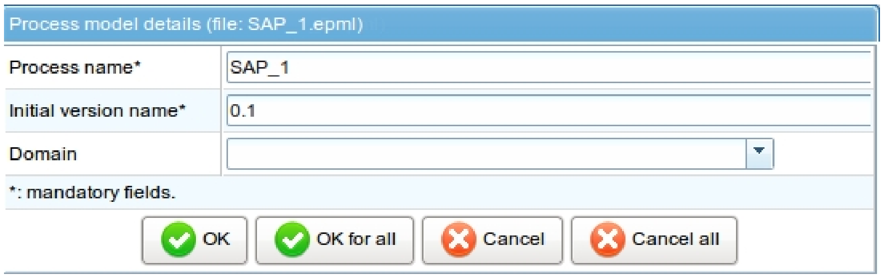
3.5 Menu

	Components
View	WebContents/macros/menu.zul
Controller	apromore.portal.dialogController.MenuController.java

This feature is enabled in connected mode only.

- Process: create or import process models, export, edit process models, edit meta-data, delete, merge selected process versions
- Evaluation:
- Comparison:
- Management:
- Presentation:

Import: users have the choice of importing a file or an archive, so many process models are imported at once. Then for each file (one if the user selected one file, or many if he selected an archive of files), the dialog boix shown in Figure 6 is opened.



Process model details (file: SAP_1.epml)

Process name*	SAP_1
Initial version name*	0.1
Domain	

*: mandatory fields.

Figure 6: Apromore import menu

Process and version names are taken from the imported file if possible. If not, the default process name is the name of the file, the default version name is “0.1”. The user is given the choice to select an existing domain or to enter a new one. The process owner is the user who performs the import task. However, authorised users can set the process owner to an existing Apromore user.

- Click on “OK”: submit the process import with the values (either default or entered).
- Click on “OK for all”: submit import of all remining processes with default values.
- Click on “Cancel”: cancel the process import.
- Click on “Cancel all”: cancel the import of all remining processes.

Export: one or more process models may be exported in any of the formats available (xpdl, epml, yawl, etc.). The export formats available include canonical and annotation formats (see Figure 7).

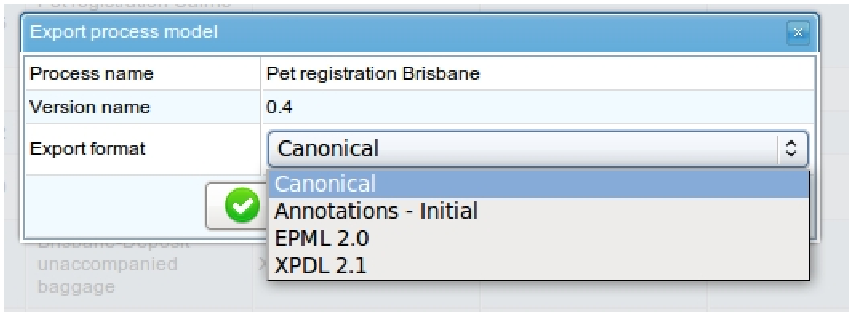


Figure 7: Export a model: format choice

When a process modelling language is selected (such as EPML, XPDL, etc.) the user is given the choice of the annotation file to use, if any available. If the APF named 'Initial' is selected, then the original native format is taken from the database as it was stored at the process version creation time. In all other cases, an NPF is generated by applying on the canonical description of the process the adapter corresponding to the format chosen by the user, and if an annotation file has been chosen, it is used by the adapter (see Figure 8).

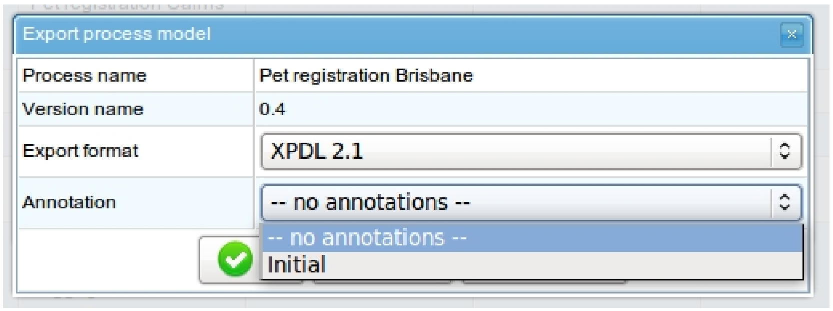
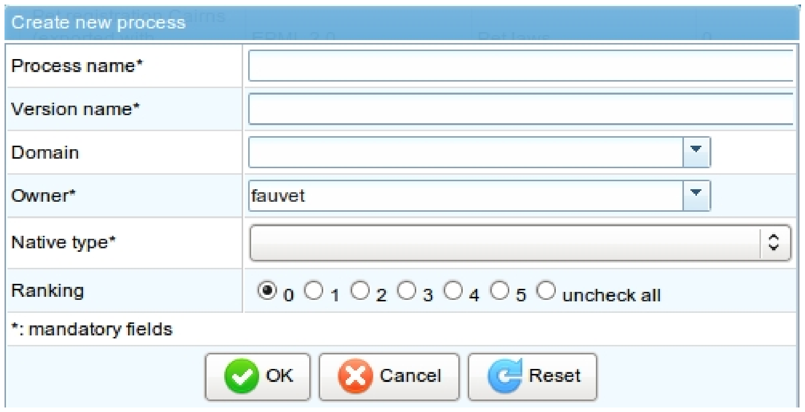


Figure 8: Export a model in a native language: annotation choice

Create: a new process can be created (see Figure 9). Users are requested to provide the process and version names, domain, and ranking for the new process version. Authorised users are entitled to set the owner to any registered users. For all others, owner is set to the user performing the creation.


Create new process

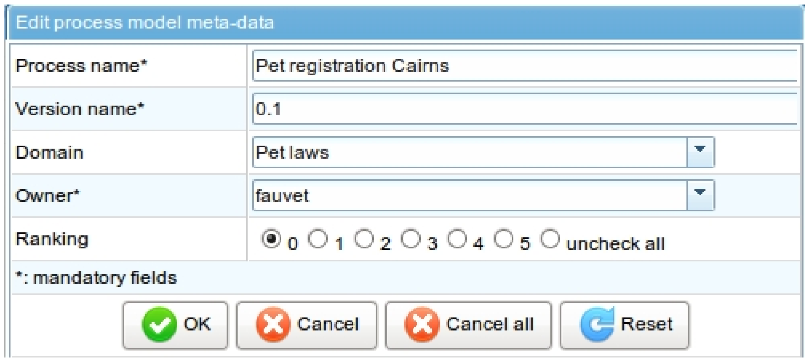
Process name*	
Version name*	
Domain	
Owner*	fauvet
Native type*	
Ranking	[✓ 0] 1 2 3 4 5 uncheck all

*: mandatory fields

Figure 9: Create model dialog box

Edit meta-data: process and version names, domain, and ranking, of a process version can be modified (see Figure 10). Authorised users are entitled to set the owner to any registered users.

- Click on "OK": submit the modifications with the entered values.
- Click on "Cancel": cancel the current modifications.
- Click on "Cancel all": cancel modifications of all remining processes.
- Click on "reset": restore the initial values.


Edit process model meta-data

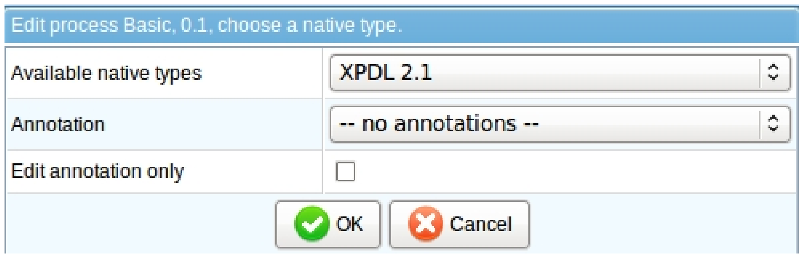
Process name*	Pet registration Cairns
Version name*	0.1
Domain	Pet laws
Owner*	fauvet
Ranking	[✓ 0] 1 2 3 4 5 uncheck all

*: mandatory fields

Figure 10: Edit model meta-data dialog box

Edit model: features for editing process models are delegated to external editors (such as Oryx). Users are given the choice of editing one or more process models, and for each of which, users can choose to edit it given one of its annotation file. The following situations can arise:

- Model, considering no annotations: see Figure 11.

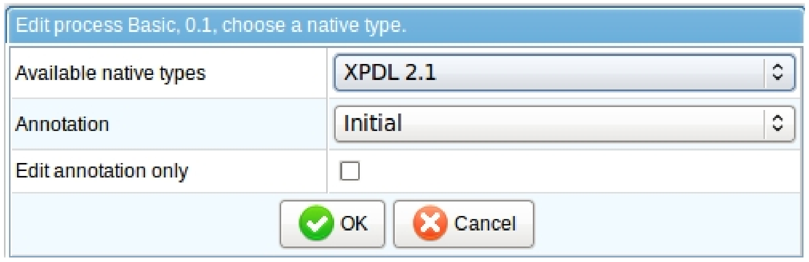


Edit process Basic, 0.1, choose a native type.

Available native types	XPDL 2.1
Annotation	-- no annotations --
Edit annotation only	☐

Figure 11: Edit model without annotations

- Model considering one of its annotation file: see Figure 12.

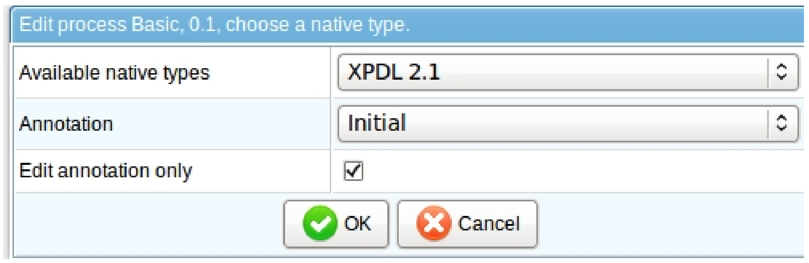


Edit process Basic, 0.1, choose a native type.

Available native types	XPDL 2.1
Annotation	Initial
Edit annotation only	☐

Figure 12: Edit model with its initial annotation file

- Annotation only: see Figure 13.



Edit process Basic, 0.1, choose a native type.

Available native types	XPDL 2.1
Annotation	Initial
Edit annotation only	☑

Figure 13: Edit model initial annotations

For each model/annotation file to edit, Apromore requests the editor to start an editing work, below are described the differents ways to finish it:

- Cancel: the user closes the editor window. By doing so, all modifications since the last "save" or "save as" are discarded.
- Save as: the editor requests Apromore to create a new process model or a new annotation file (users are asked to provide a name). In case of a process model users must also give its first version name; a new CPF is generated, associated with a new APF (automatically named "initial").

- Save: in case of annotations, the previous annotation file is overridden by the new one. In case of a process model the editor asks users to enter a version name; by giving a new version name users can save modifications in a new process version or by giving the name of the current process version users can override it:
  - New version name: the editor requests Apromore to create a new version for the process (the given new version name must not already exist). This new version is derived for the current one. A new CPF is generated, associated with a new APF which is automatically named “initial”.
  - Current version is overridden: this is possible only if the current version is a leaf in the derivation tree.

The choreography of interactions triggered by the operation edit model is detailed in Section 5.2.

3.6 Navigation

	Components
View	See Remco.
Controller	See Remco.

4 Service interfaces

Date	**Comment**
30/03/2010	Operation for new user not specified.

This section describes interface operations as they are exposed by each of Apromore services: Portal (see Section 4.1), Manager (see Section 4.2), Data access (see Section 4.3), Canoniser (see Section 4.4) and Toolbox (see Section 4.5). Figure 14 shows for each service the interface it provides defined as a WSDL description. The naming convention is that xxx_yyy.wsdl describes the interface provided by the service xxx and consumed by yyy. xxx_yyy.wsdl and all associated xschema files are located in eclipse project implementing the service xxx.

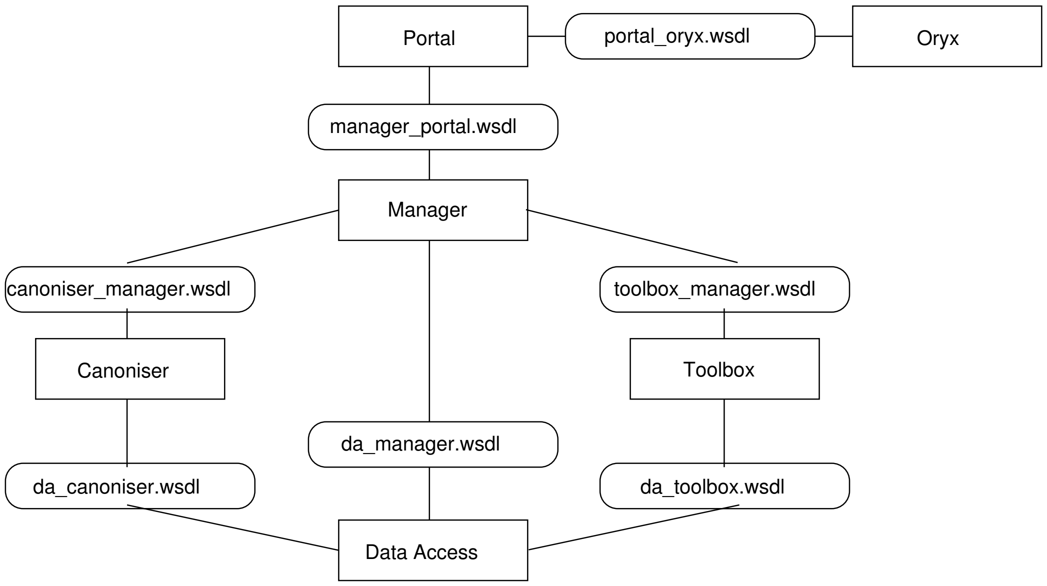


Figure 14: WSDL descriptions of service interfaces

4.1 Portal

This section documents operations exposed by the portal to the tools which support process model editing. Editing work is initiated as users choose "edit model" in the menu described in Section 3.5, page 13. First, for each process model to be edited, the portal creates an edit session which contains the information related to the process and necessary for its edition. The portal then sends the editing tool a message containing the edit session id (one message per model to be edited). This id will be used in subsequent interactions between the portal and the editor (see details in Section 5.2).

Data contained in edit sessions are:

username, processName, versionName, domain, nativeType : string
processId: integer
withAnnotation: boolean
annotations: string *if withAnnotation then annotation has a value.* */

Remark: edit sessions should have life spans which match the actual user's session (from log-in to log-out). At the moment, edit sessions are kept in the database.

ReadNative. This operation returns the NPF of a process to be edited.

ReadNative: ReadNativeInputMsg $\longrightarrow$ ReadNativeOutputMsg
/ ReadNativeInputMsg(m) requests the native description of the process version whose details are associated with the edit session id included in m. The returned message contains the result (code and message); the native description is attached. */*

WriteProcess. The WriteProcess operation exposed by the portal is meant to update an existing process in the repository by deriving a new version or overridding an existing one.

WriteProcess: WriteProcessInputMsg $\longrightarrow$ WriteProcessOutputMsg
/ WriteProcess(m) requests the process version associated with the edit session id given in m to be updated in the repository. m contains the previous name of the process version (preVersion). The native description is given as an attachment, its contains the new name of the process version (newVersion). If preVersion = newVersion, preVersion is overridden only if it is a leaf in the derivation tree. If preVersion ≠ newVersion, a new version is derived from preVersion, only if its name doesn't already exist. If successful, returns the message whose result code is 0. Otherwise returns a message whose result code is -1 and an error message. */*

WriteNewProcess. This operation creates a new process as well as its first version.

WriteNewProcess: WriteNewProcessInputMsg $\longrightarrow$ WriteNewProcessOutputMsg
/ WriteNewProcess(m) requests the process whose details are given in m to be stored in the repository. Details are process name, version name, and the edit session id associated to the process version which was originaly edited. The native description is given as an attachment. If successful, returns a message which contains a new edit session id (generated for the new process version) and a result whose code is 0; otherwise returns a message whose result code is -1 and an error message. */*

WriteAnnotation This operation is called when saving editing work on annotations.

WriteAnnotation: WriteAnnotationInputMsg $\longrightarrow$ WriteAnnotationOutputMsg
/ WriteAnnotation(m) requests the annotation associated with edit session whose id is in m to be updated in the repository. The native description is given as an attachment. If successful, returns a message which contains a new edit session id (generated for the new process version) and a result whose code is 0; otherwise returns a message whose result code is -1 and an error message. */*

WriteNewAnnotation This operation is called when saving editing work as new annotation.

WriteNewAnnotation: WriteNewAnnotationInputMsg $\longrightarrow$ WriteNewAnnotationOutputMsg
/ WriteNewAnnotation(m) requests the annotation associated with edit session whose id is in m to be created in the repository. m contains a name for the new annotation. The native description is given as an attachment. If successful, returns a message which contains a new edit session id (generated for the new process version) and a result whose code is 0; otherwise returns a message whose result code is -1 and an error message. */*

4.2 Manager

ReadXXX. Read operations are exposed for the purpose of retrieving specific informations from the database (process summaries, domains, native types, and user details). Outgoing messages are depicted in Figure 15. Classes User and Process refer to the UML class diagram given Figure 2.

Outgoing messages have a Result element which reports whether or not the associated operation successfully completed:

- Attribute code=0 means that the operation was successful (attribute error message is thus empty).

- Attribute code=-1 means that a problem occured (attribute error message explains why).

Abstract definition of operations are given below. Error conditions are only described for the first operation as the same principle applies to all.

ReadNativeTypes: ReadNativeTypesInputMsg $\longrightarrow$ ReadNativeTypesOutputMsg

/ ReadNativeTypes(im) requests the native types currently supported by Apromore. The message im is empty. If successful returns a message which contains a list of pairs of strings. Each pair <n, e> is such as n is a native type and e is the extension commonly given to files which contains process models descriptions of that type. Otherwise the returned message has a result code equal to -1 and an error message which describes error. */*

ReadDomains: ReadDomainsInputMsg $\longrightarrow$ ReadDomainsOutputMsg

/ ReadDomains(im) requests the domains, each of which associated with at least one process. The message im is empty. If successful returns a message which contains a list of domains (strings) and a result whose code is 0, otherwise a message whose result code is -1 and error message describes error (list of domains is empty). */*

ReadProcessSummaries: ReadProcessSummariesInputMsg $\longrightarrow$ ReadProcessSummariesOutputMsg

/ ReadSummaries(im) requests summary of the processes currently stored which satisfy the search expression s contained in im (s may be empty, summaries of all processes are then returned). If successful, returns a message which conforms to the class ReadProcessSummariesOutpuMsg given in Figure 15; result code is then 0. If not successful result code is -1 and the error message explains why. */*

ReadUser: ReadUserInputMsg $\longrightarrow$ ReadUserOutputMsg

/ ReadUser(im) requests details of the user whom username is contained in im. If successful, returns a message which conforms to the class ReadUserOutputMsg given in Figure 15; the result code is then 0. If not successful result code is -1 and the error message explains why. */*

ReadAllUsers: ReadAllUsersInputMsg $\longrightarrow$ ReadAllUsersOutputMsg

/ ReadAllUsers(im) requests the list of all registered user's usernames. The message im is empty. If successful returns a message which contains a list of strings and a result whose code is 0, otherwise a message whose result code is -1 and error message describes error. */*

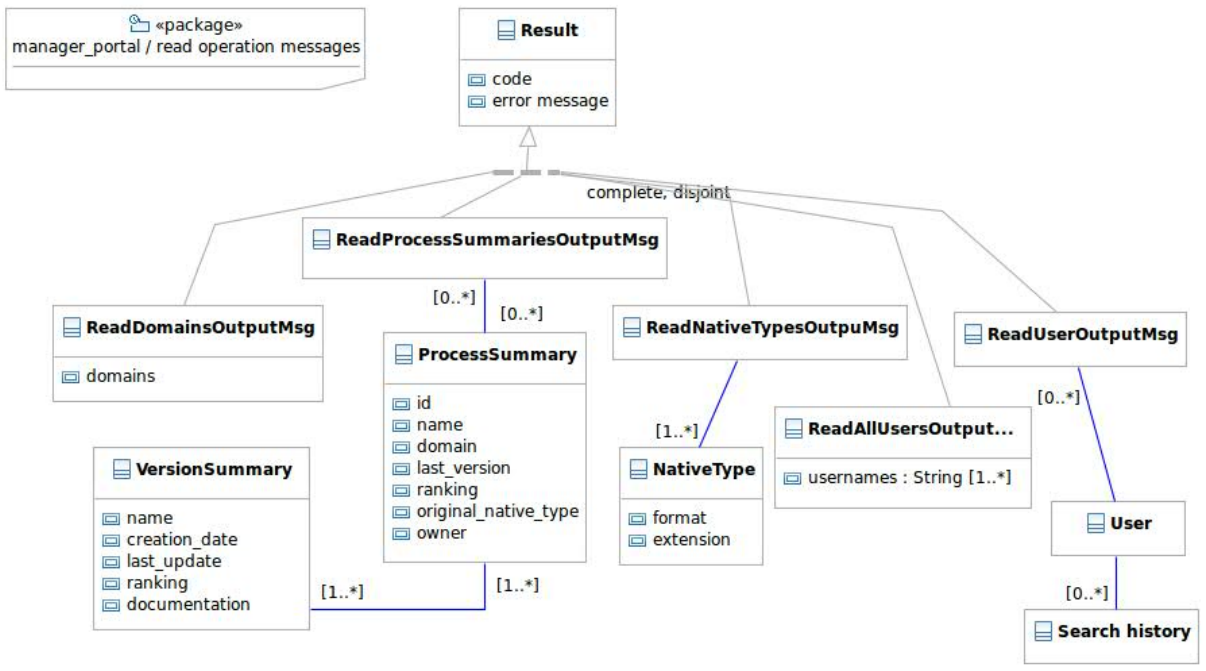


Figure 15: ReadXXX operations: outgoing message schemas

WriteUser. The operation WriteUser is meant to store new details of a user or a new user's details.

WriteUser: WriteUserInputMsg $\longrightarrow$ WriteUserOutputMsg

/ WriteUser(im) requests user information included in message m to be stored (see the class diagram depicted in Figure 16). If successful, returns result whose code is 0 otherwise returns a result whose code is -1 and an error message which explains why. */*

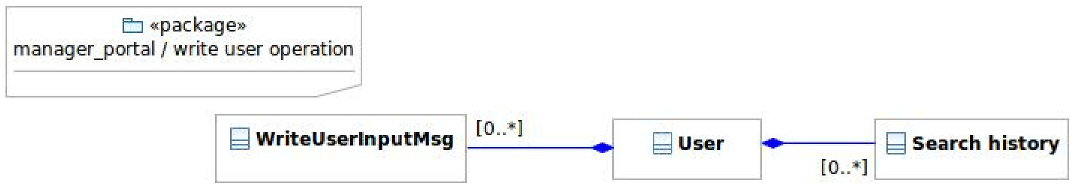


Figure 16: WriteUser: incoming message schema

ImportProcess. Process models described in any of the native formats supported by Apromore can be imported into the repository by using the ImportProcess.

ImportProcess: ImportProcessInputMsg $\longrightarrow$ ImportProcessOutMsg

/ ImportProcess(m) requests the process whose meta-data are included in the message m, and the native description given as an attachment, to be imported in the repository (see the class diagram depicted in Figure 17). If successful, returns the summary of the process (see class ProcessSummary in Figure 15) as well as a message whose result code is 0 otherwise returns a message whose result code is -1 and an error message. */*

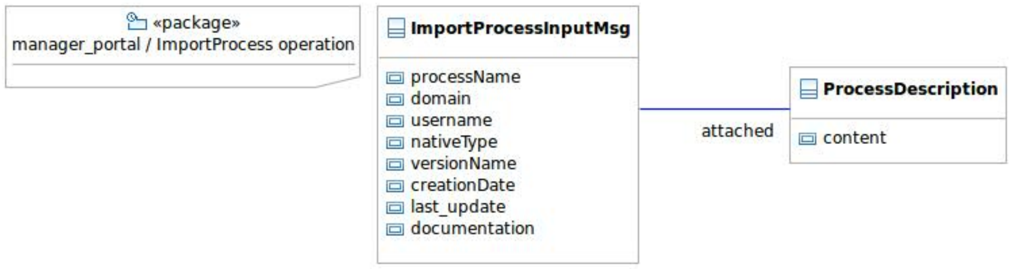


Figure 17: ImportProcess operations exposed by the manager: incoming message schema

ExportFormat. The operation ExportFormat is meant to perform exports from process models: either annotations, canonical or native formats.

ExportFormat: ExportFormatInputMsg $\longrightarrow$ ExportFormatOutputMsg
/ ExportFormat(mi) requests the export of the description in type nf for the process version identified by <p, v> (type nf, process id p, and version name v given in mi, see Figure 18). Values for nf may be: Constants.CANONICAL, Constants.ANNOTATIONS (see class Constants in package org.apromore.common) or any of the supported native type. If nf value is a native type, then mi attribute withAnnotation may be set to true, then the APF whose name is given in mi attribute annotation, will be used to generate the corresponding native description. The returned message contains the result (code and message), and if successful the native description is attached (the input message conforms to the class diagram given in Figure 18). */*

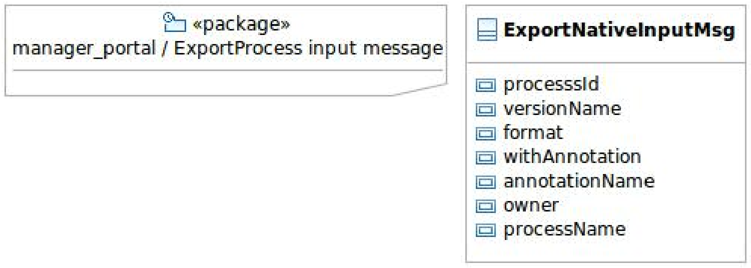


Figure 18: ExportFormat operation exposed by the manager: incoming message

UpdateProcess. This operation is to be used to store a new version or to override an existing version of an existing process.

UpdateProcess: UpdateProcessInputMsg $\longrightarrow$ UpdateProcessOutputMsg
/ UpdateProcessInputMsg(m): m contains process version meta data (which are not present in the attached NPF) and the name of the previous version (see the class diagram depicted Figure 19. */*

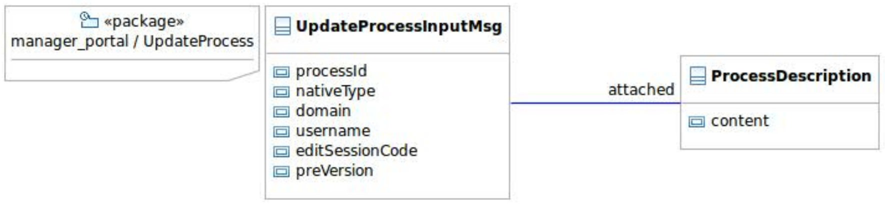


Figure 19: UpdateProcess operation exposed by the manager: incoming message schema

DeleteProcessVersion. This operation is requested by the portal for the selected processes the user applied delete on.

DeleteProcessVersion: DeleteProcessVersionInputMsg $\longrightarrow$ DeleteProcessVersionOutputMsg
/ DeleteProcessVersion(m) delete the given processes, and for each of which, its given versions as indicated in m (see the class diagram depicted in Figure 20). The returned message contains the result (code and message). */*

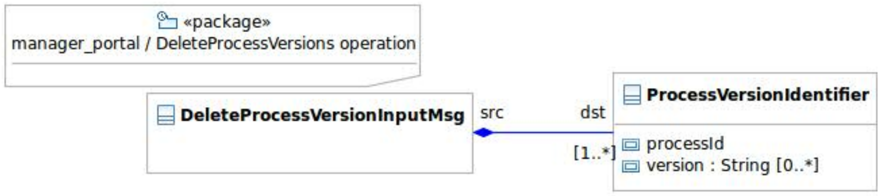


Figure 20: DeleteProcessVersion operation exposed by the manager: incoming message schema

WriteEditSession. This operation is requested by the portal for each selected process versions the user applied edit on.

WriteEditSession: WriteEditSessionInputMsg $\longrightarrow$ WriteEditSessionOutputMsg
/ WriteEditSession(m) requests the edit session whose details are in m to be stored in the database (see the class diagram depicted in Figure 21). The returned message contains the result (code and message), and if successful the code associated with the edit session. */*

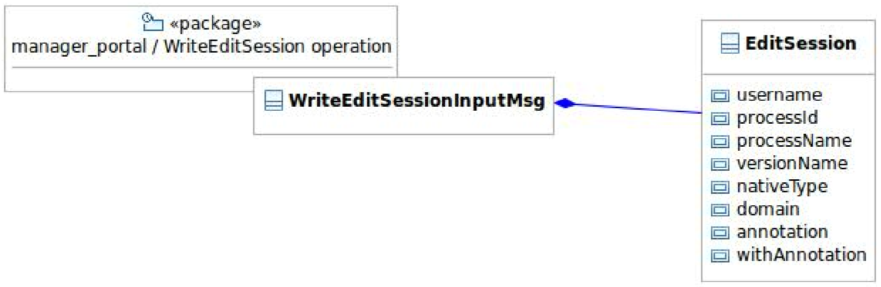


Figure 21: WriteEditSession operation exposed by the manager: incoming message schema

ReadEditSession. This operation is meant to return the native description associated with a given edit session as well as the edit session details.

ReadEditSession: ReadEditSessionInputMsg $\longrightarrow$ ReadEditSessionOutputMsg
/ ReadEditSession(m) requests the details of the edit session whose id is included in m, and the related native description. The returned message contains the result (code and message), and if successful the native description as an attachment. */*

DeleteEditSession. This operation deletes an edit session.

DeleteEditSession: DeleteEditSessionInputMsg $\longrightarrow$ DeleteEditSessionOutputMsg
/ DeleteEditSession(m) requests the edit session whose code is included in m to be deleted. The returned message contains the result (code and message). */*

4.3 Data access

4.3.1 Interface with the manager

ReadXXX. ReadXXX operations have the same specifications as those described in Section 4.2.

WriteUser. The operation WriteUser has the same specification as the one described in Section 4.2.

ReadFormat. returns a description (cpf, anf or native) of a given process version.

ReadFormat: ReadFormatInputMsg $\longrightarrow$ ReadFormatOutputMsg
/ ReadFormatInputMsg(m) returns the description of the process version whose details are associated with the code included in m (process id, version name and native type) according to the requested format (canonical, specific annotation, native description). The returned message contains the result (code and message); if successful, the native description is attached. */*

ReadCanonicalAnf. returns the canonical description of a given process version. An annotation file may be requested too.

ReadCanonicalAnf: ReadCanonicalAnfInputMsg $\longrightarrow$ ReadCanonicalAnfOutputMsg

/ ReadCanonicalInputMsg(m) requests the canonical description of the process version whose details are given in m (details are: processId and version). If m attribute withAnnotation is set to true, the APF whose name is given in m attribute annotationName is requested too. The returned message contains the result (code and message); the canonical and annotation descriptions are attached. */*

EditDataProcess.

DeleteProcessVersion. The operation WriteUser has the same specification as the one described in Section 4.2.

WriteEditSession, ReadEditSession and DeleteEditSession. These operations have the same specification as those described in Section 4.2.

4.3.2 Interface with the canoniser

StoreNativeCpf. The operation StoreNativeCpf has an incoming message which contains the process name, its native type, domain, owner's username, version name, its canonical description as well as the corresponding annotations (see Figure 22). To avoid unecessary umarschalling, canonical, annotations and native description are supplied as attachements. The operation returns the summary of the process version which has been created.

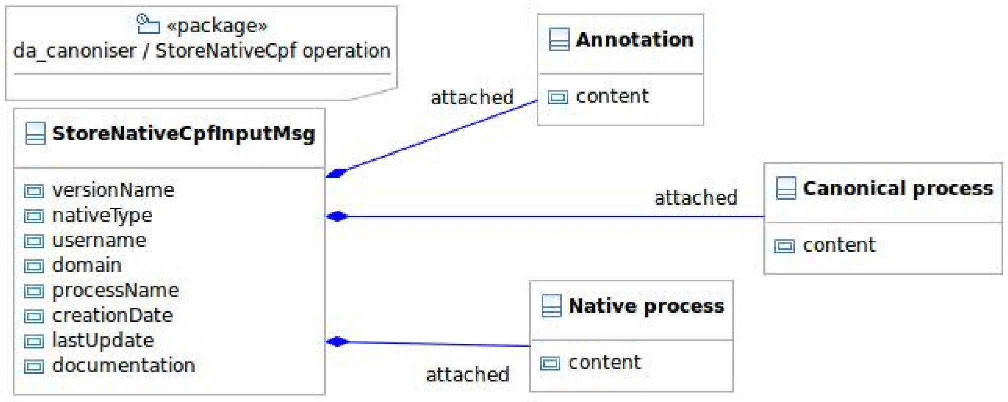


Figure 22: StoreNativeCpf-input-message operation exposed by DA: incoming message schema

StoreNativeCpf: StoreNativeCpfInputMsg $\longrightarrow$ StoreNativeCpfOutputMsg

/ StoreNativeCpf(m) requests the process whose details, native description, canonical description and annotations are attached to the message m to be stored in the repository. If successful, returns the summary of the process (see class ProcessSummary in Figure 15) as well as the message whose result code is 0 otherwise returns a message whose result code is -1 and an error message. */*

StoreNative. The operation StoreNative is meant to store the native description of a given existing process version. This operation has an input message which contains the process Id and version name, the native type of the description to be stored, and as an attachment the actual native description of the process (see Figure 22). To avoid unecessary umarschalling, the native description is supplied as an attachement.

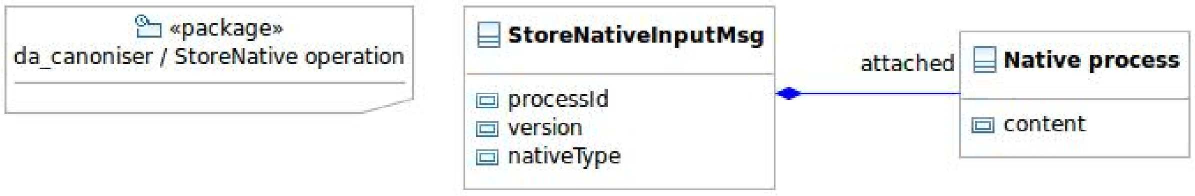


Figure 23: StoreNative-input-message operation exposed by DA: incoming message schema

StoreNative: StoreNativeInputMsg $\longrightarrow$ StoreNativeOutputMsg
/ StoreNative(m) requests the process native description of the process version to be stored in the repository. If successful, returns the message whose result code is 0 otherwise returns a message whose result code is -1 and an error message. */*

StoreVersion. This operation performs the creation of a new version for a given process version (the new version is stored as a derived version).

StoreVersion: StoreVersionInputMsg $\longrightarrow$ StoreVersionOutputMsg
/ StoreVersion(m) requests the derivation of a new version from the one identified by the process id and the version name given in m (native type and domain are given too, see details in Figure24). Other details for the new version are present in the attached NPF. If successful, returns the message whose result code is 0 otherwise returns a message whose result code is -1 and an error message. */*

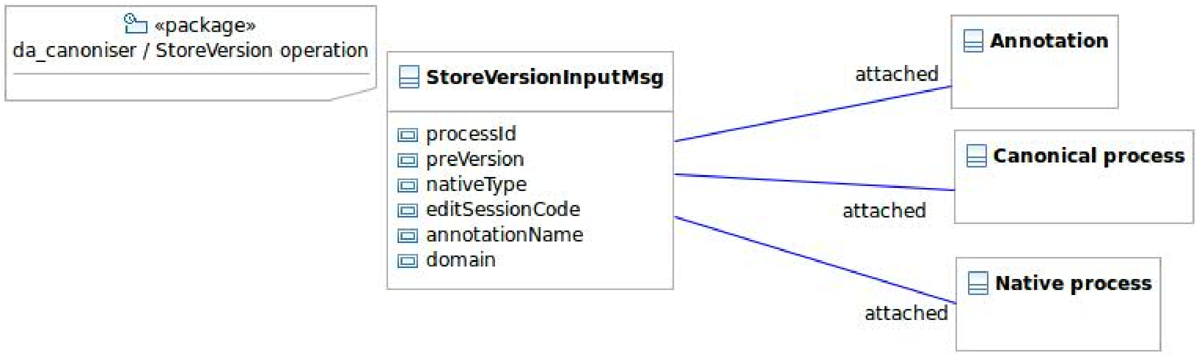


Figure 24: StoreVersion-input-message operation exposed by DA: incoming message schema

4.4 Canoniser

CanoniseProcess. The operation CanoniseProcess exposed by the Canoniser has the same specification as the operation ImportProcess exposed by the Manager (see Figure 17 for the input message structure).

CanoniseProcess: CanoniseProcessInputMsg $\longrightarrow$ CanoniseProcessOutputMsg
/ See ImportProcess provided by the manager. */*

DeCanoniseProcess. The operation DeCanoniseProcess is to be used to request a native description of a process given in the canonical format.

DeCanoniseProcess: DeCanoniseProcessInputMsg $\longrightarrow$ DeCanoniseProcessOutputMsg
/ DeCanoniseProcessInputMsg(m) given a process version identified by (pi, v), requests the decanonisation in native format nf of the process version whose canonical description is cpf (pi, v, nf are included in m and cpf is attached). See Figure 25. The returned message contains the result (code and message) and if successful, the native description as an attachment. */*

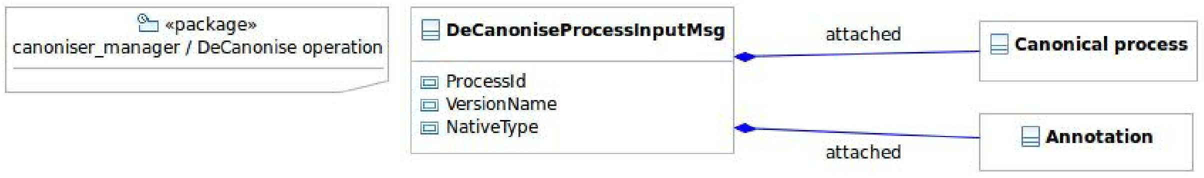


Figure 25: DeCanonise-input-message operation exposed by canoniser: incoming message schema

CanoniseVersion. This operation is requested by the manager during the process of saving an editing work into a new version of for an existing process. Input message schema needs to be revisited....

CanoniseVersion: CanoniseVersionInputMsg $\longrightarrow$ CanoniseVersionOutputMsg
/ CanoniseVersion(m) given a process identified by pi of type nf, requests the canonisation of the process version whose native description is npf (pi, nf are included in m and npf is attached). The returned message contains the result (code and message) and if successful, the native description as an attachment. */*

4.5 Toolbox

SearchForSimilarProcesses

MergeProcesses

5 Service interactions

This section details choreographies operations depending on whether they are exposed by the manager (see Section 5.1) or by the portal for the purpose of communicating with Oryx (see Section 5.2).

5.1 Operations exposed by the manager

ReadXXX Message exchanges required to achieve ReadFormats, ReadDomains, ReadProcessSummaries and ReadUser operations rely all on the same pattern depicted in Figure 26. The repository manager forwards the incoming message to the DAS and forwards its response back to the requester.

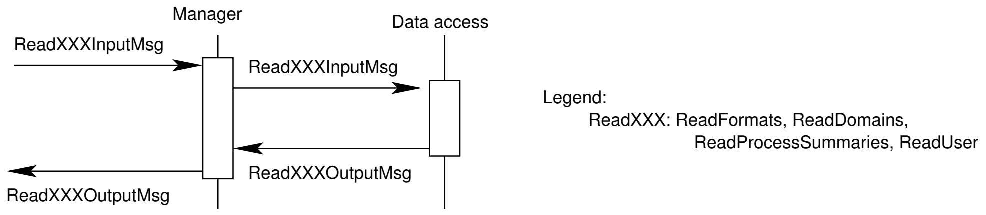


Figure 26: ReadXXX operations: choreography

WriteUser The request for executing WriteUser leads to interactions as depicted in Figure 27. At reception of the request for WriteUser, the manager forwards the DA, which in turn, stores new user information in the database and returns the result to the repository manager which returns it to the requester.

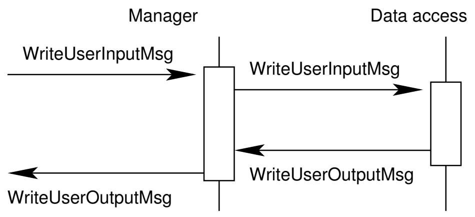


Figure 27: WriteUser operation: choreography

ImportProcess The operation ImportProcess is either enacted by the user or called by the operation CreateProcess. The request for importing a process, given in one of the native formats supported by Apromore, leads to the interactions depicted in Figure 28. When the manager receives an import request, it sends to the canoniser the message containing, among other things, the native description of the process model to be imported. The canoniser, at reception of the request from the manager, generates the canonical process and annotations for the native process and if it is successful, requests the operation StoreNativeCpf exposed by the DA, otherwise returns a result reporting the error to the repository manager. If the generation fails, the canoniser returns the result to the manager. Similarly, if the DA fails in processing the transaction, it returns an appropriate result to the canoniser which in turn forwards the result to the manager.

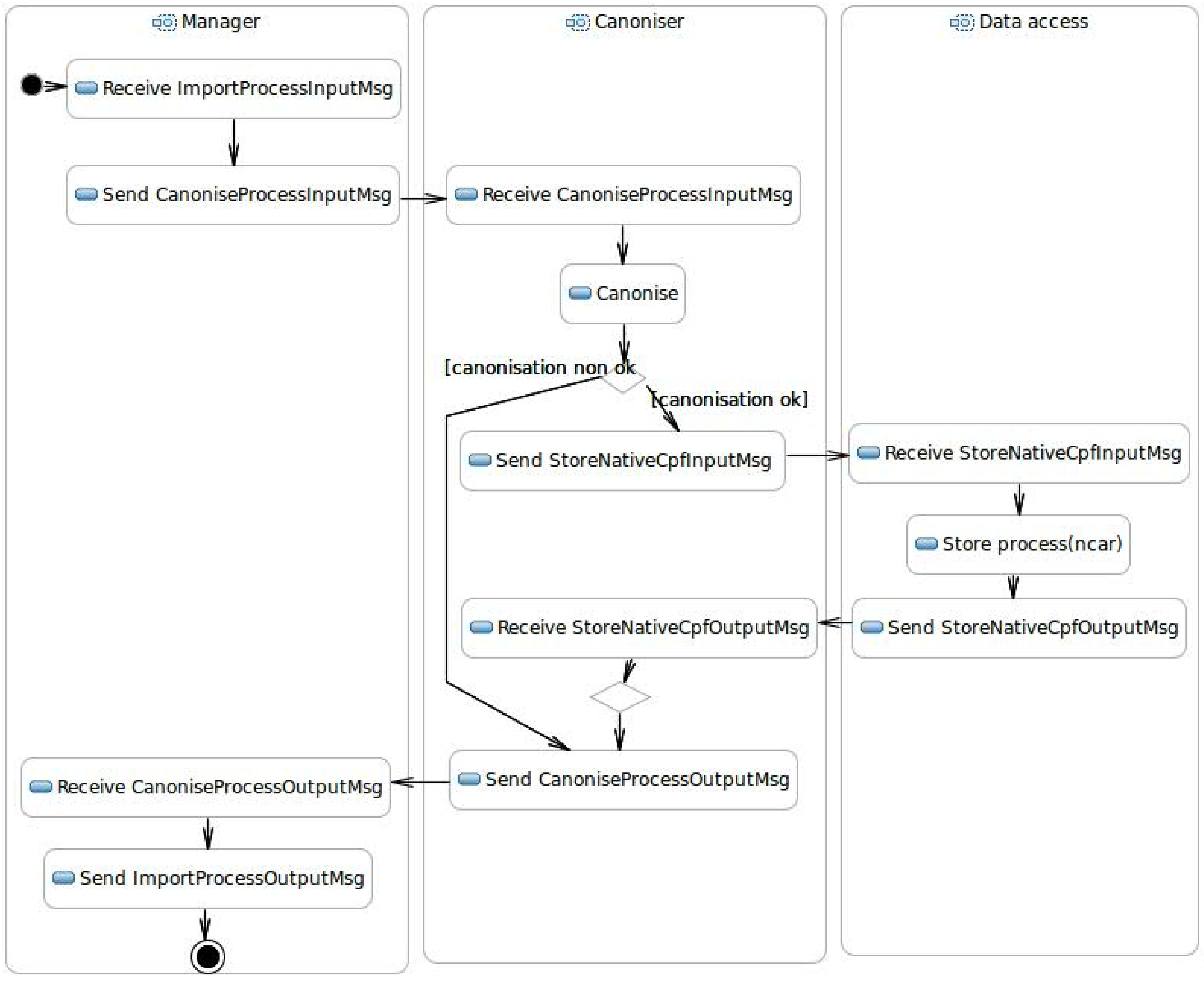

Manager
Canoniser
Data access
Receive ImportProcessInputMsg
Send CanoniseProcessInputMsg
Receive CanoniseProcessInputMsg
Canonise
[canonisation non ok
[canonisation ok]
Send StoreNativeCpfInputMsg
Receive StoreNativeCpfInputMsg
Store process(ncar)
Receive StoreNativeCpfOutputMsg
Send StoreNativeCpfOutputMsg
Receive CanoniseProcessOutputMsg
Send CanoniseProcessOutputMsg
Send ImportProcessOutputMsg

Figure 28: ImportProcess operation: choreography

ExportFormat The user is given the choice to export either the canonical format, the native format, or one of the annotation formats associated with a process version. If the requested format is canonical, annotation or native with initial annotations, then the manager forwards the message to the DA (see the first decision node in Figure 29). The requested format might not be found by the DA (for example, because the requested native format does not match the process initial native type, see the second decision node). In that case, the manager requests the DA to retrieve the process canonical format and sends it to the canoniser for generating the expected native format.

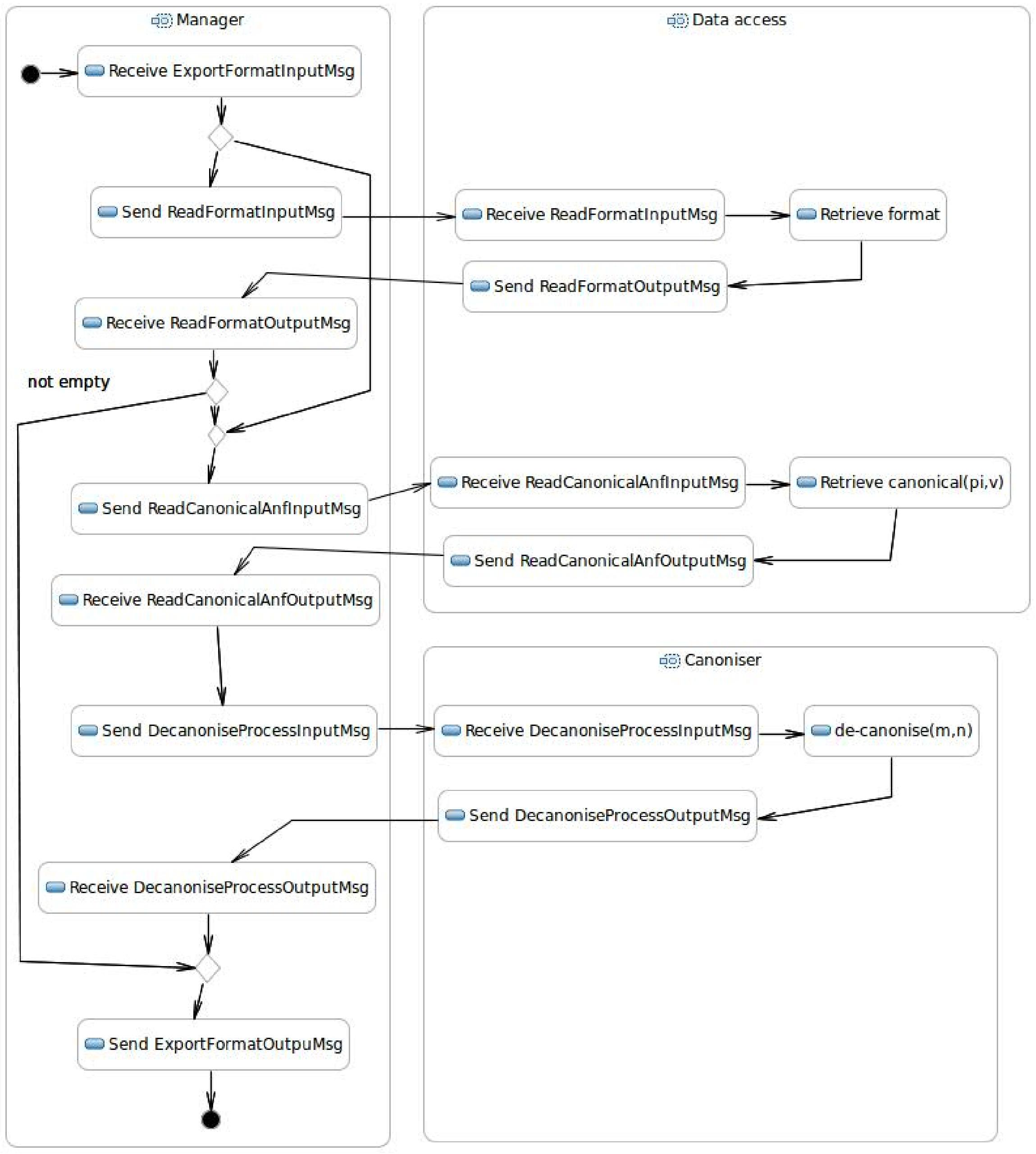

Manager
Data access
Canoniser
Receive ExportFormatInputMsg
Send ReadFormatInputMsg
Receive ReadFormatInputMsg
Retrieve format
Send ReadFormatOutputMsg
Receive ReadFormatOutputMsg
not empty
Send ReadCanonicalAnfInputMsg
Receive ReadCanonicalAnfInputMsg
Retrieve canonical(pi,v)
Send ReadCanonicalAnfOutputMsg
Receive ReadCanonicalAnfOutputMsg
Send DecanoniseProcessInputMsg
Receive DecanoniseProcessInputMsg
de-canonise(m,n)
Send DecanoniseProcessOutputMsg
Receive DecanoniseProcessOutputMsg
Send ExportFormatOutpuMsg

Figure 29: ExportFormat operation: choreography

UpdateProcess

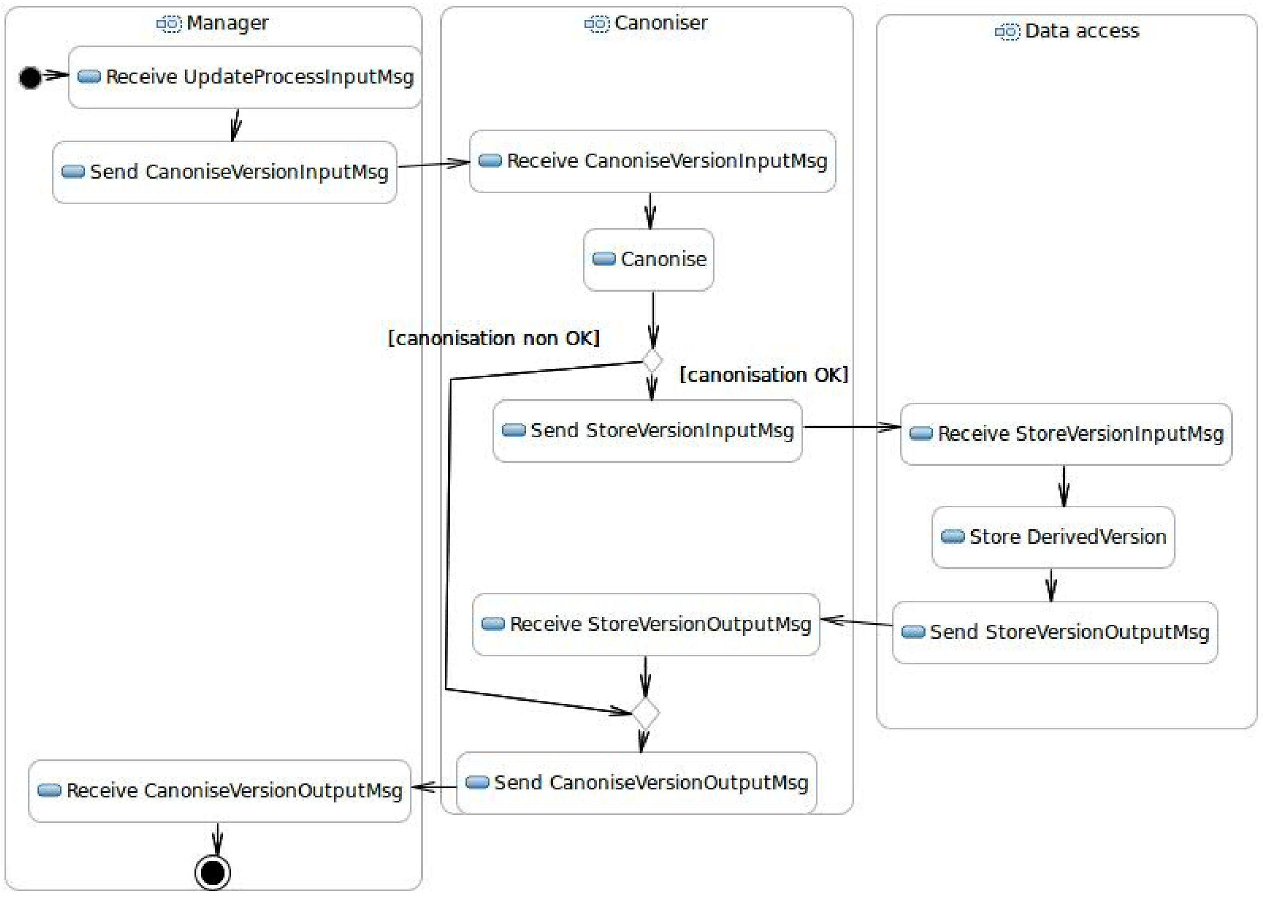

Manager
Canoniser
Data access
Receive UpdateProcessInputMsg
Send CanoniseVersionInputMsg
Receive CanoniseVersionInputMsg
Canonise
[canonisation non OK]
[canonisation OK]
Send StoreVersionInputMsg
Receive StoreVersionInputMsg
Store DerivedVersion
Receive StoreVersionOutputMsg
Send StoreVersionOutputMsg
Receive CanoniseVersionOutputMsg
Send CanoniseVersionOutputMsg

Figure 30: UpdateProcess operation: choreography

WriteAnnotation

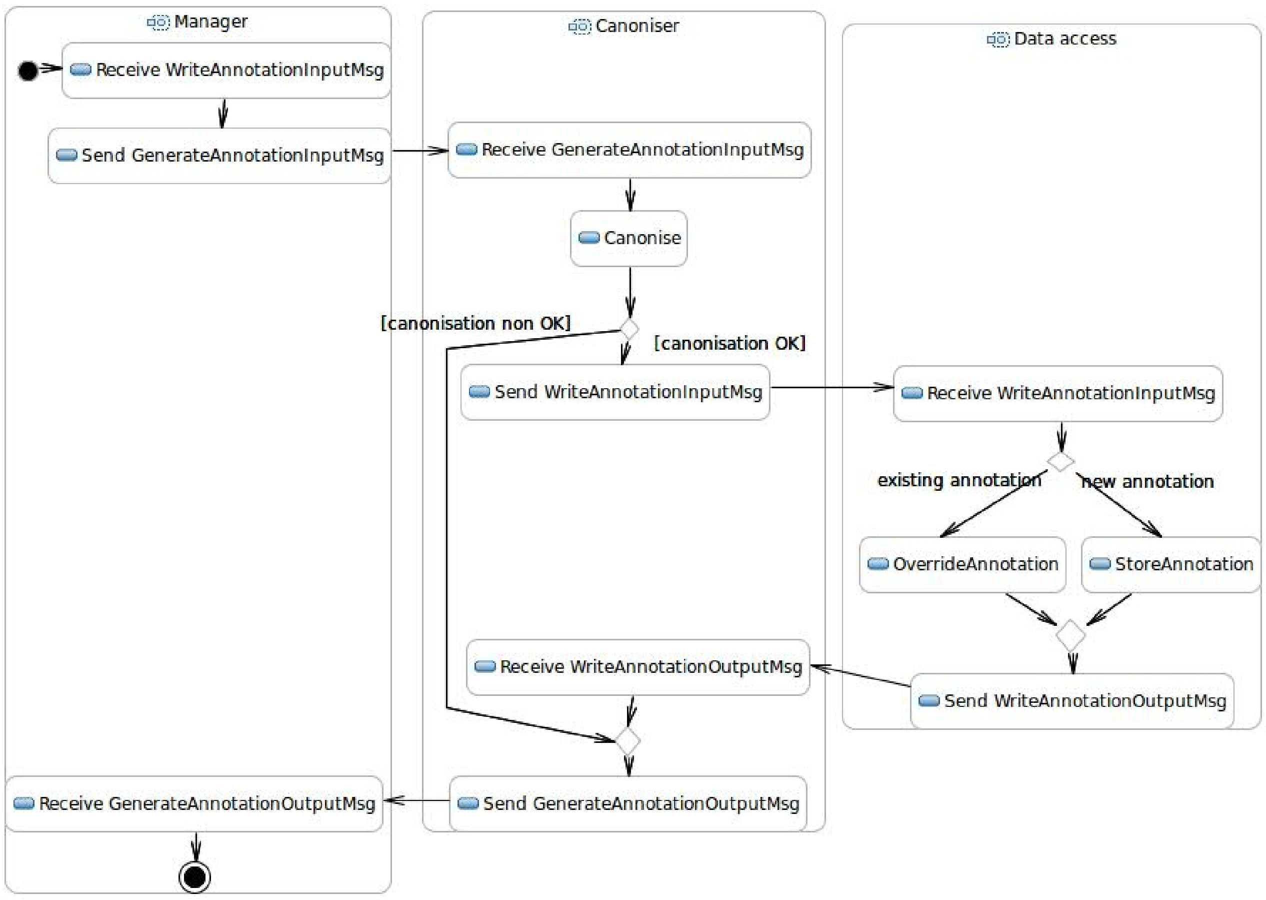

Manager
Canoniser
Data access
Receive WriteAnnotationInputMsg
Send GenerateAnnotationInputMsg
Receive GenerateAnnotationInputMsg
Canonise
[canonisation non OK]
[canonisation OK]
Send WriteAnnotationInputMsg
Receive WriteAnnotationInputMsg
existing annotation
new annotation
OverrideAnnotation
StoreAnnotation
Receive WriteAnnotationOutputMsg
Send WriteAnnotationOutputMsg
Receive GenerateAnnotationOutputMsg
Send GenerateAnnotationOutputMsg

Figure 31: WriteAnnotation operation: choreography

DeleteProcessVersion

WriteEditSession

ReadEditionSession

DeleteEditionSession

SearchForSimilarProcesses

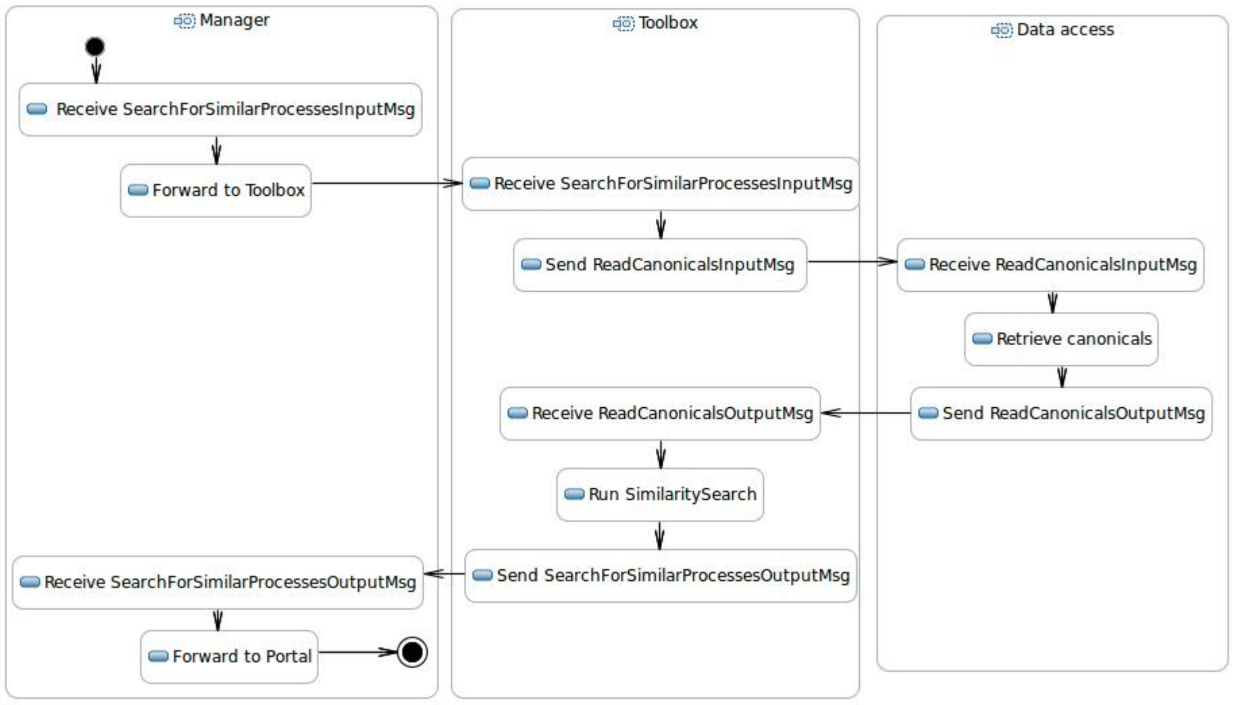

Manager
Toolbox
Data access
Receive SearchForSimilarProcessesInputMsg
Forward to Toolbox
Receive SearchForSimilarProcessesInputMsg
Send ReadCanonicalsInputMsg
Receive ReadCanonicalsInputMsg
Retrieve canonicals
Send ReadCanonicalsOutputMsg
Receive ReadCanonicalsOutputMsg
Run SimilaritySearch
Send SearchForSimilarProcessesOutputMsg
Receive SearchForSimilarProcessesOutputMsg
Forward to Portal

Figure 32: SearchForSimilarProcesses operation: choreography

MergeProcesses

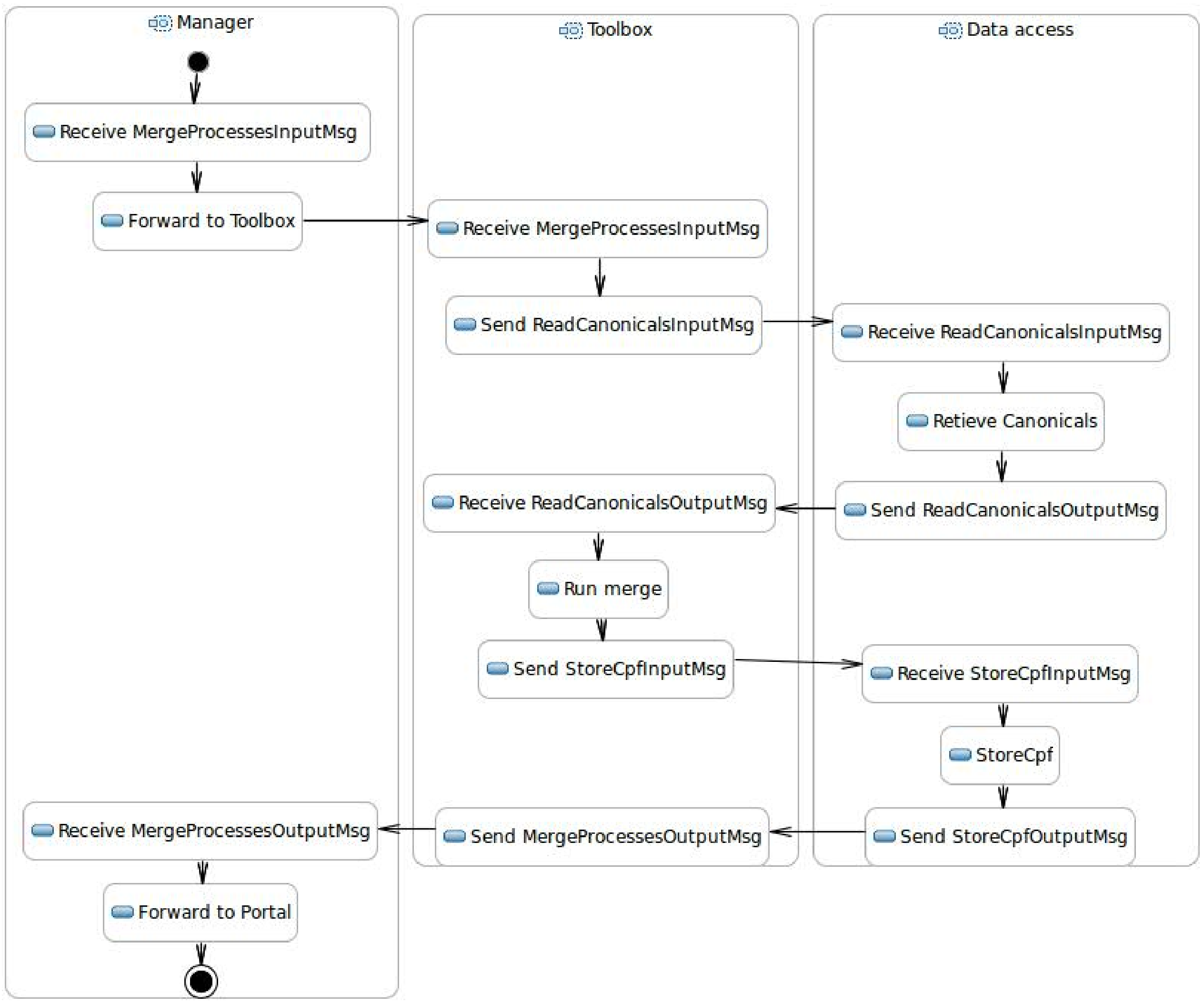

Manager
Toolbox
Data access
Receive MergeProcessesInputMsg
Forward to Toolbox
Receive MergeProcessesInputMsg
Send ReadCanonicalsInputMsg
Receive ReadCanonicalsInputMsg
Retieve Canonicals
Receive ReadCanonicalsOutputMsg
Send ReadCanonicalsOutputMsg
Run merge
Send StoreCpfInputMsg
Receive StoreCpfInputMsg
StoreCpf
Receive MergeProcessesOutputMsg
Send MergeProcessesOutputMsg
Send StoreCpfOutputMsg
Forward to Portal

Figure 33: MergeProcesses operation: choreography

5.2 Operations exposed by the portal

Edit model Edit feature allows users to edit one or more process versions at once. For each of which users are given the choice to:

- Edit the model: its structure as well as its layout. This case occurs when one annotation file is selected.
- Edit the model with no predefined layout, no annotations selected.
- Edit an annotation file only.

Figure 34 captures the choreography executed once for all process versions selected by a user. When the portal receives the event edit it associates a code with each of the selected process versions (more than one could have been selected). Then the portal send a get method to the editor with two parameters: sessionCode (whose value is the code associated with the process version to be edited) and notationsOnly (whose value is false if the user chose to edit the process model or true if she chose to edit the selected annotation only). Each time the editor receives such a get method, it requests the operation ReadNative to the portal which returns the process version details associated with the code given in the input message. If required, an annotation file is returned too. In Figure 34, the last activity to be executed is EditingWork which is detailed in Figure 35, while operations WriteEditSession is described in Section 5.1, and ReadEditSession in Section 5.1.

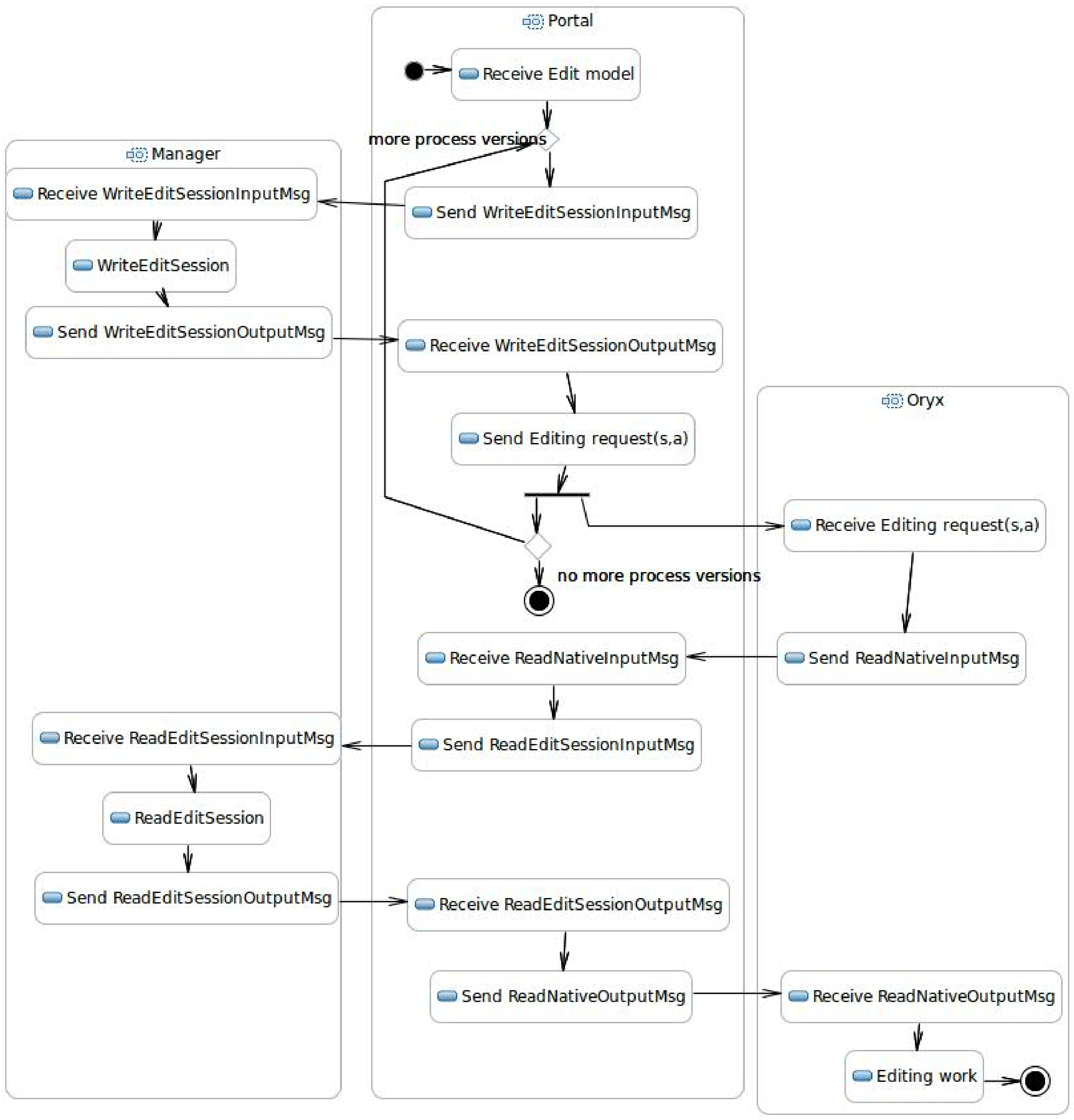

Portal
Receive Edit model
more process versions
Manager
Receive WriteEditSessionInputMsg
Send WriteEditSessionInputMsg
WriteEditSession
Send WriteEditSessionOutputMsg
Receive WriteEditSessionOutputMsg
Oryx
Send Editing request(s,a)
Receive Editing request(s,a)
no more process versions
Receive ReadNativeInputMsg
Send ReadNativeInputMsg
Receive ReadEditSessionInputMsg
Send ReadEditSessionInputMsg
ReadEditSession
Send ReadEditSessionOutputMsg
Receive ReadEditSessionOutputMsg
Send ReadNativeOutputMsg
Receive ReadNativeOutputMsg
Editing work

Figure 34: Edit model operation: choreography

Figure 35 depicts interactions engaged by Oryx editor and Portal when editing a process model and depending on user's actions (Save, Save as or Close). Save as leads to the creation of a new process as well as its first version. Save has to be chosen when one wants to override the current version or derive a new version from the current version. Users can terminate the editing work by closing the window (an alert message is then displayed warning users that modifications might be lost).

As process versions are structured as a derivation tree, some rules apply when overridding and naming versions:

- only leaf versions can be overridden
- new versions must be given names which don't exist already

When one of these rules is violated, the message WriteProcessOutMsg returned by the portal as an error code equal to -3 and a message which contains a proposal for another name for the new version.

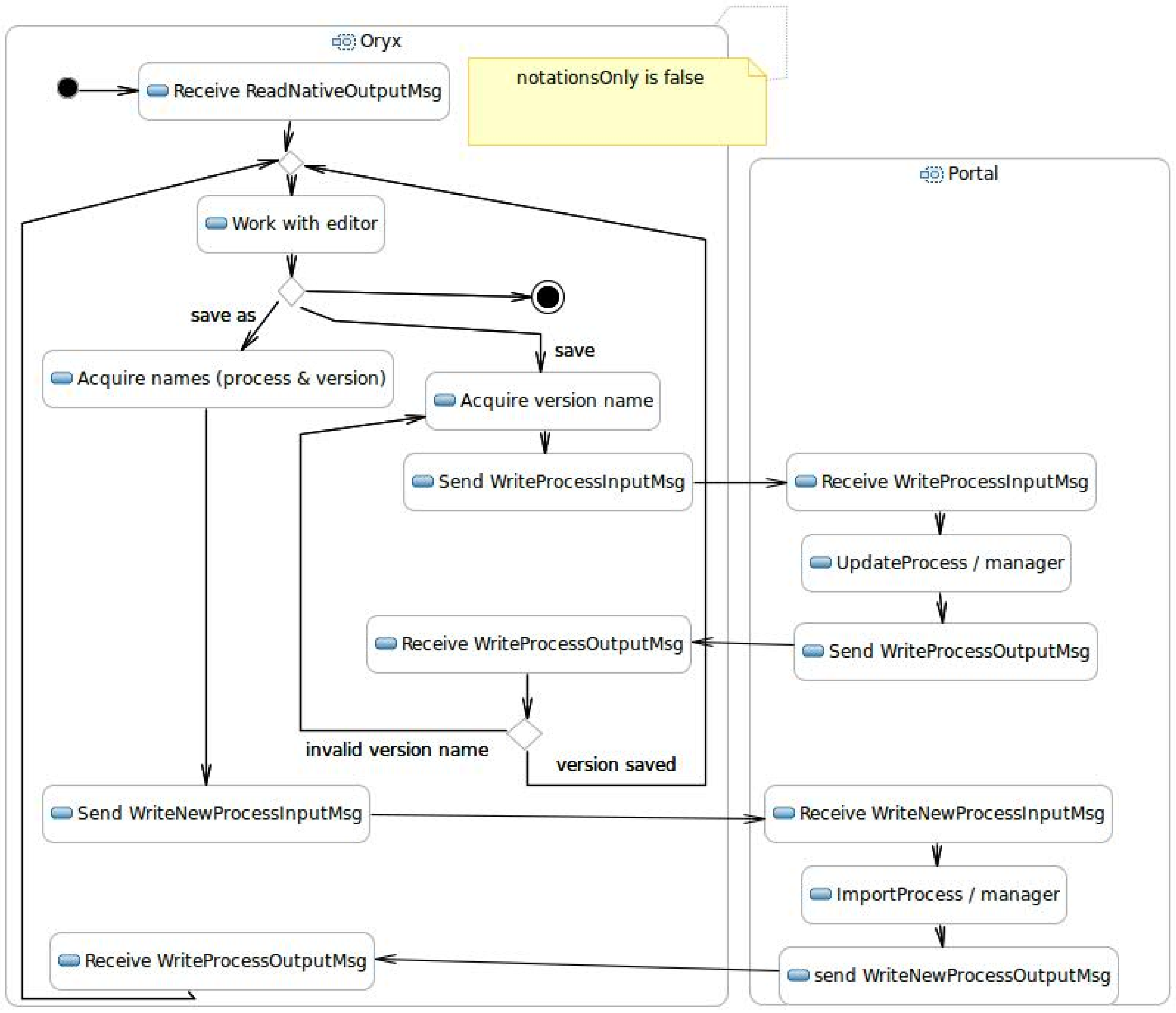

Oryx
notationsOnly is false
Receive ReadNativeOutputMsg
Work with editor
save as
save
Acquire names (process & version)
Acquire version name
Send WriteProcessInputMsg
Receive WriteProcessOutputMsg
invalid version name
version saved
Send WriteNewProcessInputMsg
Receive WriteProcessOutputMsg
Portal
Receive WriteProcessInputMsg
UpdateProcess / manager
Send WriteProcessOutputMsg
Receive WriteNewProcessInputMsg
ImportProcess / manager
send WriteNewProcessOutputMsg

Figure 35: Interactions which complete editing model: choreography

Interactions which implement editing work of annotations are similar except that when saving. As there is no support for versioning annotation files users are not asked to provide a version name (see Figure 36). In this case, Save enacts the activity Send WriteAnnotationInputMsg, while Save as enacts the one Send WriteNewAnnotationInputMsg.

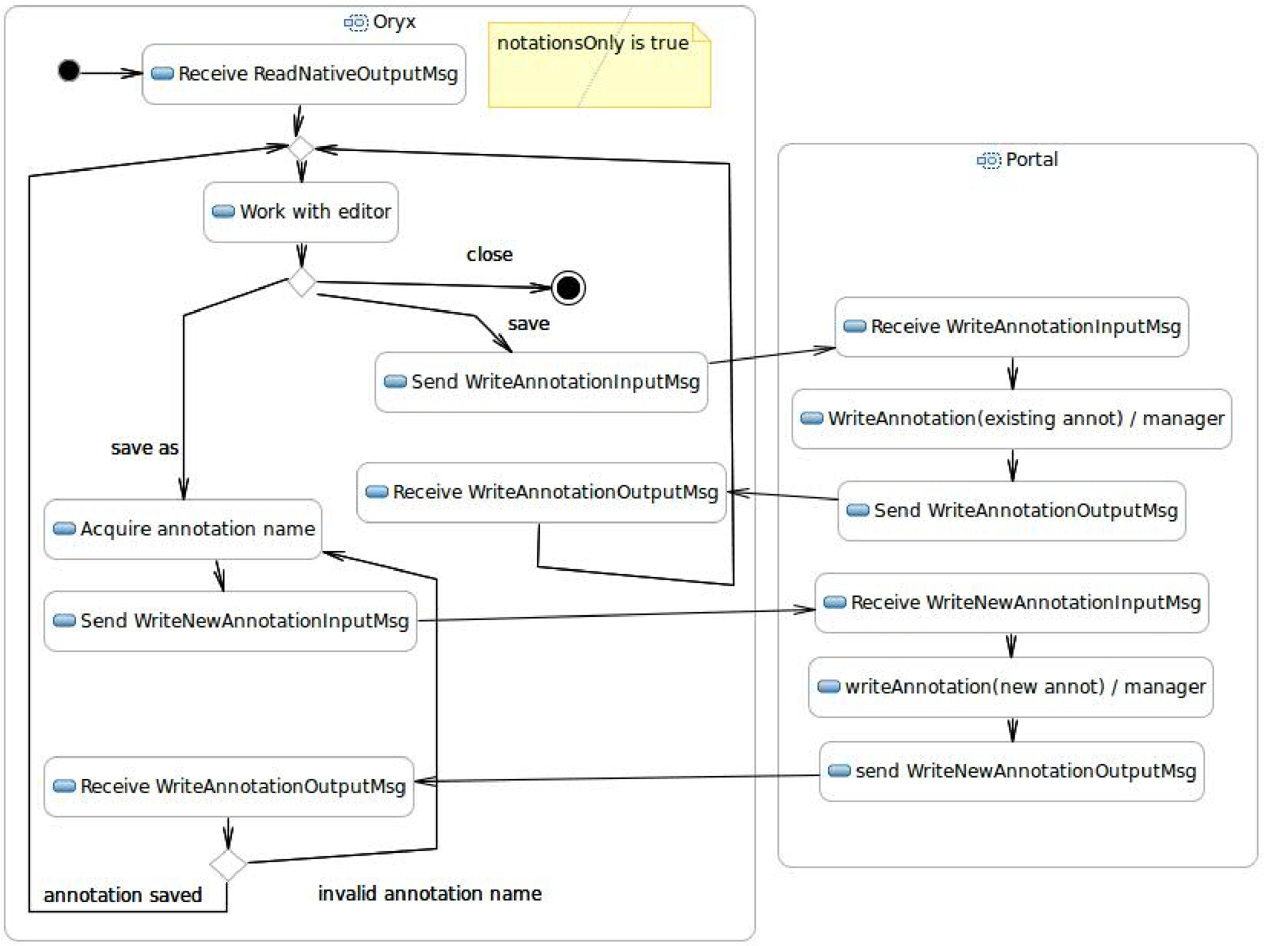

Oryx
notationsOnly is true
Receive ReadNativeOutputMsg
Work with editor
close
save
Send WriteAnnotationInputMsg
save as
Receive WriteAnnotationOutputMsg
Acquire annotation name
Send WriteNewAnnotationInputMsg
Receive WriteAnnotationOutputMsg
annotation saved
invalid annotation name
Portal
Receive WriteAnnotationInputMsg
WriteAnnotation(existing annot) / manager
Send WriteAnnotationOutputMsg
Receive WriteNewAnnotationInputMsg
writeAnnotation(new annot) / manager
send WriteNewAnnotationOutputMsg

Figure 36: Interactions which complete editing annotation: choreography

References

[1] M. La Rosa, H.-A. Reijers, W.-M.-P. van der Aalst, D. R.-M., J. Mendling, M. Dumas, and L. Garcia-Bañuelos. Apromore: an advanced process model repository. Submitted. 8

A Appendix

A.1 Search language parser

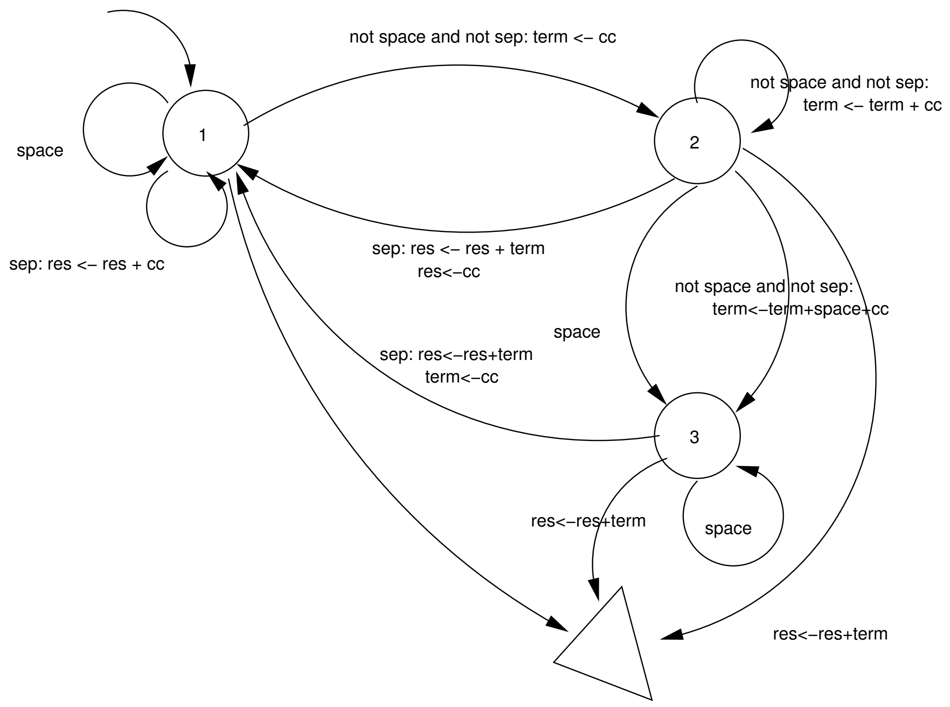


Figure 37: Search language parser: automaton