YAWL - User Manual

Version 4.1

Contents

Document Control

Arthur ter Hofstede	version 1.9	September 2008	Consolidation of previous documents, conversion to LaTeX of some of them, general cleaning and extensions (e.g. new chapter on engine).
Arthur ter Hofstede	version 1.99	October 2008	First version of chapter on resource perspective.
Michael Adams	version 2.0	July 2009	Major rewrite and additional sections to align the manual with v2.0. Public release version for YAWL 2.0.
Michael Adams	version 2.0f	September 2009	Updates for several minor Editor enhancements and addition of the Twitter Service & iGoogle Gadget.
Michael Adams	version 2.1b	June 2010	Updates for version 2.1.
Marcello La Rosa	version 2.1c	February 2011	Update for C-YAWL.
Michael Adams	version 2.2	August 2011	Revision of entire manual and additions for YAWL 2.2.
Michael Adams	version 2.3	April 2012	Updates for version 2.3.
Michael Adams	version 3.0	July 2014	Major rewrites and updates for version 3.0.
Michael Adams	version 3.0a	September 2014	Updated installation chapter.
Michael Adams	version 4.0	April 2016	Updates and additions for version 4.0.
Michael Adams	version 4.0a	April 2016	Minor typos.
Michael Adams	version 4.1	May 2016	Updates for version 4.1.

Feedback?

Any feedback regarding this manual is very much appreciated. If you find there is a topic that is missing or may not have been clearly explained, please send your feedback to `yawlmanual@gmail.com`. All

suggested improvements will be considered for incorporation into future versions of the manual.

Sources

The first version of this document (1.9) combined the following documents:

1. A "New Features" document produced by Lachlan Aldred.
2. A "Getting Started with YAWL" document by Lindsay Bradford and Marlon Dumas. This formed the basis for Chapter 3.
3. A "Editor 2.0 User Manual" document of which first versions were produced by Sean Kneipp and subsequent versions by Lindsay Bradford, Jessica Prestedge, Marcello La Rosa, and Michael Adams. This document was the original editor chapter, which has since been heavily revised and rewritten.
4. A "Data Manipulation in YAWL" document by Chun Ouyang (with some of the figures on the use of XML technology in YAWL taken from a presentation by Lachlan Aldred). This forms the basis for Chapter 5.
5. A "YAWL Engine User Manual" (Beta 8 release) document created by Sean Kneipp with subsequent additions/updates by Guy Redding, Lachlan Aldred and Michael Adams. This document provided inspiration for Chapter 6.
6. A "The Worklet Custom Service for YAWL - Installation and User Manual" document created and later revised by Michael Adams. Lachlan Aldred merged the installation manual. This document became the original Chapter 8.
7. An "Installation Manual" (Engine Beta 8.2 - Editor 1.5) first created by Sean Kneipp and with subsequent changes/corrections/extensions by Guy Redding, Lachlan Aldred, Petia Wohed, Michael Adams, Moe Wynn, and Marcello La Rosa. This formed the basis for Chapter 2. Its introduction formed the basis for Chapter 1.

Chapter 1

Introduction

This chapter provides a brief background introduction to YAWL and the YAWL Foundation.

1.1 What is YAWL?

Based on a rigourous analysis of existing workflow management systems and workflow languages, a new workflow language called YAWL (Yet Another Workflow Language) was developed by Wil van der Aalst (Eindhoven University of Technology, the Netherlands) and Arthur ter Hofstede (Queensland University of Technology, Australia) in 2002. This language was based on the one hand on Petri nets, a well-established concurrency theory with a graphical representation, and on the other hand on the well-known Workflow Patterns (www.workflowpatterns.com). The Workflow Patterns form a generally accepted benchmark for the suitability of a process specification language. Petri nets can capture quite a few of the identified control-flow patterns, but they lack support for the multiple instance patterns, the cancellation patterns and the generalised OR-join. YAWL therefore extends Petri nets with dedicated constructs to deal with these patterns.

YAWL offers the following distinctive features:

- YAWL offers comprehensive support for the control-flow patterns. It is the most powerful process specification language for capturing control-flow dependencies.
- The data perspective in YAWL is captured through the use of XML Schema, XPath and XQuery.
- YAWL offers comprehensive support for the resource patterns. It is the most powerful process specification language for capturing resourcing requirements.
- YAWL has a proper formal foundation. This makes its specifications unambiguous and automated verification becomes possible (YAWL offers two distinct approaches to verification, one based on Reset nets, the other based on transition invariants through the WofYAWL editor plug-in).
- YAWL has been developed independent from any commercial interests. It simply aims to be the most powerful language for process specification.
- For its expressiveness, YAWL offers relatively few constructs (compare this e.g. to BPMN!).
- YAWL offers unique support for exceptional handling, both those that were and those that were not anticipated at design time.
- YAWL offers unique support for dynamic workflow through the Worklets approach. Workflows can thus evolve over time to meet new and changing requirements.
- YAWL aims to be straightforward to deploy. It offers a number of automatic installers and an intuitive graphical design environment.

- YAWL's architecture is Service-oriented and hence one can replace existing components with one's own or extend the environment with newly developed components.
- The YAWL environments supports the automated generation of forms. This is particularly useful for rapid prototyping purposes.
- Tasks in YAWL can be mapped to human participants, Web Services, external applications or to Java classes.
- Through the C-YAWL approach a theory has been developed for the configuration of YAWL models. For more information on process configuration visit www.processconfiguration.com.
- Simulation support is offered through a link with the ProM (www.processmining.org) environment. Through this environment it is also possible to conduct post-execution analysis of YAWL processes (e.g. in order to identify bottlenecks).

1.2 Obtaining the Latest Version of YAWL

As new versions of the YAWL Environment are released to the public, they will be available for download at the YAWL Github repository (github.com/yawlfoundation/yawl/releases). From this site it is also possible to access the source code of all components for development purposes.

1.3 The YAWL Foundation

For up-to-the-minute information on any aspect of the YAWL Initiative, visit the YAWL Foundation Homepage (yawlfoundation.org). The site also hosts a 'Latest Builds' pages (under the 'Resources' menu) where the most recent builds of various components, incorporating bug fixes and improvements, can be downloaded between major releases. The YAWL Foundation is a non-profit organisation that acts as custodian of all intellectual property (IP) related to YAWL and its support environment.

1.4 Documentation

Apart from this user manual, there is a technical manual on YAWL and a number of case studies. These studies provide detailed examples that you may wish to consult in order to obtain a deeper understanding of the application of YAWL.

This manual does not really cover the control-flow *concepts* of YAWL in detail. One reason for this is that there are quite a few papers out there that do provide this information. We refer the reader to e.g. [10] for a justification of the extensions of Petri nets introduced for YAWL on the basis of the original control-flow patterns. The main paper on YAWL, from a language point of view, is [11]. In this paper you find a formalisation of the control-flow concepts of YAWL. More recently, a CPN formalisation of newYAWL (control-flow, data and resource perspectives) was presented in [28]. For a formalisation of the OR-join, a complex synchronisation concept in YAWL, we refer to [34]. This definition supersedes the definition provided in [11].

As mentioned above, YAWL extends Petri nets. There are a number of general introductions to Petri nets in the literature. We refer the interested reader to [22, 20].

Wil van der Aalst has written much about the application of Petri nets to workflow, see e.g. [2]. The subclass of Petri nets introduced by him, Workflow-nets, is a predecessor of YAWL. The textbook that he wrote together with Kees van Hee is highly recommended reading [9].

A recent textbook on Business Process Management (BPM), which covers the original control-flow patterns and also YAWL, was written by Mathias Weske [31]. This textbook also covers other approaches, such as the modelling standard BPMN (note that the BPMN2YAWL tool can convert these specifications to YAWL).

On the YAWL web site (yawlfoundation.org) it can be seen how the original control-flow patterns can be realised in YAWL (follow the link on Resources and then click 'patterns'). For control-flow patterns in newYAWL the reader can consult appendix A.1 of Nick Russell's PhD thesis [28].

If you would like to know more about how verification of YAWL specifications really works, we refer you to [30] and to [33]. This work forms the theoretical basis of how the verification mechanisms are realised in the YAWL editor.

In-depth discussion of YAWL's exception handling framework from a conceptual point of view can be found in [28, 24] and from an implementation aspect in [12, 13]. YAWL's worklet approach to dealing with on-the-fly changes to workflows is discussed in [12, 14].

YAWL has a close link to the Process Mining environment ProM [8], www.processmining.org. This link is for example exploited in [23] to provide simulation support for YAWL. There exists support for exporting YAWL logs to ProM which can subsequently be analysed by one of the many mining plug-ins available in this environment.

Alternative ways of presenting work lists have been addressed in [15]. In this framework users can choose a map (not just a geographical map, but also e.g. a timeline or a YAWL specification) and work items can be positioned on this map and be shown in a colour that reflects their level of urgency (a context-specific notion which can be defined for the user). It is expected that this work becomes part of the YAWL distribution in the near future.

Finally, a textbook on YAWL, called *Modern Business Process Automation: YAWL and its Support Environment* has been published by Springer (2010; ISBN: 978-3-642-03120-5).

Chapter 2

Installation

The YAWL System (also referred to as the YAWL Environment) comprises a number of web servlets and a java-based Editor desktop application. It requires a *Servlet Container* to host the servlets and a back-end database system for process data storage and archiving. Individual YAWL components may be installed manually (see Section 2.4), but there are also a number of automatic YAWL installation packages that install all the required components and allow you to be up and running with YAWL quickly and easily:

- **YAWL** has installers available for Windows, Linux and Mac OSX platforms. You should choose an installer if YAWL is to be used within a single platform environment (i.e. all components are installed on the same machine). A YAWL installer is also the right choice if you intend to learn about or to experiment with YAWL. The resulting pre-configured installation has the complete YAWL functionality and contains the same YAWL components as a *YAWL Enterprise* installation.
- **YAWL Enterprise** is how YAWL is described when installed manually on production server(s). Required third-party components, such as Apache Tomcat and PostgreSQL, are installed separately, possibly on different platforms, so that YAWL can be used as a long-running server for production purposes. See Section 2.4 for instructions on how to install an enterprise version of YAWL for multi-user production environments.

The complete YAWL environment is installed whichever installer is chosen.

Official and stable versions of the YAWL installation files are found on the Github YAWL project repository page: `github.com/yawlfoundation/yawl/releases`.

2.1 Requirements

All installers, and YAWL itself, require the Java SE Runtime Environment (JRE), 1.6 or greater (`java.com/en/download/`).

YAWL 4.1 has been successfully run on the following operating systems:

- **Windows**: XP and later;
- **Linux**: Ubuntu (9.10 and later), sidux, Debian (Etch), and other variants;
- **Mac OS X**: 10.4 and later.

2.2 Installing YAWL

The release package provides an installer for each of the operating systems Windows, Linux and Mac OSX. Their filenames are similar to those shown in Table 2.1.

	Windows	YAWL-**windows**-4.1-installer.exe
	Linux	YAWL-**linux**-4.1-installer.bin
	MacOSX	YAWL-**osx**-4.1-installer.app.zip

Table 2.1: YAWL has automated installers for three different operating systems

Preparation YAWL can be installed from an ordinary user account without administration rights as long as Java version 1.6 or greater is installed on the system.

To start the installation, simply double-click on the installation file.

The installation file needs to have read and executable permissions. Those can be assigned by right clicking the file, selecting *Properties* and setting the appropriate permissions. Alternatively the following shell command can be applied:

```
user@host:/tmp$ chmod 755 YAWL-4.1-linux-installer.bin
```

Afterwards, start the installation by double-clicking on the downloaded file or by invoking it from the command line:

```
user@host:/tmp$ ./YAWL-4.1-linux-installer.bin
```

Installation Start You will first be greeted by a Welcome screen. Pressing *Next* will take you to the License Agreement page, shown in figure 2.1. You need to accept the agreement to continue the installation.

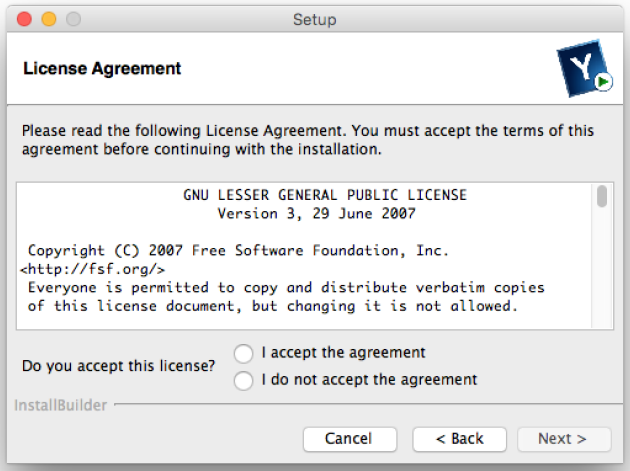


Figure 2.1: License Agreement

The next step is to select the installation directory (figure 2.2). The default location is your home directory. The installer will inform you if you're allowed to install YAWL in the selected folder. In case of Windows or Linux, folders like *C:\Program Files* or */usr/local/* are only writeable by users with administrative rights. Install YAWL into your home directory if you don't have administrative rights (this is *particularly* important for users of Windows 7 and later).

After clicking *Next*, the installer will look for a valid Java installation. The dialog box in figure 2.3 will only appear if Java has been detected on your system.

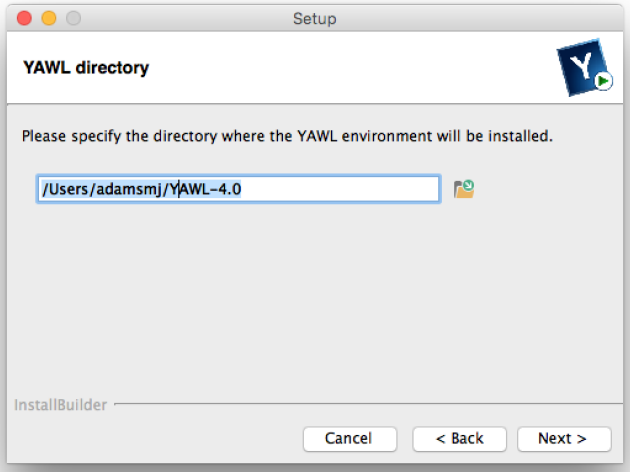


Figure 2.2: Select Installation Directory

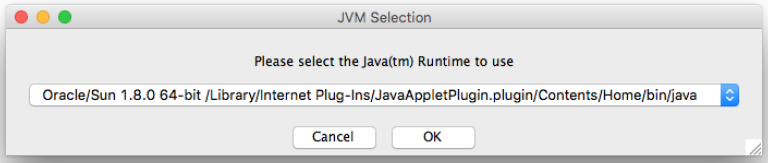


Figure 2.3: Select Java Version

The installer will show a message then exit if no Java version greater or equal 1.6 is found. If this happens, ensure the latest Java runtime[1] is installed before re-running the YAWL installer.

The YAWL installer checks the usual installation paths, like */usr/* or */usr/local/*. If you installed Java on your system but the YAWL installer is not able to find it, create the following link:

```
user@host:$ ln -s /path/to/your/java/base/dir /usr/local/java
```

Keep in mind that you need administration rights to generate the link. Afterwards, start the YAWL installation again.

You are now ready to proceed with the installation. When the installation completes, you will see the *Installation Completed* page. If you found a problem and could not complete the installation, please post the problem to the YAWL Foundation forum[2].

Database YAWL comes preconfigured with a H2[3] database.

[1] java.com/en/download/
[2] yawlfoundation.org/forum
[3] www.h2database.com

2.3 YAWL Control Panel

The Control Panel is a java desktop application that provides easy access to starting, stopping and updating the installed YAWL environment, and to start the editor, view this manual and so on. On Windows, it can be accessed via the Start Menu (in the YAWL-4.1 folder), or for all systems it can be found as *YawlControlPanel-4.1.jar* in the 'controlpanel' sub-folder of the YAWL-4.1 installation folder.

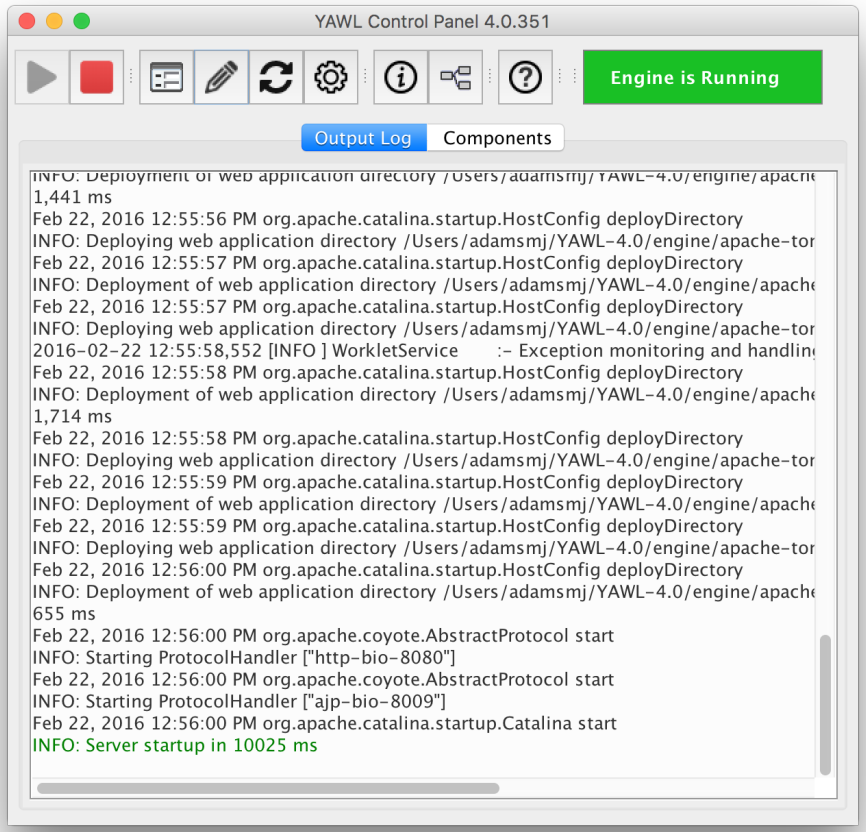


Figure 2.4: YAWL Control Panel

The Control Panel consists of a tool bar, a a status panel that reflects the current status of the YAWL Engine (i.e. Starting, Running, Stopping, Stopped), and an Output Log / Installed Components panel. The toobar buttons, as shown in Figure 2.4, are (left-to-right):

- **Start** Click to start the YAWL environment. Note that it may take 15 to 60 seconds for the start process to complete. When YAWL is starting or running, this button will be disabled.
- **Stop** Click to stop the YAWL environment. When YAWL is stopping or stopped, this button will be disabled. When starting and stopping, the status panel will update to show the current status.
- **Logon** Click the Logon button to open the YAWL logon page in your default browser. Note that this button is enabled only while YAWL is running.
- **Editor** Click the Editor button to start the YAWL process editor (see Chapter 4).

- **Updates** Click the Updates button to check for updates to existing components, or to install/uninstall components (see the Updates section below).
- **Preferences** Click the Preferences button to display the Preferences dialog (Figure 2.5).
- **Information** Click to open the user manual in your default browser.
- **Examples** Click to open a page in the browser where you can download the various example specifications and other files described in this manual.
- **About** Click this button to show the 'about' box for the control panel app. Click anywhere on the box displayed to close it.

The Output Log panel shows the real time output of the YAWL execution log.

The Control Panel can be closed at any time by simply closing its window. Closing the Control Panel will *not* affect the current YAWL execution status, unless the 'Stop YAWL Engine when Control Panel exits' option is selected in the Preferences dialog.

2.3.1 Preferences

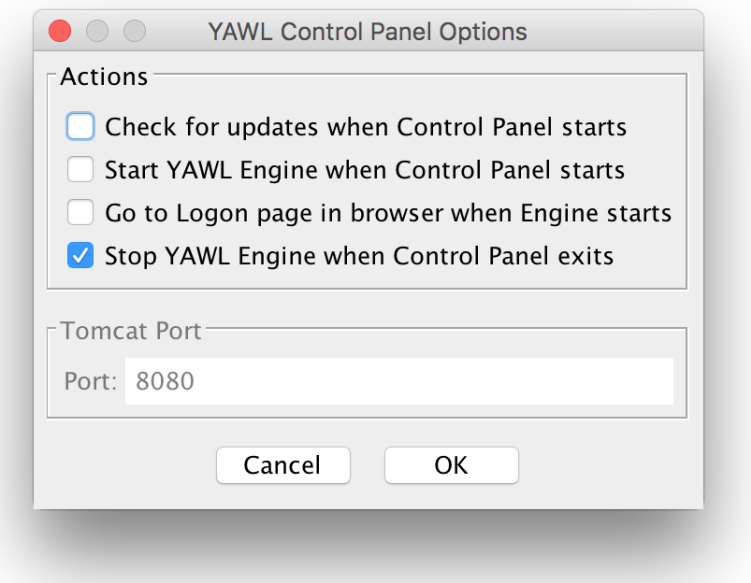


Figure 2.5: Control Panel Preferences dialog

The four actions in the Preferences dialog are self explanatory and may be chosen as desired.

The *Tomcat Port* setting in most cases need not be changed. It is provided for those cases where you may already have an application running on port 8080, in which case you need to set YAWL to run on another port. Note that the port value can only be changed when the YAWL environment is stopped.

Warning: Do not attempt to alter the port for YAWL via any other method, as a number of configuration files are required to be updated as a result of the port change.

2.3.2 Updates

The Components Panel (Figure 2.6) allows you to update, install and uninstall YAWL components. It is displayed when the Components tab is clicked, or the Updates button is clicked, or when you have selected

the 'Check for Updates when Control Panel starts' option in the Preferences dialog and there are updates available when you start the Control Panel.

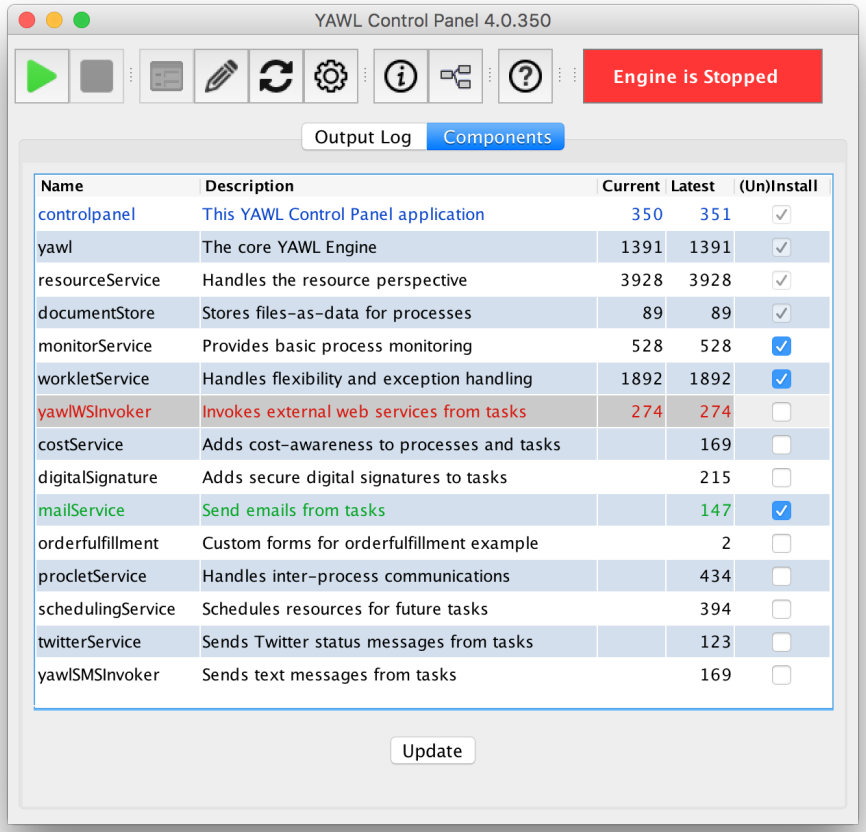


Figure 2.6: Components Panel

The Components Panel lists all available YAWL components, and for each displays a short description[4], its current build version (as installed) and the latest available build version (if the Updates button is clicked or the 'Check for Updates' preference is selected). The column on the right will show a tick for each component currently installed.

- For each available update, that is where there is a currently installed component with a build version less than the latest available version, its row be coloured blue (e.g. 'controlpanel' in Figure 2.6).
- to install a component not currently installed, click on its (Un)Install checkbox to select it. Its row will be coloured green to denote the addition (e.g. 'mailService' in Figure 2.6).
- to uninstall a component that is currently installed, click on its (Un)Install checkbox to unselect it. Its row will be coloured red to denote the deletion (e.g. 'yawlWSInvoker' in Figure 2.6).
- the first four rows (i.e. 'controlpanel', 'yawl', 'resourceService' and 'documentStore') are mandatory components that cannot be uninstalled (but may be updated).

[4]see Section 2.4.3 for more details of the purpose and use of each component

- Absolutely no information is sent from your YAWL installation to the updates server. Rather, the Control Panel downloads a file containing the latest build data for all YAWL components, then compares that to what is currently installed.

When the Update button is clicked in the Updates dialog, all components denoted blue will be updated, all denoted green will be installed, and all denoted red will be uninstalled, via this process:

1. Files for updates and installs will be downloaded. An error message will appear if the download server is unavailable. You may try again later.
2. Once the file downloading has completed, each file is verified for correctness. An error message will appear if any file fails verification. You may try again later.
3. Once verification has completed successfully for each file downloaded (if any), the Engine will be stopped if it is currently running, so that the updates, installs and uninstalls can be applied.
4. The selected changes will be applied.
5. The Engine will be (re)started.
6. Once the restart has completed, the Components Panel will be refreshed to show the results of the process.

Finally, if the Control Panel itself has been updated, it will automatically restart to the latest version.

2.3.3 Command Line Interface

The YAWL Control Panel can also be operated in 'headless' mode, via the command line. This means that you can use the Control Panel to manage a YAWL installation remotely.

The syntax is (from the *controlpanel* directory):

```
java -jar YawlControlPanel-4.1.jar -option
```

where "-option" is one of:

- **-start** Start the YAWL engine
- **-stop** Stop the YAWL engine
- **-status** Check whether the YAWL engine is running or stopped
- **-update** Update all installed components (with pending updates)
- **-versions** List installed and available components and their versions
- **-add [component]** Add the named component (and perform updates)
- **-remove [component]** Remove the named component

2.4 Manual Installation (YAWL Enterprise)

If you already have installed a servlet container, such as Apache Tomcat (version 7 or greater) and/or a preferred database, such as PostgreSQL, MySQL or Oracle (version 8.1 or greater), you may prefer to install YAWL components manually. Manual installation is also required if you want a multi-user, multi-platform, production-level installation of YAWL. This section details how to install YAWL 4.1 on a component-by-component basis.

ASIDE: To install latest builds (i.e. components that have been updated since the last formal release), please follow the instructions on the YAWL Foundation's 'Latest Builds' web page [5].

[5] yawlfoundation.org/pages/resources/latestbuilds.html

2.4.1 Installing Tomcat

YAWL mainly consists of a number of servlets, and so needs a servlet container installed to host them. We recommend Apache Tomcat be used – it is free, stable and fully tested as a YAWL host over a number of years. YAWL requires Tomcat version 7 or later.

The first step is to download the latest Tomcat version from tomcat.apache.org. The simplest way to install Tomcat in a Windows environment is to use the purpose built Windows installer provided.

For installation on Mac OSX:

1. Download the latest Tomcat binary from tomcat.apache.org.
2. Unpack the downloaded file to a destination directory of your choice.
3. If necessary, change the owner of the tomcat hierarchy to your user:

   ```
   sudo chown -R <your_username> <your root Tomcat dir>
   ```

4. Make the Tomcat scripts executable:

   ```
   sudo chmod +x <your root Tomcat dir>/bin/*.sh
   ```

Linux installation is similarly straightforward. A simple set of instructions (for ubuntu) can be found at www.howtogeek.com/howto/linux/installing-tomcat-6-on-ubuntu/

In all cases, an environment variable called "CATALINA_HOME" needs to be added, which points to the tomcat install path.

Once Tomcat is installed, two small configuration changes are required to files found in the *<your_tomcat_dir>/conf* directory:

1. **context.xml**: Locate the commented line containing *<Manager pathname="" />* and uncomment it.
2. **server.xml**: Locate the entry that begins *<Connector port="8080" protocol="HTTP/1.1"* and modify it so that it looks like this (i.e. the fifth attribute, *URIEncoding="UTF-8"*, has been added):

   ```
   <Connector port="8080"
      protocol="HTTP/1.1"
      connectionTimeout="20000"
      redirectPort="8443"
      URIEncoding="UTF-8"/>
   ```

2.4.2 Installing PostgreSQL

As a default, YAWL is configured to use PostgreSQL for database support, and this section describes how to manually install and configure PostgreSQL for YAWL (however, it is easy to configure YAWL for other database back-ends – see Section 2.4.4 for details).

Download the latest version of PostgreSQL from www.postgresql.org/download/ – there is a one-click installer available for each operating system. The following is a walkthrough for a Windows installation, but it is a similar process for other platforms. Except where otherwise mentioned, simply accept the default setting for each setup screen.

For the *Service Configuration* screen, choose the *Install as a Service* option (see Figure 2.7). Leave the account name as 'postgres' and enter any password of at least 6 characters. This will be the account the PostgreSQL service uses to run and allow connections to the database. **Note:** The *Install as a Service* option is only available on Windows systems.

On the next screen, *Initialise Database Cluster* (Figure 2.8), leave all values as they are, but enter 'yawl' (no quotes, all lower case) as the password. This is the user account that YAWL uses to connect to the database

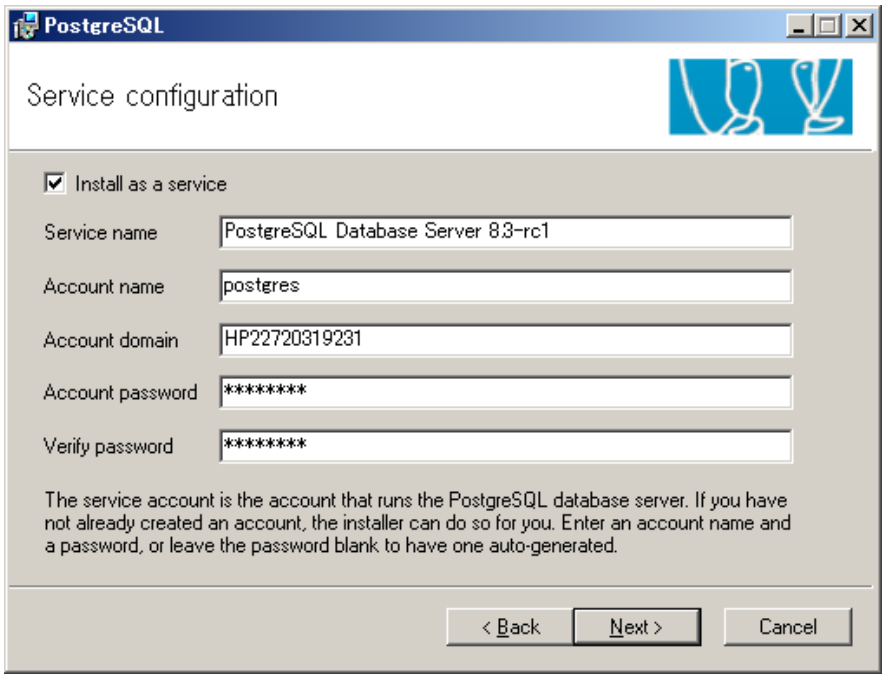


Figure 2.7: Choose *Install as a Service* and enter any password

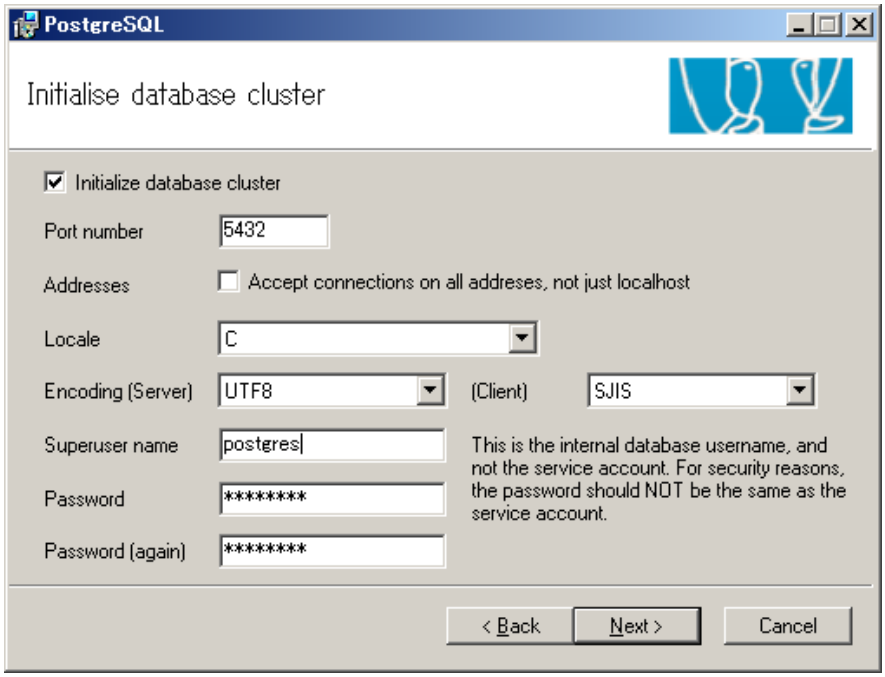


Figure 2.8: Keep *postgres* as the Superuser name, and enter *yawl* as the password

(but see Section 2.4.4 for details on how to modify the password YAWL uses). Leave the settings for all other screens at their defaults and click through to completion.

Next, run the administration tool *pgAdmin*[6], which was installed along with the PostgreSQL installation. In the *Object Browser* panel on the left (cf. Figure 2.9), double-click on the *Postgres* server, and, when prompted,

[6] www.pgadmin.org

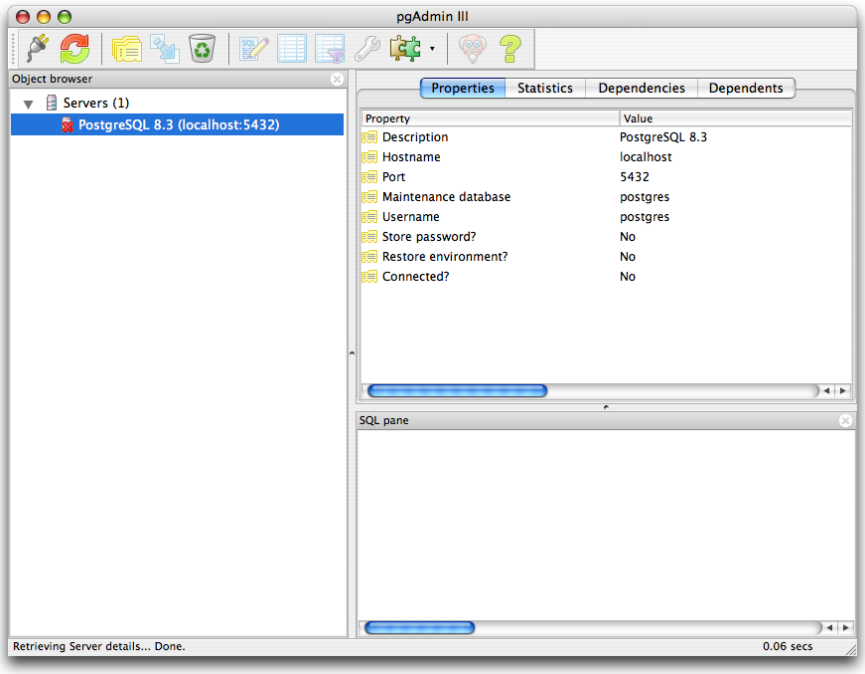


Figure 2.9: The pgAdmin tool, Object Browser panel on the left

enter 'yawl' as the password (you should only be asked for this password the first time you connect). Then, right-click on *Databases* (under Postgres) and choose *New Database* from the popup menu. In the next dialog, enter 'yawl' as the database name, leave all other fields as they are, and click OK.

That completes the installation of PostgreSQL and the admin tool pgAdmin, and the creation of the YAWL database. When YAWL is started, it will automatically create the required database tables as part of its startup process.

2.4.3 Installing YAWL Services

All the necessary YAWL files can be downloaded from the YAWL Engine download page on Github (`github.com/yawlfoundation/yawl/releases`). The latest release folder will contain these files:

- **YAWL_CoreWebServices_4.1.zip**: The core set of YAWL services, comprising the core Engine, and the Resource, Worklet, Web Service Invoker, Mail, DocumentStore and Monitor Services.
- **YAWL_OptionalWebServices_4.1.zip**: Extra services that you may find useful for particular purposes.
- **YAWL_LibraryJars_4.1.zip**: Two library jars that contain the YAWL class files only (i.e. without any third party libraries) and so can be used to embed references to the YAWL classes when developing applications. The file *yawl-lib-4.1.jar* contains every Engine and Service class file in the YAWL environment, while the file *YResourceServiceClient.jar* contains the minimum set of YAWL classes required when developing applications using the Resource Service's APIs, or when developing custom forms for your processes (see Chapter 4, Section 4.15).
- **YAWL_Standalone_4.1.jar** A basic, standalone desktop version of the Engine.
- **YAWL_SourceAndTestCode_4.1.zip** The source code for the environment.

Download the *CoreWebServices* file (and, if any of the optional services are required, the *OptionalWebServices* file). Unzip the contents of the file to *<your_tomcat_dir> /webapps* directory. When Tomcat is (re)started, it

will automatically unpack each war file into its own directory under *webapps* (cf. Figure 2.10). There are six core web services:

1. **yawl.war**: the core workflow engine.
2. **resourceService.war**: handles the allocation of tasks to resources; contains the default worklist handler; generates dynamic forms; manages codelets; manages organisational data.
3. **workletService.war**: handles dynamic flexibility and exception handling.
4. **yawlWSInvoker.war**: allocates tasks to synchronous web services.
5. **mailService.war**: sends emails base on task data to specified recipients.
6. **documentStore.war**: manages binary files passed as data values between nets and tasks during the execution of a case .
7. **monitorService.war**: provides basic monitoring capabilities for active processes.

The Resource Service is described in detail in Chapter 6. The Worklet Service is described in detail in Chapter 8. The WS Invoker, DocumentStore and Mail Services are described in Chapter 10. The Monitor Service is described in Chapter 7.

There are six optional web services:

1. **digitalSignature.war**: authenticates the information provided on a form using a digital signature via X.509 certificates and private keys.
2. **mailSender.war**: (*Deprecated*) provides a custom form for a task, from which an email can be sent.
3. **yawlSMSInvoker.war**: allows tasks to be read and modified via SMS Services.
4. **twitterService.war**: allows status updates to be sent from processes directly to a twitter account.
5. **schedulingService.war**: allows resources to be scheduled for, and allocated to, cases.
6. **procletService.war**: provides inter-process communication (cf. Chapter 9).

While all the core web services are pre-registered in the Engine when it first starts, the optional services are not, and so require manual registration after installation before they can be used. Please see Chapter 6, Section 6.4.3 for more details.

Tip: If a .war file is to be copied into the *webapps* directory to replace a file of the same name, it is advisable to first shutdown Tomcat, and delete the unpacked directory for that war, before copying in the new war file. On restart, the new war's contents will be unpacked. If the old unpacked directory is not removed, on restart the new war file *will not* unpack. If Tomcat is running when the new war is copied to the *webapps* directory, it *will* unpack and replace the old directory, but, depending on how it has been configured, may run out of resources while doing so, resulting in an *OutOfMemoryError* (see Section 2.4.5).

2.4.4 Configuring YAWL for other Databases

YAWL uses Hibernate (hibernate.org) as a database framework, which provides a transparency layer between YAWL and the back-end database used to support it. While the Enterprise version of YAWL is pre-configured to use PostgreSQL, it is a relatively simple process to reconfigure for other databases. Besides PostgreSQL, YAWL has been successfully tested with MySQL, HypersonicSQL, Oracle, Microsoft SQL Server (MSSQL), Apache Derby and H2 (used by YAWL4Study) Other database platforms are known to work well with Hibernate (see www.hibernate.org/80.html for the complete list) and so should have no trouble working with YAWL, too [7].

[7] Please pass on your experiences using YAWL with database platforms, other than those listed, on the YAWL forum.

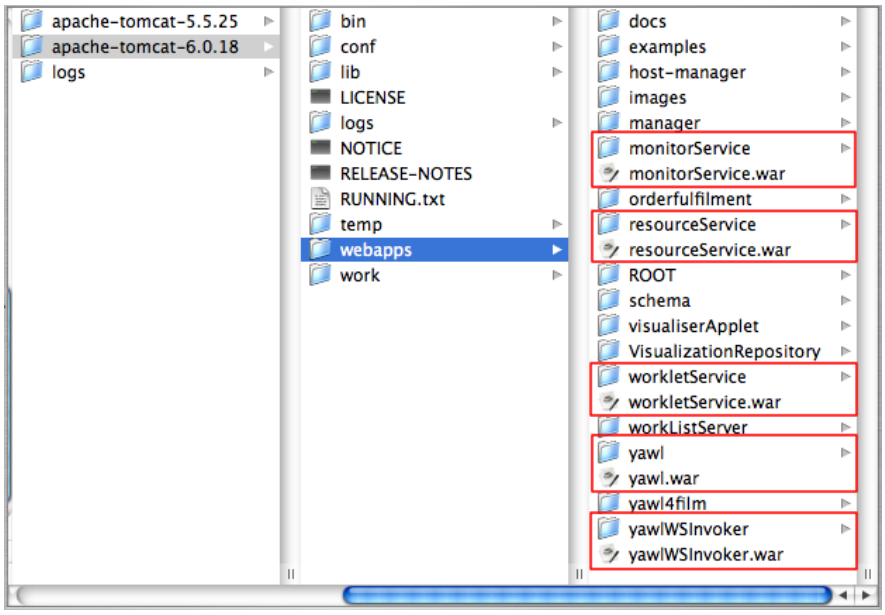


Figure 2.10: YAWL Core Services deployed in /webapps directory (OSX Example)

Each YAWL service that communicates with the database (i.e. the Engine and the Resource, Worklet, DocumentStore, Scheduling and Proclet Services) has a configuration file called *hibernate.properties* located in its *WEB-INF/classes* directory. The properties file contains a 'Platforms' section with default settings for a number of different database platforms – all except one (PostgreSQL) commented out (see Listing 2.1 for an excerpt of the properties file). To configure for a different database platform, comment out the currently enabled platform, then uncomment the platform of choice, ensuring the username and password values match the target database authorisations.

Remember to change the *hibernate.properties* file for each of the webapps mentioned above.

2.4.5 Troubleshooting Memory Problems

By default, Tomcat is configured to use the default memory parameters of the installed Java Virtual Machine (JVM). While this is sufficient for individual users, or even small groups, when larger numbers of users access YAWL concurrently, you may experience an *OutOfMemoryException* and Tomcat will freeze. If you do experience this problem, the solution is to configure Tomcat to have a larger amount of memory allocated to it when it starts.

The default memory allocation is 64 megabytes. Depending on the RAM you have available, a setting of between 256 and 512 megabytes is usually sufficient in the majority of cases.

To set the Tomcat memory allocation in a Windows environment:

1. In a text editor, create a new file and enter the line: `set JAVA_OPTS=-server -Xmx256m`
2. Save the file as */your_tomcat_dir*/bin/**setenv.bat**

To set the Tomcat memory allocation in a *nix or OSX environment:

1. In a text editor, create a new file and enter the line: `export JAVA_OPTS="-server -Xmx256m"`
2. Save the file as */your_tomcat_dir*/bin/**setenv.sh**

These instructions use 256m as an example; please replace it as necessary with the actual amount of memory you'd like to have allocated to Tomcat when it starts.

```
## HypersonicSQL

#hibernate.dialect org.hibernate.dialect.HSQLDialect
#hibernate.connection.driver_class org.hsqldb.jdbcDriver
#hibernate.connection.username sa
#hibernate.connection.password
#hibernate.connection.url jdbc:hsqldb:file:./webapps/yawl/yawl


## PostgreSQL

hibernate.dialect org.hibernate.dialect.PostgreSQLDialect
hibernate.connection.driver_class org.postgresql.Driver
hibernate.connection.url jdbc:postgresql:yawl
hibernate.connection.username postgres
hibernate.connection.password yawl
#hibernate.query.substitutions yes 'Y', no 'N'


## DB2

#hibernate.dialect org.hibernate.dialect.DB2Dialect
#hibernate.connection.driver_class COM.ibm.db2.jdbc.app.DB2Driver
#hibernate.connection.url jdbc:db2:test
#hibernate.connection.username db2
#hibernate.connection.password db2

...

## MySQL

##hibernate.connection.driver_class org.gjt.mm.mysql.Driver
#hibernate.dialect org.hibernate.dialect.MySQLDialect
#hibernate.connection.driver_class com.mysql.jdbc.Driver
#hibernate.connection.url jdbc:mysql:///yawl
#hibernate.connection.username root
#hibernate.connection.password


## Oracle

#hibernate.dialect org.hibernate.dialect.Oracle9Dialect
#hibernate.dialect org.hibernate.dialect.OracleDialect
#hibernate.connection.driver_class oracle.jdbc.driver.OracleDriver
#hibernate.connection.username ora
#hibernate.connection.password ora
#hibernate.connection.url jdbc:oracle:thin:@localhost:1521:test
```

Listing 2.1: hibernate.properties file (excerpt) with PostgreSQL enabled

Chapter 3

Getting Started with YAWL

3.1 Introduction

Nowadays, organisations are challenged to continuously improve their efficiency and to respond quickly to changes in their environment, such as new business opportunities, competition threats, and evolving customer expectations. It is not surprising then that organisations are paying more attention to capturing, analysing and improving their work practices in a systematic manner. The methods, techniques and tools to do this are collectively known as Business Process Management (BPM).

For IT departments, BPM provides an opportunity to align IT systems with business requirements, and to re-organise existing application infrastructure to better support the day-to-day operations of the organisation. BPM initiatives often translate into requirements for IT systems. Here is where workflow technology comes into play. Business process models produced by business experts are taken as a starting point by software architects to produce a blueprint for a software application that co-ordinates, monitors and controls some or all of the tasks that make up these business processes. Such software applications are called *workflows*. An example of a business process is an order-to-cash process: one that goes from the moment a purchase order for a product or service is received by an organisation to the moment the customer pays for the products, including aspects such as invoicing and shipment. After capturing this process from beginning to end, an organisation may choose to add further details about the people, legacy applications, messages and documents involved, and to deploy a workflow application to co-ordinate this process.

You can build a workflow application using general-purpose software programming platforms, e.g. as a bunch of Web applications, Enterprise Java Beans and legacy applications connected together... but this defeats the purpose of aligning the models produced by business people with the resulting IT systems. This is why one should consider an alternative approach: to develop workflow applications on top of a dedicated *workflow management system*.[1]

Many years ago, workflow was a bit of a dark art, practised by deep-pocketed companies that were able to afford expensive workflow management systems and highly specialised consultants. Today, workflow technology is widely available and its benefits and pitfalls are more widely understood. A word of warning though: while workflow doesn't have to belong to arcane masters of lore, it's also not something to trivialise. If a workflow application is not aligned with the business it's been deployed in, it can be worse than a manual, paper-based bureaucracy. It is therefore important that both business and IT stakeholders follow a sound BPM methodology before attempting to deploy a workflow application.

But assuming you've decided on a workflow solution, it's time to make a choice. You can still choose to pay for a workflow system, or you can get one for free. If you're for the latter, maybe YAWL is for you. YAWL, which stands for *Yet Another Workflow Language*, is a fully open-sourced workflow system (or

[1]The term business process management system (BPMS) is often used to refer to something similar to a workflow management system. The difference is that a BPMS supposedly offers richer functionality for analysing business processes, while workflow systems traditionally focused on the co-ordination of tasks. However, the gap between these two is narrowing, and it is difficult to differentiate modern workflow management systems and BPMSs.

"business process management system" if you prefer). Its tongue-in-cheek name belies the fact that YAWL is rather unique. It's based on a very rich workflow definition language, capable of capturing all sorts of flow dependencies between tasks. It has open interfaces based on Web standards, which enable developers to plug-in existing applications and to extend and customise the system in many ways. It also provides a graphical editor with built-in verification functionality, which helps solution architects and developers to capture workflow models and to automatically detect subtle but potentially nasty errors early-on in the piece. Finally, YAWL is arguably the most mature open-source workflow management system around. From its beginnings as an academic prototype, YAWL has evolved into an enterprise-grade workflow engine thanks to contributions from the YAWL Foundation members, and from the organisations and individuals who have used it. This demonstrated commitment from its users and community of developers also ensures the continuity of the system.

If you think YAWL might be for you, you may be wondering how to learn more. This chapter provides a gentle introduction to the YAWL workflow system. The aim of the chapter is to help people to get YAWL up and running with a minimum of fuss. The chapter doesn't cover all possible features and components of YAWL. Instead, it focuses on some essential aspects that will help you to become familiar enough with YAWL that you feel comfortable designing and executing at least simple workflows. For more information, you may refer to other chapters in this manual, the technical manual or the various academic papers and case studies available at the YAWL web site.[2].

3.2 Terminology

Before jumping in and getting our hands dirty with a real workflow example, let's briefly agree on some basic terms.

Business Process: A set of interdependent activities that need to be performed in response to a business event, to achieve a business objective. Typical examples of business processes are "complaint handling", "order-to-cash", or "credit card approval".

Workflow Application: A software application that co-ordinates the tasks, data and resources that compose a business process, in whole or part. Sometimes the term "workflow" is used as a shorthand for "workflow application".

Workflow Specification: (Also known as *Workflow Model*) A description of a business process to the level of detail required for its deployment into a workflow engine. A workflow specification defines which tasks should be performed, under which conditions and in which order, which data, documents and resources are required in performing each task, etc.

Workflow System: A system that can be used to develop and to run a workflow application. A workflow system usually includes a *process editor* to support the design of workflow models, a *workflow engine* to support the execution of workflow models, and at least one *worklist handler*.

Workflow Engine: The runtime component of a workflow system responsible for determining which tasks need to be performed and when, for maintaining execution logs, and for delegating the performance of tasks to software applications/services or to a worklist handler.

Case: (Also known as *Workflow Instance*) A specific instantiation of a workflow model as a result of an event. For example, an order management workflow is instantiated every time a new order arrives. Each of these orders leads to a different *case*.

Task: (Also known as *Activity*) A description of a unit of work that may need to be performed as part of a workflow. Workflow models are composed of tasks. Generally, a task may be either manually carried out by a person or automatically by a software application.

[2] http://yawlfoundation.org

Work item: (Also known as *Task Instance*) A particular instance of a task that needs to be performed as part of a given workflow instance.

Worklist: A list of work items.

Worklist Handler: (Also known as a *Task Management Service*) A software component that manages work items issued by a workflow engine and that assigns, prioritises and presents these work items to human participants according to policies that may be configured in the workflow model and/or at runtime.

3.3 Building a Simple Workflow Example

Designing a workflow typically begins with a process modelling exercise. A *process modelling expert* sits down with a *domain expert*, and picks their brains on "how things are done". The knowledge gained on the sequencing and nature of the work done is then transformed into an executable workflow. Let's take a look at an example transcript between a process modelling expert, Processa Maree Experta, and her cousin, Domainic Experta, who runs the credit application department of a company called Loans-R-Us.

Processa: *So, how does a credit application begin?*

Domainic: *Well, an application arrives in our office. Once we receive it, we validate the claim.*

Processa: *What happens then?*

Domainic: *We determine what credit requirements there are for the application, then we seek a credit report for the applicant.*

Processa: *So the credit report is requested after the credit requirements are determined?*

Domainic: *Mostly. Sometimes we request the credit report first. Actually, the order in which we do them doesn't really matter.*

Processa: *Ah, so both tasks could be done at the same time?*

Domainic: *Yes, I guess they could.*

Processa: *Then what happens?*

Domainic: *Once we have both the credit report and credit requirements, we can tell whether we need to do a large credit approval, or a small approval. Only senior staff here are allowed to approve large credit applications.*

Processa: *So, what makes a credit application large?*

Domainic: *If the application is for $\$5,000$ or more, it's considered large. Any lesser amount is considered a small application, and can be done by anybody in our department.*

Processa takes this transcript, dumps it on your desk and tells you to implement a workflow to match. What's more, she wants you to do it with YAWL[3].

In a nutshell, a workflow specification in YAWL describes what work needs to be done, when and by whom. Each YAWL specification is composed of one or more YAWL nets: exactly one *starting* net (also known as the root or parent or top-level net) and zero or more sub-nets. In this tutorial, we'll keep things simple and we will only consider the case of a YAWL specification composed of one net (the starting net). A net has two mandatory elements: an input condition which acts as the starting point (graphically represented like this: ▶) and an output condition which signals the end (the ■ symbol). Figure 3.1 depicts the YAWL Editor

[3]This chapter is more an overview of YAWL than an examination of its tool support. The assumed knowledge at this point is that you have installed and can begin using the YAWL toolset by following the instructions provided in Chapter 2.

with a brand new specification open and with the specification's starting net visible. Don't worry too much at this stage about the various components of the Editor – it is described in detail in Chapter 4.

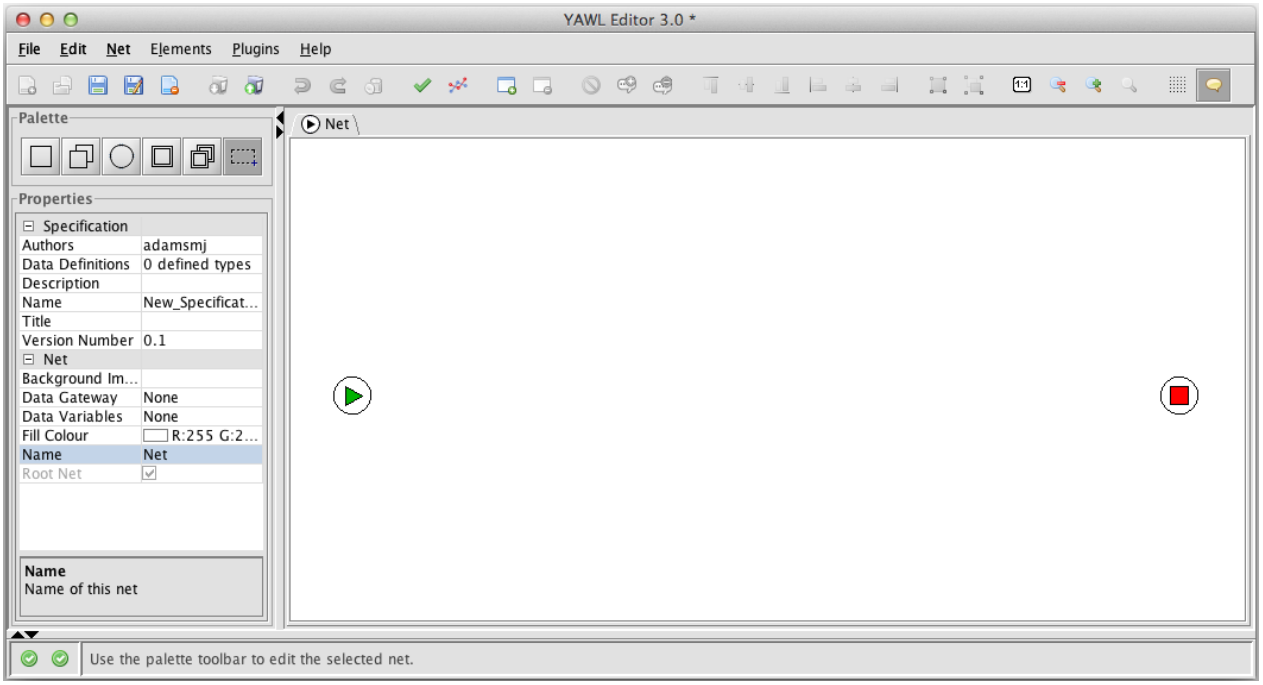


Figure 3.1: A New Specification and its Starting Net

It's time to start modelling the work to be done. Typical workflow specifications in YAWL will make significant use of atomic tasks. An atomic task (represented in YAWL as a square) models a stand-alone piece of work that is either manual or automatic, and it's here that workflow designers starts earning their money. Just how much work should a single atomic task represent? The answer is not always obvious.

Looking again at Processa's transcript, we decide that an initial atomic task is needed for receipting and validating a claim. After that, two additional pieces of work need to be done, but in no particular order. We'll add an atomic task each for determining credit requirements, and seeking a credit report. The next step requires that both credit requirements are determined, and that a credit report be ready. We need an extra task to run only once they are finished which will decide, based on the application amount, whether we then send the application on for a large or a small approval process. We expect large approvals and small approvals to have differing work requirements, so we'll model each type of approval as a separate task.

In all, we've identified six distinct pieces of work. Select the *Atomic Task* tool on the Palette at the top left of the Editor, then place six atomic tasks onto the starting net and give each a meaningful label. You should have the skeleton of a workflow that looks something like figure 3.2.

We're now ready to begin describing how the tasks in our starting net are to be ordered in their execution (known as its *control-flow*). The transfer of work between two tasks is done through a "flow". Flows are depicted within YAWL as unidirectional arrows. For a YAWL specification to be valid, every task must be tied into a net via flows that can be traced back to the net's input condition, and which will eventually lead to the net's output condition.

We'll need a flow from the input condition to the *Receive and Validate Application* task, then two flows from that task to the tasks *Determine Credit Requirements* and *Obtain Credit Report* respectively. From each of these, a flow must go to the task *Choose Approval Process*. From this task, we need a flow going to the tasks *Large Credit Approval* and *Small Credit Approval* respectively. From these last two tasks, we need flows to the final output condition.

By default a YAWL task can only have one incoming flow and one outgoing flow. When we need more incoming flows to a task, we must unambiguously state how the task should handle its inflows: should it

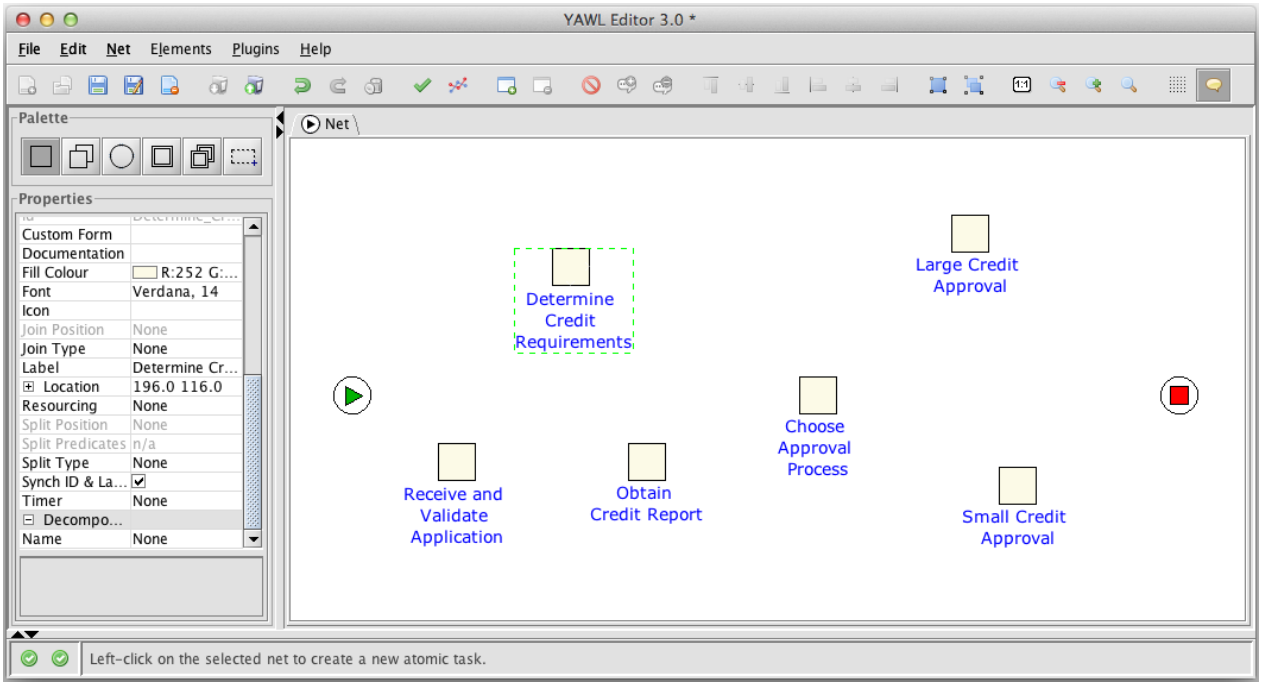


Figure 3.2: Atomic Tasks Added to the Starting Net

wait for all of them? Should it wait for only one of them? Or something in the middle? This disambiguation is done by 'decorating' the task with a *join*. A similar situation holds when a task has multiple outgoing flows. In this case, we need to decorate the task with a *split*.

Figure 3.3 lists the available joins and splits that can be used on tasks, along with a brief description of the behaviour to expect from tasks when using them.

Returning to our example, the tasks *Receive and Validate Application* and *Choose Approval Process* both require decoration. The first of these two tasks requires an AND-Split because the subsequent tasks can be done in parallel. The second task should have an AND-join, so that it waits for both preceding tasks to complete before continuing, and XOR-split decorator to signal that either of the subsequent tasks should be performed, but not both. With these splits and joins in place, we can now connect the remaining tasks as depicted in figure 3.4.

Now that we have finished specifying the control-flow perspective of our process, we need to say how information passes from YAWL to its participants (e.g. workers and external applications) and how information comes back into YAWL once they're finished. This is done by attaching a *decomposition* to each task. Every atomic task that requires work to be performed needs to have a decomposition. A decomposition may be described as a contract between the task and its 'environment', describing the data that will be assigned and updated when the task is performed and the so-called *YAWL Custom Service* (a web service designed for the YAWL environment) that will be responsible for the task's execution. Note that the YAWL Engine does not directly perform the work of the task – responsibility is always deferred to the designated YAWL Service. It is possible to define an atomic task without assigning it a decomposition: they represent so-called "empty" steps and are generally used to capture a point in the specification where there is a need to synchronise certain tasks and start a new set of tasks.

In our working example, all tasks except one require a decomposition. It is enough at this stage to simply create a decomposition per task. To do this, you need to right-click on each task and select the "Set Task Decomposition" option. For this example, we'll choose the *Default Engine Worklist* (actually the worklist handler built in to the Resource Service) as the "type of decomposition". This tells YAWL that when the task is ready to be executed, it should be displayed in the default worklist. Every instance of the task will then appear in the worklist of human participants so they may receive data relative to the task instance, work

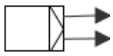
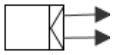
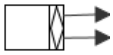
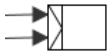
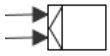
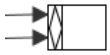
Name:	Symbol:	Description:
Split Types:		
XOR-Split		The XOR-Split is used to trigger *only one* outgoing flow. It is best used for automatically choosing between a number of possible exclusive alternatives once a task completes.
AND-Split		The AND-Split is used to start a number of task instances simultaneously. It can be viewed as a specialisation of the OR-Split, where work will be triggered to start on *all* outgoing flows.
OR-Split		The OR-Split is used to trigger some, but *not necessarily all* outgoing flows to other tasks. It is best used when we won't know until run-time exactly what concurrent resultant work can lead from the completion of a task.
Join Types:		
AND-Join		A task with an AND-Join will wait to receive completed work from all of its incoming flows before beginning. It is typically used to synchronise pre-requisite activities that must be completed before some new piece of work may begin.
XOR-Join		Once *any* work has completed on an incoming flow, a task with an XOR-Join will be capable of beginning work. It is typically used to allow new work to start so long as one of several different pieces of earlier work have been completed.
OR-Join		The OR-Join ensures that a task waits until all incoming flows have either finished, or will never finish. OR-Joins are "smart": they will only wait for something if it is necessary to wait. However, understanding models with OR-joins can be tricky and therefore OR-joins should be used sparingly.

Figure 3.3: Supported Splits and Joins in YAWL

on that data, and finally return work results to YAWL. Another type of decomposition, which we won't illustrate in this tutorial, is to associate tasks with a Custom Service that can, for example, send notifications and receive replies via SMS, or send an email, or interact with a scheduling calendar, or call an external Web Service, and so on.

The one task in our example that does not need a decomposition is the one labelled *Choose Approval Process*. This task does not need any participant interaction because the decision on whether to choose either *Large Credit Approval* or *Small Credit Approval* can be automatically determined with data made available to the workflow instance.

Congratulations, you now have an executable YAWL workflow specification. However, more effort is needed with respect to data and resourcing to achieve real utility. All YAWL can currently do with this specification is walk an unspecified user through a default path of the workflow.

3.4 Advanced Workflow Concepts

A specification capable of only walking a user through a path of a workflow is hardly going to win us any awards in workflow automation. We still have at least two major concerns to address before our specification becomes useful.

Firstly, we need to decide which participants should perform which tasks. This is discussed in section 3.4.1.

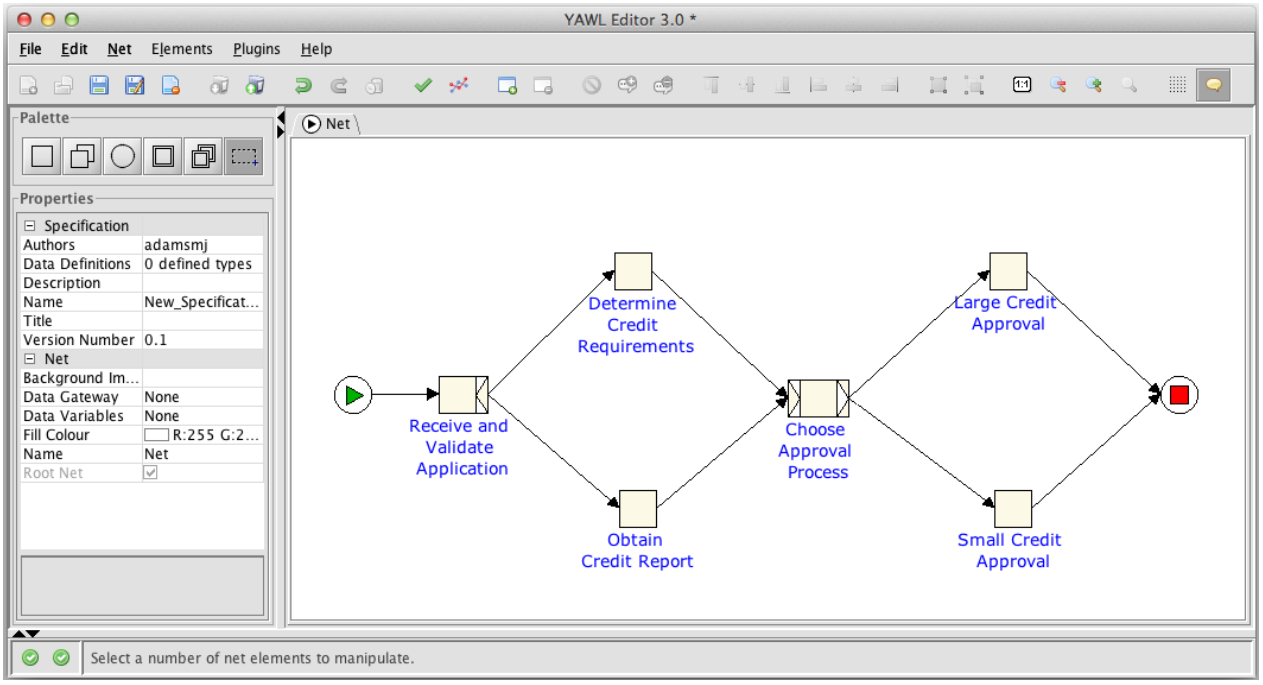


Figure 3.4: Multiple Flows Between Tasks

Secondly, we need to figure out what data these participants need from the workflow system, what data they need to supply the workflow system, and how the workflow system will use data to implement automated choice between alternatives. Concerns involving workflow data are covered in section 3.4.2.

3.4.1 Modelling Resourcing Requirements

It's time now to add detail to our YAWL specification, describing which participants should be doing particular pieces of the work specified. We'll assume that the entire workflow is to be carried out by the "Applications Department" of Loans-R-Us. All employees within this department are capable of performing the role *Credit Officer*, but a subset of these with several years of experience also perform the role *Senior Credit Officer*. Anyone with the *Credit Officer* role is allowed to approve small credit applications. Only those performing the role *Senior Credit Officer* are allowed to do the final approval of large credit applications.

We therefore have a modelling requirement where every credit officer is capable of processing a credit application right through from its receipt to approval, so long as the application is for a small amount. When it comes to the step of approval for large credit applications, however, only senior staff are allowed to do this approval.

We first need to establish an *organisation model* within a running YAWL system where we identify those participants from the Applications Department. For each of these, we assign the *Credit Officer* role. For the subset of participants recognised as senior, we also assign an extra role of *Senior Credit Officer*.

To define this organisational model you need to log into the YAWL Resource Service (for now, we'll use the generic username *admin* and password *YAWL*). Assuming you used one of the automatic installers, this is simply a matter of starting the engine (choose "Start Engine" from the options shown for the YAWL program) followed by access the Resource Service in a Web Browser (choose "YAWL Control Centre" from the options shown for the YAWL program). When you have logged in you can create new roles by choosing the "Org Data Mgt" form from the menubar, and new participants by choosing the "User Mgt" form from the menubar. This is illustrated in Figure 3.5 where the role "Senior Credit Officer" is defined and Figure 3.6 where the participant "Michael Corleone" is defined and assigned that role.

Once we have defined all the required roles and participants, we can specify resourcing requirements for

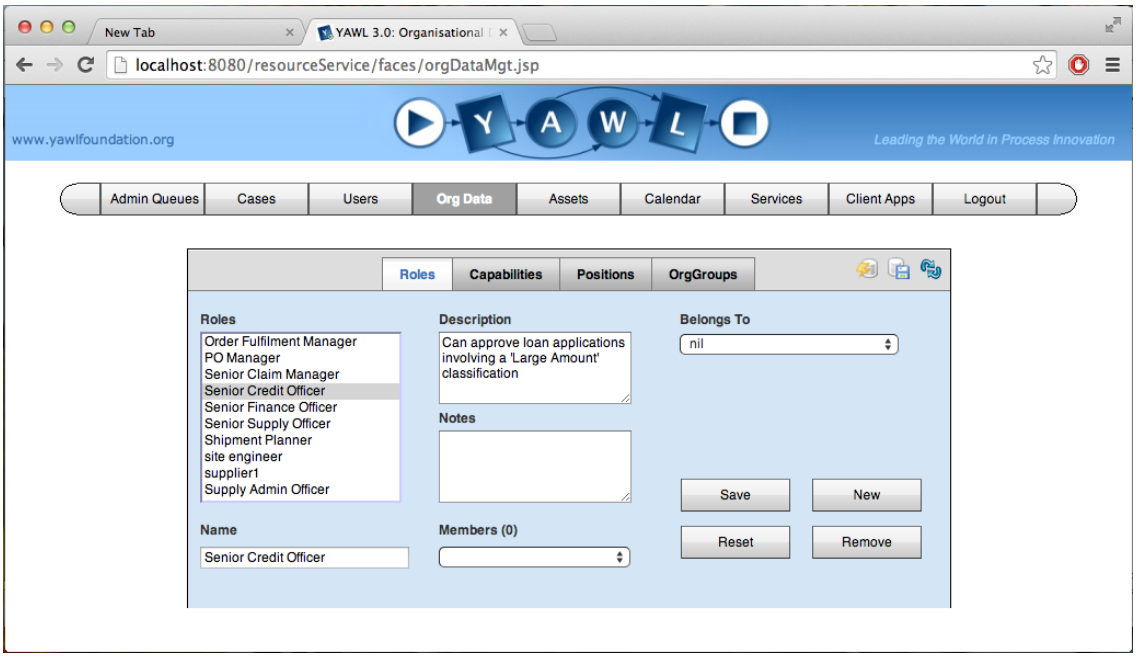


Figure 3.5: Defining a Role

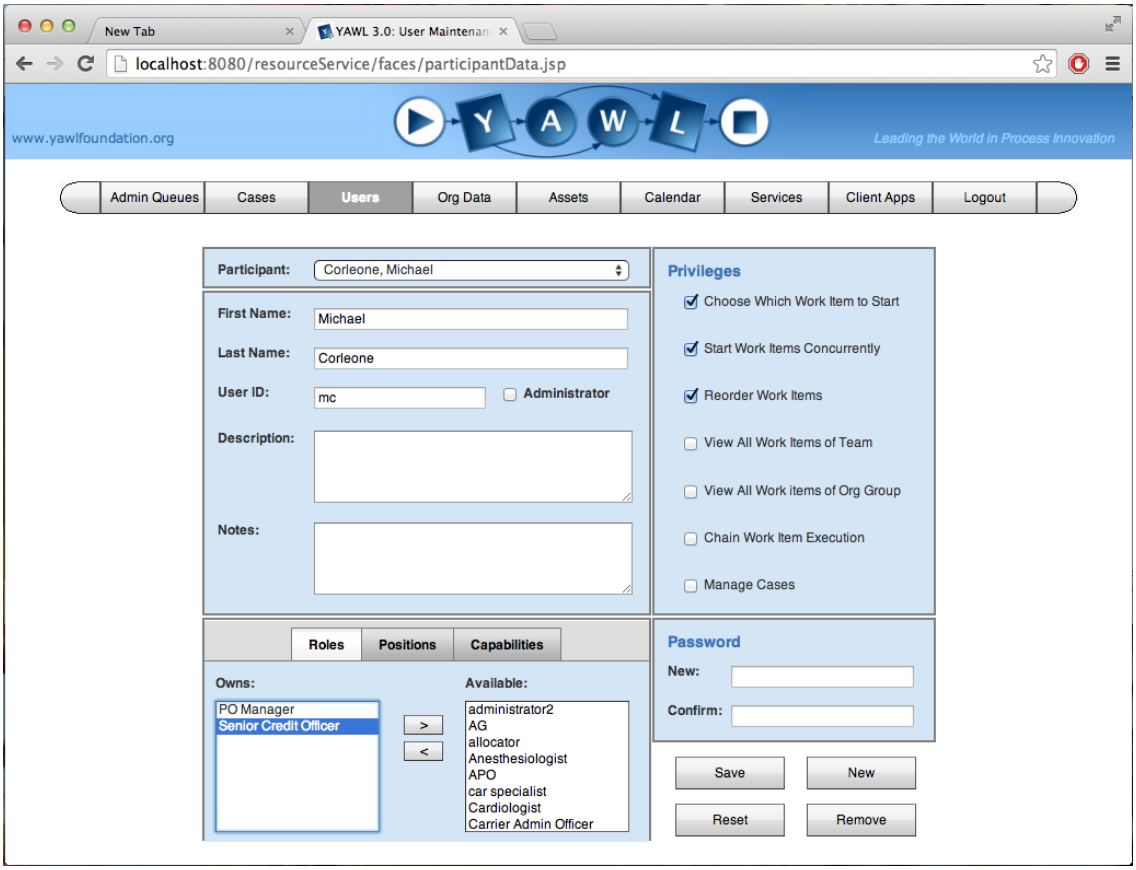


Figure 3.6: Defining a Participant

tasks. Back in the Editor, select a task with your mouse and then in the Properties window on the left click on the Resourcing property (in the Task section). Clicking on the property's button will show the Resourcing dialog for that task. Figure 3.7) shows the dialog with the *Enable System Offers* interaction strategy selected, and the *Senior Credit Officer* role added. This means at runtime the system will offer an instance of the task to all participants who are members of that role, from where one of these participants can then choose to allocate this instance to themselves and later choose to actually start working on it. This strategy (Offer: System, Allocation: User, Start: User) is a common interaction strategy for tasks to be executed by participants.

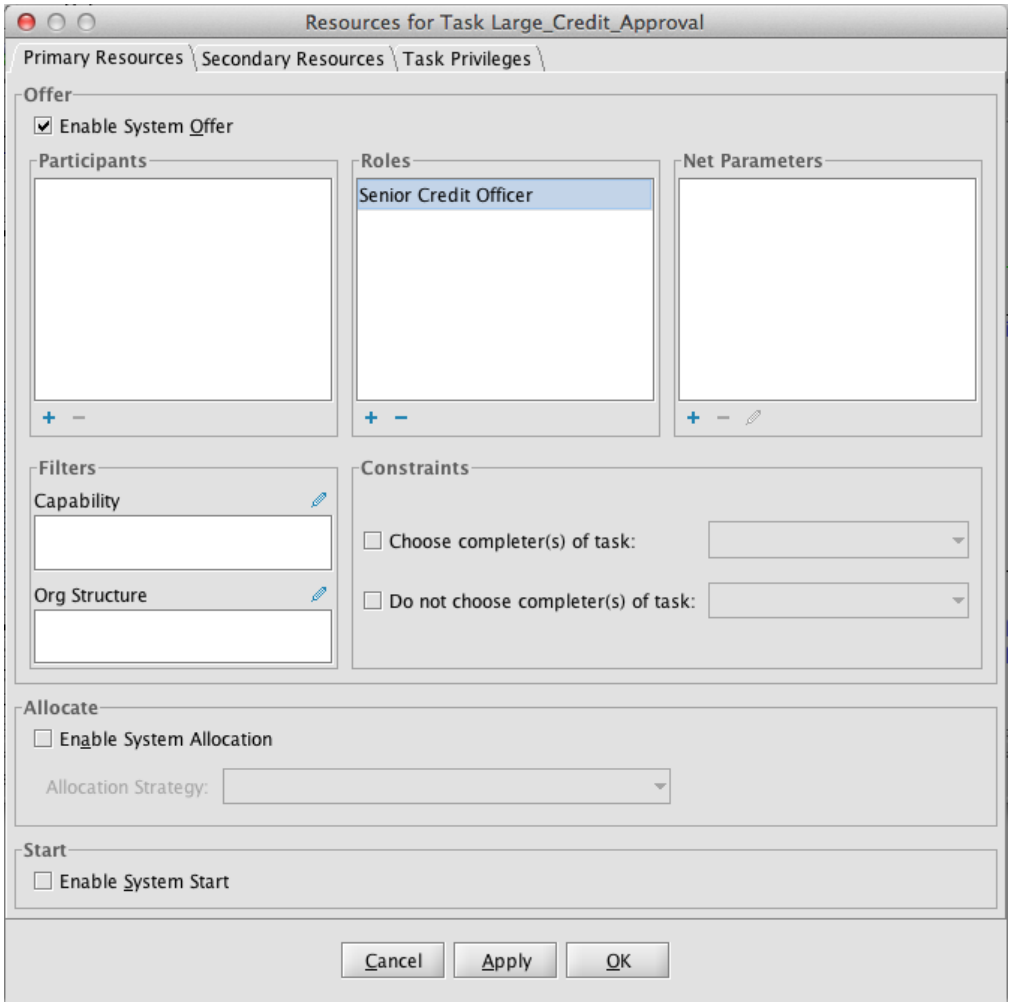


Figure 3.7: Specifying Resourcing Requirements

Resourcing requirements can become quite complex, and the YAWL environment offers comprehensive support for the vast majority of workflow resource patterns, but for the moment we will simply assign roles to the various tasks and apply the System-User-User interaction strategy.

3.4.2 Modelling Data Requirements

We now need to specify what data will be passed about during the execution of an instance of this specification. Specifically, we need to describe what data participants will need in each work item, and what data they must return to the Engine once the work item is complete. We also need to have a way of moving data about in the running workflow, including how we can use that data to automatically choose between flows in a running workflow.

We stated before that task decompositions are used to define how a running workflow interacts with the external 'environment'. In fact, all tasks of a YAWL specification that require interaction with the external environment need to be associated with a decomposition. Decompositions can have a number of *variables* (or parameters) defined for them, describing what data must be supplied to a running net or task instance, and what data that net or task instance will eventually deliver. Each variable has a *name* it may be referenced by, a *type* dictating valid values it may store, a *designation* (or usage) indicating how that data may be used, and a *scope* defining the visibility of the parameter.

Variables can belong to one of two scopes, which we'll refer to as *net scope* and *task scope*. At runtime, every net, and every task instance with a decomposition, will have data stored as a number of variables belonging to it. To get data from a net instance to a task instance within the net, or visa-versa, we require a data transfer. In YAWL, all data is passed this way – from net-level to task-level when a task instance starts, and from task-level back to net-level when the task instance completes; data cannot be directly transferred from one task instance to another.

Valid designations for a task variable are *Input*, *Output* or both *Input & Output*. A task variable with an input designation is one where we expect data to be delivered from a net-level variable to that variable at run-time. A task variable with an output designation it expected to have its data output to a containing net-level variables once a task instance has completed.

Just like task variables, net variables may have *Input*, *Output* or both *Input & Output* designations. A net-level input variable requires its data value provided to it when the net begins. A net-level output variable passes its value out when the net completes. In addition to these two options, net variables may have a *Local* designation. They are used to store intermediate data during the execution of a process instance.

Data transfer from a net to a task is achieved via *inbound mappings*. An inbound mapping is a statement that says how to transfer data from the net variable to a task input variable. Inbound mappings are evaluated when the task starts. Conversely, once the task is completed, data is moved from the task's scope to the net scope by means of *outbound mappings*. An outbound mapping is a statement that says how to move data from a task's output variable to a variable in its containing net. XPath expressions[4] are used to describe inbound and outbound mappings. Accordingly, the parameters of nets and tasks in YAWL are all encoded as XML documents.

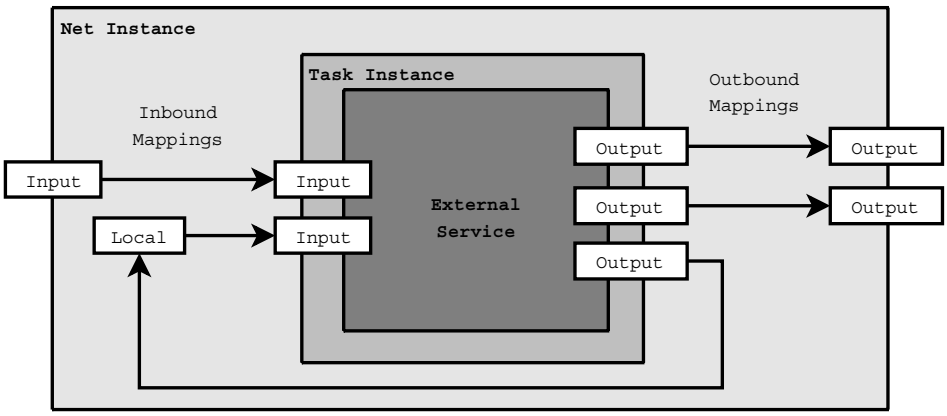


Figure 3.8: Overview of Data Transfer between a Net and Task

Figure 3.8 depicts example data transfers over the lifetime of a task instance. The task's decomposition defines two input variables and three output variables. When the task instance starts, values for its input variables are populated by evaluating the input mappings for the task, which are then passed onto the task's designated YAWL Service. The default worklist is an example of an external service, but there are many others and advanced users are able to define and add virtually any type of service that meets their need. The external service eventually finishes its execution, resulting in values being supplied to the output

[4]For more advanced workflows, XPath expressions may prove too limiting. Accordingly, YAWL allows developers to also incorporate XQuery expressions for data transfer.

variables of the task instance. The output mappings for this task instance are then evaluated, resulting in a number of variables in the task's containing net instance being updated with values from the task instance's output variables.

Now we have a basic understanding of data transfer in YAWL, let's start specifying the data transfer requirements of our workflow specification. Since all data are passed as XML documents, all data types are defined using XML Schema Language – there are over 40 in-built XML Schema data types, and YAWL allows designers to also define their own. For our example, will limit our variables to be of either XML Schema *string* or *double* simple types. We'll go through our atomic task decompositions now and add variables to each task decomposition first before we review the necessary data transfer mappings for moving data between tasks and their containing net.

Imagine that we have finished an exercise of determining what data must be passed out of YAWL at the starting of each task of our specification, and what data must be returned back into the system when each task completes. We note through the exercise that even though the tasks *Large Credit Approval* and *Small Credit Approval* are done by different parts of the organisation, they have the same data requirements, and can both use the same decomposition (which we'll call *Credit Approval*). We have a resulting variable requirement per decomposition as per figure 3.9.

Decomposition	Param-Name	Type	Designation
Receive and Validate Application	ApplicationID	string	output
Determine Credit Requirements	ApplicationID	string	input
	ApplicationAmount	double	output
Obtain Credit Report	ApplicationID	string	input
	CreditReportRef	string	output
Credit Approval	ApplicationID	string	input
	CreditReportRef	string	input
	ApplicationAmount	double	input

Figure 3.9: Variables Required for Task Decompositions

To give you some idea of how this might look, figure 3.10 is a screenshot of the Editor showing the decomposition for the task *Determine Credit Requirements* with an input variable *ApplicationID*, and an output variable *AppplicationAmount*. When running our specification, a participant will be offered a work-item for an instance of this task. They will be given an application identifier, and will work outside of the system, eventually generating an application amount for that application. Figure 3.11 shows how the default worklist displays an instance of this task for a participant to work with.

Attaining an application amount may be as trivial as reading the number from the relevant form, or as involved as considering the business's current risk exposure, running calculations, and adjusting the figure to something the insurance company is more willing to accept. The exact nature of the work to be done is left to the participant and the business rules of the organisation, and only that data relevant to progressing the workflow needs to be passed back into the system once they are done.

Because we can't transfer data directly between tasks, we need a number of local variables for our starting net. Specifically, we'll need matching *ApplicationID* and *ApplicationAmount* local parameters at the net level, and another called *CreditReportRef* that will be used by a couple of other tasks in the workflow.

With these local net variables in place, we need to specify how data is passed between the net and tasks with XPath expressions. The XPath expressions needed are fairly straightforward, in fact the Editor creates them automatically [5]. For a task input variable, we need the expression to fetch and populate the value of this variable with that of its corresponding net variable. For a task output variable, the matching net variable needs an expression to retrieve the value of the task variable. Figure 3.12 shows an example of the queries needed for the *Determine Credit Requirements* task. A similar exercise can then be conducted for the variables of the remaining tasks.

The only thing remaining to do with data in our specification is to deal with the XOR-Split in our model. XOR-splits need a boolean XPath expression to be associated with each outgoing flow of the split[6]. These

[5]These default mappings can be modified or replaced as required by advanced users.

[6]Each XOR-split and OR-split has one flow specified as the default, and is assumed to always have *true* value, to ensure that the

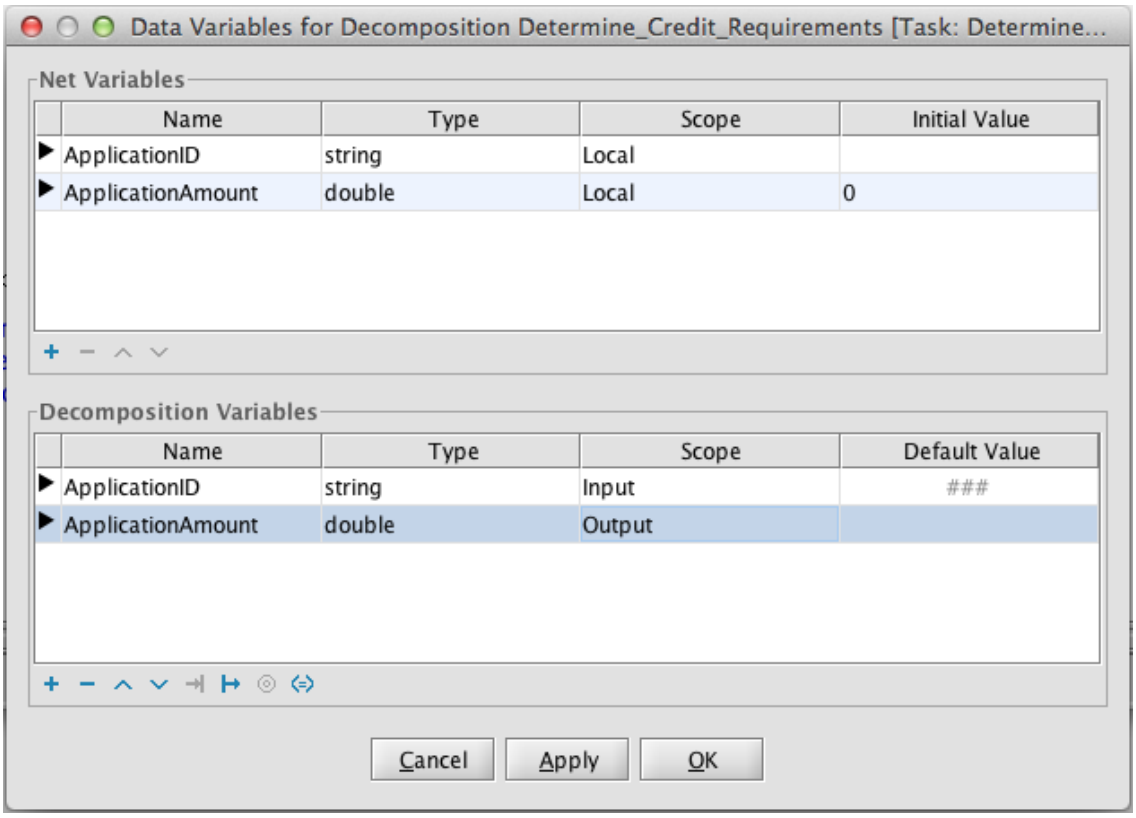


Figure 3.10: Establishing Variables for a Task Decomposition

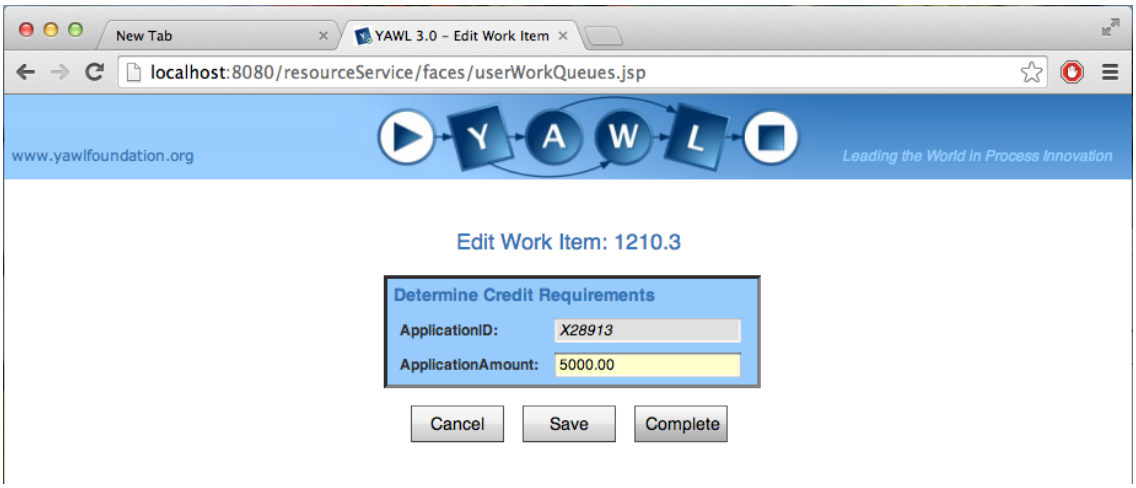
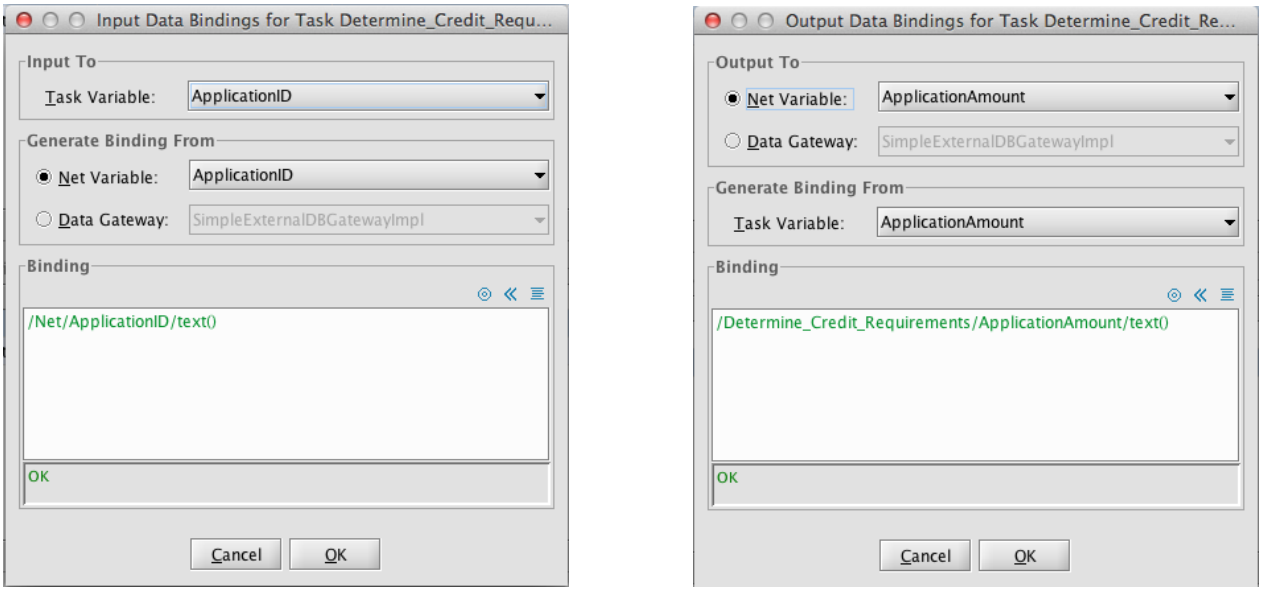


Figure 3.11: A *Determine Credit Requirements* work-item shown on a dynamic form

expressions are evaluated once a task instance completes, and so can only interrogate the state of a net at that moment. Expressions that evaluate to true indicate that a flow is to be taken. In the case of an XOR-Split, the flows have an ordered priority specified. The first flow in order whose XPath expression evaluates to true will be the only flow taken from a completed task.

The only task we need to consider in this regard is *Choose approval process*. Consulting our transcript again, approval amounts of less than $5000 are to be routed to the *Small Credit Approval* task. Anything more

workflow can continue even when all other flow conditions evaluate to false.



(a) Input Binding Dialog

(b) Output Binding Dialog

Figure 3.12: Data Mappings for *Determine Credit Requirements* Variables

requires *Large Credit Approval* to be run. The XPath expressions needed to capture this choice are shown in Figure 3.13. Once this is specified, we are done. We have a workflow specification that ensures the right work and data gets routed to the right participants at the right time.

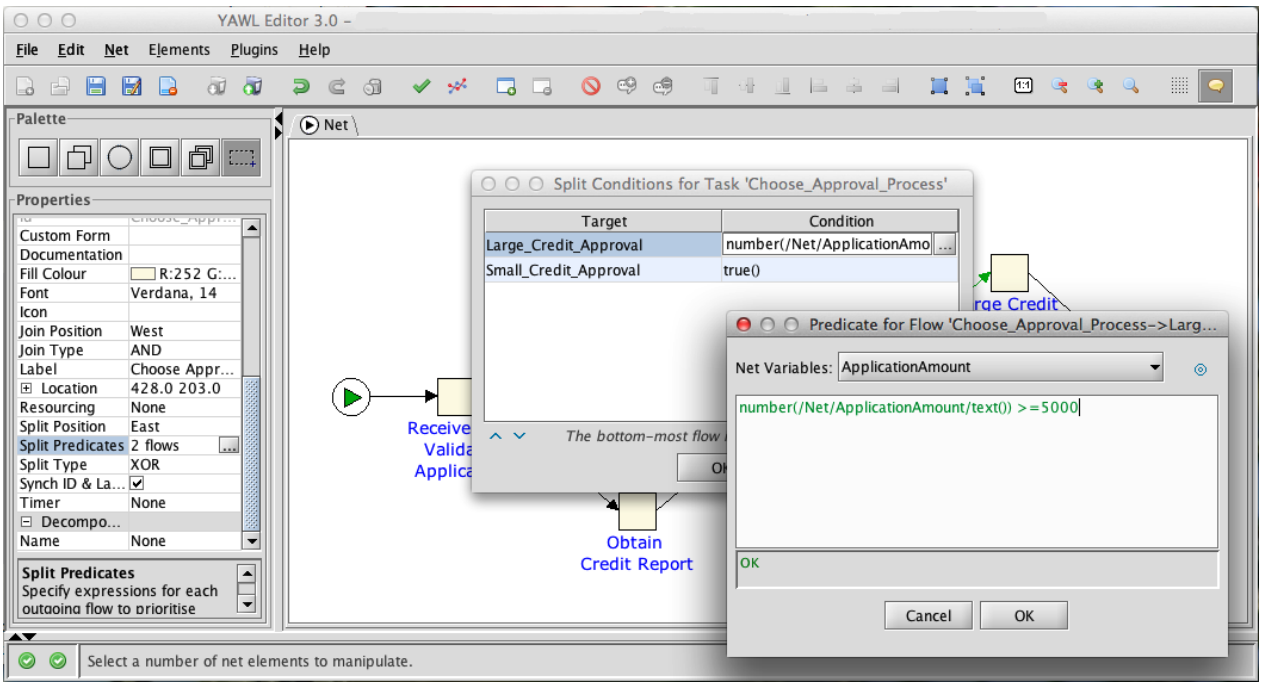


Figure 3.13: XPath predicates to choose between flows of an XOR-Split task

3.5 Where To From Here

You've now seen how we can construct a simple workflow specification for YAWL. We've used atomic tasks, with various types of splits and joins, along with resourcing and data requirements to implement a simple credit application processing workflow specification. But, we have so far only scratched the surface of what can be achieved with YAWL.

What you haven't seen yet is how larger workflows can be constructed by binding a number of nets together with composite tasks. We are also capable of iterating through a number of instances of a single task using Multiple Instance Task constructs. A single task can also be used to trigger the cancellation of current work in other parts of the workflow, which might be used for modelling a customer calling and cancelling an order that is currently being processed. Finally, we haven't described *conditions*, which represent the state a workflow is in after one task is finished but before another starts. Conditions allow us to model two or more participants competing for the same work, or a user making a decision on things that workflow systems cannot not automatically determine. An example of this would be asking a participant to decide on whether the aesthetics of some partially assembled work are appealing or not, and having the workflow coordinate further work based on that choice.

What we've also glossed over here is how to actually use YAWL's toolset. We have used version 3.0 of the Editor and of the Engine for the screenshots in this chapter. The components of YAWL can all be found int its Github repository, via the URL `github.com/yawlfoundation/yawl`. Further explanations of the use of this environment can be found in the remainder of this user manual, while there is also a *technical manual* for those that want to develop more complex applications. A number of case studies documenting the use of YAWL are also available. A forum around the components of YAWL can be accessed via the YAWL website (`yawlfoundation.org/forum`. And as mentioned earlier, YAWL is the product of several years of research into workflow patterns and formal foundations of workflow. This research, along with other informative material is available via the URL `yawlfoundation.org`.

Chapter 4

The Editor

Before a workflow model can be executed it must first be defined. This chapter describes the YAWL Editor Version 4.1, a tool for creating, editing, configuring, validating and analysing workflow specifications. New users are encouraged to read the chapter sequentially; experienced users may pick-and-choose what they need from this chapter.

Figure 4.1 illustrates the interactions among some of the major components of the YAWL environment.

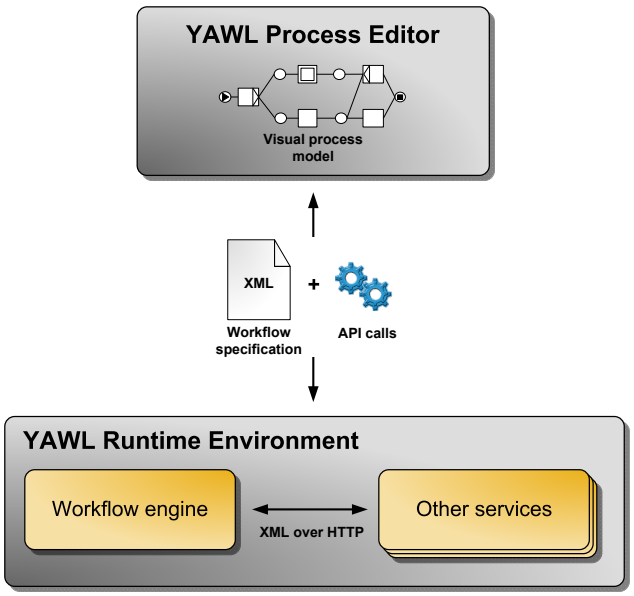


Figure 4.1: Basic YAWL Component Overview

In this chapter, the icon on the left indicates a hands-on method or instruction.

4.1 Launching the YAWL Editor

The Editor is installed along with the other YAWL System components using any of the installers described in Chapter 2. It can also be installed manually by downloading the latest version from the YAWL Github repository: `github.com/yawlfoundation/editor/releases`. Always ensure that the version of the Editor you are using matches the version of YAWL installed.

The YAWL Editor is distributed as a compressed zip archive, containing a number of Java Archive (jar) files. Unzip the downloaded zip to any directory, then double click on the *YAWLEditor4.1.jar* file to start the application (where supported). The YAWL Editor can also be started from a command line or Terminal prompt:

java –jar YAWLEditor4.1.jar

4.2 The YAWL Editor Workspace

The first time you start the YAWL Editor, you will be presented with a blank canvas, and a prompt in the Status Bar advising you to open or create a specification to begin.

Before you create your first specification, let's take a brief tour of the Editor's workspace and the elements within (the use of each element is fully described in later sections). The workspace is shown in Figure 4.2.

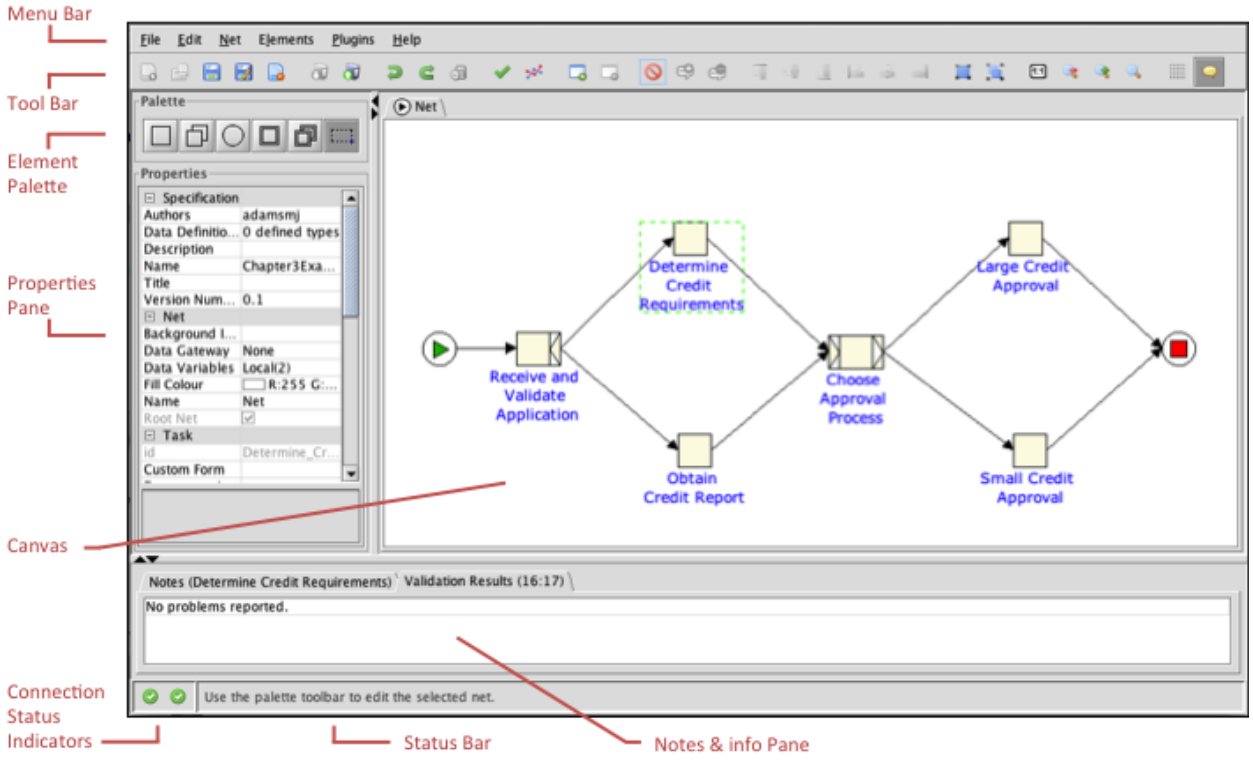


Figure 4.2: The YAWL Editor Workspace

4.2.1 The Toolbar

The Tool Bar contains ten groups of buttons that mirror most of the available menu items. Each button will be enabled or disabled at various times depending on what you are currently doing in the Editor. The purpose of each button is briefly described below. Further information can be found in later sections, when required.

File Actions

This group of buttons provides the standard file options (left to right):

- **Create** a new specification (Section 4.4.1);
- **Open** an existing specification file. Specification files will have a **.yawl** extension;
- **Save** the currently loaded specification to file (Section 4.3.3). For newly created specifications, this behaves the same as *Save As*;
- **Save** the currently loaded specification **As** a new file name;
- **Close** the loaded specification. If there are any unsaved changes, you will be prompted whether to save the file first before closing.

Transfer to/from Engine

The button on the left allows you to download a specification directly from the YAWL Engine and have it opened in the Editor for editing. Any specification currently loaded in the Engine can be opened in the Editor in this way.

The button on the right allows you to upload the specification currently opened in the Editor directly to the YAWL Engine (without having to save the file to disk).

Edit Actions

This group of buttons provides the standard Undo and Redo options as well as the option cut, copy, paste and delete the currently selected object(s).

Validation & Analysis

The first of these two buttons allows you to validate your specification against YAWL syntax and semantics (Section 4.3.3), while the second allows you to deeply analyse your specification for deadlocks and other issues (Section 4.11).

Add/Remove Sub-Net

Each YAWL specification consists of one or more nets. You can use these buttons to add a new sub-net to, or remove an existing sub-net from, your specification (Section 4.4.4).

Cancellation Set Actions

These buttons allow you to (left to right):

- Toggle on or off the management of the cancellation set of the selected task.
- Add the selected object(s) to a task's cancellation set.
- Remove the selected object(s) from a task's cancellation set.

See Section 4.6) for more details.

Alignment Actions

These buttons can be used to assist with the alignment of objects within your specification, when multiple objects have been selected. Left-to-right, they allow you to align selected objects based on top edges, centres horizontally, bottom edges, left sides, centres vertically and right sides. The first selected object is used as the reference to align the other objects to.

Resize Element Actions

These buttons can be used to increase or decrease the size of selected canvas objects.

Zoom Actions

These buttons allow you to zoom in or out on the currently selected net. The buttons are, from left-to-right, reset the net view to actual size, zoom out of the entire net, zoom in on the entire net, and zoom into the currently selected net elements. Alternately, you may zoom in and out on the currently selected net by holding down the Shift key and using the mouse wheel. Holding down the Shift + Ctrl keys and using the mouse wheel will scroll the net's view left and right.

View Options

The button on the left will toggle on or off a canvas grid to assist in aligning objects. The button on the right will toggle on or off the tool tips for all items in the Editor.

4.2.2 The Menubar

This section provides a brief overview of the YAWL Menus located along the top of the YAWL Editor. The majority of menu choices are also available via the toolbar.

File

The File menu incorporates the File, Transfer to/from Engine, and Validation & Analysis toolbar actions. Additionally, this menu also contains:

- *Open Recent*: shows a list of the eight most recent specifications loaded or saved in the Editor, so that one can be selected to be opened, saving the trouble of navigating to it via the file open dialog. If you hover the mouse over a listed file for a moment, a tip will appear showing the file's full path;
- *Print*: prints the entire loaded specification (graphically);
- *Delete Orphaned Decompositions*: allows you to permanently remove decompositions that are no longer attached to any task (see Section 4.7.2 for details of task decompositions).
- *Preferences*: shows a dialog which allows for the setting of a number of preference items.

Edit

In addition to the toolbar's Edit actions, this menu also contains items to Cut, Copy and Paste objects to/from the canvas.

Net

In addition to the Add/Remove Sub-Net toolbar actions, this menu also contains these items:

- *Set Net Background Colour*: set the background colour of the selected net.
- *Set Net Background Image*: set a background image for the selected net. Images that are smaller than the current net canvas are tiled to fit.
- *Export to PNG Image*: saves a graphical image of the current net to a png file.
- *Print Net*: prints the currently selected net (graphically).
- *Store in Repository*: Stores a copy of the current net in the Repository, from where it can be later added to other specifications (see Section 4.4.8).
- *Load from Repository*: Adds a previously stored net to the current specification.
- *Remove from Repository*: Removes a previously stored net from the Repository.

Elements

This menu contains the Alignment, Resize Elements and Cancellation Set toolbar actions. In addition, there is an item to set the fill colour for all selected objects. There is also a sub-menu titled *Decomposition* that contains the following items:

- *Store in Repository*: Stores a copy of the decomposition of the currently selected task in the Repository, from where it can be later added to other specifications (see Section 4.7.2).
- *Load from Repository*: Adds a previously stored decomposition to the current specification.
- *Remove from Repository*: Removes a previously stored decomposition from the Repository.

Plugins

The Editor allows developers to create plugin components that can be used within the Editor. Currently, two plugins ship with the Editor, one that supports Process Configuration (Section 4.19), and another that provides a toolbar for the setting of task decorators (splits and joins, see Section 4.4.3), as an alternative to accessing those settings via the Properties Pane.

Help

The Help Menu provides:

- An *About* dialog, showing the version and build date of the Editor in use. The dialog also provides web links to the YAWL online forum, and the YAWL issues list (to report any bugs or problems you may find).
- A *Check for Updates* item, which allows you to check for any updates to the Editor, and if so, provides for the downloading and automatic replacement of the current Editor with the latest version (Section 4.20).

4.2.3 Elements Palette

The Elements Palette contains six selectable buttons – five YAWL language icons and a selection tool – that allow the creation, selection and positioning of objects within your specification. The palette buttons are also accessible via a popup menu by right-clicking on any blank area of the canvas.

Once a Palette button is selected, it is possible to place objects of that type on the canvas by left-clicking the mouse button at the desired location. Each is briefly described below.

Atomic Task

Select this button to create one or more Atomic Tasks, each of which represents a single task to be performed, usually by a human participant or an external application or service.

Multiple Instance Atomic Task

Select this button to create one or more Multiple Instance Atomic Tasks, each of which allows you to run multiple instances of a task concurrently.

Condition

Select this button to create a Condition, which is a way to represent state for the Net between the execution of particular tasks.

Composite Task

Select this button to create a Composite Task, which is a container for a sub-net - with its own set of YAWL elements and constrained by the same syntax.

Multiple Instance Composite Tasks

Select this button to create a Multiple Instance Composite Task, which allows you to execute multiple instances of a composite task (i.e. a sub-net) concurrently.

Marquee Selection

Select this button to activate the Marquee Selector, which will allow you to select individual or multiple objects by clicking and dragging the left mouse button. *Note: you cannot create flows (arrows between tasks) while the Marquee Selector is selected.*

4.2.4 The Properties Pane

The properties pane provides easy access to view and modify all of the properties of the selected specification, net, decomposition and net element (task, condition or flow). Sections can be rolled-up or expanded as desired, and a brief description of each property is shown in the box at the bottom of the Properties pane. Some of the more interesting properties of each section are described below.

Specification

All of the specification level properties are set here - authors, data type definition, specification name, title (used when printing a specification) and version number. Note that a new specification is provided with a default name of *New_Specification*, which should usually be changed to something more meaningful. If it is unchanged the first time the specification is saved, it will be set to the user-provided name for the file in the *File→Save As* dialog.

Net

Individual net values are set here. A Fill Colour set here will override the default net background colour. For a specification with multiple nets, you can choose to change which net is the Root Net here. Note that the current root net cannot be unset directly, rather you must choose a sub-net to promote to root net, in which case the former root net will be automatically demoted.

Task

A Fill Colour set here will override the default element background colour. A Font set here will override the default label font, allowing you to set the font family, size and colour for individual net elements. An Icon for the selected task can be chosen from a dropdown list, and there are more than 400 standard icons to choose from (you can add your own icons to the list by placing them in a folder and providing its path via the *Preferences* dialog). Split and Join Type and Position can all be set via the corresponding dropdown lists.

The coordinates of a task can be set directly in the Location property. Resourcing for a task can be specified via its corresponding property, which opens the *Resourcing* dialog for the task (Section 4.8). This section of the Properties Pane also provides for how to set Split Predicates via the *Split Predicates* dialog (Section 4.7.4), how to set a Timer for a task via the *Timer* dialog (Section 4.13), and how to set multiple-instance attributes via the *Multiple Instance Attributes* dialog (for Multiple Instance tasks only - Section 4.4.5).

Decomposition

When a task is selected, in addition to its properties being shown, the properties of its decomposition (if any) are also shown in the Decomposition section. A new task will have a single decomposition property: Name, with a value "None", denoting it is not yet associated with a decomposition. You can set the decomposition for the selected task by making a selection from the dropdown list of the Name property. All of the current task decompositions in the specification will be listed, any one of which can be chosen for the task. Also listed is the value *New...*, which should be chosen to create a new decomposition. A decomposition can be renamed by choosing *Rename...* from the dropdown list and providing a new name. Once a decomposition has been added to a task, you can choose which Custom Service the task will be allocated to at runtime (via a dropdown list), whether it is Automated and if so, which Codelet to associate with it (if any), and the decompositions Data Variables and Extended Attributes (see Section 4.7.2).

Condition

The properties available for a condition are a subset of those shown for a task (i.e. only those net element properties relevant to a condition).

Flow

When a flow is selected, all of its properties are shown as read-only (since they are set via other dialogs) except one, the Line Style, which can be set via a selection from the dropdown list.

4.2.5 Other Components

The Canvas

The Canvas is where elements (objects) are placed to create and modify a YAWL specification.

Notes and Info Pane

This panel consists of two tabs:

- On the *Notes* pane, you can add free-form text to accompany a selected task or condition. Any text entered is accessible only at design time;
- The *Info* pane will list problems or messages that may occur while you are building your model, when you validate it and when you analyse it.

Status Bar

The Status Bar consists of three parts:

- On the left are two "Connection Status Indicators", which indicate whether there are currently valid connections to a running YAWL Engine and Resource Service respectively (required for certain design activities discussed later in this chapter). A valid connection will show a green indicator, a disconnection as a red indicator.

- Next there is a status message area that provides useful contextual hints while using the Editor.
- On the right is a progress bar, which shows the progress of file open and save events, and so is only visible during those events.

4.3 Working with YAWL Specification files

The YAWL Editor is a tool that allows you to graphically design a process specification that may be saved to file and later loaded into the YAWL Engine for execution. This section explains how to create, open and save YAWL specification files.

4.3.1 Creating a Specification

A YAWL workflow specification consists of exactly one starting net (also known as the *root* net), and zero or more sub-nets. The root net is where execution of a specification begins, while sub-nets allow you to design larger specifications in a modular hierarchy, to aid readability and re-use.

Each net will contain two mandatory components, an *input* condition (▶) and an *output* condition (■). A net must always contain exactly one of each, and they cannot be removed from the net. Net elements (*tasks* and *conditions*) are arranged between the input and output conditions and connected by directed arcs, called *flow relations*, to define control-flow, or order dependencies, between them.

To create a new specification:

- Click on the *Create New Specification* button,  , at the top left of the toolbar, or select the **File** Menu and choose **New**.

This will create a blank Net called "Net" which will be, by default, the starting (or root) net of the workflow. For details on changing the starting net, see Section 4.4.8.

4.3.2 Opening a Specification

There are three ways to load a specification into the Editor:

Opening a Specification File

To open a specification file from disk:

- Click on the *Open Specification* button,  , second from left on the toolbar, or select the **File** Menu and choose **Open**.

An *Open Specification File* dialog appears, from which you may select the file to open. All YAWL specification files end with a **.yawl** extension, and only files with that extension will be shown in the dialog.

Alternately, you may choose to reopen a recently opened file selecting the **File** Menu and choosing **Open Recent**. The eight most recently opened files will be listed in a sub-menu, from which you may choose the file to reopen.

File Drag'n'Drop

A specification file may be loaded into the Editor by selecting the file in a file management tool, such as Windows File Manager or Finder on OSX, a dragging it directly onto the (empty) canvas of the Editor.

Download from YAWL Engine

If you have a current connection to a running YAWL Engine (see Section 4.10) you may download any specification it currently has stored directly into the Editor (i.e. without accessing its disk file). To do so:

- Click on the *Download from Engine* button, , on the toolbar, or select the **File** Menu and choose **Download**.
- A small dialog will appear that lists all of the available specification files currently loaded in the connected YAWL Engine (Figure 4.3). Make a selection from the list and click **OK** to download and open the specification in the Editor.

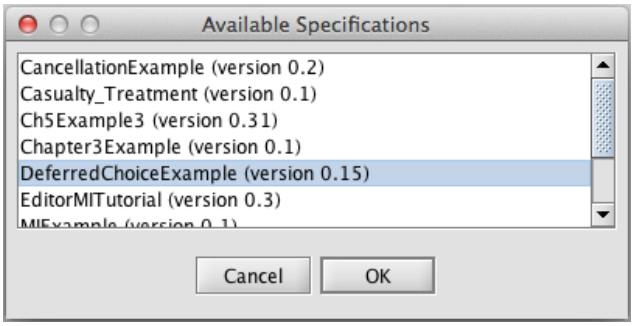


Figure 4.3: Available Specifications dialog

4.3.3 Saving and Validating a Specification

At any stage you can save your specification to a YAWL disk file (.yawl) that can be later uploaded into the YAWL Engine, or you may choose to upload the specification directly to a connected YAWL Engine, bypassing saving it to disk file. In the former case, it is recommended that you first *validate* the specification (i.e. check that it is syntactically and semantically correct), unless it is an interim save of an incomplete specification. In the latter case, the verification is done automatically as part of the upload.

Validating a Specification

To validate your specification:

1. Click the Validate button, , on the toolbar or click **File** on the Menu and choose **Validate**.
2. If problems are detected, they will be listed in the Info Pane at the bottom of the Editor with details of any inconsistencies that would prevent the specification from successfully executing in the YAWL Engine. The Info Pane tab label also shows the time the last validation was run. Figure 4.4 shows an example. If a message ends with "<more...>", you may click on it to see a popup with more information about the problem and possibly a suggestion on how to fix it. Click anywhere to dismiss the 'more information' box.

Saving a Specification File

To save your specification to file to disk, click on the Save button, , on the toolbar or click **File** on the Menu and choose **Save**. If set in the Preferences Dialog, the 'File Save Options' dialog appears (Figure 4.5).

The 'File Save Options' Dialog has the following options:

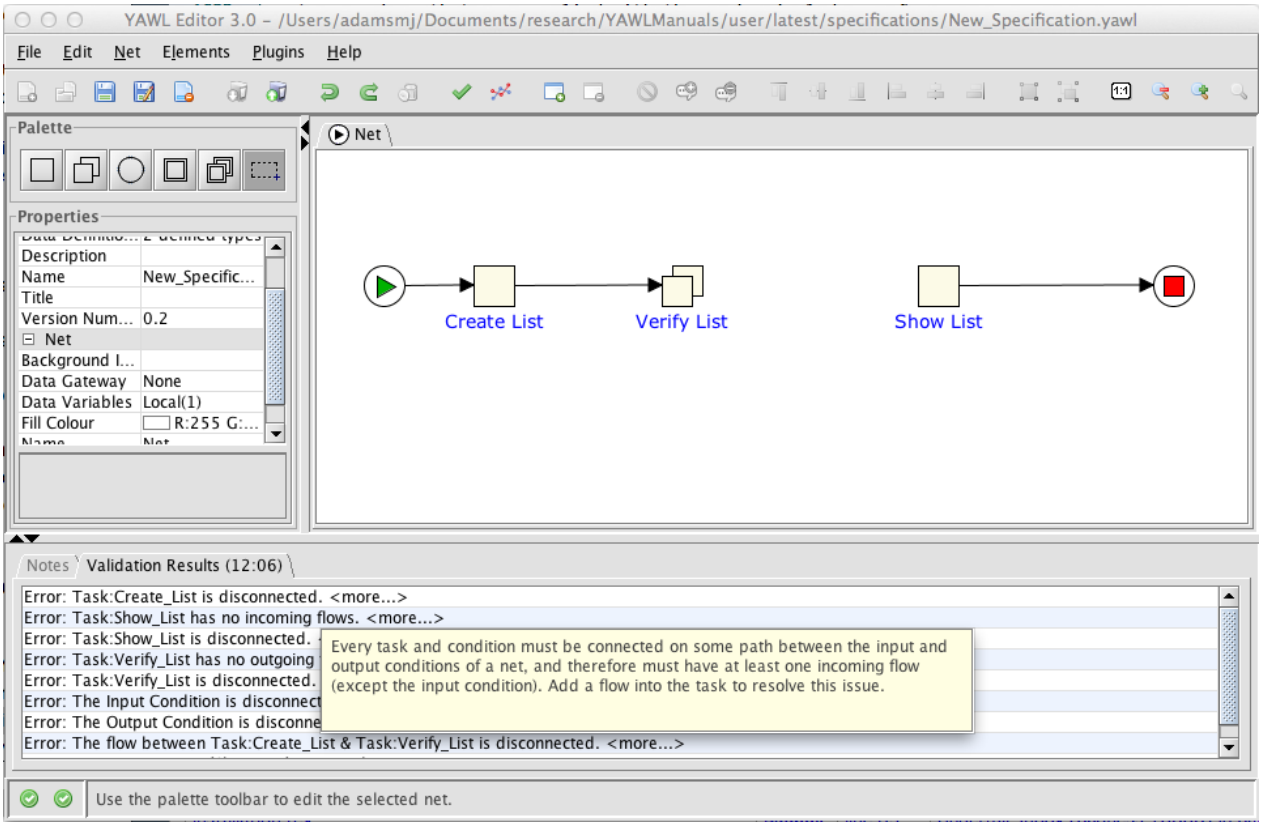


Figure 4.4: An invalid specification

- **Verify specification** When selected, will run a verification each time the specification is saved to file. Checking this option is recommended, unless you a doing many interim saves of specifications you know to be incomplete.
- **Analyse specification** When selected, will deeply analyse the specification each time it is saved to file. See Section 4.11 for more details of the Analysis process. (Note: analysis may take some time for large and/or complex models, and should be unchecked for incremental saves of such models).
- **Auto increment minor version number** When selected, the minor part of the specification's version number will increment each time the file is saved. Checking this option is recommended.
- **Backup previous version** When selected, the last saved version of the specification will be copied to a file of the same name, but with a *.bak* extension. Checking this option is recommended.
- **Keep all previous versions** When selected, the previous version of the specification will be copied to a file of the same name, but with its version number appended to it, so that an archive of each and every version of a specification can be maintained. For example, if you have a specification called 'InsuranceClaim.yawl', and have chosen to auto-increment the minor version number, each time a version of the file is saved, a copy of the previous version will be created called 'InsuranceClaim.0.1.yawl', 'InsuranceClaim.0.2.yawl', and so on.
- **Show this dialog for each save** When selected, the dialog is shown whenever a file is saved, allowing you to change the settings as desired. Since these settings are usually changed infrequently, it is recommended that this option be unchecked. All of the settings in this dialog can also be accessed via the Preferences Dialog whenever necessary.

Once the dialog is completed, click **OK** to save the file. This saved specification file can now be loaded into a running YAWL Engine and executed (See Chapter 6).

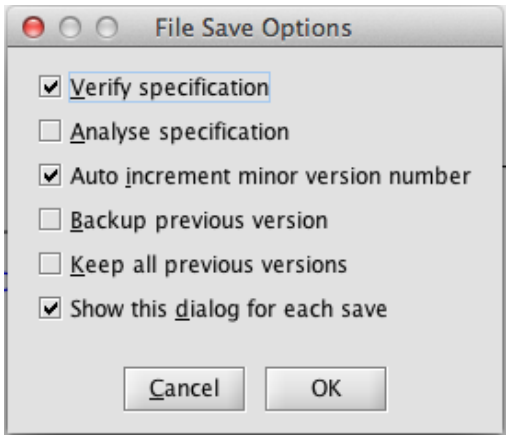


Figure 4.5: File Save Options dialog

Upload to YAWL Engine

You may upload a valid specification directly to a connected YAWL Engine. As part of the upload, you can also choose to remove any previous versions of the specification from the Engine, and/or to start of new instance of the specification in the Engine.

To do so:

- Click on the *Upload to Engine* button, , on the toolbar, or select the **File** Menu and choose **Upload**.
- A small Upload Options dialog will appear that lists the following options (Figure 4.6):

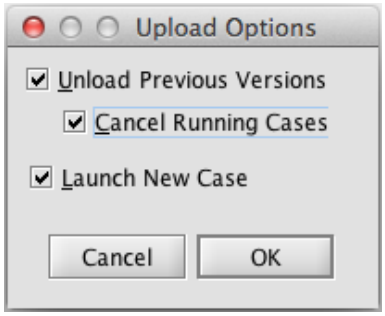


Figure 4.6: Specification Upload Options dialog

  - **Unload Previous Versions:** The Engine allows multiple versions a particular specification to co-exist in the Engine, so that any currently running instances may continue under previous versions of a specification, while new instances can be started under new versions. When this option is selected, all previous versions of the current specification will be removed from the Engine, subject to whether the following option is checked.
  - **Cancel Running Cases:** If you choose the Unload Previous Versions option above, you may also choose to cancel any currently running cases that are instances of those versions. If you choose this option, all currently running instances of previous versions of the specification will be cancelled, before the removal of those previous versions. If you do not choose to cancel running cases, then only those previous versions that ***do not*** have instances of them currently running will be removed (because the Engine requires a loaded specification for all currently running instances).

– **Launch New Case:** When selected, the current specification will be loaded into the Engine and a new instance of it will be launched.

Make the appropriate selections in the dialog then click **OK** to upload the currently loaded specification into the Engine. The specification will first be automatically verified, and the upload will proceed only if there are no validation errors. If the specification is valid and you chose to launch a new case, a message will appear with the case id of the launched case, if successful, or an error if there were any problems.

NOTE: The Engine requires that all specification layout information (i.e. element locations, colours, fonts, icons. etc.) be removed from a specification when it is uploaded. Because of this, the Editor will save the layout information to repository when the specification is uploaded. If the specification is later downloaded, its layout information will be automatically reattached by the Editor. If you download a specification from the Engine to a computer other than that which it was uploaded from, the layout information will not be available to the Editor, and so a default layout will be calculated.

4.4 The Control-Flow Perspective

A workflow specification can be broadly divided into three aspects or perspectives: *control-flow* (how and when execution of a specification progresses through its various elements), *data* (how and when data values are passed and populated), and *resource* (what human and non-human resources are required to perform the work defined within the specification).

This section presents a hands-on walk-through that covers the basic control-flow components of the YAWL language, and how to use those components to develop a YAWL specification for a simple business scenario from beginning to end (the data and resource perspectives are discussed in the sections following this one). Designing the control-flow of a specification begins with choosing the required components from the *Palette* and laying them out on the *canvas*.

The scenario that we will follow throughout this section involves designing and implementing a workflow for a student who has just completed their secondary study and is now looking to start their career. The scenario will present the student with two choices: to either enrol in a University and begin tertiary education, or undertake private study that will eventually lead to getting a job.

Look for the "hands-on" icon to the left of instructions for specific details of practical examples.

4.4.1 Create a New Specification

Create a new specification in the Editor as described in the previous section, then:

1. In the *Specification* section of the Properties Pane, edit the various properties as you feel appropriate (optional). For the moment, you can ignore the *Data Definitions* property.
2. Rename the Net by editing the *Name* property in the *Net* section of the Properties Pane. Change the name of the Net to "My Career". This Net will be the root net for our scenario.
3. You are now ready to start constructing your specification.

4.4.2 Atomic Tasks

An atomic task represents a single unit of work, which may involve interaction with a user via a worklist or with an external application or web service.

To add an Atomic task to the canvas:

1. Click on the *Atomic Task* button, ▢, in the Palette, or right click in an empty area of the canvas and choose **Atomic Task** from the popup menu.

   - TIP: When you move the mouse pointer over the canvas, it will change to signify which palette element currently chosen.

2. Position your mouse just to the right of the Input Condition (the ▶ symbol, which signifies the starting point of your process) on the canvas, and click the left mouse button once to place an Atomic Task.

3. Every task that will perform work at runtime must be associated with a *decomposition*. To set the decomposition of this task, first click inside it to select it. Then, go to the *Decomposition* section of the Properties Pane (you may have to scroll to the bottom) and click the *Name* property. Select *New...* from the dropdown list to display a dialog for you to enter a name for the new decomposition.

Set the decomposition name to "Begin My Career", then click the OK button.

- The decomposition Name property's dropdown list will show all the current decompositions in the specification, along with entries to create a new decomposition, rename an exist one, or remove a decomposition from a task (by choosing *None*).
- By default, a new task takes on the label of the decomposition that it is associated with (several tasks are allowed to share the same decomposition). Once you've created your task, you are free to relabel the task to whatever you like. This can be done by entering a new label for the task via the *Label* property in the *Task* section of the Properties Pane, or by double-clicking on the task an entering a new label in the dialog that appears. This will not change the name of the decomposition with which the task is associated.

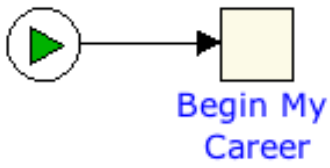


Figure 4.7: An established flow relation

4. Connect the Input Condition to your Atomic Task with a *Flow Relation*, as shown in Figure 4.7. A flow relation, most often referred to simply as a *flow*, is a directed arc (arrow) from one net object to another – it signifies the 'flow' of execution between objects. To draw a flow, first locate a flow connector (they appear as small crosses as you hover your mouse over the sides of net objects) on the Input Condition. Hold the left mouse button down over the flow connector and draw a flow by dragging the mouse to a flow connector on the edge of the Atomic Task (which will appear when the mouse hovers over the edges of the task). Release the mouse to complete the flow.

   - Tip: A flow connector is only shown when it is valid to draw a flow connection between the objects. Invalid connection attempts include those between two conditions, or multiple flow to/from tasks without the appropriate split and/or join (see the next section for more details).

   That's it! Your Atomic Task is set.

Repeat the process to insert the following Atomic Tasks in order: 'Go to University', 'Get A Job', 'Career Started'. Draw flows between them, and finally between the 'Career Started' task and the Output Condition (the ■ symbol, which signifies the concluding point of your process). You should end up with a completed net like the one in Figure 4.8.

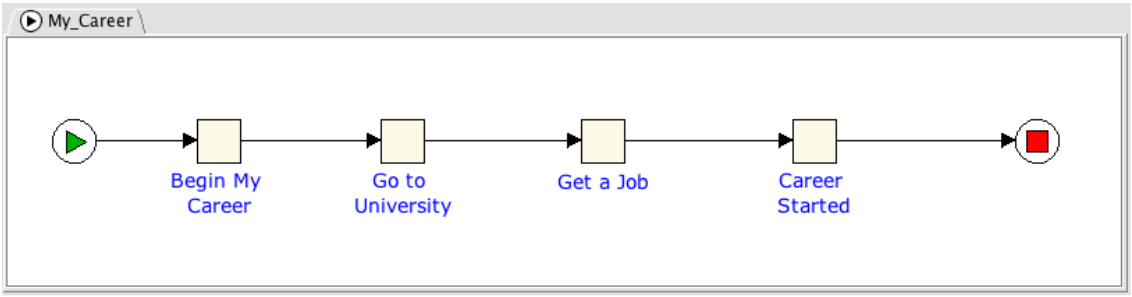


Figure 4.8: The "My Career" Net

5. Finally check the validity of specification by clicking on the Validate Specification button, ✔, in the Menu Toolbar or click on **Specification** in the Menu and choose **Validate**. If all things are going to plan, then you will receive a confirmation message in the Notes Pane at the bottom of the Editor reporting no problems were found.

4.4.3 Task Decorators

By default, each task may have exactly one incoming flow and exactly one outgoing flow, which allows only for simple, sequential control-flow arrangements. However, there are usually points in a net where it is desirable to *split* the flow into a number of flow paths, and others where it is necessary to *join* a number of paths into one path. Decorating a task is the process of adding a split and/or join to the task.

By adding a split decorator to a task, you are specifying that when the task completes, it will be succeeded by one or more tasks. Here are the choices for a task's split decorator:

- **No split**: The task has no split decorator, and so will have exactly one outgoing flow;
- **AND split**: The task may have one or more outgoing flows. When the task completes, it will activate *each and every* outgoing flow;
- **XOR split**: The task may have one or more outgoing flows, each with an associated boolean condition. When the task completes, it will activate *exactly one* outgoing flow – the first that has its condition evaluate to true, or the designated default flow if none of the other flow conditions evaluate to true;
- **OR split**: The task may have one or more outgoing flows, each with an associated boolean condition. When the task completes, it will activate *each* outgoing flow that has its condition evaluate to true, or the designated default flow if none of the other flow conditions evaluate to true;

By adding a join to a task, you are specifying at what point the task will become available for execution through the completion of one or more preceding tasks flowing into it (depending on the type of join). Here are the choices for a task's join decorator:

- **No join**: The task has no join decorator, and so will have exactly one incoming flow;
- **AND join**: The task may have one or more incoming flows, and will activate only after *each and every* incoming flow is activated (through the completion of the task at the other end of each flow);
- **XOR join**: The task may have one or more incoming flows, and will activate as soon as *one* incoming flow is activated (through the completion of the task at the other end of the flow);
- **OR join**: The task may have one or more incoming flows, and will activate only after each and every incoming flow *that can possibly be activated* has activated. Basically this means the completion of each and every task at the other end of a flow leading into the OR-join that has started or may possibly start at some future time. More on the OR-join in later sections.

For more detailed information on join and split types, please consult the YAWL Book or the technical papers on the YAWL website.

To create a split or join:

1. Select a task.
2. In the *Task* section of the Properties Pane, select the *Split Type* property and choose the required split type, and/or the *Join Type* property and choose the required join type, from the property's dropdown list. You can use the *Split Position* and *Join Position* properties to set the orientation of the decorator (i.e. which edge of the task to attach the decorator to).
    - TIP: A decorator toolbar can be displayed via *Plugins→Toolbars→Task Decorators* from the main menu. This toolbar has buttons to set the split and join type(s) and their position(s) for the selected task as a convenient alternative to setting them via the Properties Pane.

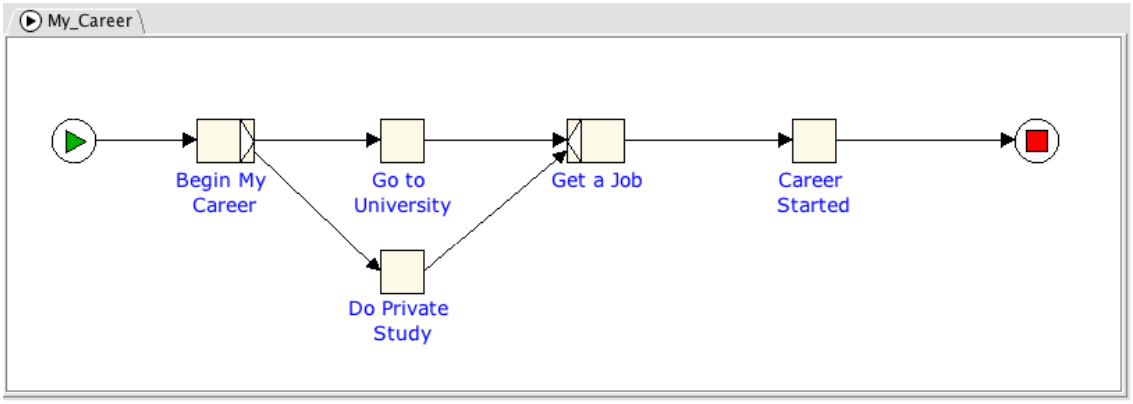


Figure 4.9: Adding an XOR Split and Join

In our example, select the "Begin My Career" task and set its *Split Type* property to **XOR** split. Its position will default to the eastern edge of the task, as per Figure 4.9.

Create a new Atomic task called "Do Private Study". This task will represent those students that choose not to go to University.

Finally, select the "Get A Job" task and decorate it with an **XOR** join. Its position will default to the western edge of this task.

3. Split and Join decorators allow you to connect several Flow Relations from and to your task respectively.

Create a flow relation from "Begin My Career" to "Do Private Study", then create another flow relation from "Do Private Study" to "Get A Job", as per Figure 4.9. Notice how the decorators allow more than one flow to be created to/from tasks. Also note that decorators provide multiple flow connection points along their edges, which are provided so that you can position flows as desired (it is also permissible to have several flows connect to the same connection point on a decorator).

4. Don't forget to check the validity of your specification.

Hint: If you are having trouble with positioning your tasks, the alignment tools are a big help.

Now, when the "Begin My Career" task has been completed, a choice must be made on which of the two tasks ("Go To University" or "Do Private Study") will be followed (via the XOR Split). How that choice is made will be explained a little later. "Get a Job" will become available after the completion of the task selected via the XOR split.

4.4.4 Composite Tasks

Composite tasks are placeholders for sub-nets. That is, you can create another workflow in a separate net, which is represented in the first (or *parent*) net by the composite task. When a composite task is activated in the parent net, control branches to the sub-net; when the sub-net completes, control passes back to the parent net via the completion of the composite task.

To create a Composite Task:

1. Click on the Composite Task button, , in the Palette or right click on an empty part of the canvas and choose **Composite Task**.

We are going to replace our existing "Go to University" Atomic Task with a Composite Task. Click on the "Go to University" Atomic Task and click the trash bin on the toolbar or press the Delete key on the keyboard to remove it from the net (when you delete a task, any flows connected to it are also removed). We will add in the new composite task next.

2. Place your Composite Task in your Net. *Tip: use the arrow keys on your keyboard to move/adjust the task to the desired location.*

Reconnect the flows from "Begin My Career" to the new Composite Task, and from new Composite Task to "Get a Job".

3. Create a new sub-net for the composite task by selecting the task, then go to the *Name* property in the *Decomposition* section of the Properties Pane, and provide a name for the new sub-net (by selecting *New...* from the dropdown list). A new sub-net will be created (a separate tab will appear for it on the canvas) and associated with the composite task.

We are going to call this new Net "Attend University".

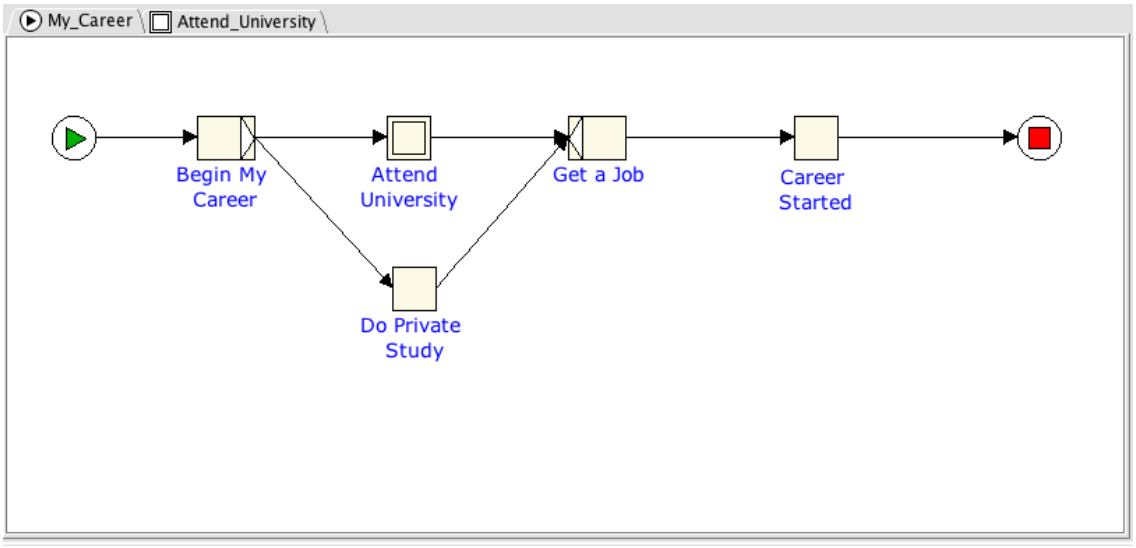


Figure 4.10: Root net with "Attend University" Composite Task

- A sub-net is a particular type of decomposition for a Composite Task, and so shares a number of the same kinds of properties with decompositions of Atomic Tasks.
- An alternative method to create a sub-net is by clicking on the *Create a New Net* button, , on the tool bar, or by clicking on the **Net** Menu and choosing **Add Net**. You should set the new sub-net's *Name* property to something more meaningful, then return to your original (root) Net (click on its tab), click on your Composite Task and choose the newly created sub-net from the dropdown list in the Decomposition's *Name* property.

Choose the "Attend University" net by clicking on its tab, or right-clicking on the Composite Task and selecting 'Go to sub-net' from the popup menu. You can now design the content of your new "Attend University" sub-net.

Create the following Atomic Tasks in order and then link them with flows, and don't forget to check for validity:

- Enrol
- Do Subjects
- Pass All Subjects
- Get Degree

The resulting nets are shown in Figures 4.10 and 4.11.

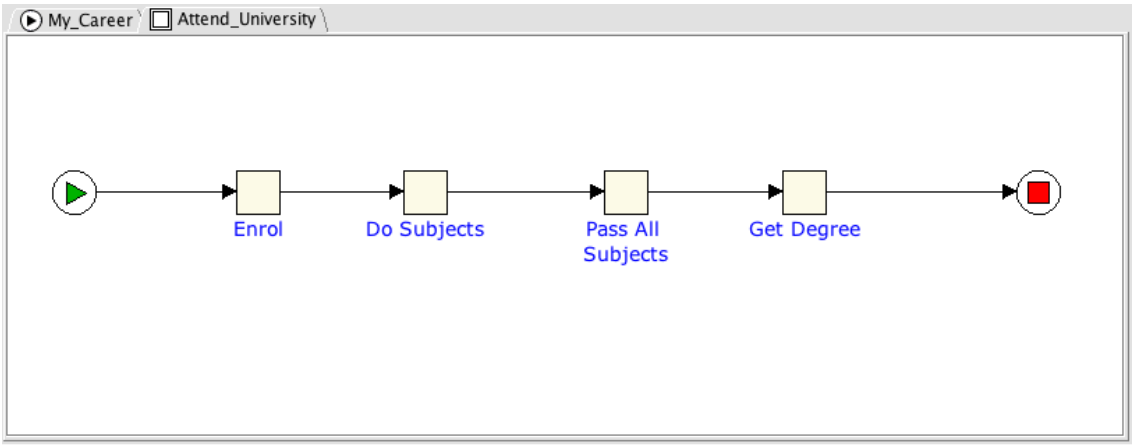


Figure 4.11: The "Attend University" sub-net

4.4.5 Multiple Instance Atomic Tasks

Multiple Instance Atomic Tasks (MI Tasks) allow you to execute multiple instances of a task concurrently. That is, from a single MI Task added to a net at design time, a number of instances of that task are instantiated at runtime, based on parameters set for the task and the data available to it.

To create a Multiple Instance Atomic Task:

1. Click on the Multiple Instance Atomic Task button, , in the Palette or right click in an empty part of the canvas and choose **Multiple Atomic Task**.

Go back to the "My Career" Net. We are going to replace our existing "Do Private Study" Atomic Task with a Multiple Instance Atomic task, so click on the "Do Private Study" Atomic Task and delete it. We will add in the new Multiple Instance Atomic task next.

2. Place a new Multiple Instance Atomic Task in your Net.

Set the name of this task by selecting the Decomposition's *Name* property and choosing the same decomposition as before, "Do Private Study", from the drop-down list.

Reconnect the flow relations from "Begin My Career" to "Do Private Study", and from "Do Private Study" to "Get A Job", as per Figure 4.12.

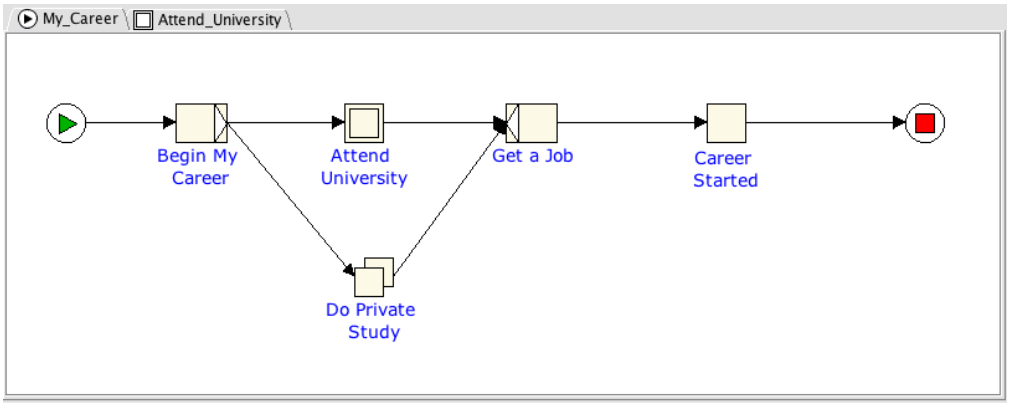


Figure 4.12: Adding a Multiple Atomic Task

3. You will now need to set the parameters of the Multiple Instance Task, which, because it will create multiple instances of the task at runtime, needs a few more values set than for a simple atomic task. Select the task and choose the Task's *M-I Attributes* property. Click the action button of the property (the small button to the right of the property's input field) to show the M-I Attributes dialog (Figure 4.13).

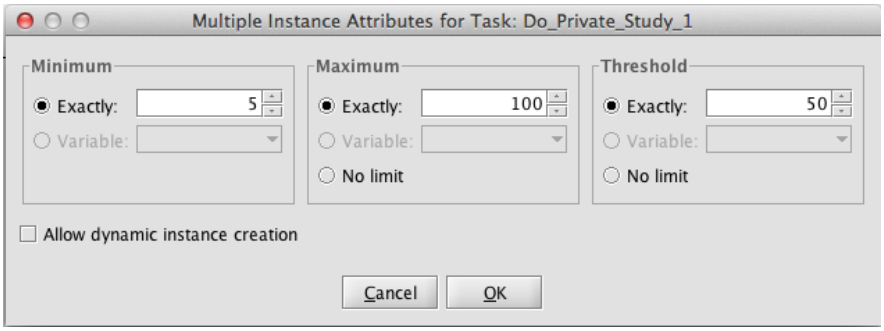


Figure 4.13: M-I Attributes dialog

(a) **Choose the Instance Creation mode**. Notice the *Allow dynamic instance creation* checkbox at the bottom of the dialog. When this checkbox is unselected, the instance creation mode is set to *Static*, which means the number of task instances to be started cannot vary once the task begins execution at runtime. When the checkbox is selected, the instance creation mode is set to *Dynamic*, which means the same number of task instances (as static mode) are started initially, but new instances of the task may be started dynamically at runtime (i.e. after task execution has begun), up to the value entered in "Maximum Instances" (see also Section 6.7.3 on page 172). In either mode, the number of task instances created at runtime for the task will be between the values given for "Minimum Instances" and "Maximum Instances".

Leave the *Allow dynamic instance creation* checkbox unselected (i.e. in static mode).

(b) **Set the Minimum Instances value.** This is the minimum number of instances of this task that will be started when the task is activated.

Set the Minimum Instances to 5.

(c) **Set the Maximum Instances value.** This is the maximum number of instances of this task that can be created from this task.

Set the Maximum Instances to 100.

(d) **Set the Threshold value.** The moment all task instances have completed, or if the number of instances created exceeds the Threshold the number specified for the Threshold have completed, the multiple instance task itself will be considered complete, and will trigger relevant outgoing flows from this task.

Set the Continuation Threshold to 50.

(e) Click **OK** to save your M-I attributes.

With the values set in this scenario, we have specified that the "Do Private Study" task can have a maximum of 100 instances created, a minimum of five instances will be created, and once 50 instances (or all those started if less than 50) have completed, the outgoing flow relation to ''Get A Job" will trigger. The actual number of task instances started at runtime for a process will depend on the data supplied to it; more on this later, and on the use of variables for M-I values, in Section 4.7.6, after the basics of the data perspective have been introduced.

4.4.6 Multiple Instance Composite Tasks

Multiple Instance Composite Tasks allow you to execute multiple instances of a sub-net, represented by the multiple instance composite task, concurrently at runtime, based on the parameters set for the task and the data available to it.

To create a Multiple Composite Task:

1. Click on the Multiple Composite Task button, , in the Palette or right click in an empty part of the canvas and choose **Multiple Composite Task**.

Go to the "My Career" Net. We are going to replace our existing "Do Private Study" Multiple Instance Task, with a Multiple Composite task, so click on the "Do Private Study" Task and delete it. We will add in the new Multiple Composite task next.

2. Place your Multiple Composite Task in your Net.

Reconnect the Flow Relations from "Begin My Career" to the new Multiple Composite Task, and from the new Multiple Composite Task to "Get a Job".

3. Next, create a new Net by clicking on the New Net button, , on the tool bar, or click on **Net** in the Menu and choose **Create Net**, or by creating a new decomposition for the task, as we did for the Multiple Instance Atomic Task previously.

4. Give the new Net a name.

We are going to call this new Net "Study Privately".

5. On the root net, right click on your Multiple Instance Composite Task and choose **Go to sub-net** (or simply click on the "Study Privately" tab).

6. You will now need to set the parameters of the Multiple Composite Task, in the same manner as those set previously for the Multiple Instance Atomic Task, via the Task's *M-I Attributes* property.

Set the Minimum Instances to 5, the Maximum Instances to 100, the Threshold to 50, and leave the Instance Creation type as "Static".

7. Click **OK**.

8. You can now complete your new "Study Privately" Net represented by your Multiple Instance Composite Task.

Create the following Atomic Tasks in order and then link them (and the Input and Output Conditions) with Flow Relations:

- Read a Book
- Feel Smarter

4.4.7 Conditions

Conditions represent states of the workflow that exist between the completion of a task and the starting of the next, and so are always located between two tasks. To create a Condition:

1. Click on the Condition button,  , in the Palette or right click on an empty part of the canvas and choose **Condition**.

Go to the "Study Privately" Net. We are going to place a Condition after the "Read a Book" atomic task, to determine whether we gained any knowledge from the book (that is, we will query the state of our process at that point).

2. Place your Condition in your Net and set its *Label* property.

Call this Condition "Knowledge Gained?".

3. Now link the Condition to the tasks of the net using flow relations.

Select the flow relation between the Read a Book atomic task and the Feel Smarter Atomic Task and delete it.

Create a flow relation from the "Read A Book" task to the "Knowledge Gained?" condition.

4. Create a flow relation from your condition to a task.

Set the flow relation from the "Knowledge Gained?" condition to "Feel Smarter" atomic task.

5. Create another flow relation from your condition to another task to signify the two possible flows from the condition.

Before we create our second flow relation from our condition, first create another atomic task and call it "Look for Easier Book".

Add an XOR join decoration to the "Read a Book" task, with a West position.

Then, create a Flow Relation from the "Knowledge Gained?" condition to the "Look for Easier Book" task, and another Flow Relation from that task back to the XOR join of the "Read A Book" atomic task, as shown in Figure 4.14.

Hint: to curve a Flow Relation, select it and set its Line Style property to either "Bezier" or "Spline", then move the drag point which appears on the Flow Relation to create the desired curve. You can add more drag points by right-clicking on the Flow Relation.

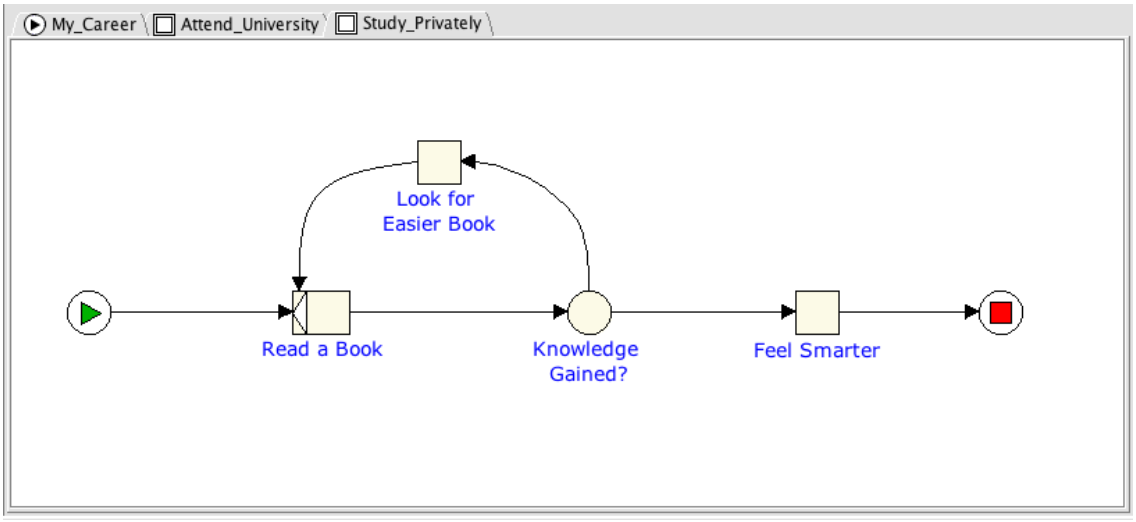


Figure 4.14: The "Study Privately" sub-net

The *Knowledge Gained?* condition in Figure 4.14 shows an example of a *Deferred Choice* construct. When the condition is reached during execution of the process, both of its outgoing flows are activated (a condition may have any number of incoming and outgoing flows). This results in both the "Look for Easier Book" and "Feel Smarter" tasks appearing in the user's work list, allowing the user to make a (deferred) choice between the two. As soon as the user chooses the appropriate task for execution, the other task is immediately withdrawn and is removed from the work list.

4.4.8 Changing the Starting Net

At any stage you can change the starting (root) Net of the specification. Every net in a specification has a *Root Net* property. You can use this property to promote a sub-net to be the new root net for the specification. To do so, select the appropriate sub-net and tick the property's checkbox. Note that you can't demote the current root net directly (its property is greyed out), since that will leave the specification without a root net, but promoting a sub-net will automatically demote the current root net to sub-net status.

Notice that the root net has an input condition symbol, ⊙, in its title tab. All sub-nets have a composite task symbol, ▣, in their title tab.

The Net Repository

The **Net** menu has three items that allow you to save, load and remove nets to and from the repository. This means you may save any net (root or sub-net) and all of its contents to the repository. Later, you can load any saved net from the repository into another specification. In this way, a saved root net can become a sub-net in a different specification, and vice versa.

To add the current net to the repository, select **Net**, then **Store in Repository...** from the menu. In the dialog that appears, give the stored net a name (the current net name is provided, but may be changed as desired), and a meaningful description, then click OK to save. The net and all its content will be saved to the repository under the name provided.

To retrieve a previously saved net from the repository, select **Net**, then **Load from Repository...**, select the name of the desired net from the list, then click OK. The net will be added to the current specification as a sub-net, which you can then promote to root net or assign to a composite task as required.

Finally, existing nets can be removed from the repository at any time by selecting **Net**, then **Remove from Repository...**, choosing the name for the net to remove from the list of stored nets, then clicking OK.

4.5 Changing the Appearance of Your Specification

At any time, you can change the appearance of the specification, and its nets and elements, on the canvas to aid in readability. This section discusses those features.

4.5.1 Flow Relations

Relocation You can reconnect flow relations to other elements of a net, or different points on the same element by selecting the flow, and dragging one of its connecting ends from one net element or position to another. If a connection is possible to some other element, connection points will become visible as described earlier. Release the mouse button to attach the flow to its new home.

Select the Flow Relation between "Look for Easier Book" and "Read a Book", and move its point end from the top of the task's XOR join to its side, as depicted in Figure 4.15.

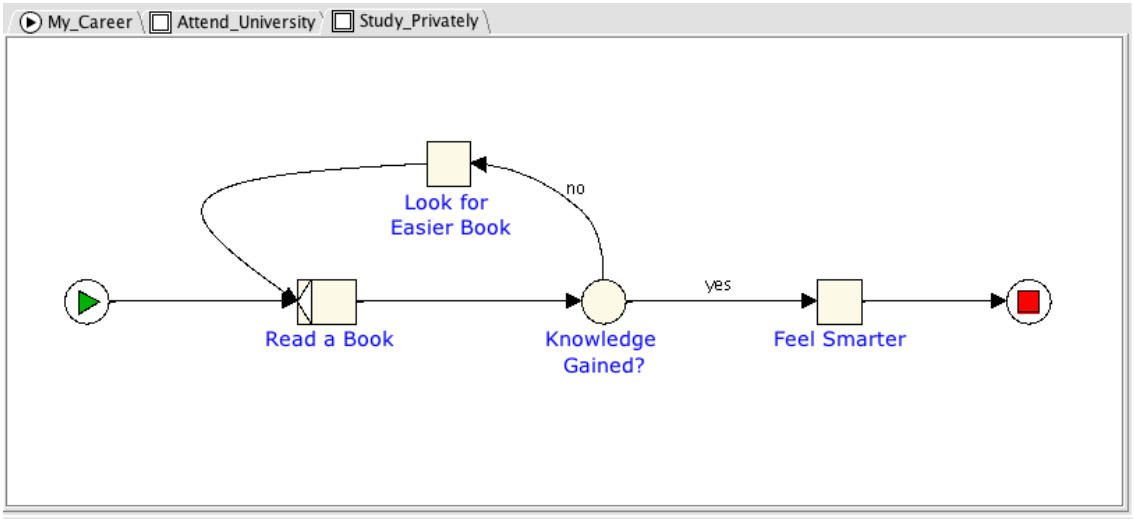


Figure 4.15: Adding bends and labels to Flow Relations

Adding Labels As well as tasks and conditions, it is also possible to add labels to flows. To do so, double click on a flow. A small text input box will appear over the flow. Type your desired text, and commit the flow label by pressing the ENTER key. You may then drag that flow label around to position it as desired.

Place labels on the two outgoing flow relations from the "Knowledge Gained?" condition. Attach the label *yes* to the flow relation going from the "Knowledge Gained?" condition to the "Feel Smarter" atomic task. Attach the label *no* to the flow relation going from the "Knowledge Gained?" condition to the "Look for an Easier Book" atomic task. Drag the labels about to a desired position, much like what's been done in Figure 4.15.

Line Style There are three line styles available for flows, *Orthogonal* (straight edged), *Bezier* (curved with sharper corners) and *Spline* (curved with smoothed corners). A line style can be set for a flow by selecting the flow, then choosing the style from the flow's Line Style property dropdown list, or right clicking on the flow and choosing the style from the popup menu. Additional bend points can also be added via the popup menu. Figure 4.16 shows flows using two different line styles. The outgoing flow from "Look for an Easier Book" has been given the spline style, while the remaining flows are all orthogonal, resulting in sharp edged bends on flows, such as the one running from the "Knowledge Gained?" condition to the "Look for an Easier Book" task.

4.5.2 Editing Multiple Objects

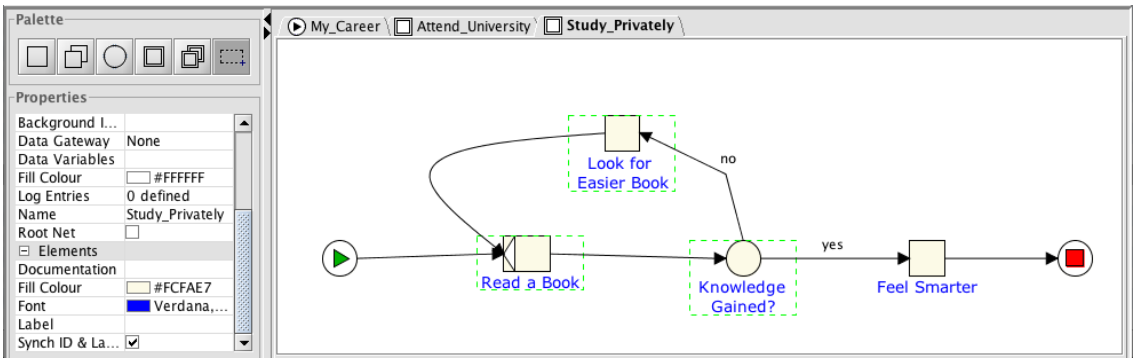


Figure 4.16: Selecting Multiple Objects

You can select more than one object at a time and edit their common properties in one action by using the Marquee Selection tool. See Figure 4.16.

1. Select the Marquee Selection tool, , from the Palette.
2. Click on the first object that you want to edit, then hold down the Shift key and then click on the other objects that you want to include. Alternately, click and drag the Marquee tool to include multiple items in the drag rectangle. To select all the elements in the net, you may also use Ctrl-A.
3. Now, choose **Edit** from the menu bar to select Cut, Copy or Delete options, or choose **Elements** from the menu bar then select Alignment, Set Colour, Increase Size, Decrease Size and options to change the appearance of the selected objects. The resizing options can also be accessed via the relevant tool bar buttons.

Depending on the types of elements selected, the Properties Pane also changes to display only those properties common to the selections, from which a property change will be applied to each selected element (for example, label fonts can be set for multiple elements in this way). Note also that whenever you have selected a number of net elements, pressing one of the arrow keys will move the selected elements a small distance in the direction of the arrow key.

4.5.3 Colours and Fonts

For nets, the default background colour can be set (i.e. applied to all nets) via the *Preferences* dialog by choosing *File... Preferences* on the top menu bar (see Section 4.9 for details). To set the background colour of individual nets, overriding the default colour, choose *Net Background Colour...* from the *Net* menu, or set the net's *Fill Colour* property.

For tasks and conditions, the default fill colour (i.e. for all newly added tasks and conditions) can also be set via the *Preferences* dialog. For individual tasks and conditions, select it then set its *Fill Colour* property. Several selected tasks and/or conditions can have their fill colour set at the same time by choosing *Set Selected Fill Colour...* from the *Elements* menu.

For tasks and conditions, the default font family, size and colour can also be set via the *Preferences* dialog. The default font settings can be overridden for individual tasks and conditions via their *Font* property (Figure 4.17).

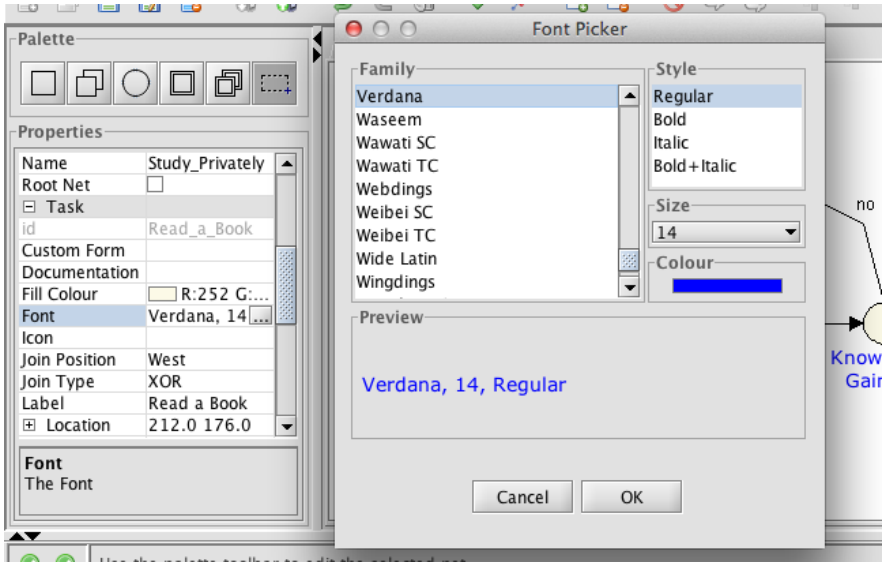


Figure 4.17: Font dialog

4.5.4 Task Icons

You can add, change or remove an icon to any atomic task, to signify or identify its intended action or to aid in readability. You are free to assign any icon, there are over 400 to choose from. Icons have no effect on how the task is instantiated at runtime.

To add or change a task icon, select any single atomic task, then select its *Icon* property. Choose the desired icon from the dropdown list. To remove an icon, select 'None' from the dropdown list.

Using Custom Icons You may also plug in and use your own icon sets. Icons must be of the PNG file format, and be a maximum of 24 × 24 pixels to render properly within task boundaries.

The editor will load user-supplied task icons from the location specified for them in the *File Paths*pane of the *Preferences* dialog (cf. Section 4.9); if never specified, the location defaults to the directory:

```
<editor_installation_path>/YAWLEditorPlugins/TaskIcons
```

All icons found in the specified location are added to the dropdown list of the *Icon* property for selection.

If a specification is loaded into the editor that contains a reference to an icon that cannot be found, a special "broken" icon will render in its place, as depicted in Figure 4.18.

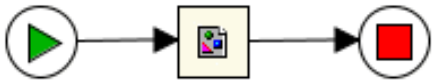


Figure 4.18: A task specifying an icon that the editor cannot locate

4.5.5 Task Indicators

Task indicators are mini-icons that appear across the top of a task to provide a visual cue regarding certain settings that have been applied to the task. An example of a task with all three possible indicators is shown in Figure 4.19.

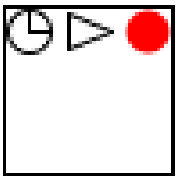


Figure 4.19: Task Indicators

The three task indicators (left to right in Figure 4.19) are:

- **Timer**: This task has had a timer set (see Section 4.13).
- **Automated**: This task has been set as automated (see Section 4.12). If this automated task also has a codelet specified, it will be filled green (see Section 4.12.1).
- **Cancellation Set**: This task has had a cancellation set defined (see Section 4.6).

These task indicators are rendered on top of any icons set for the task.

4.6 Cancellation Sets

Cancellation Sets allow you to nominate any number of tasks, conditions and/or flow relations (which, when they join two tasks directly, contain an *implicit* condition that is not visible on the net) for cancellation, upon the completion of a specified task. That is, once a specified task has completed execution in a workflow instance, all other net elements within that task's nominated cancellation set (if any) are deactivated.

To create a Cancellation Set for a task:

1. First select the task to create the Cancellation Set for.
2. Choose **View Cancellation Set** from the **Elements** menu, or click the View Cancellation Set toolbar button. The task will fill with a grey colour to indicate that this is the task that 'owns' the cancellation set currently on view.

Create a new "Purchase Book" specification as shown in Figure 4.20, noting that the "Get Book Details" task has an AND-split and the "Pay" task has an XOR-join. In this example, we are going to purchase a book by placing an order with three different sellers. As soon as the first seller fills the order, we want to cancel the other two orders. To achieve this, we create a cancellation set for each "Order" task that includes the other two "Order" tasks. We will step through creating a cancellation set for the "Order from Amazon" task – the other two are created in a similar manner.

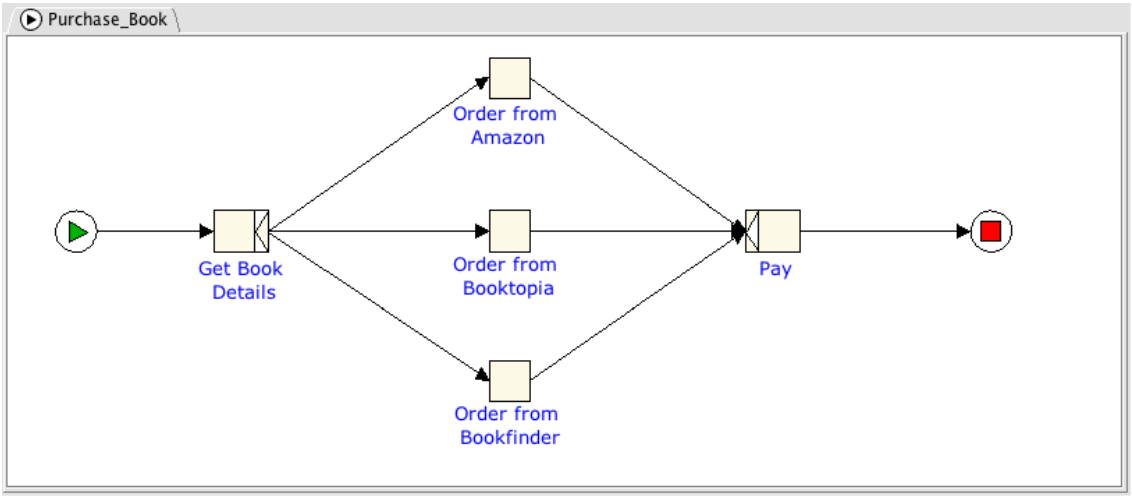


Figure 4.20: The Purchase Book specification

Select the "Order from Amazon" task and view its Cancellation Set using either method described above.

3. Next, choose a task, condition or flow (and thus an implicit condition) to add to the Cancellation Set. Hold down the shift key to select more than one object for cancellation.

Select the "Order from Booktopia" and "Order from Bookfinder" tasks, and the flow relation preceding each of them.

4. Click on the Add Selected Items to Visible Cancellation Set button,  , on the toolbar, or choose **Elements→Add to Cancellation Set** from the menu. Items will be given a red border to indicate they belong to the cancellation set of the grey-filled task (see Figure 4.21).

Add the selected tasks and flow relations to the cancellation set.

5. Once you have established the cancellation set, you can again choose **View Cancellation Set** from the Elements menu, or click the View Cancellation Set toolbar button,  , to toggle off the cancellation set view. The selected elements will return to their original colours, and a cancellation set task indicator will appear (a red dot in the top right) in the cancelling task to indicate that the task has a cancellation set defined.

Notes about Cancellation Sets:

- We gave the "Get Book Details" task an AND-split and the "Pay" task an XOR-join because we know that when one of the "Order" tasks completes, the other two will be cancelled, so in every case only one incoming flow to the "Pay" task will activate. Since we want the process to complete, we must add the join type that will activate the task when a single incoming flow activates: the XOR-join. If an AND-join had been used here, it would wait until all three incoming flows were activated, which in this case is never going to happen, and would result in a *deadlock* of the workflow instance. However, without the careful setting of cancellation sets for all three intermediate tasks, the net would represent an example of an *unsound* net, which basically means the net may complete while there were still active tasks within it. Great care needs to be taken when mixing split and join types, and when defining cancellation sets, so that the execution of the net behaves precisely as intended.

- A Cancellation Set that has been created will remain in the specification, regardless of whether you have the **View Cancellation Set** option ticked.

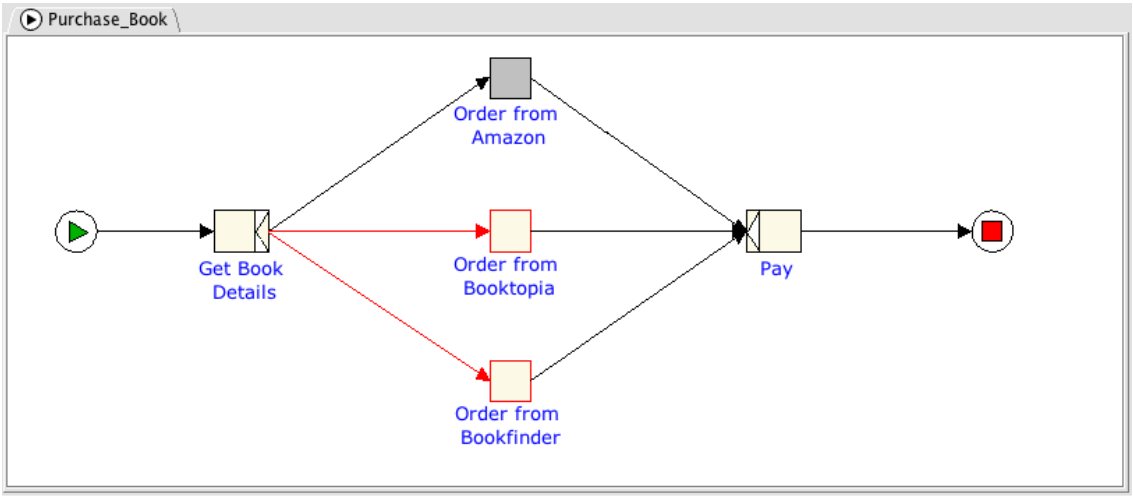


Figure 4.21: A Cancellation Set specified for the "Order from Amazon" task

- You can create multiple Cancellation Sets in your workflow, by selecting another task and choosing the **View Cancellation Set** option. Only one cancellation set may be viewed at any one time.
- All flows leading to or from (explicit) conditions are not valid cancellation set members. Neither are the Input and Output conditions. The editor will ignore them if you select them for inclusion in a task's cancellation set.
- A task may be included as a member of its own cancellation set.
- The reason for including preceding flows of a task in a cancellation set is this: If a flow relation connects two tasks directly, then it is said to contain an *implicit* condition. If there is a condition object in the model between two tasks, so that the connection is task - flow - condition - flow - task (for example, the "Knowledge Gained?" condition in Figure 4.15), it is said to be an *explicit* condition. In either case, when a task completes, it passes 'control' to the condition preceding the next task in the flow. When the next task is started, it takes 'control' from its preceding condition (whether implicit or explicit). If there is a chance that the tasks in a cancellation set may not have started when the owner task of the set completes, then cancelling those tasks will have no effect – it is their preceding conditions that have 'control' and so they are the elements that must be cancelled. By including both tasks and their preceding conditions, we are ensuring that the desired cancellation will occur, regardless of whether the tasks in the set are currently executing.

To remove an element from a task's Cancellation Set:

1. First, make sure you have the **View Cancellation Set** option selected for the task.
2. Select the element(s) for removal.
3. Click on the *Remove Selected Items from Visible Cancellation Set* button, , on the toolbar, or choose **Elements→Remove from Cancellation Set** from the menu.

4.7 The Data Perspective

A workflow process that describes the control-flow perspective only is of limited use. To get any real work done, data must be created, manipulated, transformed and passed to various users, applications and services throughout the process. The section discusses the YAWL Data Perspective.

In general terms, each net in a YAWL specification defines and controls a set of data variables that have values assigned to them during its life-cycle. Certain net variable values are passed to task-level variables using a *binding* expression or parameter when the task instance begins, with values passed backed from task-level to net-level using another binding expression when the task instance completes. Net-level variable values are also used at runtime to determine the choice of flow(s) to take from an XOR-split or OR-split through the evaluation of a *Split Predicate* associated with each flow.

All data is defined and stored as *XML Schema*. That is, all net-level and task-level variables store XML Schema data type values, and all binding and split predicate expressions use a combination of XQuery and XPath expressions to evaluate the data.

A great deal of effort has gone into designing the Editor to make it as easy as possible to create the variables and expressions required for a specification. In many cases, you don't need to be exposed to the underlying XML at all, but of course the Editor allows you to also "get your hands dirty" in XML whenever you need to.

4.7.1 Creating Net-Level Variables

You can add variables to a net to store information relating to that net that tasks within the net may need to read or update.

To add a variable to a Net:

1. Select the *Data Variables* property in the Net section of the Properties Pane, then click its Action button to show the Data Variables Dialog for the Net.

We will be adding Net variables in the "Attend University" net. Go to the "Attend University" net and choose its *Data Variables* property, then open the Data Variables Dialog (see Figure 4.22).

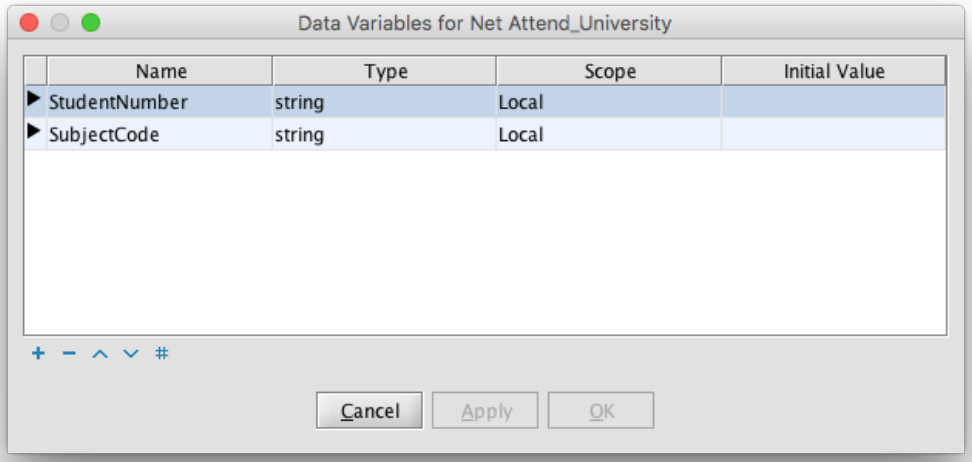


Figure 4.22: Adding "Attend University" Net Variables

2. Click the + button on the toolbar under the list area to begin adding a new variable. Enter a Name for the variable, choose the appropriate Type and intended Scope of the variable from those listed, then click **OK** to save the variable and close the dialog.

Add a new net variable: enter "StudentNumber" for the *Name* of the variable, leave the *Type* as

"string", the *Scope* as "Local", and the *Initial Value* blank. Create another net variable with the Name "SubjectCode", Type "string", Scope "Local" and Initial Value blank (scope types will be explained a little later). Click **OK** to save the new variables.

4.7.2 Creating Task-Level Variables

Task-level variables are not added to a task directly, but rather to its decomposition.

Task Decompositions

By choosing the *Name* property in the Decomposition section of the Properties Pane, or by entering a new name in that property, you are identifying which *task decomposition* will be associated with the task. A task decomposition describes the variables 'handled' by the task.

Like nets, tasks have decompositions where you can specify variables and a name to associate with the task. Unlike nets, which cannot share decompositions within a specification, there is a 1:N relationship between task decompositions (scoped to the entire specification) and their tasks (scoped to nets), meaning that a number of tasks within a specification may share the same decomposition.

Besides variables and a name, task decompositions also allow the workflow designer to identify which web service the decomposition should invoke in a running workflow engine when the task is initiated, and whether the decomposition will create manual (i.e. human-actioned) or automated (non-human-actioned) tasks. When two tasks share the same decomposition, we are saying that the same activity is required in two different places in the workflow (the two tasks may be named the same or differently, but they will share the same underlying definition of work).

From the Decomposition *Name* property, you can use the dropdown list to select an existing decomposition, or alternately you can select *New...* to create a new one that will become the task's decomposition (Figure 4.23).

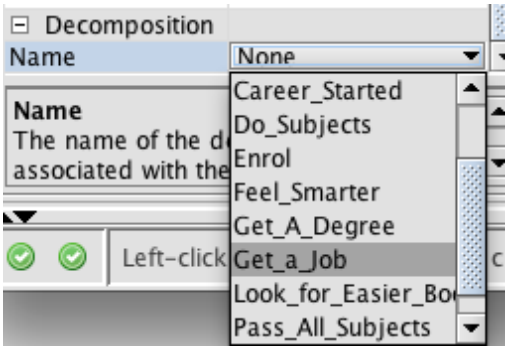


Figure 4.23: Select Task Decomposition detail (example "Get a Job" task)

You can add variables to a task decomposition (referred to as *task-level variables*) to store specific information relating to the task(s) associated with it, in a similar way to adding variables to a net. Task-level variables have several uses. One use is for transferring information between users and the process instance. A second use is for passing data between web services and/or external code and/or applications that the YAWL Engine invokes and the net the task resides in.

For example, if your task is called "Purchase a Book", you may want to store the name and/or ISBN of the book being ordered, so that information can be sent to a book store's web service.

To add a task-level variable to a decomposition:

1. First select the task that will require the variable.

We will be setting up variables for the "Enrol" task. Go to the "Attend University" Net and select the "Enrol" task. We have already created an empty decomposition for this task earlier in this chapter.

2. Select the Decomposition's *Data Variables* property, then click the Action button to show the Data Variables Dialog. When the dialog is invoked for a decomposition, it shows both the net-level and task-level variable lists (Figure 4.24).

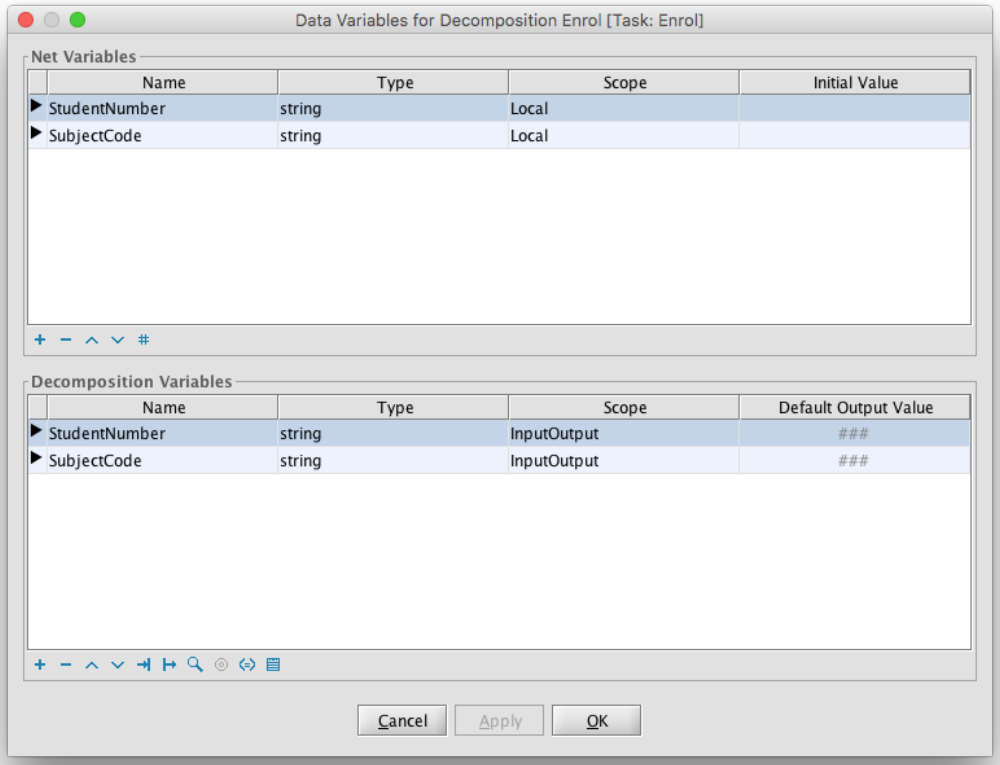


Figure 4.24: Adding Task-Level Variables

3. Enter the Name of your variable, choose the Type of the variable and its Scope from those listed, then click **OK** to save the new variable and close the dialog.

Enter "StudentNumber" for the name of the variable, leave the type as **string**, and the scope as "Input & Output". Create another variable for the same task called "SubjectCode" with type **string** and scope of "Input & Output". Click **OK** to save and close. The "Enrol" task now has two variables, "StudentNumber" and "SubjectCode" (Figure 4.24).

The Decomposition Repository

The **Elements** menu has a **Decomposition** item which provides three sub-menu items that allow you to save, load and remove decompositions to and from the repository. This means you may save any decomposition and all of its variables to the repository. Later, you can load any saved decomposition from the repository into another specification. In this way, a stored decomposition can be reused in other specifications.

To add the currently selected decomposition to the repository, select **Elements**, **Decomposition**, then **Store in Repository...** from the menu. In the dialog that appears, give the stored decomposition a name (the current decomposition name is provided, but may be changed as desired), and a meaningful description, then click OK to save. The decomposition and its content will be saved to the repository under the name provided.

To retrieve a previously saved decomposition from the repository, select **Elements**, **Decomposition**, then **Load from Repository...**, select the name of the desired decomposition from the list, then click OK. The decomposition will be added to the current specification, and will appear in the Decomposition Name property's dropdown list, from where it can be assigned to any task as required.

Finally, existing decompositions can be removed from the repository at any time by selecting select **Elements**, **Decomposition**, then **Remove from Repository...**, choosing the name for the decomposition to remove from the list of stored decompositions, then clicking OK.

4.7.3 Data Bindings

A *data binding* (also called a data parameter or mapping) defines how a value is assigned to a variable, and how a value is passed between net-level and task-level variables and vice-versa. Both Input and Output Parameters can be assigned to any tasks (depending on their scope) to allow the passing of data between nets and their tasks, and between tasks and YAWL Engine, users and web services. Data may also be assigned to and from net and/or task variables directly from an external data source (more later in this section).

Defining Bindings with XQuery

Bindings may be defined using *XQuery* expressions[1]. Input Bindings use an XQuery expression to specify a value (generally drawing on one or more net-level variable and/or static values) that can be passed to a single selected task-level variable. Output bindings use an XQuery expression to specify a value (generally drawing on one or more task-level variable and/or static values) that can be passed to a single selected net variable.

For example, if a task is called 'Lookup Book', then an Input binding could pass the name of the book from a net-level to a task-level variable, whereas the Output binding of that task may produce the corresponding ISBN for that book back to a net-level variable.

To add an **Input** binding:

1. Select the task to add the binding to.

   We will be setting up Input bindings for the variables that we created in the Creating Task-Level Variables section above. Go to the "Attend University" Net and select the "Enrol" task.

2. Select the task's Decomposition Data Variables property, an open its dialog.

   If you haven't set up Task variables for this task yet, please return to Section 4.7.3) to do so.

   In the Data Variables Dialog, notice that the decomposition variables are currently shown italicised – this indicates that we are yet to provide the required data bindings for those variables. Select the "StudentNumber" variable from the list of existing decomposition variables, then click the Input Bindings dialog button on the lower toolbar *(Hint: To select the whole row, click on the 'arrowhead' to the row's left)*. The Input Bindings dialog will appear.

   The dialog consists of 4 main parts (top to bottom):

[1] An examination of the XQuery language is beyond the scope of this chapter; good XQuery learning resources can be found at www.w3schools.com/xquery/default.asp and www.xquery.com/developers/

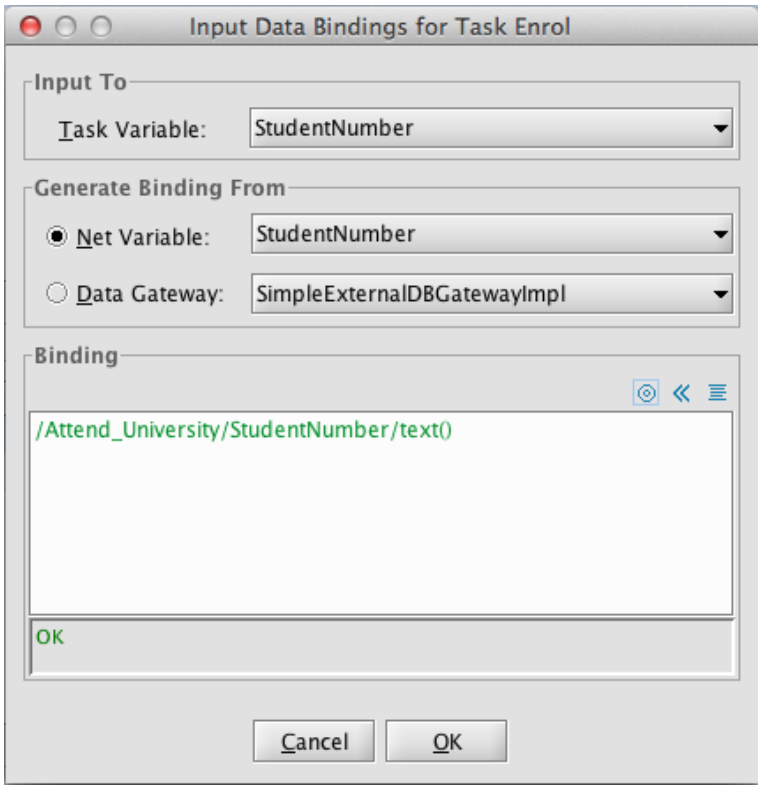


Figure 4.25: The Input Bindings Dialog (with auto-generated binding)

- The *Input To* panel contains a dropdown list of all task variables defined for the selected task, allowing you to choose which variable to provide an input binding for. The variable that was selected on the Data Variable dialog when the Input Binding Dialog button was clicked will be selected initially in the *Input To* list.
- The *Generate Binding From* panel allows you to select a source variable to use as the basis for an automatically generated binding. Two lists are available, Net Variables and External Data Gateways (more on these a little later). If there is a net variable of the same name and data type as the selected task variable, it will be initially selected in this list. If you intend to enter your own XQuery expression manually for the binding, you can ignore the contents of this panel.
- The *Binding* panel, where the XQuery expression for the binding is entered. There are three toolbar buttons in this panel:
  - The **Auto Binding** button: click this button to generate and insert a valid binding from the selected *Generate Binding From* source.
  - The **Reset** button: click this button to revert back to the XQuery that existed when the dialog was opened (if any).
  - The **Auto Format** button: to reformat the layout and indentations of large XQuery expressions.
- The validation message pane.

If you are familiar with XQuery syntax, then you can add an XQuery expression directly to perform the assignment of values to Input variables. When entering an XQuery, it is continuously validated for syntactic "well-formedness" and, when it meets that criterion, further validated to ensure it is assigning a value of the correct data type for its target variable. Remember to watch the message pane at the bottom of the dialog for helpful messages. Valid expressions will be denoted by a green "OK" in the message pane.

Check that the "StudentNumber" task-level variable is selected in the *Input To* panel, and the "StudentNumber" net-level variable is selected in the *Generate Binding From* panel, then click the Auto Binding button. Notice that an appropriate XQuery has been generated that extracts the value of the selected variable from the net-level. Click **OK** to save the binding and exit the dialog.

Now, select the "SubjectCode" decomposition variable, and add an input binding for it using the same technique as described above. Again, when done click **OK** to save the binding and exit the dialog.

To add an **Output** Binding:

1. Select the task to add the binding to.

We will now be adding the Output bindings for the variables that we created in the *Creating Task-Level Variables* section previously. If you haven't already done so, go to the "Attend University" Net and select the "Enrol" task.

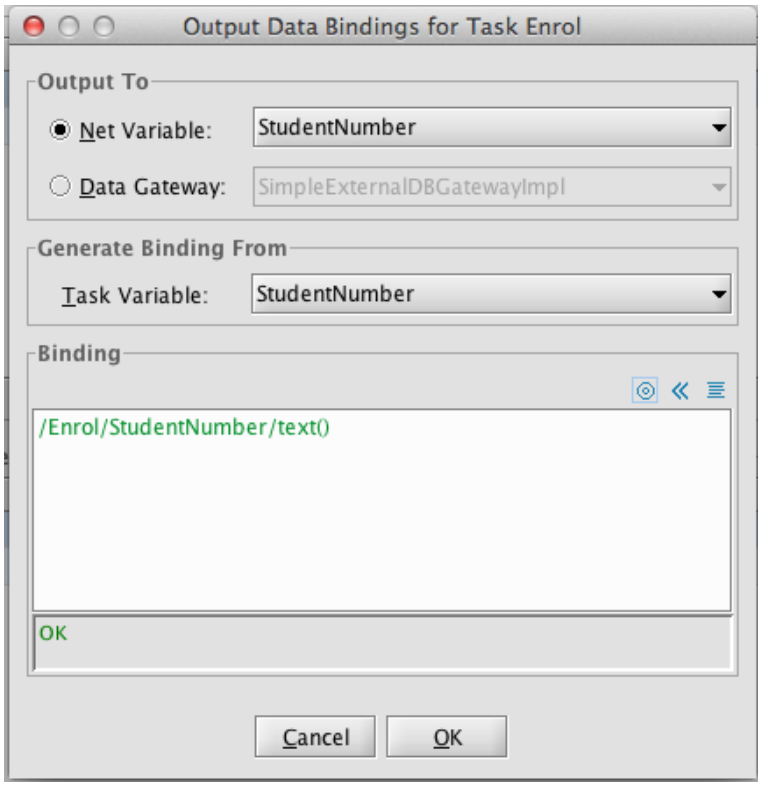


Figure 4.26: The Output Bindings Dialog (with auto-generated binding)

2. Select the task's Decomposition *Data Variables* property, an open its dialog.

If you haven't set up Task variables for this task yet, please return to Section 4.7.3) to do so.

In the Data Variables Dialog, select the "StudentNumber" variable from the list of existing task-level variables, then click the Output Bindings dialog button on the lower toolbar. The dialog will appear.

Like the Input Bindings dialog, the Output Bindings dialog consists of 4 main parts (top to bottom):

- The *Output To* panel allows you to select a target variable to provide an output binding for. Two lists are available, Net Variables and External Data Gateways. If there is a net variable of the same name and data type as the selected task variable, it will be initially selected in this list.
- The *Generate Binding From* panel contains a dropdown list of all task variables defined for the selected task to use as the basis for an automatically generated binding. The variable that was selected on the Data Variable dialog when the Output Binding Dialog button was clicked will be selected initially in this panel. If you intend to enter your own XQuery expression manually for the binding, you can ignore the contents of this panel.
- The *Binding* panel, where the XQuery expression for the binding is entered. There are three toolbar buttons in this panel:
  - The **Auto Binding** button: click this button to generate and insert a valid binding from the selected *Generate Binding From* source.
  - The **Reset** button: click this button to revert back to the XQuery that existed when the dialog was opened (if any).
  - The **Auto Format** button: to reformat the layout and indentations of large XQuery expressions.
- The validation message pane.

If you are familiar with XQuery syntax, you can add an XQuery expression directly to perform the assignment of values to Output variables. When entering an XQuery, it is continuously validated for syntactic "well-formedness" and, when it meets that criterion, further validated to ensure it is assigning a value of the correct data type for its target variable. Remember to watch the message pane at the bottom of the dialog for helpful messages. Valid expressions will be denoted by a green "OK" in the message pane.

Check that the "StudentNumber" variable is selected in the *Output To* panel, and the "StudentNumber" task variable is selected in the *Generate Binding From* panel, then click the Auto Binding button. Notice that an appropriate XQuery has been generated that extracts the value of the selected variable from the task-level. Click **OK** to save the binding and exit the dialog. Notice that in the list of Decomposition Variables the "StudentNumber" variable is no longer italicised, denoting that it now has valid input and output bindings defined

Now, select the "SubjectCode" decomposition variable, and add an output binding for it using the same technique as described above. Again, when done click **OK** to save the binding and exit the dialog.

Smart Data Bindings

Smart Data Bindings provide an easy way to create data bindings between net-level and task-level variables. Let's see an example of how they work.

1. If your following on from the previous section, open up the Data Variables Dialog for the "Enrol" task of the "Attend University" net, and remove both task-level variables *(Hint: Use the – button on the tool bar at the bottom of the variable list)*. Click **Apply** to save the changes *without* closing the dialog.
2. Now, re-add the two decomposition variables (see Section 4.7.3 for a reminder if needed).
3. You should now have two task-level variables of the same name and data type as the two net-level variables. Notice that the task-level variables are italicised to remind you that they don't yet have the required data bindings defined.

4. Click the *Smart Data Bindings* button, , on the toolbar under the task-level variables list. Both the Input and Output bindings for both variables have been automatically generated – you can check by looking in the Input and Output Bindings dialogs for each task-level variable.

Smart Data Bindings works by searching for a net-level variable of the same name and data type of each decomposition-level variable that has missing data binding(s). If it finds a match, it will auto-generate the required bindings.

Adding Task Variables using Drag'n'Drop

The editor provides a third and even easier way to create task-level variables: the drag'n'drop method.

1. Let's delete our task-level variables again from the Data Variables Dialog for the "Enrol" task of the "Attend University" net. Click **Apply** to save the changes without closing the dialog.
2. Now, say you want to bind the "StudentNumber" net-level variable to the "Enrol" task. Simply select the net-level variable, then holding the mouse button down, drag and drop the variable onto the task-level variable list.
3. A task-level variable of the same name and data type, with InputOutput scope and with all bindings generated, is created. Do the same for the other net-level variable, "SubjectCode". Click **OK** to save the changes and close the dialog.

Once you have dropped the net-level variable into the task-level variable list, you are free to change the name and/or scope of the new task-level variable, but changing the data type may mean that any bindings are no longer valid.

Note that at any time while the Data Variable dialog is open from the Decomposition *Data Variables* property, the net-level variables list can also be edited (i.e. new net-level variables can be added, and existing ones updated or removed).

The Bindings Summary Dialog

You can view a summary of all input and output bindings for a task's variables from the Data Variable Dialog by clicking on the *Quick View Bindings* button, , on the task-level variable list toolbar. An example of the dialog can be seen in Figure 4.27.

The dialog presents a read-only view, but the toolbar under each list provides two buttons, the first for opening the selected binding in the appropriate Bindings dialog for editing, the second for removing the binding from the variable (which won't be necessary in most cases).

Now that we have an understanding of net-level and task-level variables, and how to create bindings to map values between them, we can revisit the earlier example from Section 4.7.3, where we created two local variables for the sub-net "Attend University". By creating them with *Local* scope they are actually different variables than those of the same name created in the outer (parent) net "My Career". If you wanted to map the values of those variables from the parent net to the sub-net, then their scope in the sub-net will need to be changed from *Local* to *Input Only* (since they are not to be updated in the sub-net's tasks), then perform the appropriate mapping between the parent net and the "Attend University" composite task, following one of the methods described above.

Defining Parameters using External Data Sources

As an alternative to mapping parameter values from net-level to task-level and back again, task (and net) variables may be assigned values directly from an external data source on starting and be directly mapped

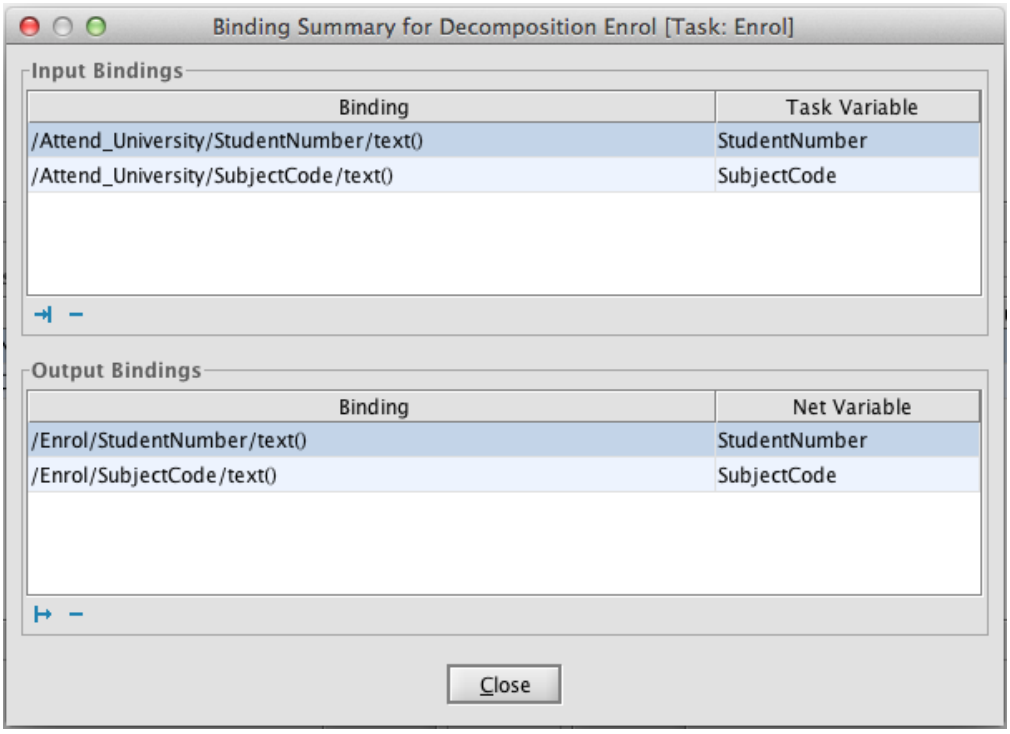


Figure 4.27: The Bindings Summary Dialog

back on completion. External data sources are accessed via a specific *External Data Gateway*. For task data bindings, the list of available external data gateways shown in the Input and Output Bindings dialogs, from where one can be chosen as an alternate source and/or target of task variable values. For the net-level, gateways can be selected via the Net's *Data Gateway* property dropdown list.

SimpleExternalDBGateway is an example gateway that ships with YAWL. Any other gateways implemented by developers will also be listed (see *Note* below).

A net-level external data gateway may also be chosen, so that each time a net is started for a specification the chosen data gateway will be called to populate its net-level variables, and when each net completes, the data gateway will be called to update output values from the case back to the specified external data source.

Note: Specific data gateways must be created for each specification that wishes to access an external data source directly. How to create a data gateway is outside the scope of this manual. Please refer to the YAWL Technical Manual for more details.

Notes about bindings

- For simple assignments, such as those in Figures 4.25 and 4.26, the XQuery expressions for input bindings follow the form {/name_of_net/name_of_variable/text()} and are bound to a task variable, while those for output bindings follow the form {/name_of_task/name_of_variable/text()} and are bound to a net variable.
- An *Input* Scope mode means that the variable requires a value (from a net-level variable or external data gateway or a literal) to be mapped into it when its task starts (via an input binding). An *Output* Scope means that the variable is required to map a value from it (to a net-level variable or external data gateway) when its task completes (via an output binding). An *InputOutput* Scope combines both requirements.
- In addition to *Input*, *Output* and *InputOutput*, net-level variables may have *Local* Scope, which signifies accessibility within the net but not external to it. Thus, sub-nets require net-level variables with modes

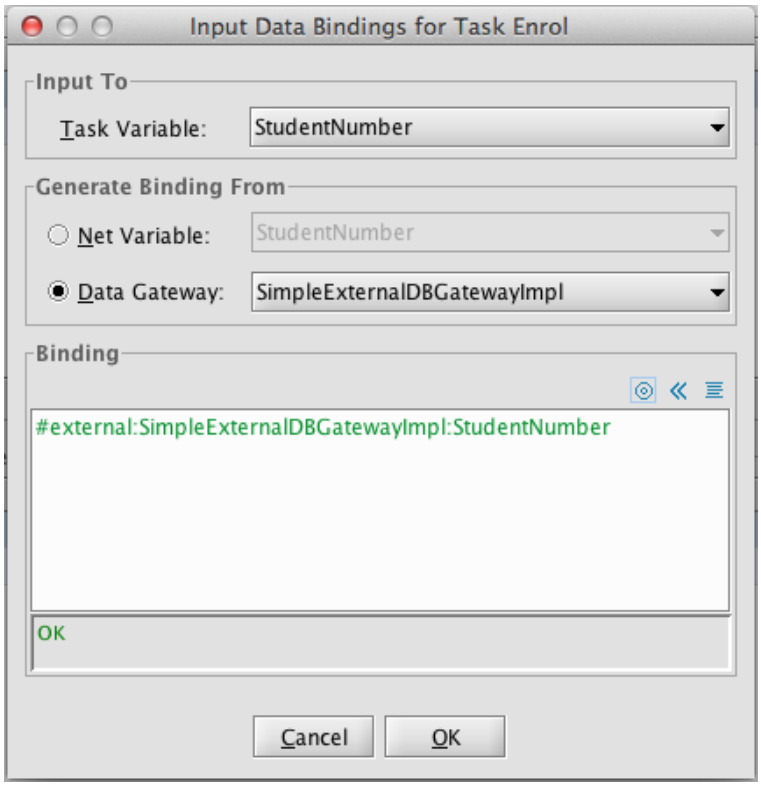


Figure 4.28: Task input bindings with external data gateway mapping

other than *Local* to support data passing to and from their parent nets. A root (or top-level) net with variables of type *Input* or *InputOutput* will, when started, request values for those variables from a user via a form, before the first task in the net is activated. No action is taken for *Output* scopes set for root net variables.

- Input bindings may be created only for variables of scopes *Input* or *InputOutput*. For task-level variables, all input variables must have a binding that maps a value to it.
- Output bindings may be created only for variables of scopes *Output* or *InputOutput*. For task-level variables, all output variables must have a binding that maps a value from it.
- A single task may map some values via XQuery and others via external data gateways. That is, data can be sourced from either net-level variables, external data gateways or both for the variables of a single task.
- It is possible to combine several task-level output variables into a single, complex data type to pass to a net-level variable of that data type. That is, there need not be a 1:1 mapping between task-level output variables and net-level variables, the only requirement is that there is some binding expression(s) to contain bindings for every task-level variable with output scope.

4.7.4 Split Predicates

When dealing with tasks that have XOR and OR splits, we need some way of defining which flow(s) should be activated at runtime. This is achieved by associating a boolean XQuery expression with each flow. At runtime, the flow expressions are evaluated and:

- if the split type is an OR-split, *each* flow that has an expression that evaluates to true will be executed.

- if the split type is an XOR-split, the *first* listed flow that has an expression that evaluates to true will be executed.

Since it is possible that all flow expressions may evaluate to false in a given process instance, XOR and OR splits must nominate a *default* flow, which will activate if all of the other flow expressions evaluate to false, to ensure that the workflow does not deadlock (i.e. is not blocked at that point from proceeding and eventually completing). Default flows are defined by prioritising the order in which the various flows of a split are evaluated – the one prioritised last in the order becomes the default flow.

To update the split predicates of a task that has a split, select the task and choose the Task *Split Predicates* property. The Split Conditions dialog appears, which lists the targets of the outgoing flows of the split and each flow's corresponding predicate (or flow condition).

The arrowed buttons at the bottom of the list allow you to reorder the evaluation sequence of the predicates, so that the default predicate (the one you want to have activated when all others fail) can be placed at the bottom of the list. Carefully ordering the evaluation sequence is especially important when dealing with an XOR-split, because only the first that evaluates to true will be activated, and all subsequent flows will be ignored.

The currently selected flow in the dialog will be identified by being highlighted green in the Net (Figure 4.29).

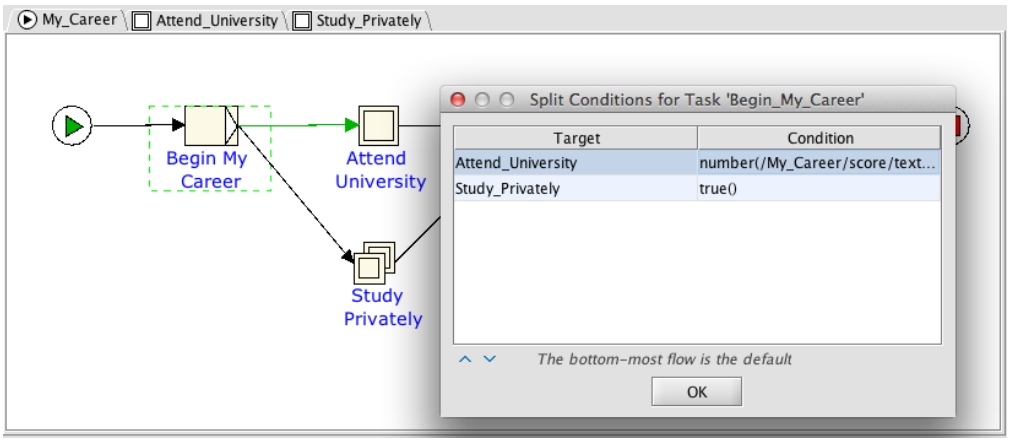


Figure 4.29: Split Conditions Dialog

Predicates typically include chosen net-level variables so that their current values can be evaluated, thus allowing contextual decisions to be made. Because split predicates are evaluated after the completion of the task with the split, its variable are no longer available and so can't be included in predicate expressions.

A predicate for a particular flow can be entered directly into the dialog, or you can click the Action button next to a selected predicate to open the Update Predicate dialog (Figure 4.30). You can enter a predicate as a boolean XQuery expression directly, and/or use the Generate Predicate button, ⚙, to help build a data expression using the selected net-level variable from the dropdown list. Click **OK** to save the changes and close the dialog, then **OK** again on the Split Conditions dialog. (Note that a net-level local integer variable called 'score' has been introduced to the net in the example for the purposes of showing how to create a boolean XQuery expression for a flow predicate; it is not used again in this tutorial).

Timer Predicates

Timer predicates are special (non-XQuery) expressions that may be used as flow predicates. For each task that has a timer associated with it (cf. Section 4.13) an implicit, net-level timer-state variable is created and maintained at runtime. At any particular time during the execution of the net, a timer-state variable can have one of four values (Table 4.1).

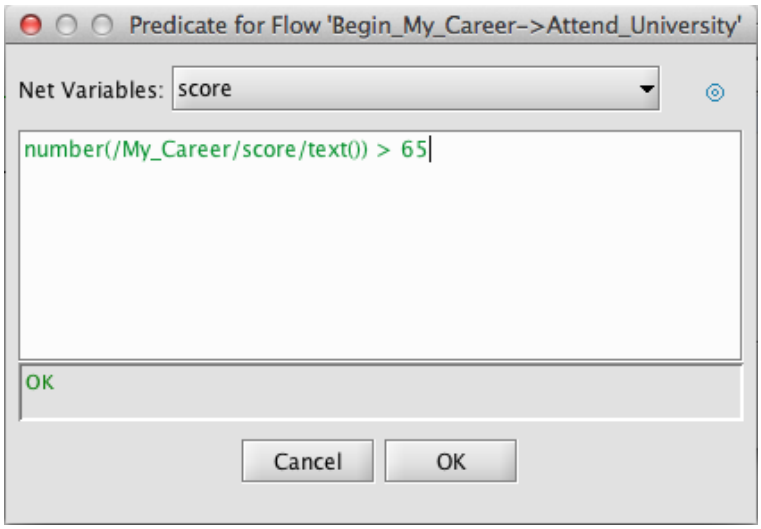


Figure 4.30: Predicate Update Dialog

dormant	Timer has not yet started
active	Timer is running
closed	Task completed before timer expired
expired	Timer expired before task completed

Table 4.1: Valid states of timer state variables

A timer predicate can have one of two operators, **=** (equals) or **!=** (not equals), and takes the form:

timer(*name_of_task*) operator '*timer-state*'

For example, assuming a task called 'Enrol' has a timer, then on any outgoing flow from an OR-split or XOR-split on any task in the net that also contains the 'Enrol' task, the following example timer predicates are valid:

- timer(Enrol) = 'dormant'
- timer(Enrol) ='active'
- timer(Enrol)='expired'
- timer(Enrol) != 'expired'

Notes:

- Timer predicates are case-sensitive (including the 'timer' keyword, the name of the task and the state value).
- Timer predicates are specialised split predicates, and are available only for use as split predicates.

4.7.5 Data Type Definitions

As mentioned above, YAWL uses XML Schema to define data documents that are passed from net to task and back during the life of a workflow instance. There are over 40 simple XML Schema data types (string, integer, boolean, etc.), all of which are supported by YAWL.

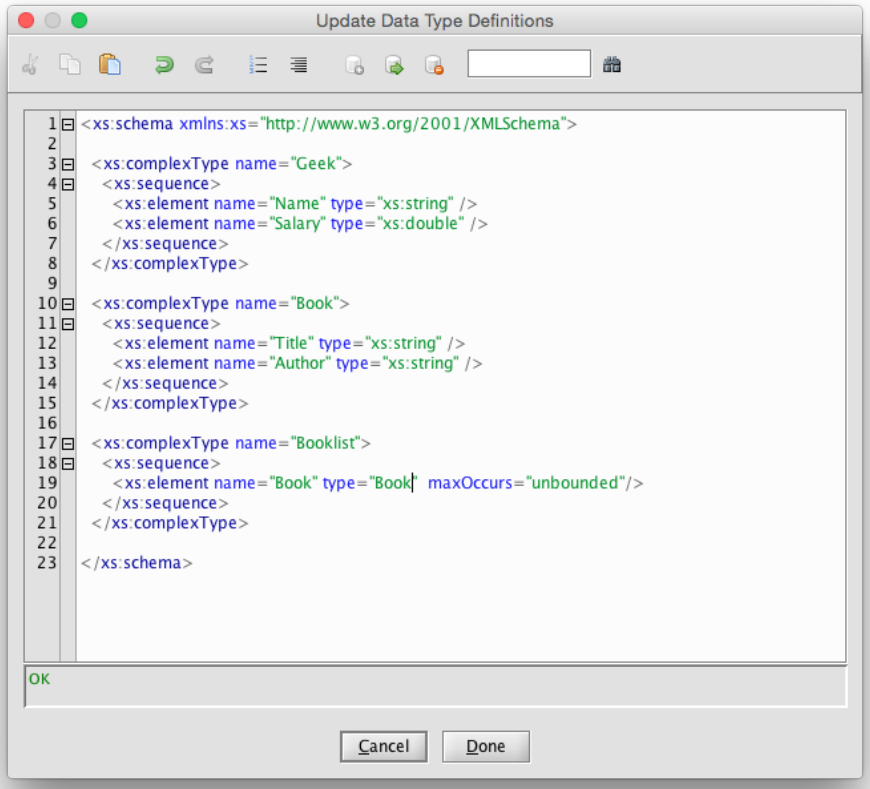


Figure 4.31: Adding the "Geek", "Book" and "Booklist" complex data types

While the simple data types are often sufficient to support process needs, the Editor also provides for the definition of user-defined simple and complex data types, which may be added to a specification and then used to define variables based on those types. To define a new type for a specification:

1. Select the *Data Definitions* property in the Specification section of the Properties pane, then click on the Action button (at the right of the property) to open the Data Definitions Dialog.
2. Enter an XML Schema Data Type Definition into the dialog. (See Figure 4.31). Any valid XML Schema definitions are acceptable.
3. As you enter your schema, it is continuously validated. Remember to check the messages pane at the bottom of the dialog for helpful messages about current errors in the schema, including the line and column numbers where the error occurs. An example of this is shown in Figure 4.32. When the schema passes validation, the pane will show a green "OK" message and the **Done** button will be enabled. The schema may now be used to define Net or Task data variables in your specification.

Open the Data Type Definitions dialog and type in the XML text that appears in Figure 4.31.

The Data Type Definition dialog comes with its own toolbar (Figure 4.33). From left to right, the buttons are:

- Cut, Copy, Paste text;
- Undo, Redo changes;

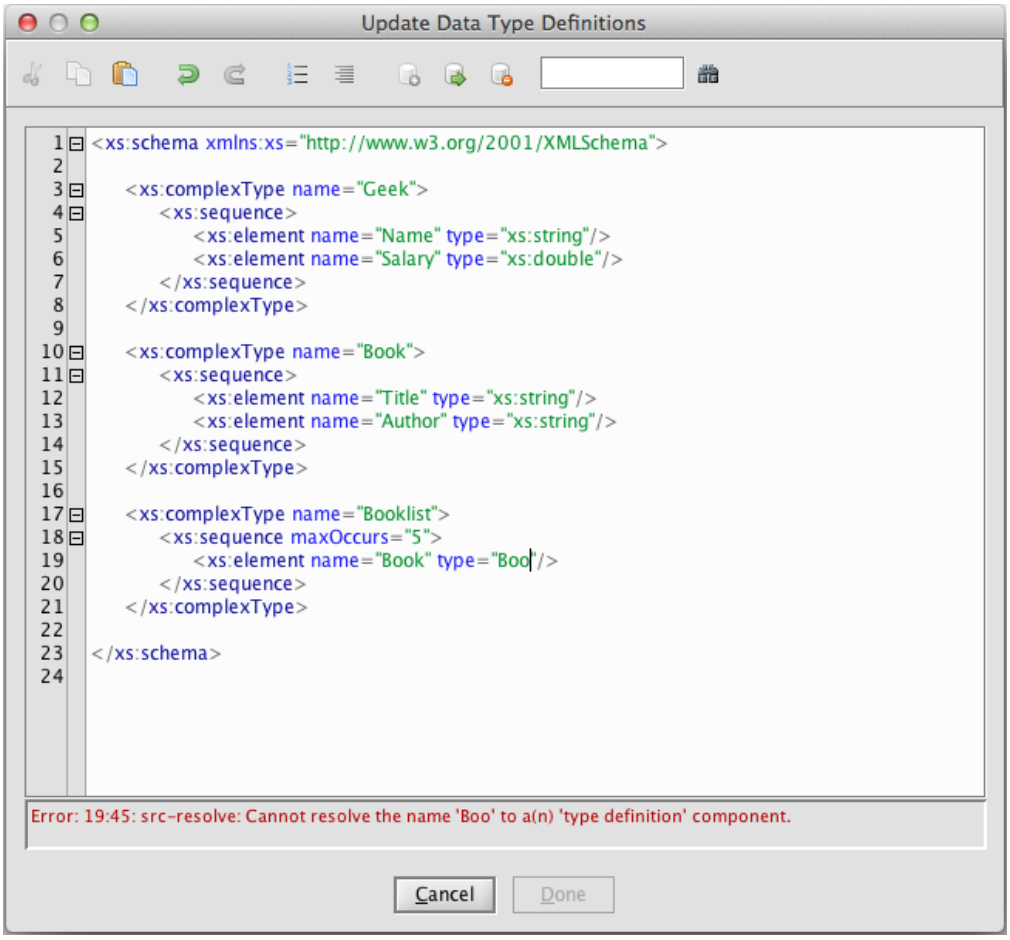


Figure 4.32: Example of an invalid data type definition

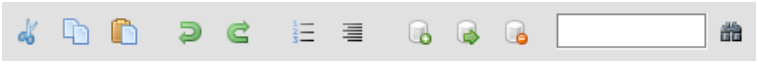


Figure 4.33: The data type definition dialog toolbar

- Toggle the viewing of line numbers on and off;
- Format text (fix indentation etc.);
- Store, Load and Remove selected definitions to and from the Repository – this allows data definitions to be reused in other specifications (see below).
- Search for text.

The example in Figure 4.31 creates a complex data type called “Geek” that has two separate sub-components, “Name” and “Salary” of type “string” and “double” respectively. Types called ‘Book’ and ‘Booklist’ are created in the same way. As depicted in Figure 4.34, the new data type “Geek” is available to choose from the list of available types when creating a task or net variable.

Net variables with a usage of “Local” can have initial values specified for them, as depicted in the same figure. As with the data type definition dialog, validation errors will appear when the initial value text is invalid for its data type schema.

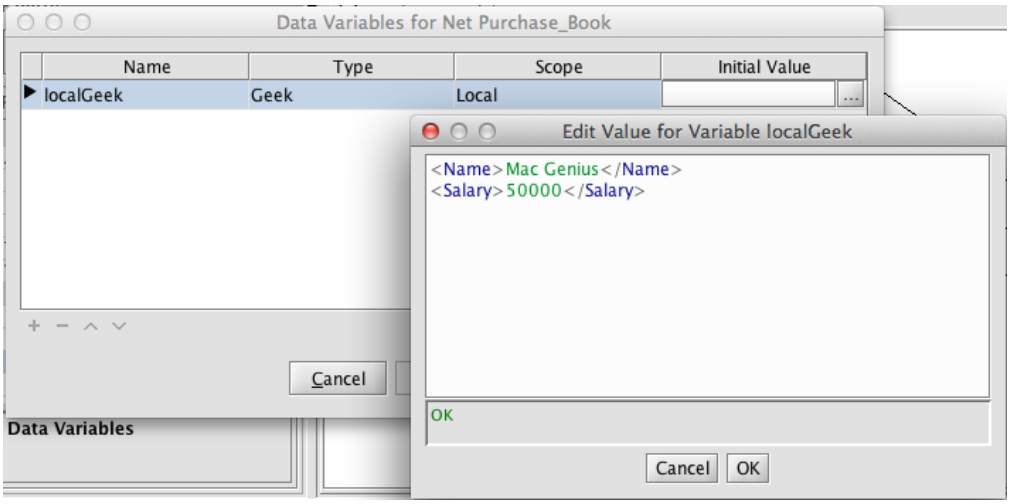


Figure 4.34: A "Geek" net variable with a valid initial value

The Data Type Definition Repository

As mentioned above, the Data Type Definition dialog provides three buttons on the dialog tool bar that allow you to save complete or partial data type definitions to the repository and later retrieve them for other specifications.

To add a data type definition to the repository, first select it in the dialog's editor then click the 'Add to Repository' button ( ) Add a label and meaningful description for the definition to the dialog that appears, then click OK to save.

To retrieve previously saved data type definitions from the repository, first place the cursor at the desired insertion point, then click the 'Load from Repository' button ( ). Select the label for the desired definition from the list, then click OK. The previously saved definition will be inserted at the cursor.

Finally, existing data type definitions can be removed from the repository at any time by clicking the 'Remove from Repository' button ( ), selecting the label for the definition to remove, then clicking OK.

4.7.6 Multiple Instance Queries

Now that we have an understanding of data bindings and XQueries, we can revisit, from a data perspective, binding definitions for the particular requirements of Multiple Instance (MI) Tasks. In general terms, an MI task will contain a number of variables, one of which is an Input or InputOutput variable of complex data type, typically a list of some other data type (but more complex constructions are of course supported too), which is 'marked' at design time as the *formal multiple instance input variable*. When the task is instantiated at runtime, that variable's data is *split* by the YAWL Engine into a number of logically distinct data values. The Engine then uses those values to instantiate a number of work items (task instances), one for each distinct data value, assigning that value to the bound task-level variable (any other task-level variables have their data assigned to them in the usual way). When the MI task completes, it gathers all the individual pieces of data from the formal input variable of each task instance and reconstructs the complex type variable so that it can be mapped back to a net-level variable (usually, but not necessarily, the same one that the data was mapped from).

To illustrate the operation of MI tasks, with particular emphasis on the data perspective, we will use the "List Builder" specification shown in Figure 4.35, which begins by compiling an 'order' – a list of book titles. It then creates a number of MI task instances, one for each book title in the list of books. Once all the MI task instances complete, the updated list is recomposed and shown in the final task.

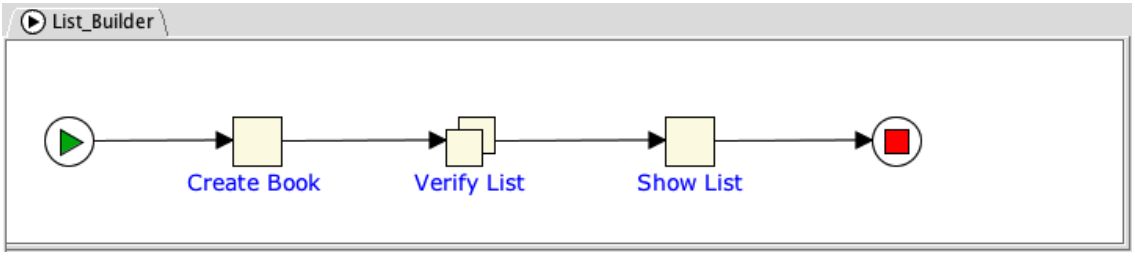


Figure 4.35: Example specification with a Multiple Instance Task

To prepare this specification, place two atomic tasks and one MI task onto the canvas, and join then with flows as shown in Figure 4.35. Now, we need to define a complex data type to store the entire book order. Select the *Data Definitions* property in the Specifications section of the Properties Pane, and enter the following two complex type definitions:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="BookOrder">
    <xs:sequence>
      <xs:element name="title" type="xs:string" />
      <xs:element name="price" type="xs:double" />
      <xs:element name="inStock" type="xs:boolean" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="BookList">
    <xs:sequence>
      <xs:element maxOccurs="unbounded" name="order" type="BookOrder" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

The first defines a complex type called 'BookOrder', which is a record with three simple type fields. The second defines a complex data type called 'Booklist', which consists of an array of one or more elements called 'order', of BookOrder type ('unbounded' means there is no upper limit on the number of order records we can include in the book list). Once this is added, we can start populating the data perspective of the specification:

- Create a net-level data variable called **MasterList** of type **BookList** and scope **Local** (Tip: In the 'Type' dropdown, all types are listed alphabetically, case-insensitive). Give the variable this initial value (Tip: click the Generate button, ⊙, to build the 'scaffolding' for the value):

  ```
  <order>
     <title>YAWL User Manual</title>
     <price>0.00</price>
     <inStock>false</inStock>
  </order>
  ```

  When entered correctly, the green OK message is shown to denote that it is a valid complex value to assign to the Masterlist variable of complex type 'BookList', since it defines values for the elements of one BookOrder (see Figure 4.36). It is important that an initial value is provided for this variable, because our definition of the 'BookList' type specifies that it will contain at least one element (that is, because it doesn't include a 'minOccurs=0' clause). If there was no initial value specified for this type, the specification would fail schema validation at runtime – in other words, the Engine would reject the specification.

- Add a decomposition to the first atomic task, and call it 'Create Book List'. Go to the Decomposition's *Data Variables* property, open the Data Variables Dialog, and drag the net-level MasterList variable to

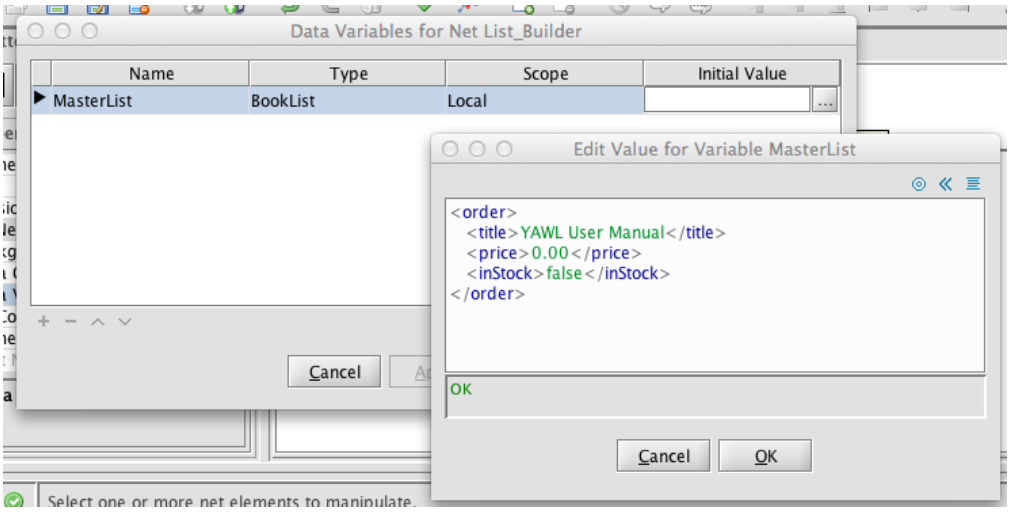


Figure 4.36: Creating the 'MasterList' net-level variable

the Decomposition Variables list to create a variable called **MasterList** of type **BookList** and scope **InputOutput** (the mappings between the net-level and task-level variables are automatically generated). Rename the task-level variable 'bookList' to differentiate it from the net-level variable. This task will allow a user to add any number of book orders to the master book list.

- Add a decomposition to the other atomic task, and call it 'Show List'. Add a variable to the decomposition similarly to the first task, except that its scope should be set to **Input**, and so only an input binding will be generated. This task will show the user the results of any data changes done during the execution of the MI task, thus its variable is input only (meaning that the values are 'display-only' for this task).

- Add a decomposition to the MI task and call it 'Verify List'. To the decomposition, add a variable called **MasterList** of type **BookList** and scope **Input & Output** by dragging it from the net-level list, then rename it to 'bookList' like we did earlier. Notice that for this task, the tool bar at the bottom on the Decomposition Variables list has a 'Mark as MI' button, , appears. This button only shows for MI tasks. Each MI task requires one variable to be marked as the formal MI variable, that is the variable that will provide a list or set of data values which can be split into individual values for passing to individual (multiple) instances of the task. Our variable of 'BookList' type fits that bill.

- With the 'bookList' Decomposition variable selected, click the 'Mark as MI' button. Four changes are immediately visible:

  - the variable's name has changed from 'bookList' to 'bookList_Item', to signify that each created task instance will handle one item of the overall book list;
  - the variable's type has changed from 'BookList' to 'BookOrder', which is the data type of each item in the book list;
  - the variable row is highlighted blue, to denote that it is the marked MI variable for this task; and
  - the 'arrowhead' icon on the left side of the variable row has changed to an MI Task icon.

- The input and output parameter mappings for MI tasks are a little different to atomic (single instance) tasks. In addition to the mappings to pass the initial and final data values between net-level and task-level, two more queries are required, one which defines how to split the list of values into individual components, and the other which defines how to join those components back into the composite list. Fortunately, those additional queries are also generated automatically when you mark a variable as the MI variable for the task.

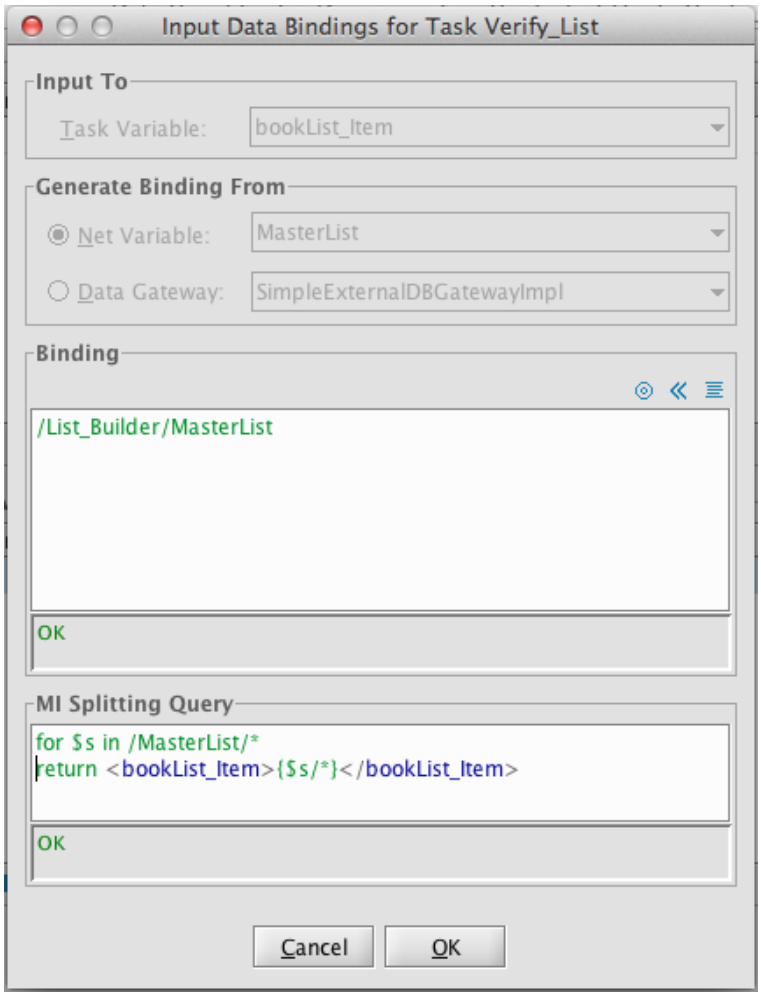


Figure 4.37: Input Binding and Splitting Query generated from the 'MasterList' variable

- Let's review the generated input queries for the formal MI variable. With the 'bookList' task-level variable selected, click the 'Input Bindings' button. Notice that the dialog has an additional input field for the 'MI Splitting Query', which shows the generated query (Figure 4.37). Notice also that for the Input Binding the entire 'MasterList' value is specified, rather than its child content (i.e. there's no '*' at its end). The MI Splitter Query is an XQuery expression that is used to take the list variable mapped in and split it into a number of 'child' elements, one for each member of the list. In natural language, the query reads: "for each child element in the list variable, return that element". In our example, the generated XQuery is:

  ```
  for $s in /MasterList/* return <bookList_Item>{$s/*}</bookList_Item>
  ```

  The $s part is a query variable – all query variables start with a $ sign followed by one or more characters ($s is a convention for a loop variable, but other names are, of course, allowed). The return value for each $s in our query would by default start and end with 'order' tags (the name of the element in the 'BookList' date definition); since our variable is called 'bookList_Item', we have to replace the 'order' tags with 'bookList_Item' tags, so the return value of our XQuery is the *contents* of the order (`{$s/*}`), but surrounded by 'bookList_Item' tags. The final result is a number of 'bookList_Item' values, each corresponding to one 'order' element in MasterList. Click OK to close the Input Bindings Dialog.

- Now let's take a look at the generated output queries for the formal MI variable. With the 'bookList' Decomposition variable selected, click the 'Output Bindings' button. Notice that the dialog has an addition input field for the 'MI Joining Query', which shows the generated query (Figure 4.38). In our example, the generated query puts all the instance query results into a list (in this case, a list of 'order' elements) ready for mapping back to the net-level variable. The aggregate query will look exactly like this in most cases. ($j is used simply to differentiate it from $s in the splitter query, but its name is unimportant).

- Click OK to close the Output Bindings Dialog, and OK again to close the Data Variables Dialog.

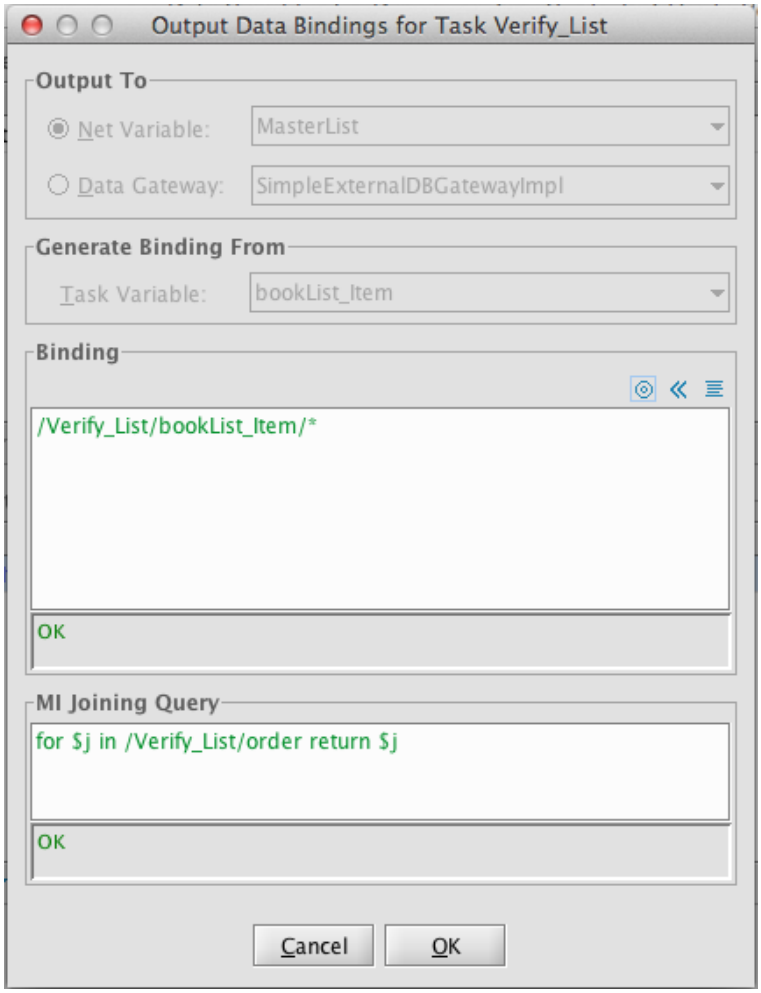


Figure 4.38: Output Binding and Joining Query generated for the 'MasterList' variable

- The final requirement is to specify the instance attributes for the MI task. Select the 'Verify List' task, then the Task *M-I Attributes* property. Set minimum instances to 1, maximum instances to 20, threshold to 5 and 'static' instance creation type. These settings mean that between 1 and 20 task instances will by started from this MI task at runtime (depending on the number of book orders in the book list), the MI task will complete when 5 instances complete (or when all complete if less than 5 were started) and new instances may not be dynamically started after task execution begins.

When this specification is executed, it will first allow the user to specify a number of book orders, then will split the details of each into a corresponding number of MI task instances, one for each order. The price and availability of each order can be updated within its own task instance. When all (or the threshold) of MI

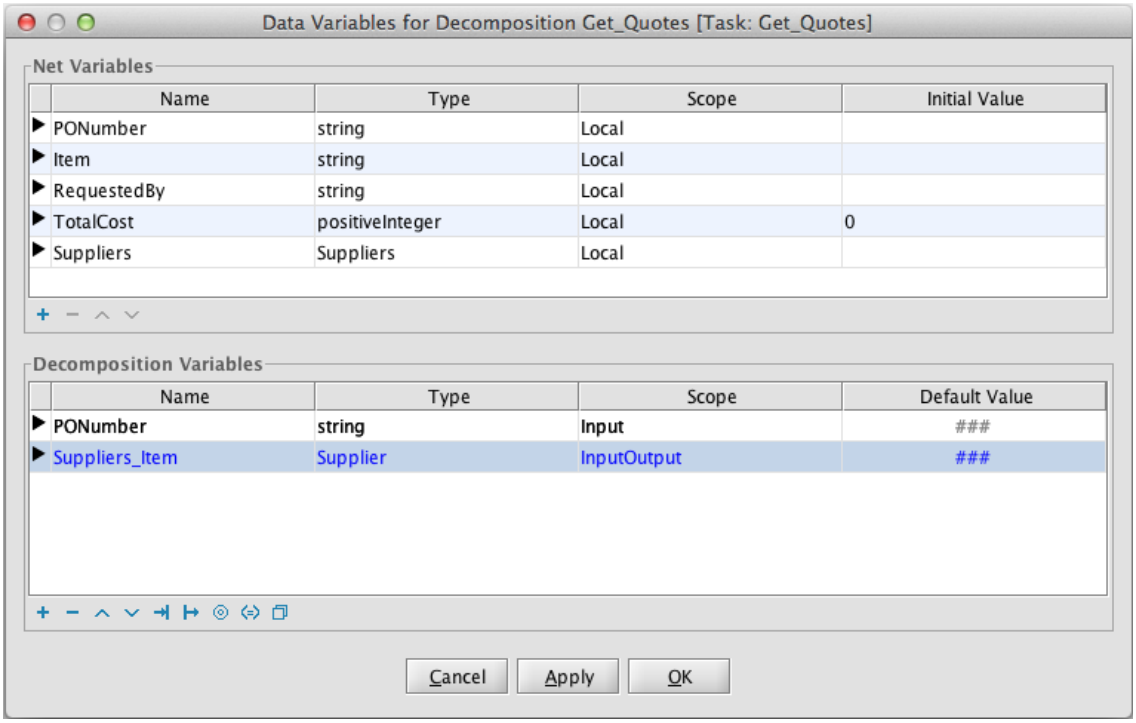


Figure 4.39: Update Parameters dialog, 'Get Quotes' task

task instances have completed, their data will be aggregated back into the list for display in the third and final task.

For a second example, consider a process that raises a purchase order to purchase a particular product. The business rules for this process require that at least three quotes are needed for any purchase, from which one is chosen to fill the order. A multiple instance task called *Get Quotes* is used to gather the product quotes from suppliers. The Data Variable dialog for the *Get Quotes* task can be seen in Figure 4.39.

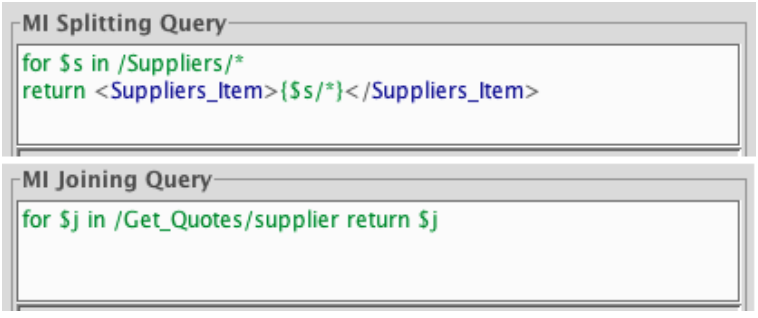


Figure 4.40: Instance Detail dialog for the 'Get Quotes' MI task

The task has two variables defined: 'PONumber' (the purchase order number) and 'Suppliers_Item' (the details of a single supplier). Notice that 'PONumber' is defined as *Input Only*, and is mapped from the net-level variable of the same name – this means that the purchase order number will be displayed within each of the work items created from the MI task. Notice also that while a list of Suppliers is mapped in from a net-level variable, the marked MI variable maps to a single supplier – one for each task to be created. The MI queries for the task can be seen in Figure 4.40, and are very similar to those generated in our first example.

When the instance creation type is set to **dynamic**, then new instances of the MI task may be created at

runtime *after the task execution has begun and split into its instances*. There is a button on the default worklist handler to accommodate this (see Chapter 6), which will be enabled for dynamic MI tasks until such time as the maximum instances bound has been met.

A more complex example of MI Task, which involves mapping the resulting data from the task back to a different net-level variable than the one that supplied the MI data to the task, can be found at the end of Chapter 5.

4.8 The Resource Perspective

The resource perspective allows you to allocate available resources to tasks, specify rules that should be used for those allocations, place certain constraints on who may or may not start a task, assign certain task privileges to resources, and much more. To enable the resource perspective in the Editor, a valid connection to the YAWL Resource Service must first be established (see Section 4.10 for more details).

Once a connection with the Resource Service has been established, any manual atomic task with a decomposition (a task is manual by default, and only becomes automated when explicitly checked as automated in the Properties Pane) can be allocated resources via the *Resourcing* property in the Task section of the Properties Pane. Selecting this property will launch the Resourcing Dialog (Figure 4.41).

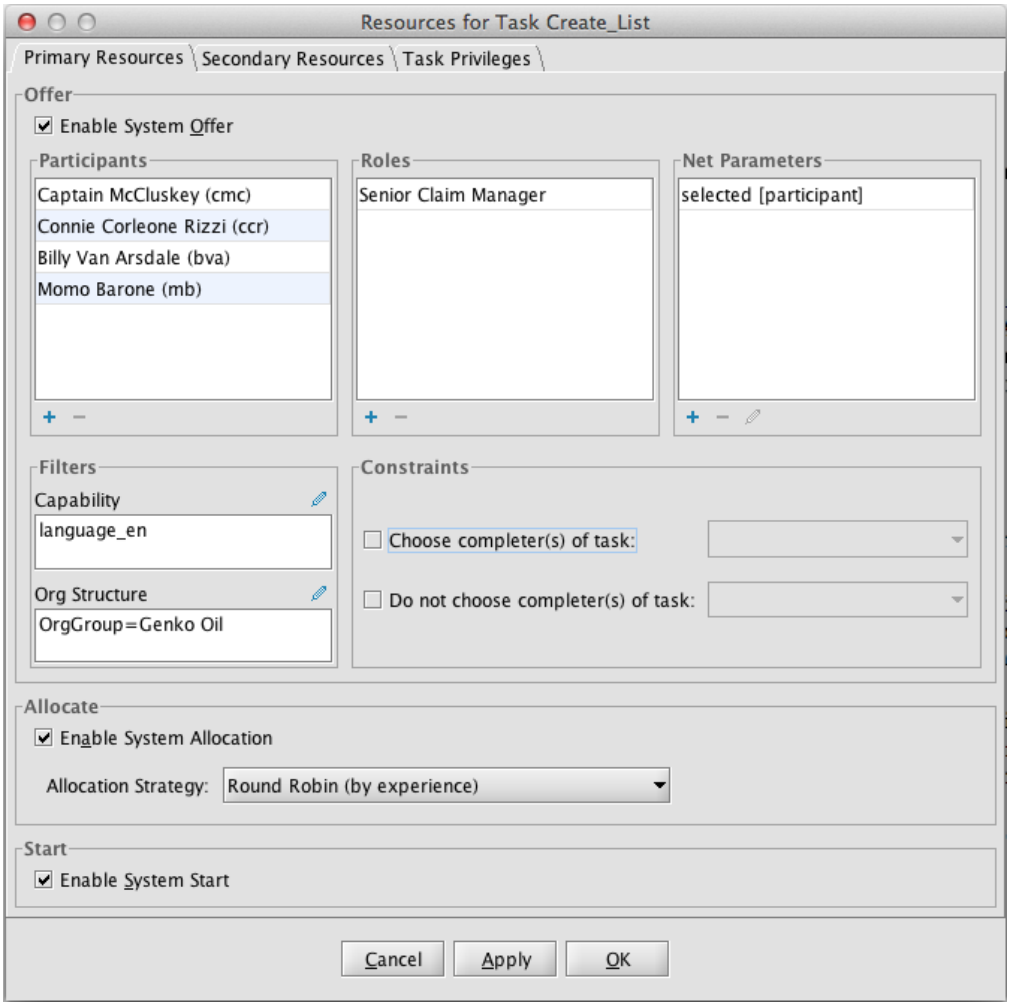


Figure 4.41: Resourcing Dialog, Primary Resources tab

A task may be allocated both human (i.e. a person) and non-human (e.g. rooms, vehicles, equipment, materials, etc.) resources. Further, resources are divided into primary and secondary resources. Primary resources are human resources that own a work list, so that allocating a task to a primary resource means that a work item (i.e. a runtime instantiation of the task) will (potentially) appear on that resource's work list. Secondary resources consist of both human and non-human resources, and are those resources that are required to assist in performing the work of the task, but do not manage the task directly (as the primary resource does). Thus, each manually resourced task, when executed, will have exactly one primary resource, and zero or more secondary resources.

To summarise:

- Human secondary resources don't see the work item on their work list.
- Non-human secondary resources don't have a worklist.
- A human resource may be a primary resource for some work items and a secondary resource for others.
- A non-human resource can never be a primary resource.

At runtime, when a work item is started, all of its allocated secondary resources (if any) are marked busy. When a work item is completed or cancelled, all of its busy secondary resources are released. All busy and release events are stored in the process logs. If a secondary resource is not available when a work item start is actioned, the work item may start anyway or may be blocked from starting, depending on a configuration setting (see Chapter 6 for more details).

The following sections describe how to specify resourcing requirements for a task using the Resourcing dialog.

4.8.1 Primary Resources Tab

For the majority of tasks, the Primary Resources tab of the Resourcing Dialog will be the only tab that needs to be populated. The first decision to be made is to determine the *interaction strategy* for work items of the selected task. There are three interaction points: offer, allocation, and start, and each interaction point requires a decision to be made on whether it will be *User* or *System* processed.

The Primary Resources tab comprises three sections, one for each interaction point.

Offer Interaction

If we choose that work items are to be *Offered* by the System (by selecting the 'Enable System Offer' check box in the Offer section of the tab), then we can specify the primary resources that will automatically be offered work items of the task at runtime. If the check box is not selected – a user initiated offer – at runtime an Administrator will need to manually choose the resource(s) to offer the work item to.

For a System offer, we can specify an initial set of primary resources (called the *distribution set*) that will be offered work items of the selected task at runtime by the System. The distribution set may consist of any number of participants and/or roles[2].

To add participants, click the Add (+) button on the toolbar under the *Participants* list. A dialog showing the complete list of all participants known to the Resource Service is displayed, from which selections may be made (Figure 4.42). Multiple participants may be selected concurrently. Also, you can quickly filter the list be typing characters into the 'Filter' field at the top of the dialog – only those names that contain the same sequence of letters as those typed will be shown in the list. Click OK to add the selections to the Participants list on the underlying tab. *Roles* may be added in the same way. To remove selections, select one or more from the list then click the remove (–) button on the toolbar.

[2] A role is essentially a set of participants.

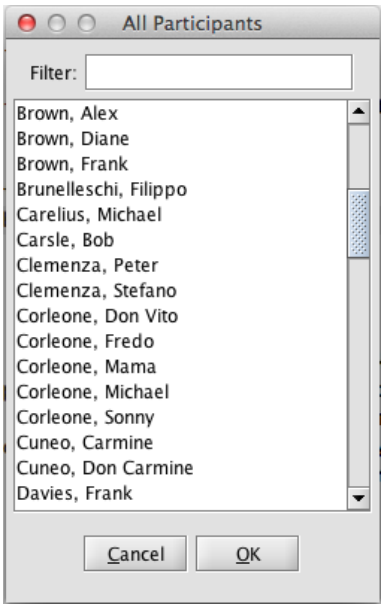


Figure 4.42: All Participants selection dialog

In the *Net Parameters* section, you may nominate one or more net-level variables that at runtime will contain a value of either the userid of a participant or the name of a role to be added to the distribution set. This is known as *deferred allocation*. *All* net-level variables that are of string type will be available for addition to the Net Parameters list. To select a net-level variable to use for deferred allocation, click the Add (+) button on the toolbar under the Net Parameters list. A small dialog will appear (Figure 4.43) that displays the list of appropriate net-level variables, from which one may be selected. Then, specify in the 'Will refer to' panel whether at runtime the variable's value will refer to a Participant's userid, or to a Role's name. Click OK to save the selection.

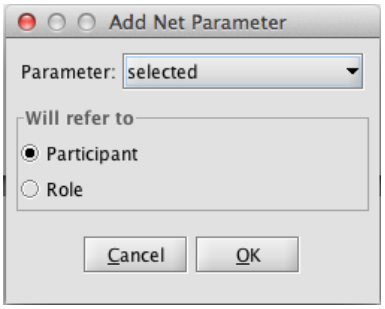


Figure 4.43: Add Net Parameter dialog

In the *Filters* section we can filter the distribution set of participants, roles and net parameters that have been described above. Filtering can be done over capabilities, positions and/or organisational groups, so that for example any participants in the distribution set that do not hold a specified capability will be discarded from the set.

To set a capability filter, click the Edit button, , on the top-right of the Capability Filter panel. The Add/Edit Filter dialog is shown which consists of two panels, a list of all the Capabilities known to the Resource Service, and a panel for composing Filter Expressions (Figure 4.44). A filter expression may be composed by first double-clicking on a Capability from the list (which will copy it to the expression panel). Composite expressions can be formed using the buttons on the bottom toolbar:

- Logical AND (&) – both sides must be true

- Logical OR (|) – at least one side must be true
- Undo (⌫) – delete the last entered capability or logical operator
- Remove (✕) – clear the entire expression

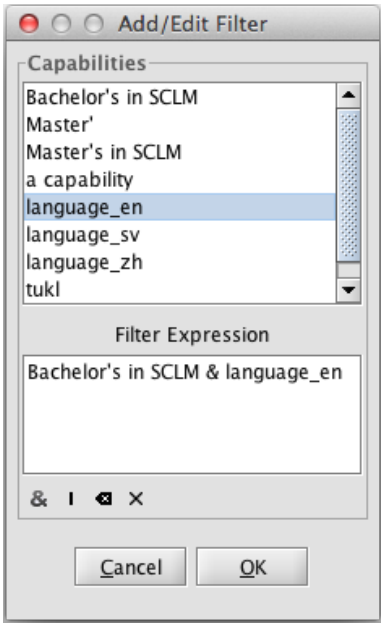


Figure 4.44: Capability Filter dialog

For example, the expression in Figure 4.44 will match only those participants in the distribution set that have a Bachelor's degree in SCLM *and* English language skills specified in their capability set.

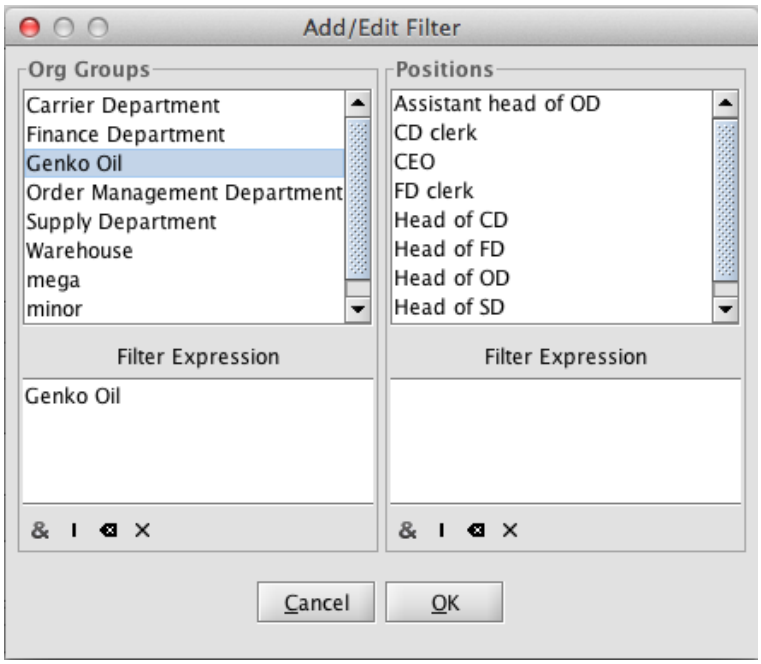


Figure 4.45: Org Structure Filter dialog

The Org Structure filter works in a similar way, except that you may specify expressions based on Org Groups and/or Positions that appear in the organisation structure managed by the Resource Service (Figure 4.45). If both types are specified, they both must be true for a participant to be retained in the distribution set. Finally, if both a Capability and an Org Structure filter is specified, then all expressions must be true for a participant to remain in the distribution set.

The final part in the Offer section of the Primary Resources tab is the *Constraints* section. Here you can:

- **allow** the work items of the selected task to be offered only to participants who are members of the specified distribution set *and* have previously completed work item(s) of another task (as specified) in the current process instance (this is referred to as the 'familiar task' pattern). To do so, select "Choose completer(s) of task:" then choose the previously completed task from the dropdown list.
- **prevent** the work items of the selected task from being offered to participants who are members of the specified distribution set *and* have previously completed work items of another task (as specified) in the current process instance (this is referred to as the 'separation of duties' or 'four-eyes' pattern). To do so, select "Do not choose completer(s) of task:" then choose the previously completed task from the dropdown list.

The dropdown list will contain all tasks that precede the current task (i.e. the one we are setting resources for) in the specification's control-flow, including those in any previous sub-nets, that were handled by the Resource Service (the Default Worklist). Of course, if you choose to specify both constraints for a task, the choices are mutually exclusive, and you will be prevented from choosing the same task for both constraints.

Allocate Interaction

If we choose that work items are to be *Allocated* by the System (by selecting the 'Enable System Allocation' check box in the Allocate section of the tab), we can specify how work items will be allocated to a particular participant, that is we can select the allocation strategy from the dropdown list that will be used to choose a single participant to be allocated a work item from the set of participants offered the work item. If the check box is not selected – a user initiated allocation – at runtime any participant who has been offered the work item can manually choose whether to commit to being responsible for the performance of the work item (i.e. to allocate the work item to themselves).

The available allocation strategies are:

- Round Robin (by time): chooses the participant in the set who has not been allocated a task instance for the longest time;
- Round Robin (by least frequency): chooses the participant in the set who has been allocated an instance of this task the least number of times in the past;
- Round Robin (by experience): chooses the participant in the set who has been allocated an instance of this task the most number of times in the past;
- Fastest to Allocate: chooses the participant in the set who, on average, has recorded the shortest time from being offered an instance of this task to allocating it to themselves.
- Fastest to Start: chooses the participant in the set who, on average, has recorded the shortest time from being allocated an instance of this task to starting it themselves.
- Fastest to Complete: chooses the participant in the set who, on average, has recorded the shortest time from starting an instance of this task to completing it.
- Fastest Resource: chooses the participant in the set who, on average, has recorded the shortest time from being offered an instance of this task to completing it.
- Random Choice: randomly chooses a participant from the set (the default); and

- Shortest Queue: chooses the participant in the set who has the least number of task instances currently in their work queue.

Developers can 'plug-in' new allocators and make them available both at design time and at runtime; see the YAWL Technical Manual for details.

Start Interaction

If we choose that work items are to be *Started* by the System (by selecting the 'Enable System Start' check box in the Start section of the tab), then the work item, once it is allocated to a participant, will be immediately started. If the check box is not selected – a user initiated start – then the user who has been allocated the work item will choose to manually start working on it at a time of their choosing.

4.8.2 Secondary Resources Tab

On the Secondary Resources Tab, the set of secondary resources that are needed to assist in the successful performance of the task may optionally be specified. Resources may be selected in each of the four categories by clicking the Add (+) button on the toolbar under each list. Individual participants and/or 'assets' (non-human resources) may be chosen at most once, while roles and categories (i.e. of non-human resources) may be selected more than once. For example, the selected resources in Figure 4.46 include two entries for the role 'Surgical Nurse', meaning that two nurses are required.

A category of non-human resources provides a similar relation to that of a role to participants – it is a grouping of resources. Each category may also have listed a number of subcategories. Some examples may be:

- *office machines* denotes any non-human resource in the office machines category, including all those listed in any subcategories.
- *office machines -> copiers* denotes those non-human resources in the 'copiers' subcategory of the office machines category.
- *office machines -> no category* denotes those non-human resources in the office machines category that are not further subcategorised.

If a participant chosen as a secondary resource also becomes the primary resource for a task, then their demarcation as primary resource takes precedence.

4.8.3 Task Privileges Tab

Participant-Task privileges (i.e. privileges that apply only for this task) may be set on the Task Privileges Tab. For example, we can specify here whether participants are allowed to suspend the execution of work items of the selected task, or to skip them, and so on.

Individual privileges may be set by selecting them (i.e. checking their check box). You may further restrict a privilege to individual participants and/or roles by first checking the Restrict Privilege box for the relevant privilege, then adding participants and/or roles as desired. If a privilege is allowed but not restricted, it means that all participants and roles are granted that privilege for the selected task.

For example, Figure 4.47 shows that the "Allow work item reallocation with retained state" privilege is granted to any participant who performs an instance of this task, while the "Allow work item to be piled" privilege is also granted, but restricted to an individual participant and to members of a particular role. *Piling* means that, if a participant chooses to pile an instance of the task at runtime, that participant will be automatically allocated that instance *and all future instances of the task for all future instances of the process containing the task*, until such time as piling is turned off for that task by the participant or an administrator.

More details on resource allocation and authorisation can be found in Chapter 6.

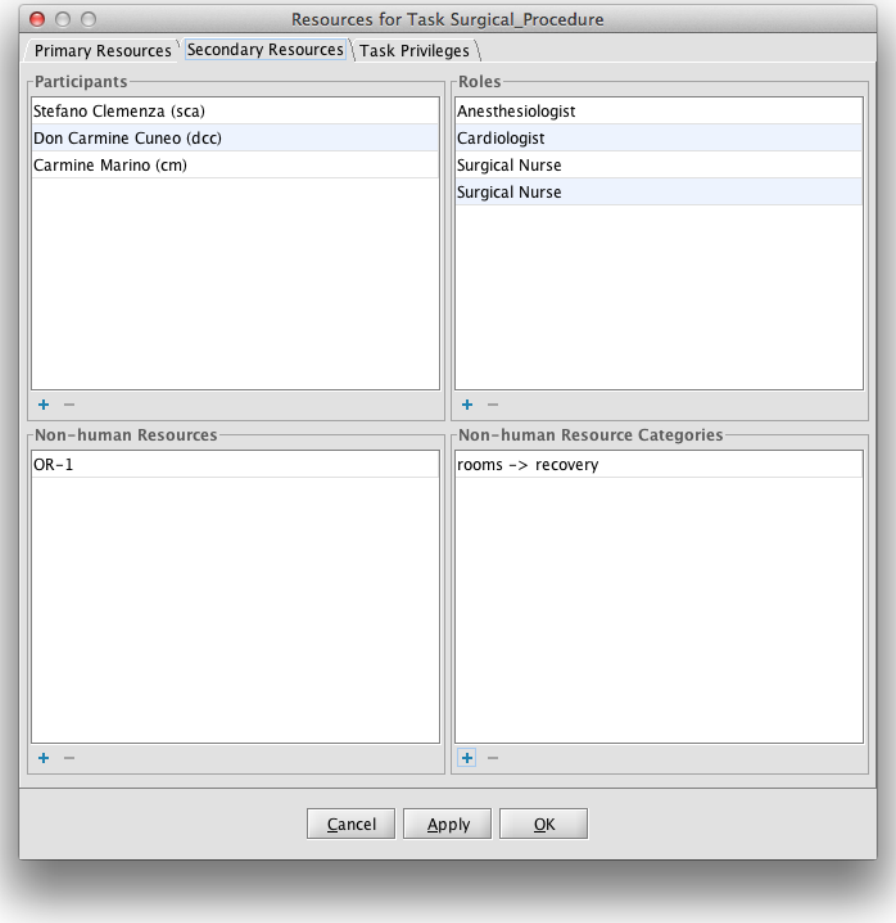


Figure 4.46: Resourcing Dialog, Secondary Resources tab

That completes the review of the three perspectives, control-flow, data and resource, and how they are supported in the Editor. The following sections provide details of other Editor features.

4.9 The Preferences Dialog

The Preferences dialog co-locates the default settings for a number of system-wide properties. The dialog can be accessed by clicking on the **File** Menu and choosing **Preferences...**, and is shown in Figure 4.48. (Tip: You can also open the Preferences dialog by clicking on either of the connection indicators on the left of the status bar.

The dialog is divided into five panes, which can be selected from the list on the left:

- **Connections**: The Connections pane is used to set parameters for connecting to a running YAWL Engine and Resource Service (see Section 4.10).
- **Analysis**: The Analysis pane is used to set parameters applicable to the Verification Analysis feature of the Editor (explained in detail in Section 4.11).

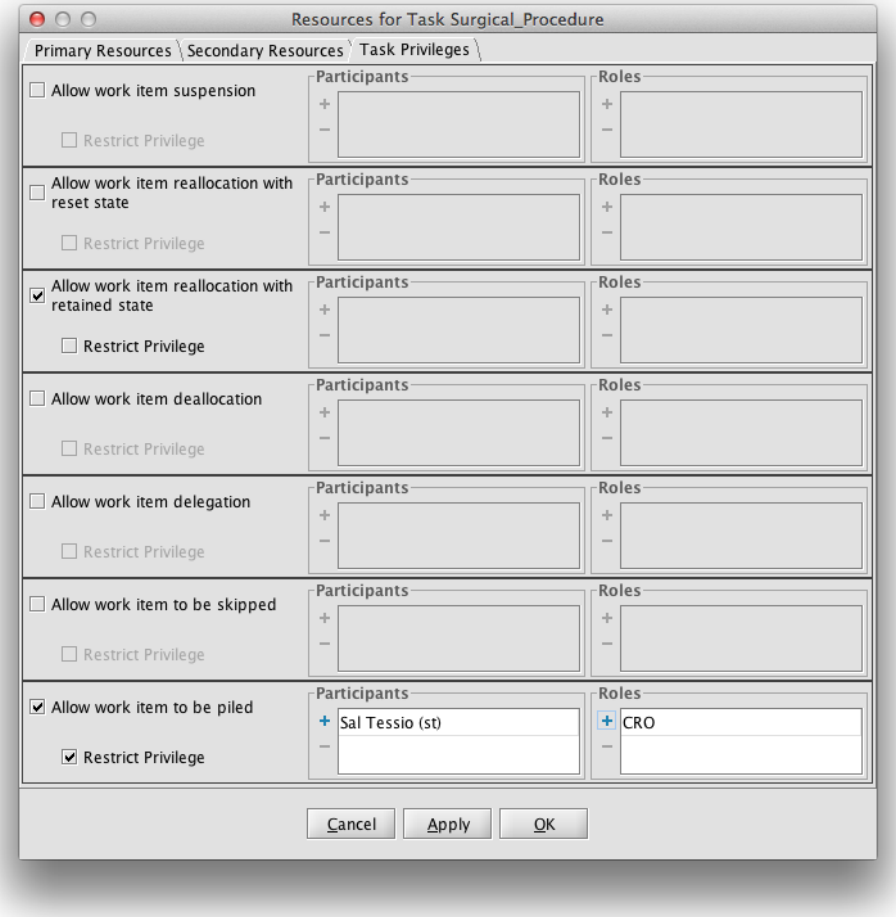


Figure 4.47: Resourcing Dialog, Task Privileges tab

- **File Paths**: The File Paths pane is used to set the directories for the location of user-defined task icons (Section 4.5.4) and the WofYAWL analysis tool (Section 4.11), and the files that contain definitions for User-Defined Extended Attributes (Section 4.18).

- **Save Options**: The Save Options pane is used to set the various parameters involved when saving a specification to disk file (cf. Section 4.3.3). It also contains one additional option: *Reload most recent specification on startup*. When selected, the specification that was opened in the Editor when it last closed will be reopened next time you start the Editor.

- **Defaults**: The Defaults pane is where parameters for default net background colour, element fill colour and label font are set. This pane also provides the following two preferences:

  – **Show Description panel in properties pane**: When checked, a description of the selected property appears at the bottom of the Properties Pane. Uncheck this option to hide the description panel.

  – **Check for updates when Editor starts**: When checked, each time the Editor starts it will check to see if there are any new updates available for the Editor and, if there are, will display the details in a dialog with options for downloading and updating (see Section 4.20 for details on checking for updates). Uncheck this option to not have this check occur each time the Editor starts.

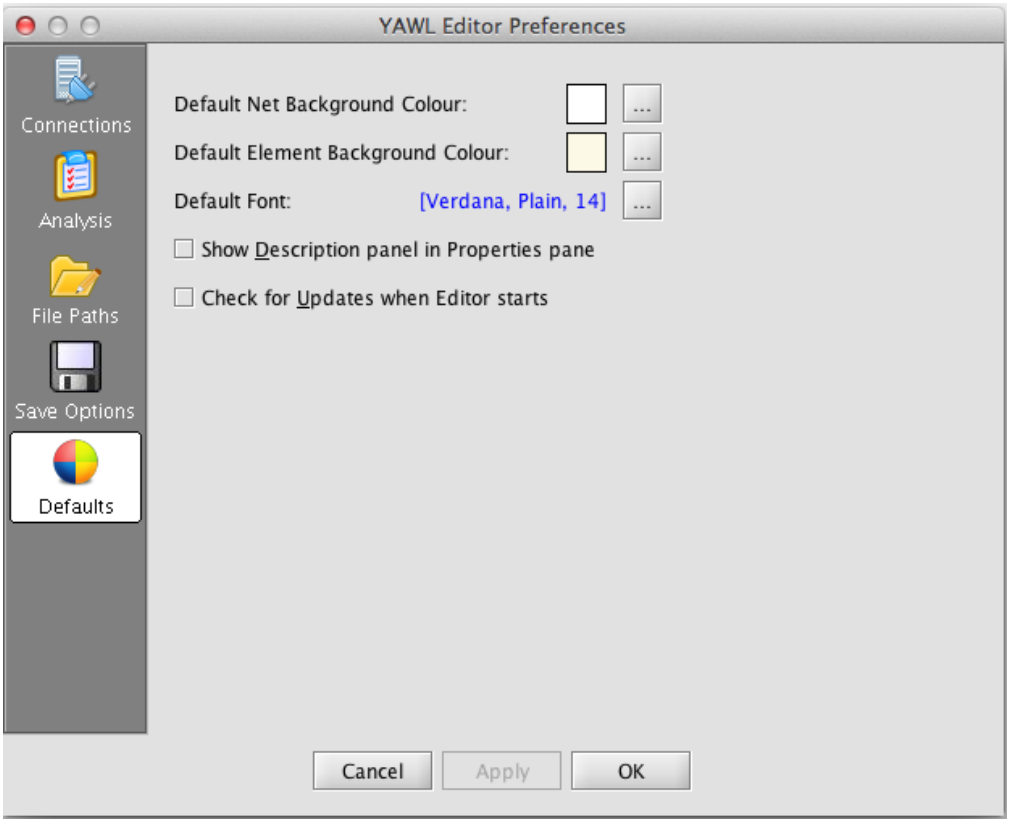


Figure 4.48: The Preferences Dialog, Defaults pane shown

4.10 Connections

4.10.1 Connecting to the YAWL Engine and Resource Service

Why connect to the YAWL Engine? At design time, each atomic task in a YAWL specification must be associated with a 'service' that will be responsible for performing the work of the task at runtime. If you do not explicitly specify a service for a task, the task will be assigned to the *Default Worklist Handler*, which by default is provided as part of the Resource Service. If you want to associate a task with a service other than the Default Worklist Handler, you can make the appropriate choice from a list in the Decomposition's Custom Service property (see the following section). The Editor populates the list of available services by making a call to a running YAWL Engine, which returns the list of services currently registered with it. Before this can occur, the Editor must first establish a valid connection to a running Engine.

Why connect to the Resource Service? To use the organisational data managed by the Resource Service, for example to assign tasks to Participants or Roles, a connection between the Editor and the Resource Service needs to be established. The Editor populates the Resources Dialog with all the necessary organisational data needed.

On Editor startup, a connection to a running Engine and Resource Service is attempted. If successful, the connection icon at the left of the status bar will turn green (the left icon represents the Engine connection, the right icon the Resource Service connection). If unsuccessful (perhaps because the Engine is not running, or the connection parameters are incorrect) the connection icon(s) will show red. After the Editor has started, an connection parameters may be updated via the Preferences Dialog.

To establish a connection with a running YAWL Engine or Resource Service, or to change the connection parameters:

1. Click the **File** Menu and choose **Preferences ...**.

2. On the Preferences Dialog, choose 'Connections' from the list on the left (Figure 4.49). Here you can specify the host name (default: localhost) and port (default: 8080) that represent the locations of the Engine and Resource Service. Most times the defaults are sufficient, but may be changed if the Engine or Resource Service are installed remotely, or if the port needs to be changed because the default port was already in use. You may also specify an alternate user name and/or password to use for the connection. Note that the user must have administrator privileges. A default user **editor** with password **yEditor** are pre-installed for use.

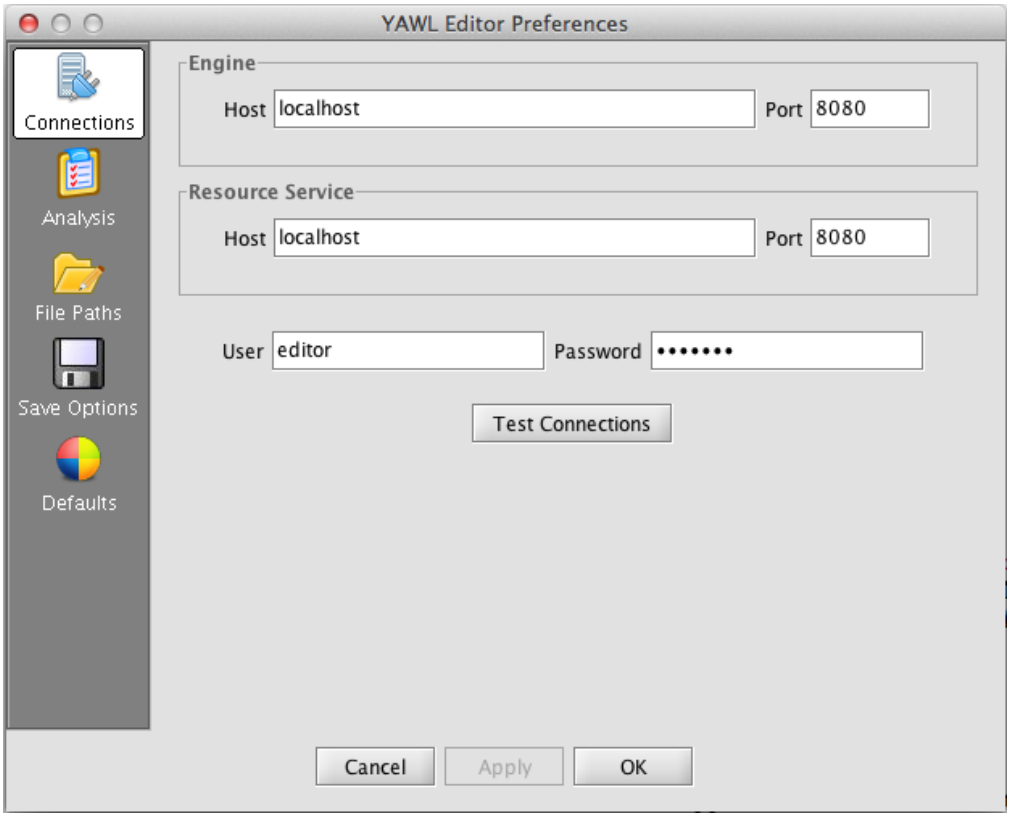


Figure 4.49: Preferences Dialog, Connections panel

For convenience, a **Test Connections** button is provided, which will allow you to test the connection using the supplied parameters before you commit them. Click OK to finalise the connection settings.

4.10.2 Connecting a Decomposition to a registered YAWL Service

Each task decomposition within a workflow must be associated with a custom YAWL service that has been registered with the YAWL Engine. By associating a task decomposition with a custom service, all task instances based on that decomposition will be passed to that custom service at runtime for processing – that is, the custom service is responsible for performing the work of the task instance.

For example, a decomposition may be set up to place an order with an external company. Upon execution of any task using this decomposition, data could be transmitted via a Web Service invocation to this company.

To have a decomposition invoke custom YAWL service, click on a task with the decomposition, then select the Custom Service property in the Decomposition section of the Properties Pane. From the dropdown list, choose the desired custom service for this decomposition.

If the Editor is connected to a valid running YAWL Engine instance, the Custom Service dropdown list will

contain entries for all custom YAWL services the engine has registered. The Resource Service is listed as the 'Default Worklist Handler', and is the service assigned to the decomposition by default (i.e. unless you change it). If there is currently no Engine connection, only the Default Worklist (of the Resource Service) will be available.

When you select a YAWL Service, the Editor will query the running engine for the mandatory input and output variables required by the service (if any), and will populate the decomposition variables of the selected task with those variables. Core custom services that are supplied with the engine include one for RPC-Style Web Service Invocation (WSInvoker Service), and one for supporting flexibility and exception handling for YAWL processes (Worklet Service). Optional services include an SMS handler, a digital signature service and a email sender service.

4.11 Specification Analysis

Verification of specifications for the engine only determines whether the engine will be able to successfully load and begin execution of the specification. In contrast, the analysis tool can be used to test for deeper issues in the specification.

The analysis toolbar button,  , or the matching **Analyse** menu item under the **File** menu allows workflow designers to analyse their specifications. A number of potential problems with a workflow can be automatically spotted with analysis. Examples include spotting potential deadlock situations, unnecessary cancellation set members, and unnecessary or-join decorators (at run-time, or-joins require significant processing effort, and should be removed or replaced with other join types if they are not actually needed).

A configuration dialog for specification analysis is available in the Preferences Dialog via the **File**, **Preferences** (see Figure 4.50). Note that many of the options are disabled by default since they are resource intensive and may take some time to complete for large and/or complex workflows. It is recommended that analysis of such specifications not be done incrementally, but rather at planned checkpoints during specification development.

Because analysis make take a long time and is very resource intensive, it may appear that the Editor has frozen during an analysis (although it is very unlikely that this will actually be the case). To provide some feedback about the progress of the analysis, a dialog will appear which shows updates and messages (see Figure 4.51). At any time, you may click the 'Stop' button in the dialog to abort the analysis. The dialog also includes a checkbox that, when checked, will keep the dialog open after the analysis completes so that messages may be noted. This functionality can also be controlled via a setting in the Analysis preferences (see Figure 4.50).

If the optional YAWL specification analysis utility, **wofyawl.exe**, is supplied in the same directory as the Editor, an extra set of options will be enabled in this dialog, allowing more analysis options than those supplied by default. The utility must be compiled for specific architectures[3]. The current version of the Editor needs version 0.4 of the utility.

4.11.1 Verification Analysis Explained

This section provides a brief overview of verification analysis in YAWL. Verification is concerned with the design time detection of certain undesirable characteristics in process models.

Extensive research has been conducted in the area of workflow verification. One of the pioneers of this work is Wil van der Aalst. He formally defined the notion of *soundness* as a correctness notion for workflow nets. This class of Petri nets, which does not support OR-splits/joins, Multiple Instance Tasks and cancellation regions, forms a predecessor of YAWL. Informally speaking a workflow is sound iff [9]:

- The net has the *option to complete*. That means that from every reachable state the final state, where there is a single token in the output condition, can be reached.

[3] WofYAWL is currently only available for Windows environments.

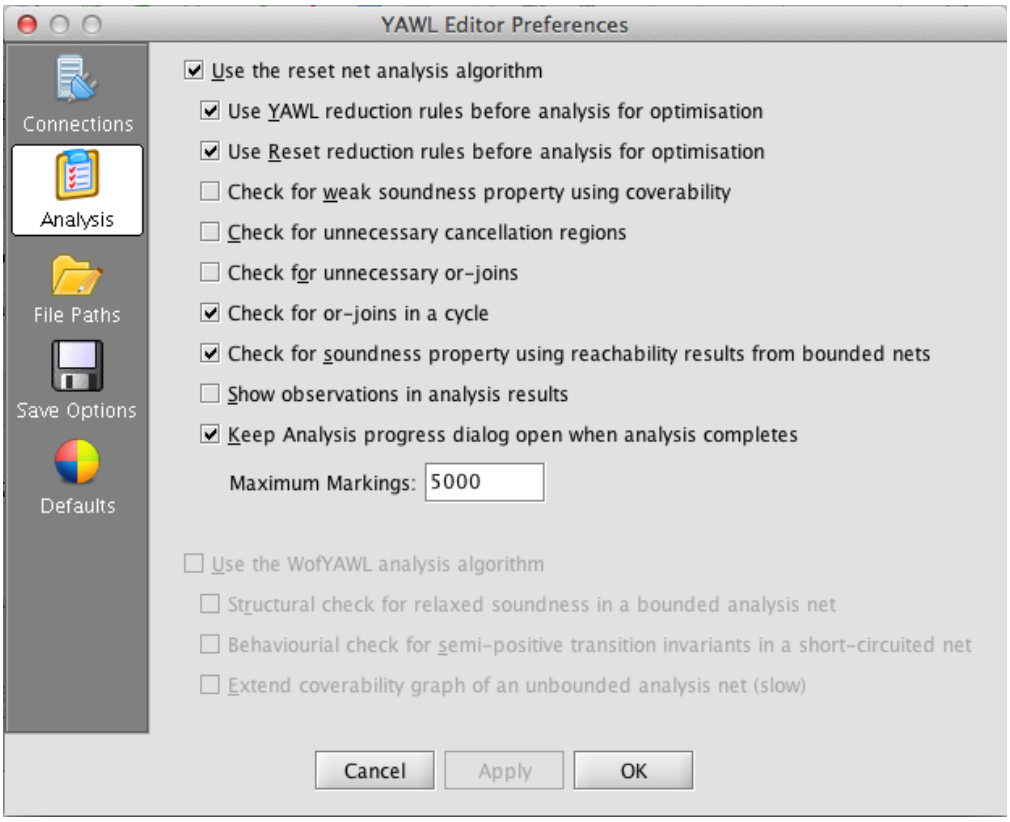


Figure 4.50: Preferences Dialog, Analysis panel

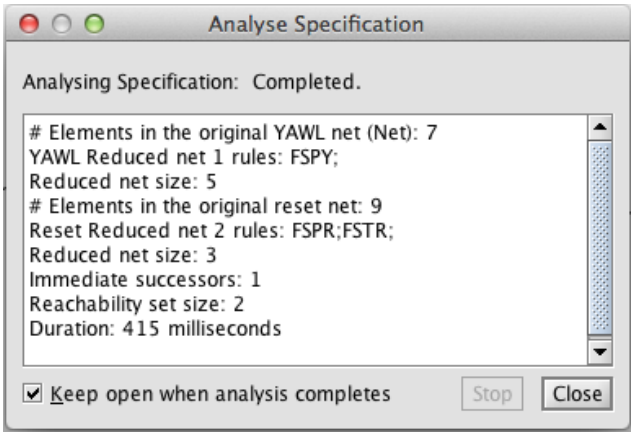


Figure 4.51: Analysis progress dialog

- The net has *proper completion*. This means that when the output condition is marked (i.e. has a token) there are no other tokens anywhere else in the net.
- The net has no *dead tasks*. These are tasks that cannot be executed in any scenario.

For YAWL, the notion of *weak soundness* was introduced as it can be theoretically proven that soundness is not decidable [33]. For a finite state space, we can simply try and check all reachable states, but this is obviously not possible when this state space is very large or infinite. In those cases, we can check whether it is possible to reach the final state from the initial state. Hence, the question becomes does a scenario exist where we reach the final state?

The richer concepts offered by YAWL also introduce additional correctness notions. For example, an analyst may have used an OR-join where an XOR-join or an AND-join could have been used. This is not desirable for computational reasons, but also because it makes the process model harder to understand. Hence, the YAWL environment will check whether all OR-joins are *immutable*. Another correctness notion is that of *irreducible cancellation regions*. Here it is checked whether certain conditions or tasks can be removed from a cancellation region as they will never contain a token or will never be active when the associated cancellation task executes.

The YAWL editor offers two different approaches to automated verification. One approach [33] is based on the theory of Reset nets (this is built into the Editor). The other approach [30] uses Petri net theory and in particular, the concept of transition invariants. For this latter approach the program **wofYAWL** is to be used. These approaches are different in that there are workflow specifications where one of them can pick up an error which the other approach cannot.

In the Analysis panel of the Preferences Dialog you can choose what type of verification the YAWL editor needs to perform. As some forms of verification may require quite a bit of time it is important to choose the right approach and generally speaking, it is probably best not to verify every intermediate version of a specification.

The screen shot shown in Figure 4.50 shows the options one can choose for the analysis based on Reset nets. This form of analysis supports the use of reduction rules. YAWL reduction rules can be applied to the net and Reset net reduction rules can be applied to the Reset net that results from the mapping of a YAWL net. The application of these reduction strategies may significantly reduce the workflow that needs to be analysed, hence it may significantly reduce the time that verification takes. Note that there is overhead associated with performing the reductions themselves. Also worth mentioning is that the soundness check is supported for workflows with a finite state space. The editor caps the state space at a certain number of states (via the 'Maximum Markings' setting, by default 5000) and tries to construct the reachability graph for the workflow. If the upper bound is not exceeded the soundness of the workflow can be determined with certainty.

Refer to [30] for a detailed explanation of the concepts behind the wofYAWL approach to verification.

4.12 Automated Tasks

Any atomic task in YAWL that is associated with the Resource Service (i.e. the default association if the task is not explicitly associated with another service) can have its decomposition defined as *manual* or *automated*. A task with a manual decomposition is a task that is intended to be executed by a human resource, e.g. a participant in the Resource Service's organisational model. A task with an automated decomposition is a task that is not offered to any resource but is executed by the system. This type of task can be used to manipulate the content of net variables, from simple data assignments to complex reports generation. Alternately it may be associated with a *codelet* – a discrete piece of code that is executed, optionally using the input variables of the task, and assigning any results to the chosen output variables of the task.

Both task types are handled by the Resource Service, but the behaviour of an automated task differs as follows:

- on enablement, it is automatically checked out of the engine (thus having priority over manual tasks in a deferred choice) and its input parameters are parsed;
- if a codelet has been specified, it will be executed using the task's variables as required; then
- it is automatically checked in and its output variables are mapped back to the corresponding net variables.

A task is designated as manual by default, but can be set as automated by selecting the 'Automated' property in the Decomposition section of the Properties Pane. When the Automated checkbox is checked, the task will display an *automated* indicator, and the Task 'Resourcing' property will be disabled (since human resources cannot be assigned to automated tasks).

Data manipulation can be achieved by using the task's variables and their Input and Output bindings. Bindings are generally used to copy the content of a net variable to a task variable and back again, but they may also contain an XQuery expression that uses static values, or the values of other variables, to copy data between net variables and task variables.

4.12.1 Codelets

Essentially, a codelet is a discrete Java class, managed by the Resource Service, that may be enacted by an automated task at runtime. When an automated task is enabled during process execution, and it has a codelet associated with it, the input parameters of the task are passed to the codelet, it is executed, and any results are passed back to the task via its output parameters.

There are currently six codelets available by default in the Editor (additional codelets can be added using a pluggable interface; see the *YAWL Technical Manual* for directions on how to add user-defined codelets):

- **ExampleCodelet**: a simple example designed to demonstrate the usage of codelets;
- **XQueryEvaluator**: accepts an XQuery as an input parameter, evaluates it (using other input parameters as required by the XQuery), and produces the result to an output parameter.
- **RandomWait**: accepts a time unit parameter ('H', 'M' or 'S') and a maximum value (type 'long'), and waits for a random amount of time units between zero and the maximum value.
- **ShellExecution**: accepts an input parameter containing a command line of an external program, and runs it, waiting for it to complete and returning the result (if any) via an output parameter.
- **SupervisorInfo**: accepts the userid of a user and returns the userid of that user's supervisor (based on the installed organisational data model - see Chapter 6 for more information).
- **TaskCompleterInfo**: Gets the name and userid of the participant who completed a specified atomic task in the current case.

To associate a codelet with a task decomposition, first mark the decomposition as automated by selecting the Decomposition **Automated** property and checking the box. When checked, the Codelet property is enabled; selecting that property opens the *Set Codelet for Automated Decomposition* dialog, listing the available codelets (Figure 4.52). Note that a valid connection to the Resource Service is required for this list to be populated with codelets (see Section 4.10). The dialog lists the available codelets, together with a description of what each one does and the task variables required to successfully execute it. For example, if ExampleCodelet is chosen, the automated task requires 3 variables to be created: input variables 'a' and 'b', and output variable 'c' (all of type 'long'). These variables are automatically added to the task decomposition when the codelet is selected for a task – at runtime, if the required variables values are not present, the codelet will be unable to successfully complete (the task will still complete successfully, however).

The codelet repository has been designed as 'pluggable', so that designers and developers can easily add new codelets to perform various operations, which will immediately be available to process designers via the dialog above, as long as there is a valid connection to the Resource Service.

4.13 Task Timers

Any atomic task can be assigned a timer behaviour. To do so, select the Timer property in the Task section of the Properties Pane. The dialog in Figure 4.53 will appear.

From this dialog it is possible to set an activation type (when the timer should begin) and an expiration value for the timer. The timer can be activated either when a task is enabled (i.e. is offered or allocated) or when it starts. These have different meanings according to the type of task – manual vs. automated.

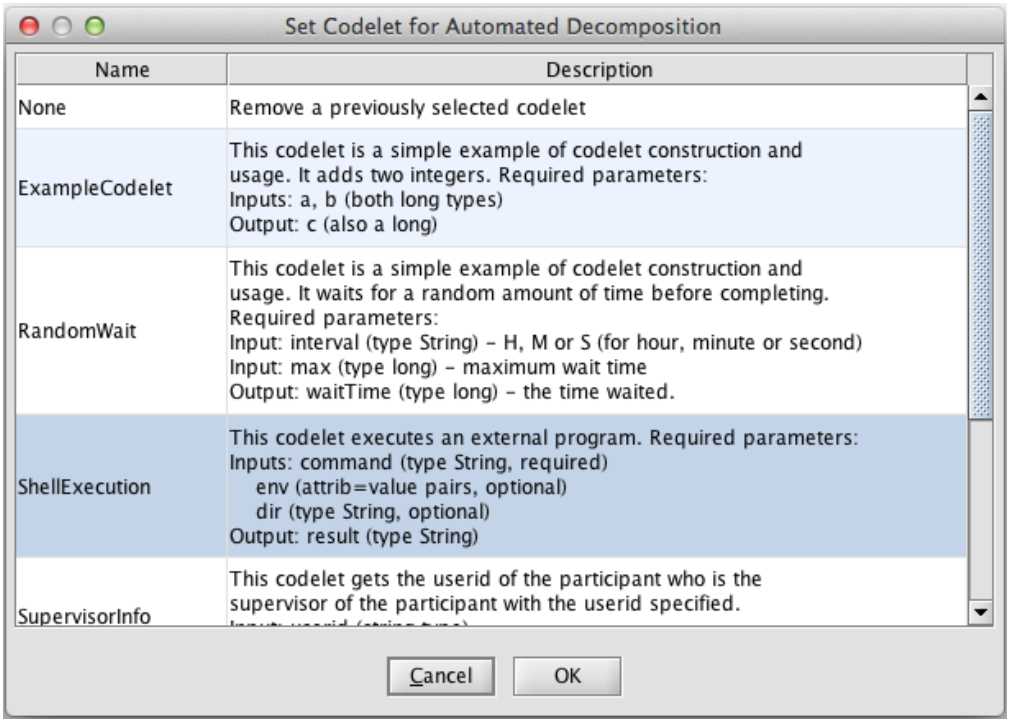


Figure 4.52: The Set Codelet dialog

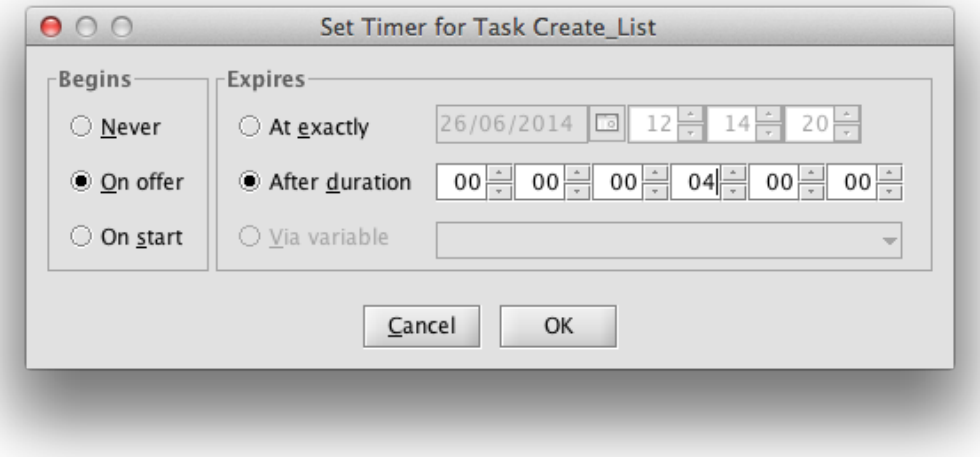


Figure 4.53: Timer Dialog

4.13.1 Activation on enablement

- In the case of a manual task, as soon as the task is enabled, the timer begins and it remains live so long as the specified expiry time is not reached. During this time frame, the task will follow the normal resource assignment policy. In other words, it will be offered and can be allocated and started. Once the timer expires, the task instance will complete no matter what its current status is (offered, allocated, started). A possible danger of this behaviour is that a work item might expire while being edited by a user, in which case any modifications the user makes after that time are lost. The default

YAWL worklist shows a "time until expiry" value for each timed task.

- In the case of an automated task, the timer works as a delay, i.e. the automatic execution of the task instance created by an automated task is delayed until the specified expiry time is reached. Once the timer expires, the task is immediately executed and completed.

4.13.2 Activation on starting

- In case of a manual task, the timer begins only when the task has started. Therefore, the task will be first offered, then allocated, and once it is started the timer begins. Again, there is a risk of the timer expiring while the task is being edited by a user.

- This option is does not apply for an automated task.

4.13.3 Expiry value

The expiry value of the timer specifies for how long the timer will live after being activated. Expiry values can be expressed in either of two ways:

- Using a specific date and time, which means the timer will expire at the specified moment. To set a specific date and time, choose the 'At exactly' radio button in the 'Expires' section the Set Timer dialog (Figure 4.53) and enter the required values. Care should be taken when setting a specific time value for timers – if it happens that the specified value is earlier than the moment the task is actually enabled or started (depending on when it is set to activate), then the YAWL Engine will recognise that the timer has already expired and immediately complete the work item before it has a chance to appear on a worklist.

- Using a Duration value. A Duration is one of the built-in simple XML Schema data types, and is used to represent a period of time. When a Duration type is used as a timer expiry value, the timer will expire exactly when that period of time has passed since the work item was enabled or started (depending on when it is set to activate). To set a specific duration, choose the 'After duration' radio button in the 'Expires' section the Set Timer dialog (Figure 4.53) and enter the required value in the six 'spin' fields, representing from right-to-left, seconds, minutes, hours, days, months and years (Tip: hover the mouse over each field to see the unit of time it represents). Care should be taken when specifying a duration that includes a 'months' value, since for example 2 months may mean a different number of days depending on what month it is started in.

In addition to the methods described above, timer parameters may also be set at runtime via a declared net-level variable of type YTimerType (a YAWL built-in complex type), so that values can be supplied and late-bound to a task's timer settings. To use this deferred approach:

1. Create a net-level variable of type YTimerType.

2. Click on a task, then select its Timer property. Choose the 'Via variable' option and select the net-level variable created in Step 1 (when 'Via Variable' is selected, all the options in the 'Begins' section are disabled because those options will now be set within the YTimerType variable).

3. Map the net-level variable to another (previous) task in the process, which will be used to capture the required values from a user at runtime. Those values will be used to set the timer parameters on the task selected in Step 2. The values requested are:

   - **Trigger**: when should the timer start? There are two valid trigger values, *OnEnabled* and *OnExecuting*.

- **Expiry**: when should the timer expire? This value can be either a date/time string (for example 12/12/08), which will be interpreted as a specific moment, or as a Duration value, which will be interpreted as a period of time. A Duration value is expressed in the following form:

  PnYnMnDTnHnMnS

  All values start with P (for Period) followed by a non-negative number of years, months, days, then T (for time), followed by a non-negative number of hours, minutes and seconds. The seconds value can have a decimal point and as many digits following the point as required (e.g. to specify fractions of a second). Any zero value parts can be omitted. Valid examples: P1Y4M3DT23H55M1.5S, P2M3D, PT10S.

An example of how a variable of YTimerType appears in a dynamic form at runtime can be seen in Figure 4.54.

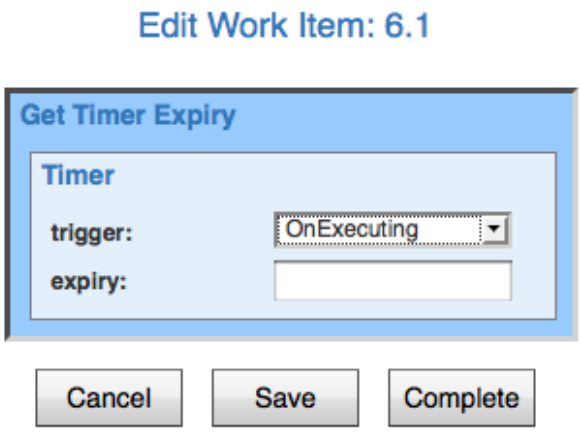


Figure 4.54: Example of a YTimerType variable rendered on a dynamic form

To remove a timer from a task, select the task's Timer property, select the 'Never' option in the 'Begins' section of the dialog, then click OK.

4.14 Document Type – passing files as data

A YAWL built-in complex datatype called *YDocumentType* can be used to upload, store and download files of any description for a process instance. To use this feature, declare a net-level variable of *YDocumentType* and then map it to and from task-level variables of the same type in the usual way (cf. Section 4.7.2). At runtime, users will be able to upload and download files that will be stored as variables in the process instance (Figure 4.55).

At runtime, when a process instance completes, the file can either be archived or removed from the store, depending on a configuration setting in the *DocumentStore* service (cf. Section 6.1). Note that the *DocumentStore* service needs to be available at runtime to support this feature (see Sections 2.4.3 & 10.1).

4.15 Custom Forms

When a task is associated with the default worklist handler (i.e. the Resource Service), then at runtime the data within the task instance may be selected for viewing and/or updating. By default, the Resource Service uses a built in "dynamic forms" component, which generates appropriate but generic data editing forms

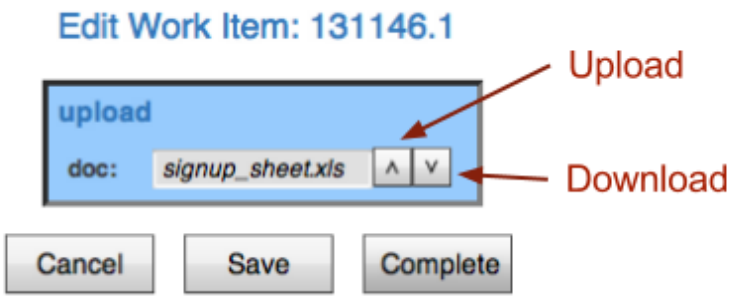


Figure 4.55: Example of a YDocumentType variable rendered on a dynamic form

designed for maximum flexibility and that can display data parameters of any type. However, their generic look and feel may not be appropriate in all cases, for example where an organisation has a standardised set of forms for their business processes, and would like their web-based forms to match that standard. In such cases, a *Custom Form* may be user-defined and associated with a task by specifying a URL to the form. At runtime for such a task, the Resource Service will package up the task data and send it to the custom form for display and/or editing (depending on how the form has been defined). On submission of the form by the user, the data is extracted from the form by the Service and passed back to the task in the same manner as dynamic forms. Custom forms may be built using any web-based technology, such as JSF, Javascript, .NET, PHP, or any other browser-based environment that can receive data, use it to populate form fields, update the data with user inputs, and pass control back to the calling service.

To set a custom form for a task, select the Custom Form property in the Task section of the Properties Pane. In the dialog that appears, enter the absolute URI of the custom form (see Figure 4.56). To remove a custom form association, open the dialog and enter a blank URI (i.e. remove the URI from the dialog and click OK).

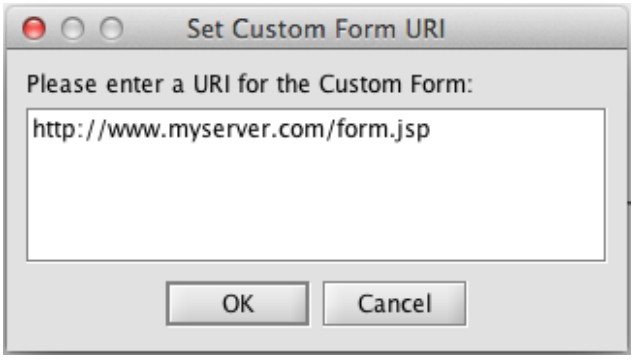


Figure 4.56: Custom form dialog

See the YAWL Technical Manual for information regarding the creation and configuration of custom forms.

4.16 Task Documentation

A task may be annotated with descriptive text that will be viewable on a user's worklist at runtime. This text, or *documentation*, can be used to describe details about each instance of a task.

To add documentation to a task, select the 'Documentation' property from the Task section of the Properties Pane. A dialog will appear into which the text can be added (Figure 4.57).

XQueries that reference net-level variables can be embedded in the text, and are evaluated at runtime when the task is enabled. For example, the text in Figure 4.57 will resolve to "This claim has high priority" at

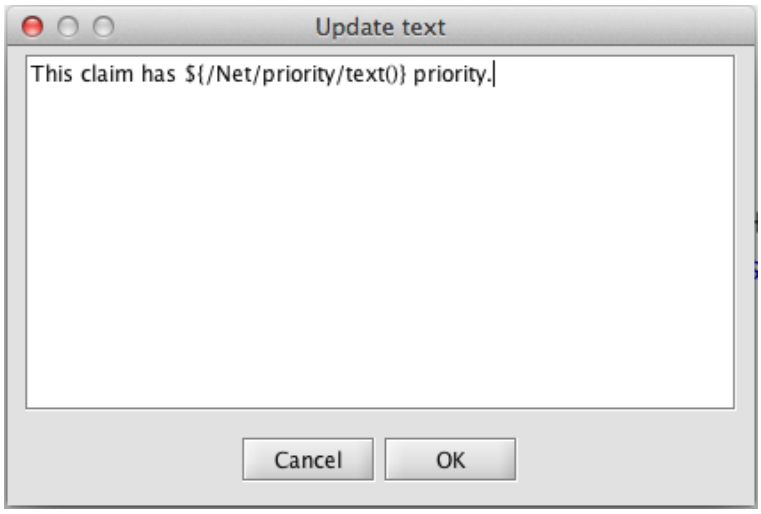


Figure 4.57: Set Task Documentation dialog

runtime, assuming there is a net-level variable called 'priority' that has a value of 'high' when the task is enabled.

At runtime, the task documentation can be annotated on the fly so that, for example, messages can be passed between administrators' and users' work lists regarding the task.

4.17 Configurable Logging

The YAWL process logging framework keeps a record of all aspects of each process execution, including all control-flow, resourcing and data aspects, which can later be analysed. As part of the logging framework, provision is made for *configurable logging*, that is the ability to log messages (known as *Log Predicates*) along with process data during process execution.

Log Predicates may be defined, for each process, that will be logged at the start and completion of each net, at the start and completion of each work item, and when data values are assigned to and from each net and task variable. A log predicate can contain any text, and may also contain *embedded values* describing current values for certain aspects of the process. Embedded values take the form **$**{***keyword***}, and different sets of values are available depending on whether the log predicate refers to a net, task or variable.

The available embedded values for a net-level log predicate can be seen in Table 4.2, while those at the task-level are listed in Table 4.3, and those at the variable-level are listed in Table 4.4.

${now}	Current date and time
${date}	Current date
${time}	Current time
${decomp:name}	Name of the net
${decomp:spec:name}	Name of the specification that contains the net
${decomp:inputs}	Names of any input parameters for the net
${decomp:outputs}	Names of any output parameters for the net
${decomp:doco}	Documentation supplied for the net
${decomp:attribute:*attribute_name*}	The value of the named extended attribute

Table 4.2: Available embedded values for net-level log predicates

If the resource service is handling the task, some extra embedded values become available for use within workitem level log predicates. The resource service will pre-parse the log predicate, replacing the embedded

${now}	Current date and time
${date}	Current date
${time}	Current time
${spec:name}	Name of the specification containing this task
${spec:version}	Version of the specification containing this task
${spec:key}	Internal identifier of the specification containing this task
${task:id}	Runtime identifier of the task
${task:name}	Name of the task
${task:doco}	Documentation supplied for the task
${task:decomposition:name}	Name of the net that contains the task
${item:id}	Runtime identifier of the work item
${item:handlingservice:name}	Name of the service responsible for the work item
${item:handlingservice:uri}	URI of the service responsible for the work item
${item:handlingservice:doco}	Documentation supplied for the service responsible for the work item
${item:codelet}	Name of the codelet to be executed for the work item (if any)
${item:customform}	URI of the custom form to be displayed for the work item (if any)
${item:enabledtime}	Date and time the work item was enabled
${item:firedtime}	Date and time the work item was fired
${item:startedtime}	Date and time the work item was started
${item:status}	Current status of the work item
${item:timer:status}	Current status of timer set for the work item (if any)
${item:timer:expiry}	Expiration time of timer set for the work item (if any)
${item:attribute:*attribute_name*}	The value of the named extended attribute
${expression:*xquery_expression*}	The evaluated value of the XQuery expression specified. The expression may reference work item variable data values

Table 4.3: Available embedded values for work item-level log predicates

${now}	Current date and time
${date}	Current date
${time}	Current time
${parameter:name}	Name of the variable
${parameter:datatype}	Data type of the variable
${parameter:namespace}	Data schema namespace of the variable
${parameter:doco}	Documentation supplied for the variable
${parameter:usage}	Whether the variable is input, output or both
${parameter:ordering}	Index of the variable's order compared to the other task variables
${parameter:decomposition}	Name of the decomposition that contains the variable
${parameter:initialvalue}	Initial value of the variable (if any)
${parameter:defaultvalue}	Default value of the variable (if any)
${parameter:attribute:*attribute_name*}	The value of the named extended attribute

Table 4.4: Available embedded values for variable-level log predicates

values it recognises with actual values, before passing the log predicate to the engine for final parsing of the workitem-level embedded values described above. The available embedded values for tasks handled by the resource service can be found in Table 4.5. Other custom services may also provide their own custom embedded values for log predicates – consult the documentation of custom services in the Technical Manual for details.

${participant:name}	Name of the participant handling the work item
${participant:userid}	Userid of the participant handling the work item
${participant:offeredQueueSize}	Number of items in the handling participants offered queue
${participant:allocatedQueueSize}	Number of items in the handling participants allocated queue
${participant:startedQueueSize}	Number of items in the handling participants started queue
${participant:suspendedQueueSize}	Number of items in the handling participants suspended queue
${resource:initiator:offer}	Whether the workitem's offer interaction was initiated by the *system* or by a *user*
${resource:initiator:allocate}	Whether the workitem's allocate interaction was initiated by the *system* or by a *user*
${resource:initiator:start}	Whether the workitem's start interaction was initiated by the *system* or by a *user*
${resource:offerset}	A comma separated list of the names of all participants the workitem was offered to
${resource:piler}	The name of the participant piling the workitem (if any)
${resource:deallocators}	A comma separated list of the names of all participants who have deallocated the workitem
${resource:allocator}	Name of the system-based allocation algorithm used to allocate the workitem
${resource:roles}	A comma separated list of the names of all roles the workitem was offered to
${resource:dynParams}	A comma separated list of the names of all dynamic parameters used in the offer set evaluation for the workitem
${resource:filters}	A comma separated list of the names of all filters used in the offer set evaluation for the workitem
${resource:constraints}	A comma separated list of the names of all constraints used in the offer set evaluation for the workitem

Table 4.5: Additional resourcing embedded values for work item-level log predicates

Log predicates are optional, and so may be left empty if desired. Net-level log predicates can be added via the Log Entries property in the Net section of the Properties Pane, while task-level log predicates can be added via the Log Entries property in the Decomposition section of the Properties Pane. To add a task-level variable log predicate, open the Data Variables dialog from the Decomposition properties, select the variable then click the *Log Entries* toolbar button, . In each case, the Log Entries dialog appears where you can enter log predicates that will be logged when the relevant net, task or variable begins and ends. Figure 4.58 shows an example at the variable level, indicating that correctly entered embedded keywords appear green, while incorrect ones appear red (if not corrected, they will not be parsed at runtime but will appear as literal strings in the logs).

4.18 Extended Attributes

The Editor offers a means for defining extended attributes to be associated with task decompositions and variables. There are a default set of attributes supplied for task decompositions, and a default set for task variables, the values of which may be set at design time; at runtime, the values (for the most part) will effect the display parameters of the dynamic form generated by the Resource Service's default worklist handler for the work item.

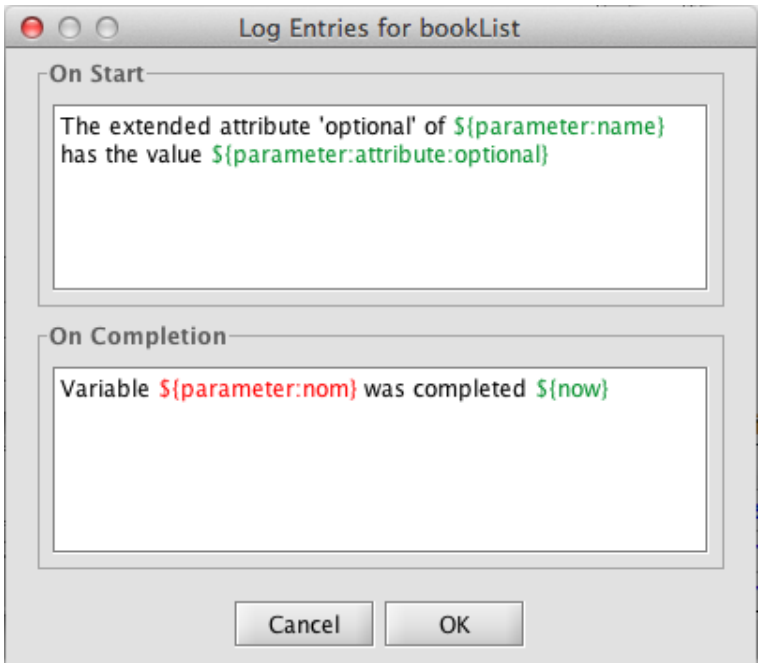


Figure 4.58: Entering variable-level log predicates

Additionally, designers may add their own extended attribute definitions to the Editor at design time, for runtime interpretation via custom classes and services. That is, the Engine and/or Resource Service will interpret and act on any values set for the default extended attributes, while the effects of values set for user-defined extended attributes are defined by developers of custom services. See Section 4.18.2 for information on defining your own extended attributes.

4.18.1 The Default Extended Attributes

Table 4.6 lists the default extended attributes for decompositions. For the most part, the attributes affect the work-item level of the dynamic form displayed (i.e the form itself). Note that where an variable-level attribute of the same name as a decomposition-level attribute exists, the variable-level attribute takes precedence.

Attribute	Sets	Default
Background Alt Colour	The background colour of alternating form panels	light blue (#D3E0FC)
Background Colour	The main background colour of the form	blue (#97CBFD)
Font	Font family, size, style & colour for labels	browser default, 12, plain, black
Header Font	Font family, size, style & colour for headings	browser default, 14, Bold, dark teal (#3277BA)
Hide Banner	Suppress display of the YAWL banner, if true	false
Justify	Justify text in text fields	left
Label	Label text for form header	task name
Page Background Colour	The background color of the page	white
Page Background Image	The background image url for the page	none
Read Only	Field values can't be modified, if true	false
Title	Title of form	Edit Work Item: *item id*

Table 4.6: Default decomposition-level extended attributes

Decomposition Extended Attributes can be updated by selecting the Extended Attributes property in the Decomposition section of the Properties Pane, then setting the desired values in the dialog that appears (see Figure 4.59 for an example).

Table 4.7 lists the default variable level extended attributes. For the most part, the attributes affect a single variable field on the dynamic form displayed.

Attribute	Sets	Default
Alert	A tailored validation error message	auto-generated message
Background Colour	The background colour of the field	yellow (mandatory); white (optional)
Blackout	Show field blacked out (unviewable), if true	false
Font	The font family, size, style & colour for the field & label	browser default, 12, plain, black
Fraction Digits*	Number of digits to show after the decimal point	none
Hide	Hide the field from view, if true	false
Hide If	Hide the field from view, if the XQuery value evaluates to true	false
Image Above	URL for an image to show above the field	none
Image Above Align	Alignment for image above the field	left
Image Below	URL for an image to show below the field	none
Image Below Align	Alignment for image below the field	left
Justify	Justify text in the field	left
Label	Text for the field label	variable name
Length*	Exact number of characters required	undefined
Line Above	Draw a horizontal line above the field, if true	false
Line Below	Draw a horizontal line below the field, if true	false
Max Exclusive*	One less than the upper range of valid values accepted	undefined
Max Inclusive*	Upper range of the valid values accepted	undefined
max Length*	Maximum number of characters required	undefined
Min Exclusive*	One more than the lower range of the valid values accepted	undefined
Min Inclusive*	Lower range of the valid values accepted	undefined
Min Length*	Minimum number of characters required	undefined
Optional	Field does not require a value, if true	false
Pattern*	Field value must match the value pattern	undefined
Read Only	Field value can't be modified, if true	false
Skip Validation	Don't validate value against schema, if true	false
Text Above	Insert given text above the field	undefined
Text Below	Insert given text below the field	undefined
Text Area	Render a text area instead of a text field (text fields only)	
Tool Tip	A tip to show when the mouse hovers over the field	auto-generated tooltip
Total Digits*	Total number of digits expected (numeric values only)	undefined
Whitespace*	Normalise whitespace in the given value	undefined

Table 4.7: Default variable-level extended attributes

Variable Extended Attributes can be updated by selecting the Data Variables property in the Decomposition section of the Properties Pane, selecting the appropriate task-level variable from the Data Variables Dialog that appears, then clicking the Extended Attributes toolbar button, , and setting the desired values in the Extended Attributes dialog.

Notes about extended attributes:

- Those attributes marked with an asterisk (*) in Table 4.7 mirror XML Schema *facets* that may be set for values as part of the type definition for a field. These facet attributes are only available for simple data types. If a extended attribute facet has a value, and the type definition also has a value for the facet, the extended attribute value takes precedence. Note that not all facets make sense for all data types –

where a value is set for an extended attribute that mirrors a facet, and the field in question does not support that facet, then the value is ignored. Please refer to an XML Schema reference for more information about facets and their application to different data types (for example, http://www.w3.org/TR/xmlschema-2/#rf-facets and http://www.w3schools.com/schema/schema_facets.asp).

- Values for the text-above, text-below and label attributes may include embedded XQuery/XPath expressions that reference the work item's data. See the Section 4.18.3 for examples.
- The Read Only attribute only applies to fields that would otherwise not be read-only. That is, a variable with usage type *Input Only* will display as read-only regardless of the value of the Read Only attribute, so the attribute will only apply to variables of usage types *Input & Output* or *Output Only*.
- For attributes that require a URL value, absolute URLs must be used. A simple solution is to create a subdirectory in the tomcat/webapps directory (called, for example, 'images') and place the images within it. Then, the absolute URL would be, for example, http://localhost:8080/images/myImage.png (assuming tomcat is installed locally).
- To avoid confusion, care should be taken when overriding the background colour of a field, that the colour used for validation errors (#FFCCCC), or colours close to it, are not used.
- A mechanism exists for extended attribute values to be set and/or modified at runtime – please see the Technical Manual for more information.

4.18.2 User-Defined Extended Attributes

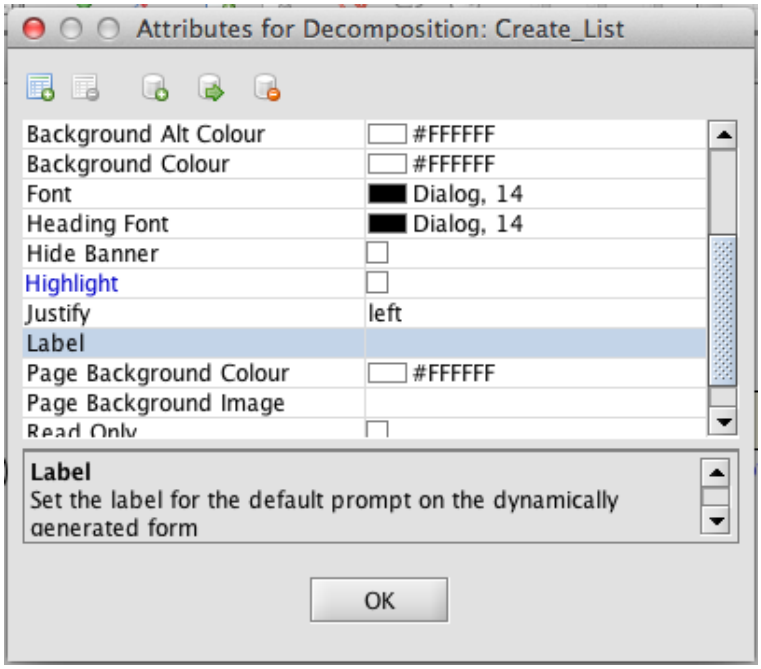


Figure 4.59: Decomposition-level extended attributes dialog

User-defined extended attributes are specified in property files, one for decomposition-level attributes and one for variable-level attributes. By default, the Editor will look for attribute property files in the folder:

```
<editor_install_path>/attributes/
```

In that folder, the Editor will look for a file named *decomposition.properties* for decomposition-level properties, and a file named *variable.properties* for variable-level attributes.

However, the property files can be placed in any folder and use any filename – in which case, the location of the particular property file must be set in the File Paths section of the Preferences dialog (**File... Preferences**). The Editor will always look for the file in the last folder/filename set (or the default path/filename if never set).

In the Extended Attribute dialogs, the default attribute set is listed alphabetically in black text, while any user-defined attributes are listed alphabetically in blue text (case sensitive). The property files can be edited in any text editor (the YAWL Editor will need to be restarted to pick up any changes). Attributes are defined as *key=value* pairs; the key represents the name of the attribute, and the value its type. The available attribute types are:

- **boolean**: may be given a value of true or false. Rendered as a checkbox in the Extended Attributes dialog.
- **string**: a simple text string. Rendered as a text field.
- **color**: a colour value. Renders as a text field which shows a hex colour value.
- **font**: a valid font name. Renders as a text field which shows the font details.
- **integer**: a valid integer value. Renders as a text field. Invalid entries are ignored.
- **double**: a valid double value. Renders as a text field. Invalid entries are ignored.
- **enumeration**: a set of string or numeric values. Renders as a dropdown dialog.
- **xquery**: a valid XQuery expression.
- **text**: a long text value.

When the field of an attribute with color, font, xquery or text type is selected, a small 'Open Dialog' button appears on the right-hand side of the field. When clicked, a dialog appropriate to the type will be displayed. Enter or select an appropriate value from the dialog, then click OK to save the value to the attribute. Figure 4.60 shows the dialog that appears for attributes of type *font*.

The following is an example of a property file that defines the attributes description, help, mode, refresh and showDetails (comments are indicated by a line starting with #).

```
#Decomposition Attributes
#Wed May 14 17:35:42 AET 2014
description=xquery
help=string
mode=enumeration{normal,final,pending}
refresh=xquery
showDetails=boolean
```

Only those values that are changed from their defaults will be saved to the specification file. For all types except *enumeration*, the default value is considered to be an empty field. For enumerations, the first listed value is considered the default. To denote an empty value as the first in an enumerated list, list the enumeration like this example (i.e with no first value):

```
mode=enumeration{,normal,final,pending}
```

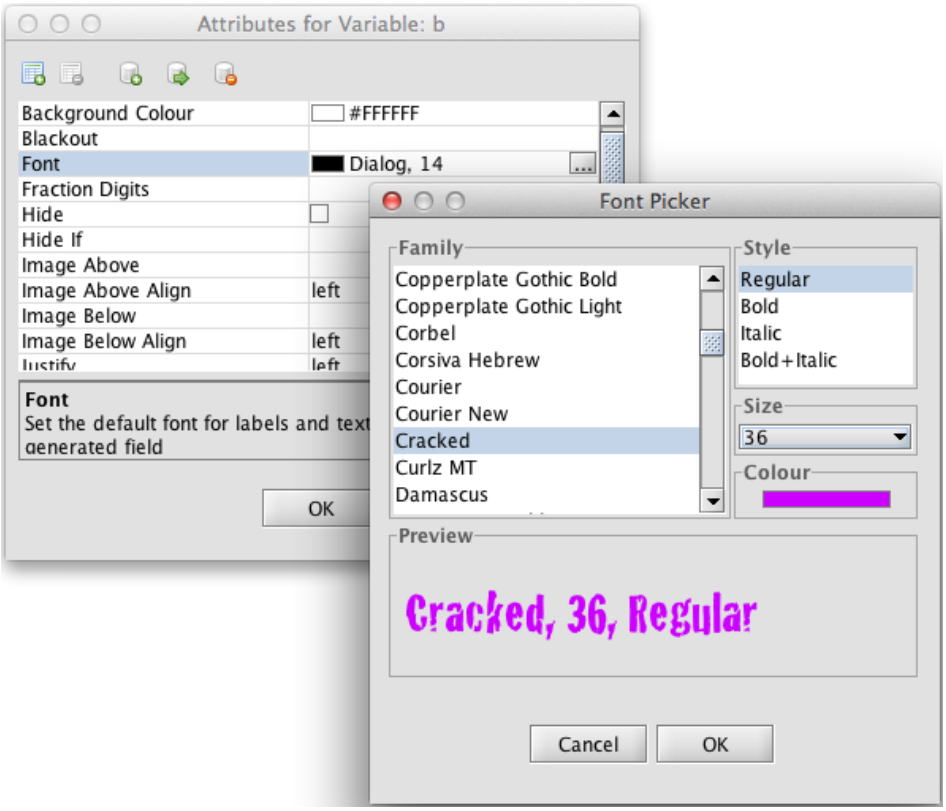


Figure 4.60: Extended attributes font dialog

Adding Attributes via the Dialog

As an alternative to creating user-defined attributes via a text editor, they can also be added or removed directly via the Extended Attributes dialog. The tool bar of that dialog has two sets of buttons, the first of which are buttons for adding and removing user-defined attributes (cf. Figure 4.59).

To add a user-defined attribute, click the add button ( ) and in the dialog that appears add a name and data type for the new attribute. For enumeration types, a field will appear for you to define the enumeration's list of possible values (Figure 4.61). Click OK to save.

To remove a user-defined attribute, select it then click the remove button ( ). Note that only user-defined attributes may be removed.

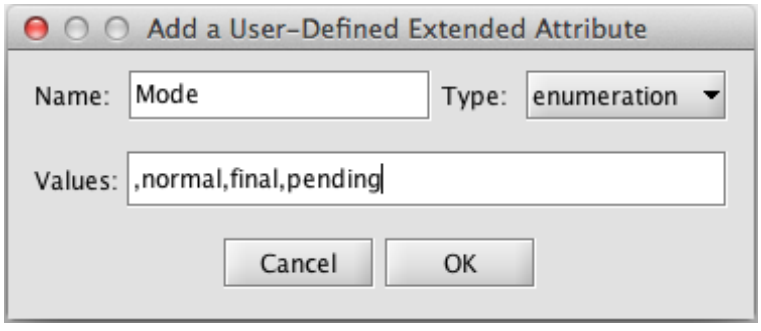


Figure 4.61: Add User-Defined Extended Attribute dialog

The Extended Attribute Repository

The second set of buttons on the Extended Attributes dialog tool bar allow you to save attribute values to the repository and later retrieve them for other decompositions and variables.

To add attributes and their values to the repository, click the 'Add to Repository' button (), add a label and description of the values to the dialog that appears, then click OK to save. All of the attributes in the dialog that have non-default values will be saved to the repository under the label provided.

To retrieve previously saved values for attributes from the repository, click the 'Load from Repository' button (), select the label for the desired values from the list, then click OK. All of the values previously saved to the repository under the selected label will be added to the dialog (and will overwrite any existing values set for those attributes).

Finally, existing attribute-value sets can be removed from the repository at any time by clicking the 'Remove from Repository' button (), selecting the label for the set to remove, then clicking OK.

4.18.3 Extended Attribute Example

To show a small sample of what can be done with extended attributes, a simple specification called *EternalQuestion* will be used. The specification consists of two atomic tasks; the first requests an answer to the eternal question from a user, while the second shows the answer provided (in an Input Only variable). Figure 4.62 shows the process model, while Figure 4.63 shows how the dynamic form for the *Answer* task is displayed by default.

Figure 4.62: The EternalQuestion specification

We would like to change the look of the form for the Answer task from the generic default to something more appropriate for this process, so values are provided for external attributes as follows:

- At the decomposition level:
  - **Background Colour**: white (#FFFFFF)
  - **Page Background Image**: `http://localhost:8080/images/guide2.jpeg`
  - **Hide Banner**: true
  - **Title**: a space (to suppress the title display)
  - **Label**: a space (to suppress the task-name label display)
- At the variable-level (the 'answer' variable):
  - **Font**: Capitals, 20, plain, dark green (#009900)
  - **Image Above**: `http://localhost:8080/images/guide.jpeg`
  - **Image Above Align**: center
  - **Hide**: true (to suppress display of the text field)
  - **Text Below**: The answer to the meaning of life, the universe and everything is ${/Answer/answer/text()}

The effects of setting the attribute values listed can be seen in Figure 4.64. Remember that nothing has changed in the specification except for the setting of the extended attributes listed.

Notice that we have hidden the display of the field itself, because we have embedded the variable's value in the text-below attribute, via an XPath expression. This is a quite powerful construct, allowing usages like:

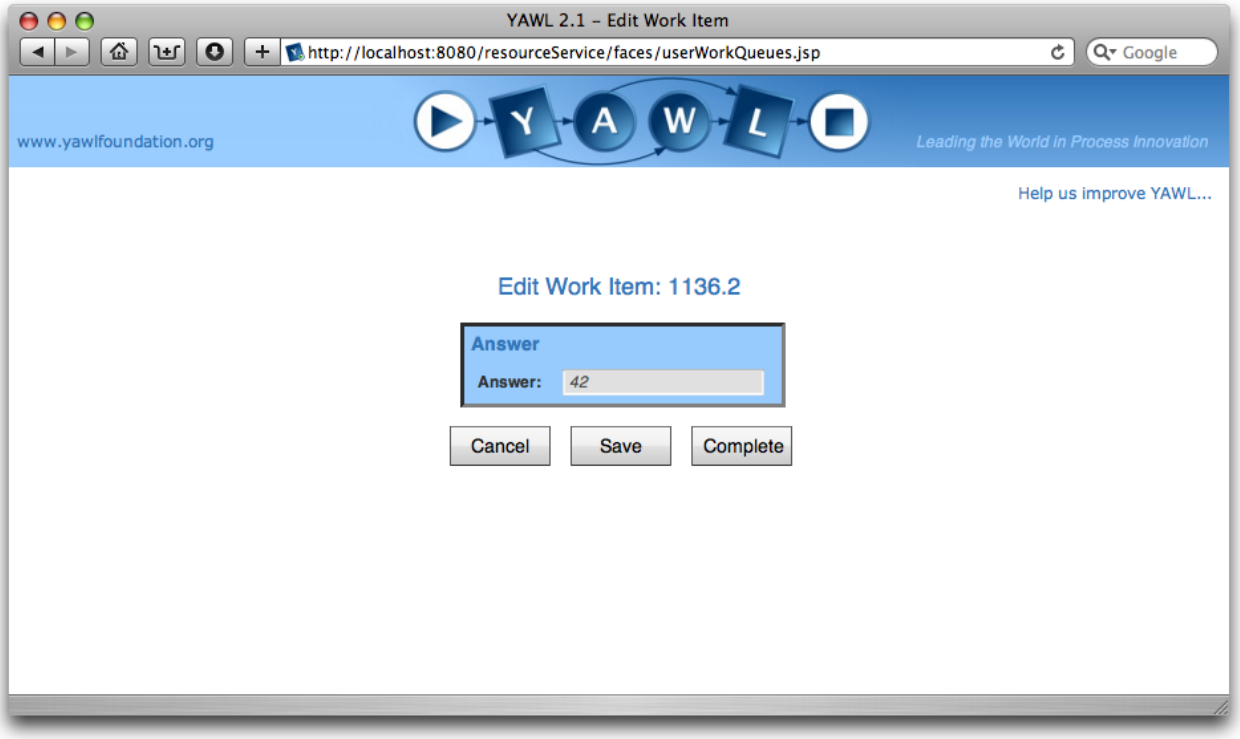


Figure 4.63: The default dynamic form for the Answer task

- "Total charge: ${number(/Order/subtotal/text()) + number(/Order/tax/text())}"
- "You ${if (number(/Exam/score/text()) > 50) then 'passed' else 'failed'} the exam!"
- "${if (boolean(/Question/response/text())) then 'Correct.' else 'Sorry, try again.'}" (assumes the response variable has a value of 'true' or 'false').

4.19 Configurable YAWL

A configurable process model is an integrated representation of multiple variants of a same business process in a given domain, such as multiple variants of an insurance business process operating in different markets. A configurable process model offers the following benefits over traditional process models:

- eliminates redundancies in a process family,
- fosters standardization and reuse of proven practices,
- enables a clear distinction between commonalities (those parts that are shared by all process variants) and variants (those parts that are specific to certain process variants) in a process family,
- can be configured to meet specific requirements, such as those of a new organization, product or brand.

The YAWL Editor supports the definition of *configurable YAWL* (C-YAWL) models via an Editor plugin. A C-YAWL model is a YAWL model where some tasks are annotated as *configurable*. These configurable tasks represent the 'variable' parts of the process model, and are distinguished by a thicker border from the remaining tasks representing commonalities. Let's have a look at the example C-YAWL model in Figure 4.65.

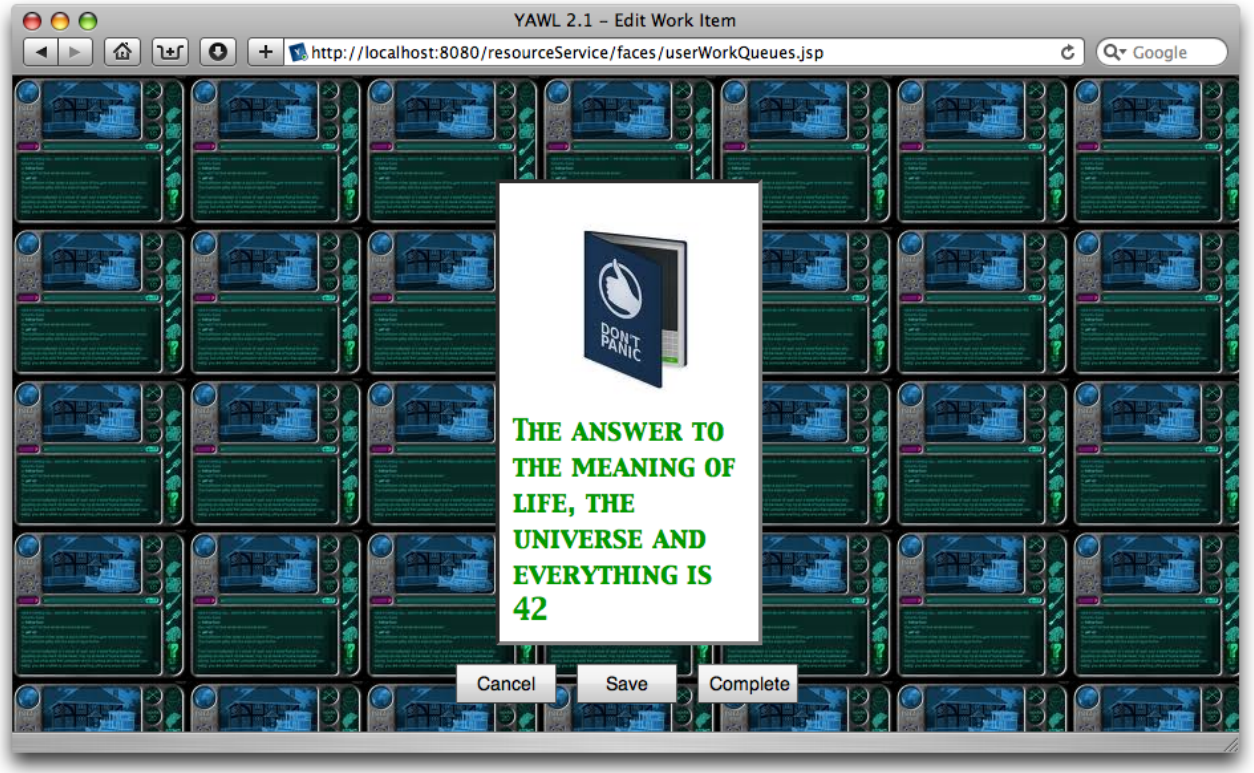


Figure 4.64: The dynamic form for the Answer task with certain extended attributes set

This model depicts a travel requisition process (you can find it in the YAWL distribution under the folder `C-YAWL models`). In this example, all tasks but the task labeled "tau" are configurable.

Configuration is achieved by restricting the behavior of a C-YAWL model. Configurable tasks can be restricted via the notion of *ports*. A configurable task's joining behavior is identified by one or more *input* ports, whereas its splitting behavior is identified by one or more *output* ports.

The number of ports for a configurable task depends on the task's routing behavior:

- AND-split, AND-join and OR-join are each identified by a single port.
- XOR-split and XOR-join are identified by one port for each outgoing/incoming flow.
- an OR-split is identified by a port for each combination of its outgoing flows.

For example, task "Submit Travel Form for Approval" has two input ports: one from task "tau", the other from task "Check and Update Travel Form", and three output ports: one towards task "Approve Travel Form Admin", one towards task "Reject Travel Form" and one towards task "Request for Change", while the latter task only has one input port and one output port.

To make a task configurable, select it, then from the **Plugins** menu select **Process Configuration** > **Task** > **Set Task Configurable**, or click the toolbar button on the Process Configuration toolbar (Tip: to make the toolbar visible, from the **Plugins** menu select **Toolbars** > **Process Configuration**).

For example, let's make task "Submit Travel Form for Approval" configurable. Now we are ready to configure this task.

All ports are *activated* by default. You can configure them by either *blocking* or *hiding* them:

- Input ports can be hidden or blocked. Hiding an input port corresponds to making the task silent, i.e. the task will still be executed but its decomposition will be removed and thus the task won't have any

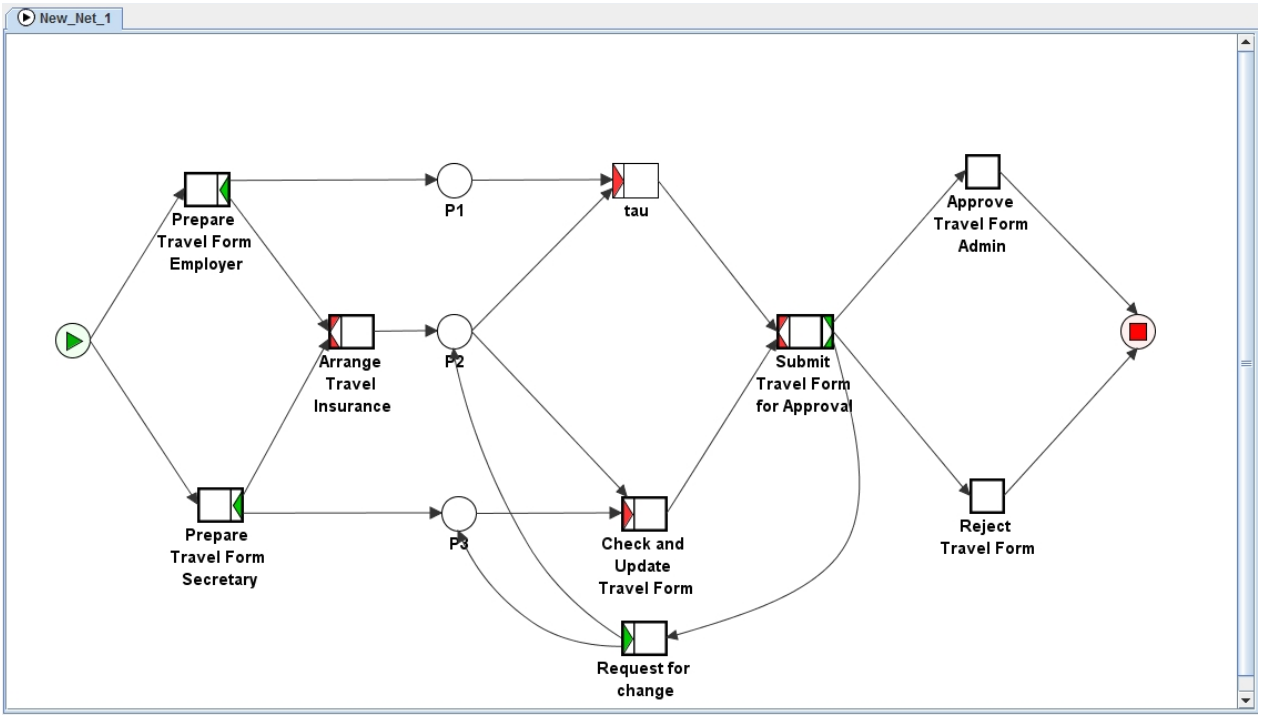


Figure 4.65: A C-YAWL model for travel requisition

observable behavior. Blocking an input port corresponds to inhibiting control to the task via that port, i.e. it will no longer be possible to execute the task via that port.

- Output ports can only be blocked. This means that the outgoing paths from that task via that port will be disabled.

Moreover, you can block the cancellation region assigned to a task (this means that the task's region will be removed altogether), and restrict the parameters of a multiple instance task. Specifically, you can reduce the maximum number of allowed instances, increase the minimal number of allowed instances and the continuation threshold, and change the dynamic creation of instances to static.

The configuration of input ports, output ports, cancellation regions and multiple instance parameters, can be accessed from the **Plugins**, and then selecting **Process Configuration** > **Task** > **Input Ports...**, **Output Ports...**, etc, or via the corresponding toolbar buttons. Figure 4.66 shows the dialog window for the output ports configuration of task "Submit Travel Form for Approval". You can select a single port or multiple ports by using the Shift or Ctrl keys, and then press the **Hide** or **Block** button to configure the selected ports, or **Activate** to rollback a configuration. It is also possible to set default configuration values for each port, by pressing the button **Set Defaults**. Once default configuration values have been assigned to a port, the button **Use Default** will become available when selecting that port, and you can configure that port by using its default value. All configuration settings (including default values) are stored in the YAWL specification upon saving, so this information will be preserved after closing the model.

Let's block the output port of task "Submit Travel Form for Approval" towards task "Request for change", and let's make task "Check and Update Travel Form" silent, by hiding its input port. With the first operation we deny requests for changes after a travel form has been submitted, by blocking the flow into the loop path, while with the second operation we deny the possibility of checking and updating a travel form, although in this case we don't block the flow through the hidden task.

The YAWL Editor can show a preview of the resulting configured net by greying out all model fragments which have been removed. You can do this from the **Plugins** menu via **Process Configuration** > **Net** > **Preview Process Configuration**, or by clicking the **Preview Process Configuration** button, , from the tool

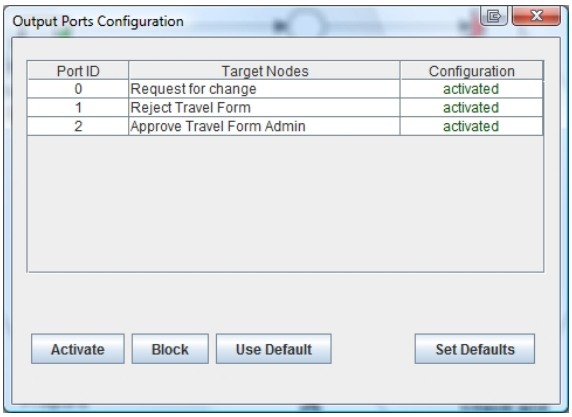


Figure 4.66: Configuring the input ports of a task.

bar. Figure 4.67 shows the preview of the configuration in our running example. Task "Request for change" and its input and output arcs have been greyed out, whereas task "Check and Update Travel form" is still in the model (this task will actually be replaced with a silent task once the configuration has been committed). To remove a preview, simply click again on the Preview toolbar button.

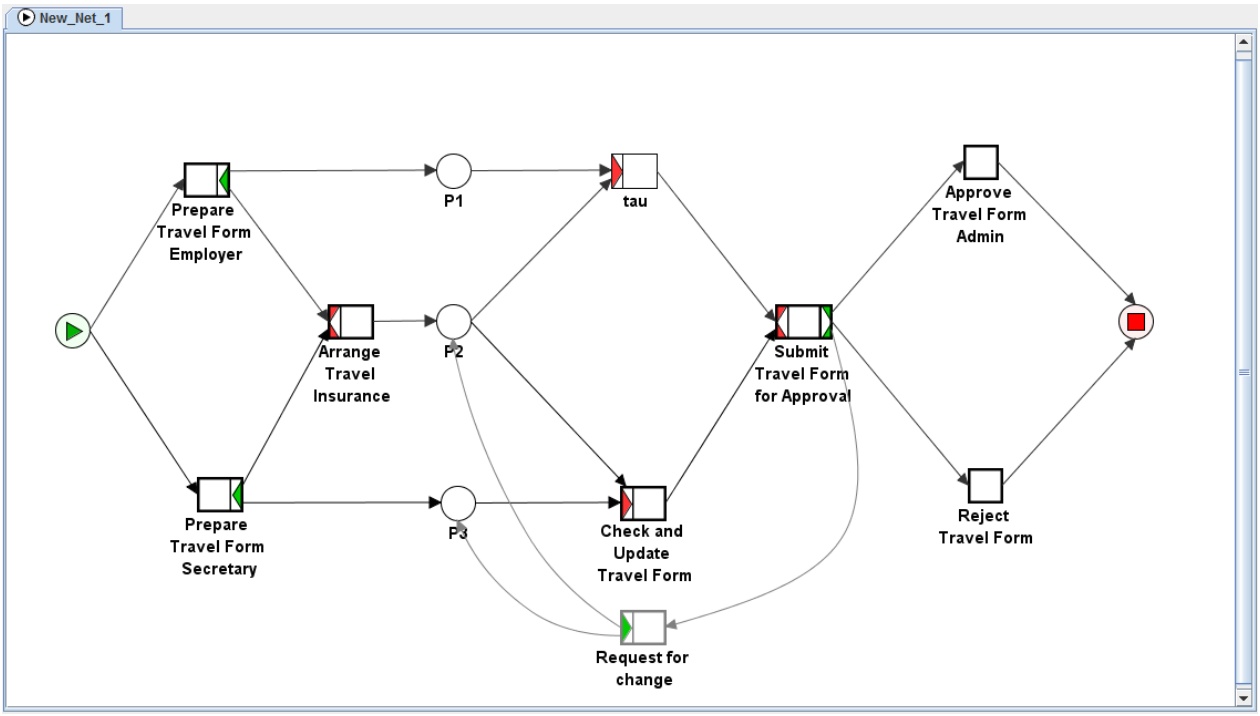


Figure 4.67: The preview of a process model configuration.

To commit a configuration you need to press the **Apply Process Configuration** button, , from the toolbar or from the **Plugins** menu **Process Configuration** > **Net** > **Apply Process Configuration**. This operation generates an *individualized* YAWL model, i.e. a regular YAWL model where:

- all blocked tasks and cancelation regions are removed,
- tasks attached to a hidden input port are replaced by a silent task bearing label "_tau",

- the parameters of all configurable multiple instance tasks are restricted according to the configuration settings,
- all configurable tasks are turned into normal tasks.

Note that since a task can have multiple input ports (e.g. in the case of an XOR-join), those input ports that are not hidden will not be replaced with a silent task. For more information on how the configuration of hidden ports work, please refer to the Process Configuration book chapter of the YAWL Book [17].

Figure 4.68 shows the individualized YAWL model for the travel requisition example. Task "Request for change" and its connecting flows have been removed, whereas task "Check and Update Travel Form" has been replaced by a silent task labelled "_tau". If you inspect this task, you will see that it is no longer associated with a decomposition. Let's undo the commitment of this configuration to revert its effects. This can be done simply by pressing again the toggle button **Apply Process Configuration** on the toolbar.

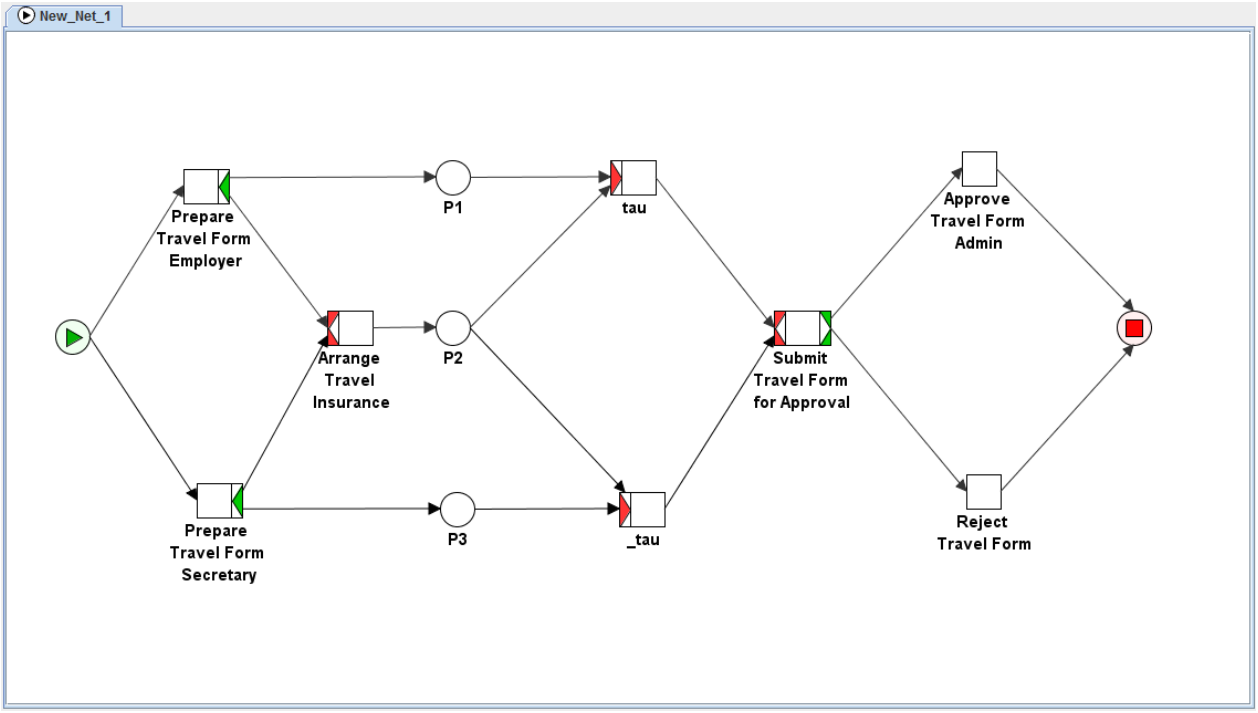


Figure 4.68: The result of committing a process model configuration.

While hiding a port is a safe operation, blocking a port may lead to behavioural anomalies such as deadlocks. If we now blocked the input port of the task "Check and Update Travel Form", we could cause a deadlock in condition "P3" because if task "Prepare Travel Form Secretary" fired, a token would remain stuck in that condition. So it would be wise to also block the input port of task "Prepare Travel Form Secretary", so that "P3" could never get control. The YAWL Editor provides a Configuration Correctness Checker which can be used to prevent possible behavioral issues already during configuration, by excluding those combinations of blocked ports that will lead to an unsound individualized YAWL model. In this way you don't need to individualize a process model to find out only later that the configuration you used led to an unsound model.

The Configuration Correctness Checker needs an external tool, namely Wendy [18], which must be located at the path specified in the Process Configuration Settings, which can be found from the **Plugins** menu **Process Configuration** > **Preferences**, or by clicking the Preferences button, , on the Process Configuration toolbar. There are Wendy distributions for Windows 32bit, MacOS and Linux available for download from the YAWL SourceForge project.

The Configuration Correctness Checker can be enabled at any time from the **Plugins** menu **Process Configuration** > **Net** > **Check Configuration Correctness**, or by clicking the Check Correctness button, , on the Process Configuration toolbar. This will invoke Wendy to analyse the current model, and the output from this tool will appear in a pop-up window (see Figure 4.69). This process may take some time depending on your machine's characteristics and on the degree of parallelism of your YAWL model (i.e. how many combinations of tasks there exist that can be executed in interleaved parallelism). However you only have to run this process once, once you have completed the design of your process and are ready to configure it. You can also interrupt this process at any time by closing the window.

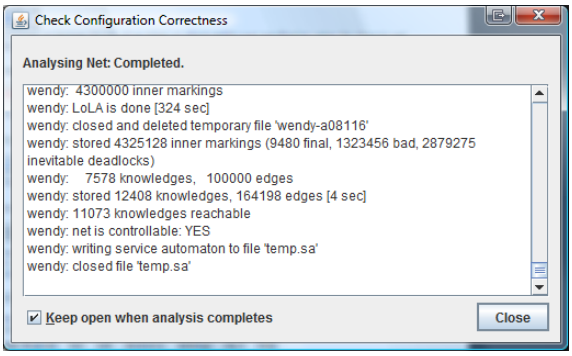


Figure 4.69: Output from Wendy.

Now launch the Configuration Correctness Checker for the travel requisition process. Once the analysis has been completed, the YAWL Editor will be able to automatically block or activate further ports in an interactive way, i.e. as you configure your model. Let's now block the input port of task "Check and Update Travel Form", which was previously just hidden. The Editor will react to that configuration by also blocking the input port of task "Prepare Travel Form Secretary", to avoid possible deadlocks in the individualized model. This is notified to the user via a dialog window.

Figure 4.70 shows the preview of the resulting individualized model. You can see that condition "P3" will also be removed from the net now, as a result of applying this configuration.

Automatic completion also works when you activate ports that were previously blocked. Try now to activate the input port of task "Prepare Travel Form Secretary". Since the input port of "Check and Update Travel Form" is blocked, a token may again get stuck in "P3"; so the Editor will also activate the latter port.

The following parameters related to C-YAWL models can be customized from the **Plugins** menu **Process Configuration** > **Preferences**, or by clicking the Preferences button, , on the toolbar:

- **Set new elements configurable**: new tasks are set as configurable automatically/manually (default option),
- **Preview process automatically**: the preview of configured processes is always switched on—the corresponding button on the tool bar becomes inactive/the preview can be switched on or off manually (default),
- **Deny blocking input ports**: input ports cannot be blocked/can be blocked (default),
- **Allow changing default configurations**: default configuration values can be changed (default)/cannot be changed.

For more information on C-YAWL, please refer to the Process Configuration book chapter of the YAWL Book [17] and go to www.processconfiguration.com. For technical details, you can also read the scientific paper [16].

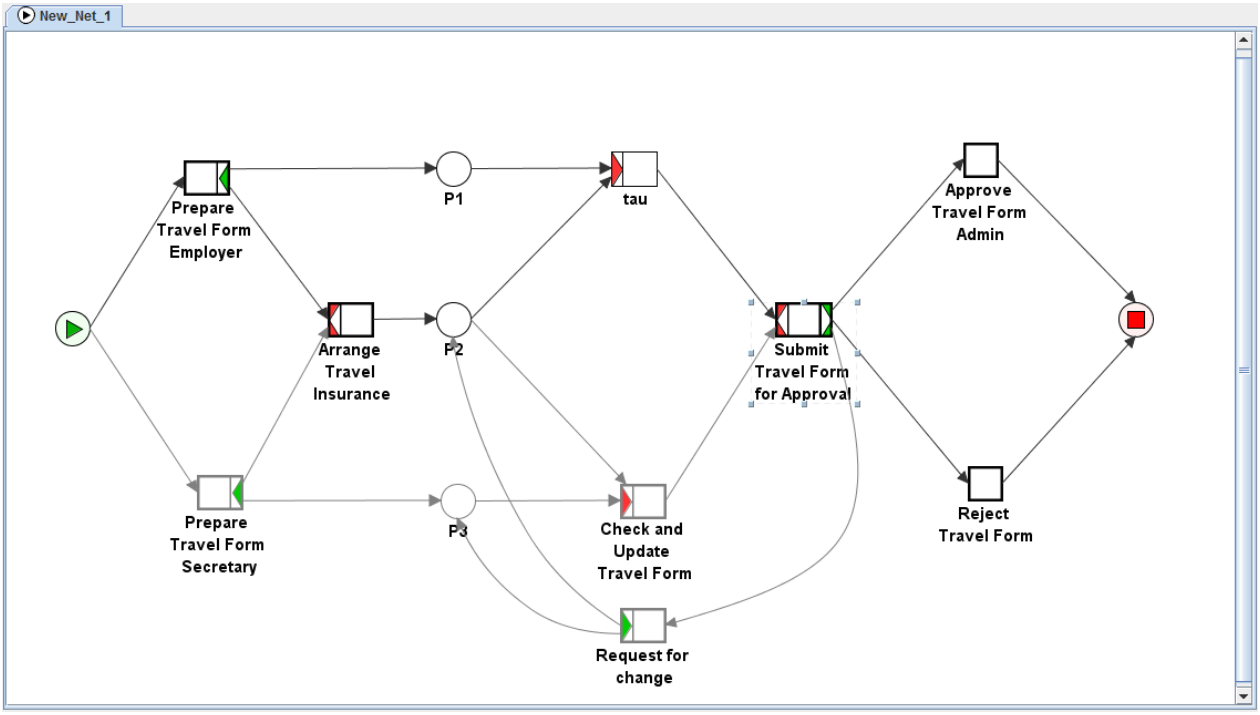


Figure 4.70: Preview of the configuration after blocking the input port of task "Check and Update Travel Form"

4.20 Checking for Updates

At any time, you may check to see if there are any updates of the YAWL Editor for downloading and installing (a valid internet connection is required).

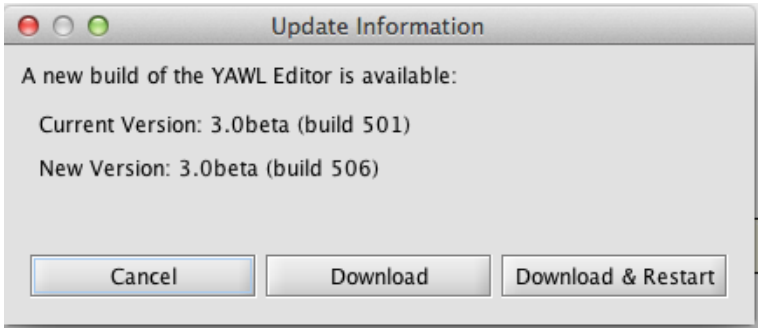


Figure 4.71: Update Information dialog

To check for updates, select **Help**, then **Check for Updates...** from the menu bar. The update server will be contacted and the latest list of components downloaded, then compared to the installed components locally (*i.e.* ***no*** *information is sent to the upload server*). If there are updates available, the *Update Information* dialog will appear (Figure 4.71) and display information about the currently installed version and the latest version. You have three options:

- **Cancel** Close the dialog and take no further action
- **Download** Download the updated components to a directory of your choice (a File...Save dialog will appear). You can later apply the updates manually to your installed version of the Editor.

- *Download & Restart* The updates will be downloaded, your installed version will be automatically updated, then restarted to pick up the changes.

When either of the 'Download' options are chosen, the downloads will begin and you will see a progress bar in the update dialog. Note that only the components that are newer than the ones installed are downloaded. After the restart (if chosen), a message will display regarding the new version.

If there has been a major version update, the entire Editor package will need to be downloaded and the option to install and restart will be disabled. Instructions will be shown on how to install the new major release version.

If there are no updates available for your installed version, a message to that effect will appear.

4.21 About Dialog

To view information about the currently installed Editor version, select **Help**, then **About...** from the menu bar. The build major version, build number and build data and time will be displayed. The dialog also provides two web links, one which will take you to the YAWL forum to read or participate in discussion about YAWL, and the other which will take you to the YAWL issues list, where you can report any issues or problems with the Editor.

Click anywhere on the About dialog to close it.

Chapter 5

How to Manipulate Data in YAWL

5.1 Introduction

Compared to most of the existing workflow management systems which use a propriety language for dealing with data, YAWL completely relies on XML-based standards like XPath[1] and XQuery[2] for data manipulation. This document provides some insights into data manipulation in YAWL, in terms of *data visibility* (defining data elements), *data transfer* (passing data between workflow components and exchanging information with the environment), data related issues such as *data-based conditional routing* and handling of *multiple instance data*. Readers are assumed to have knowledge of YAWL and its supporting tools: the YAWL engine (see Chapter 6) and the YAWL editor (see Chapter 4).

5.2 Data Visibility

In YAWL, all data are represented as XML documents. Figure 5.1 depicts an example of a YAWL net specifying a simple trip booking process. The data at the net level are written in an XML document with a root element named *PerformBooking* (i.e. the name of the net), while the data at the task level are written in an XML document for each task.

Next, data elements are stored in variables. There are: net variables for storing data that need to be accessed and/or updated by tasks in a net, and task variables for storing data that needs to be accessed and/or updated only within the context of individual execution instances of a task. Note that the task variables of a composite task are conceptually the net variables of the corresponding subnet.

YAWL applies strong data typing. Data types are defined using XML Schema. The YAWL Editor provides all XML Schema simple data types for variable definition. These include some basic types such as *boolean*, *string*, *double*, *decimal*, *long*, *integer*, *date* (in the format of *yyyy-mm-dd*) and *time* (*hh:mm:ss*). Based on the above, users may also provide their own XML Schema to define more complex data types. Figure 5.2 shows the XML schema of a user-defined data type for element "CustInfo" which consists of both customer name and target start date for the booking trip process depicted in [11].

Data usage (or *scope*) is also part of a variable definition. There are: *input and output* variables, *input only* or *output only* variables, and *local* variables. In general, data are written to input variables and read from output variables. Local data usage is applicable to net variables only. The local (net) variables are used to store data that can be manipulated only *internally* within the scope of the corresponding net.

Finally, a local variable may be assigned an initial value at design time, while an output-only variable may be assigned a default value at design time. Further details will be given in the next section (5.3).

[1] XML Path Language (XPath) 1.0. W3C Recommendation, 16 November 1999.
[2] XQuery 1.0: An XML Query Language. W3C Working Draft, 4 April 2005.

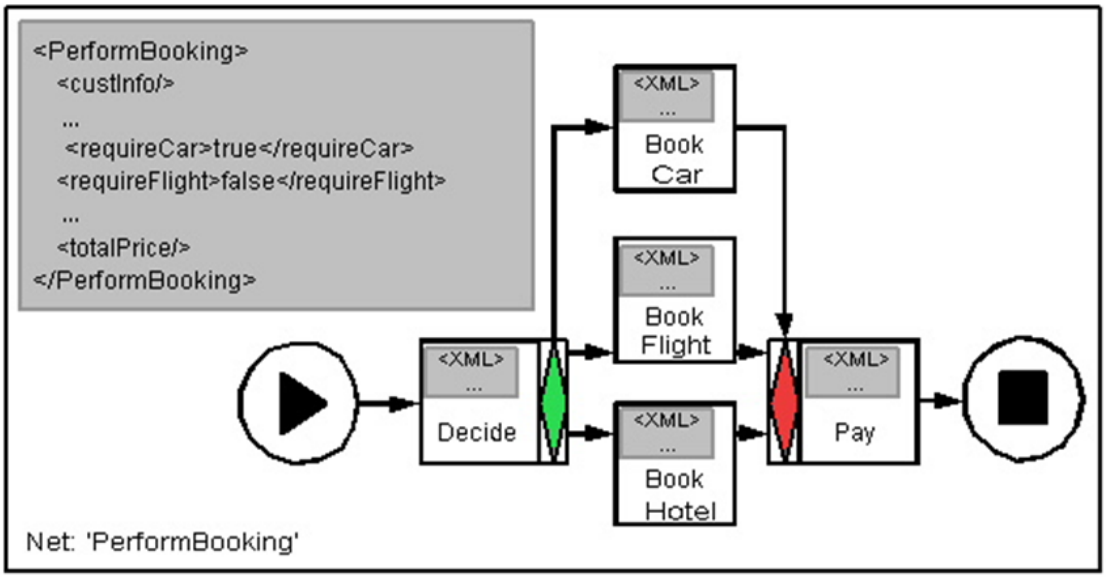


Figure 5.1: A YAWL net "PerformBooking" and its data representation

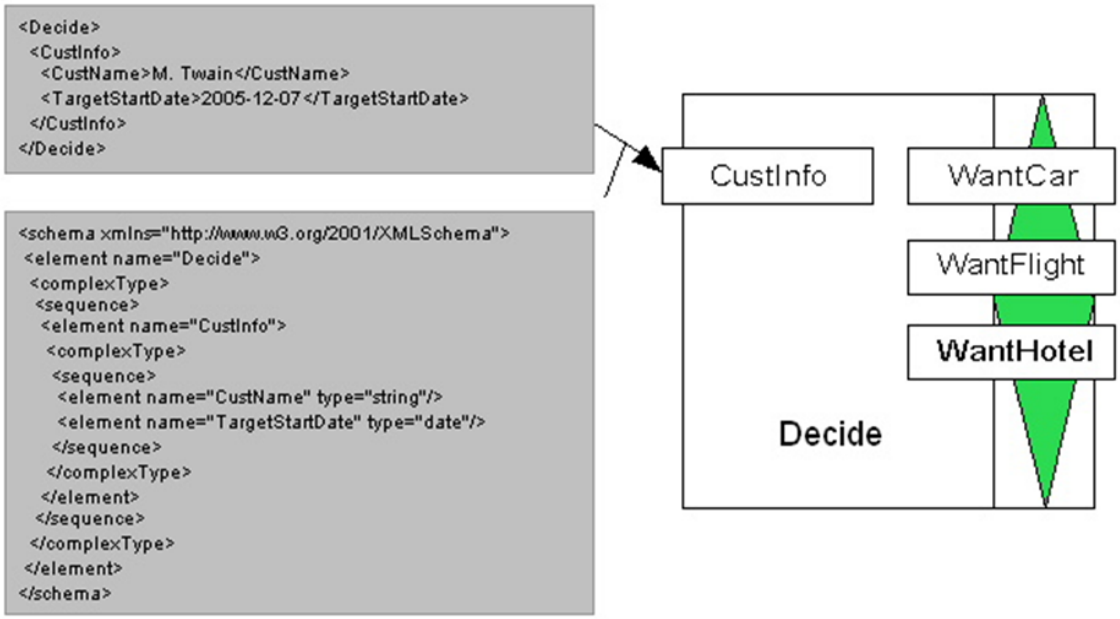


Figure 5.2: XML Schema of a user-defined data type for "CustInfo"

Note In Chapter 4, Sections 4.7.1 to 4.7.2 illustrate how to define net and task variables in the YAWL Editor.

5.3 Data Transfer

5.3.1 Internal and External Data Transfers

YAWL supports data passing between variables, which can be considered internal data transfer, and data interaction between a process and its operating environment (i.e. workflow engine, users and web services),

which can be considered external data transfer.

Internal data transfer is always conducted between nets and their tasks (which themselves may or may not be composite) using XQueries[3]. Note that YAWL does not support *direct* data passing between tasks. This is because each task variable is local to the task that the variable belongs to, i.e., it is not accessible by other tasks. Assume task A and task B in net N. To pass data from task A (e.g. variable V_a) to task B (e.g. variable V_b), an appropriate net variable of N (e.g. V_n) must be available to convey data from V_a to V_b. In the YAWL editor, each task can be assigned an *input parameter* and/or an *output parameter* (depending on the specified 'Scope' type) which define internal data transfer associated with that task. Input Parameters use an XQuery to extract the required information from a net variable and pass such information to the corresponding task variable, while output parameters define data passing in the opposite direction. With reference to the process depicted in Figure 5.1, Figure 5.3 shows an example of passing the customer information from the net level to the task level (task "Decide").

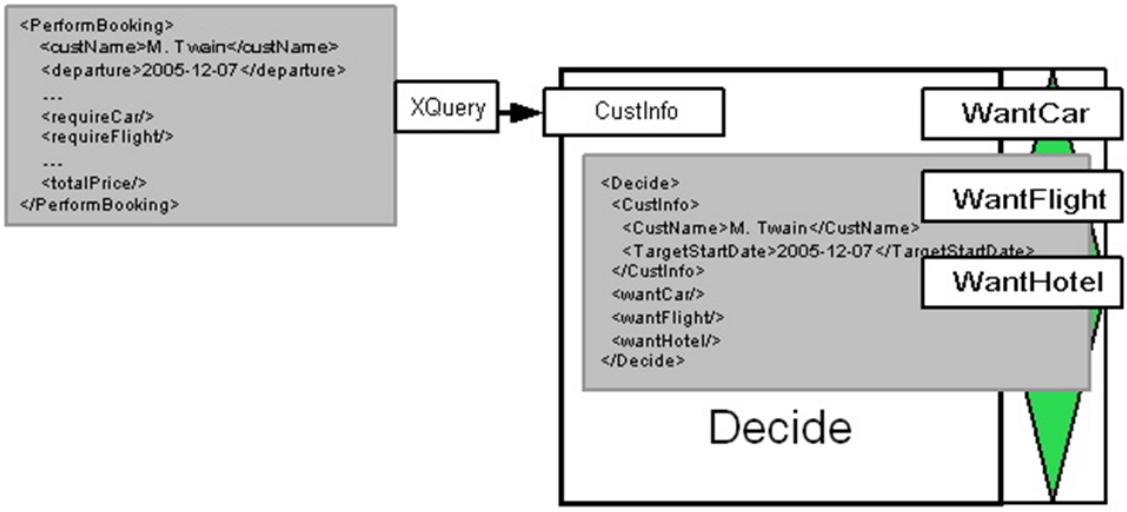


Figure 5.3: An example of data transfer from net "PerformBooking" to task "Decide"

External data transfer does not apply to any local variable or any variable of a composite task. This is because the local variables cannot be observed externally, and the variables of composite tasks serve as intermediate variables for passing data from the higher level to the lower level of a process and vice versa (e.g. between a net and the tasks in its subnets). When data are required from the external environment at run time, either a web form will be generated requesting the data from the user or a custom service will be invoked that can provide the required data.

5.3.2 Valid and Invalid Data Transfers

To ensure correct data transfer, YAWL defines a set of transfer rules for variables of different data scope. Each input variable, except those associated with the top-level net (root net), must have data supplied from the corresponding net variables (which could be a single net variable or an aggregation of any number of net variables), via an input parameter definition. An input variable of the top-level net gets data supplied from the environment (e.g. a user input) once an execution of the net is started; a local variable of the top-level net may be assigned an initial value at design time (data assignment). Each output variable, except those associated with composite tasks, requests data from the environment once the corresponding net or task is executed. An output variable associated with a composite task gets data via the net data in its subnet, using an output parameter definition. Otherwise, output variables are used to supply task data to corresponding net level variables (in internal data transfers).

[3]XQuery 1.0: An XML Query Language. W3C Working Draft, 4 April 2005.

Input and output variables combine scopes. No (internal) data transfers are allowed to local variables of subnets and no (external) data transfers are allowed between local variables and the external environment.

Note In Chapter 4, Section 4.7.3 illustrates how to pass data between variables in the YAWL Editor. Three things are worth noting:

1. The YAWL Editor enforces correct data transfers between variables in the parameter definition for each task. For example, it is not possible to define data passing between local variables, or to set more than one input/output parameter on a single variable associated with a task.

2. In the YAWL editor, by running specification validation, a user can check whether or not required data transfers are missing. For example, if an input variable (V_{in}) of a task (T) does not have any data mapping specified, a validation message will indicate that problem.

3. When specifying XQueries in the parameter definitions of a task, the YAWL editor may automatically generate a compatible XQuery or indicate whether a user-defined XQuery has valid or invalid syntax. The semantics validation of XQueries is performed at run time by the YAWL engine. A semantic error may result in a *Schema Validation Problem* and the execution of the process may fail.

5.4 Data-related Issues

5.4.1 Data-based Conditional Routing

When tasks have XOR or OR splits, which branch to choose is determined by conditional expressions associated with flows. These conditions are boolean expressions that involve data within the process. The data may determine the evaluation results of the conditions and therefore influence the operation of the process.

In YAWL, the branching conditions are specified as XPath[4] boolean expressions in the *split predicates* for tasks with XOR or OR splits. The branches (flows) whose conditions (predicates) evaluate to true will be executed by the YAWL engine (all true branches for an OR split; the first true branch from an XOR split). Also, for each task with an XOR or OR split, there is always a default flow that will be taken if none of the other flow predicates evaluate to true. We consider separately below tasks with XOR splits and the tasks with OR splits.

As an example, Figure 5.4 shows the XPath expression, which is specified for task "Decide", for choosing the branch of "Book Car" in the "PerformBooking" process depicted in Figure 5.1.

For a task with an XOR split, all (conditional) flows are specified in a list, and their predicates are evaluated in the same order as they are presented in the list. Since an XOR split allows only one flow to be chosen, when the engine reaches the first flow predicate that evaluates to true, the corresponding flow will be chosen and the rest of the list will be not be evaluated. However, if the engine reaches the bottom of the list, the bottom-most flow will always be chosen as the default, and it is not necessary to evaluate the predicate associated with that flow. Therefore, the default flow of a task with XOR split is similar to the concept of an "otherwise" clause defined in most programming languages.

For a task with an OR split (e.g. the task "Decide" in Figure 5.4), all flows with their predicates are also presented in a list. However, an OR split requires that all flows whose predicates evaluate to true are taken. Therefore, the engine will evaluate all flow predicates, and only if none of them evaluate to true will the bottom-most flow be taken as the default (despite the false evaluation result of its predicate).

Note In Chapter 4, Section 4.7.4 illustrates how to specify flow predicates for tasks with XOR or OR splits in the YAWL Editor. Two things are worth noting:

1. Only net variables are allowed to be used in specifying flow predicates. This is because the flow

[4]XML Path Language (XPath) 1.0. W3C Recommendation, 16 November 1999.

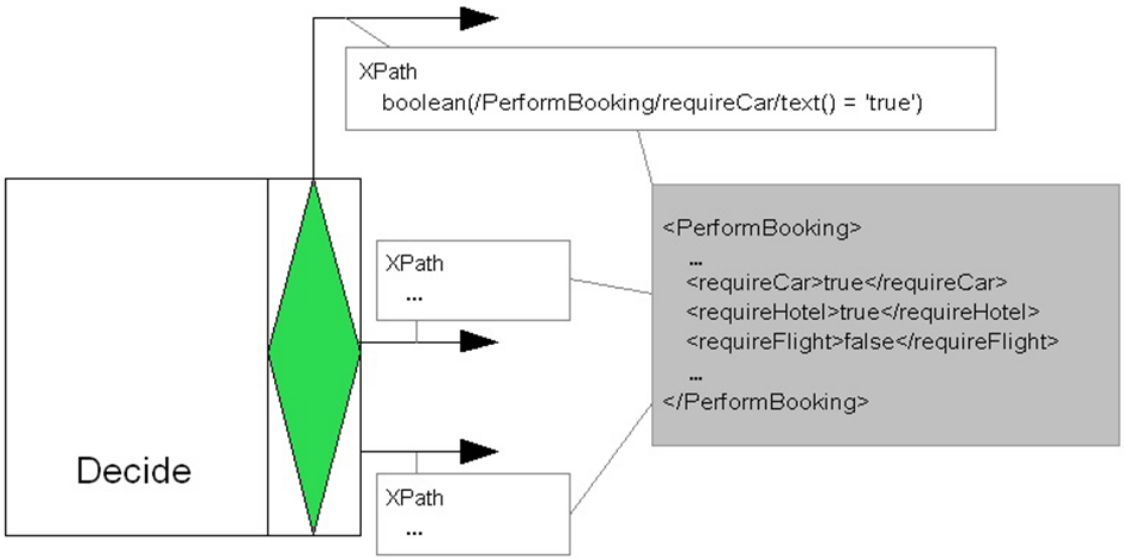


Figure 5.4: XPath expression for choosing the branch of "Book Car" at task "Decide"

evaluation for a task with XOR or OR split is conducted after completing the execution of the task, and therefore the task variables are no longer available.

2. Similarly to the XQuery validation, the syntax validation of XPath expressions can be performed in the YAWL editor. The semantics validation is however a different matter. XQuery is very flexible on what is considered to be a valid boolean value:

   - An empty condition will evaluate to false (however, the editor's 'Split Predicates' dialog doesnt allow empty expressions).
   - A valid XPath expression to any 'node' is treated as true, even if the value of the node is a string or integer, for example (i.e. testing the node's existence rather than its value).
   - A single boolean value is treated as that value (i.e. 'true()' or 'false()').
   - If it is a single string, then if it is zero length it is treated as false, otherwise it is treated as true. For example, 'abc' and 'false' will evaluate to true.
   - If it is a single numeric value, then if the value is NaN or zero it is treated as false; otherwise it is treated as true. For example, -37 and 432 will evaluate to true.

   So care should be taken when entering split predicate expressions.

5.4.2 Multiple Instance Data

There are two categories of data associated with multiple instance tasks. One is the task attribute data which define the *maximum* and the *minimum* number of instances allowed as well as the *threshold* value. The other is the multiple instance data which are specific to individual execution instances of tasks within a single workflow case [26]. Below we describe how to handle multiple instance data in YAWL.

YAWL supports both the designated multiple instance tasks and the isolation of data elements between task instances. However, the handling of multiple instance data is far from trivial. Data at the higher level needs to be split over the instances and after completion of the instances aggregated to data elements at the higher level [5]. A set of four XQueries are used to pass multiple instance data between different levels. These are: the *accessor query* for manipulating the overall multiple instance data before the unique values are split out (to individual execution instances); the *splitter query* for separating the unique values

from the overall multiple instance data and passing a unique value to each instance; the *instance query* for transforming the XML document returned on completion of an instance to a form that is suitable for aggregation; and the *aggregate query* for finally generating an overall result and passing the result to the higher level on completion of the multiple instance task.

In Chapter 4, Section 4.7.6 illustrates how to specify multiple instance data manipulation, e.g., the four types of XQueries, in the YAWL Editor. The XQueries automatically generated by the Editor are sufficient in the vast majority of cases.

5.5 Illustrative Examples

Below are three illustrative examples which cover the data perspective of YAWL. The first example is a revised version of the "Credit Rating Process" taken from the "Oracle BPEL Process Manager: Quick Start Guide" (*10g Release 2*. May 2005). The next two examples are the first two "Make Trip Processes" that can be found in [5].

5.5.1 Example 1: Credit Rating Process

This is a simple process that provides a credit rating service. When running this process, the client (user) is asked to provide his/her social security number. The process takes the number and returns credit rating. There are two situations. If the client's social security number starts with 0, a fault reporting "Bankruptcy" will occur. Otherwise, a credit rating (e.g. 560) will be given. From the above, the data associated with this process are: 1) client's social security number (*ssn*), 2) credit *rating* (560), and 3) *fault* ("Bankruptcy").

YAWL Specification

Figure 5.5 shows the YAWL net specifying the above credit rating process. There are three labelled tasks: ReceiveSSN for requesting a social security number from the client; ReportFault for reporting a "Bankruptcy" fault; and DecideRating for providing the credit rating 560. There is also an unlabelled task, which has an XOR join; this is an example of a *routing* (or empty) task – that is a task without decomposition – and is used here to ensure the net is 'sound'.

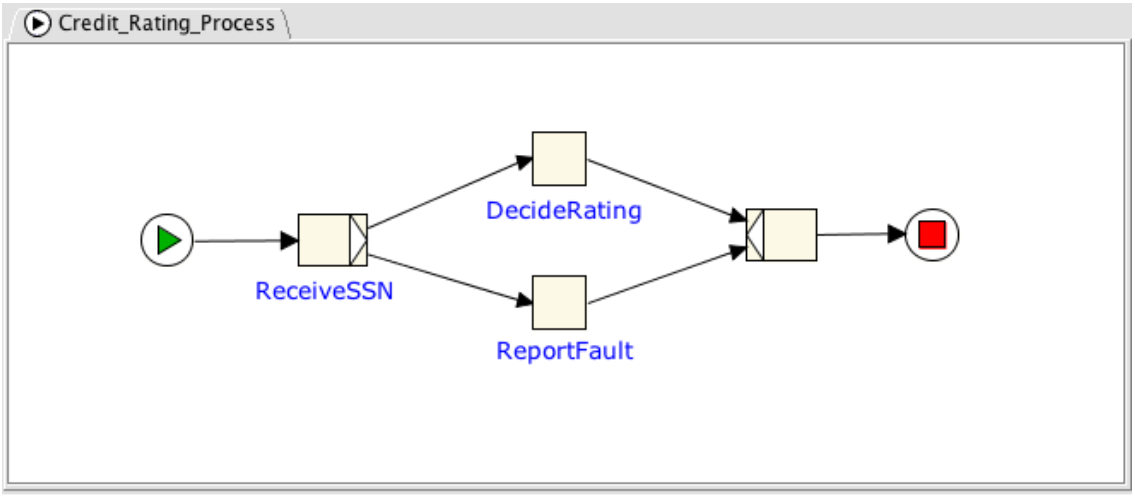


Figure 5.5: The "Credit Rating Process" net

Figure 5.6 shows three net variable definitions for the "Credit Rating Process" net. All are defined as local

variables in order to avoid any data interaction with the external environment at the net level. Also, both *rating* and *fault* are assigned an initial value.

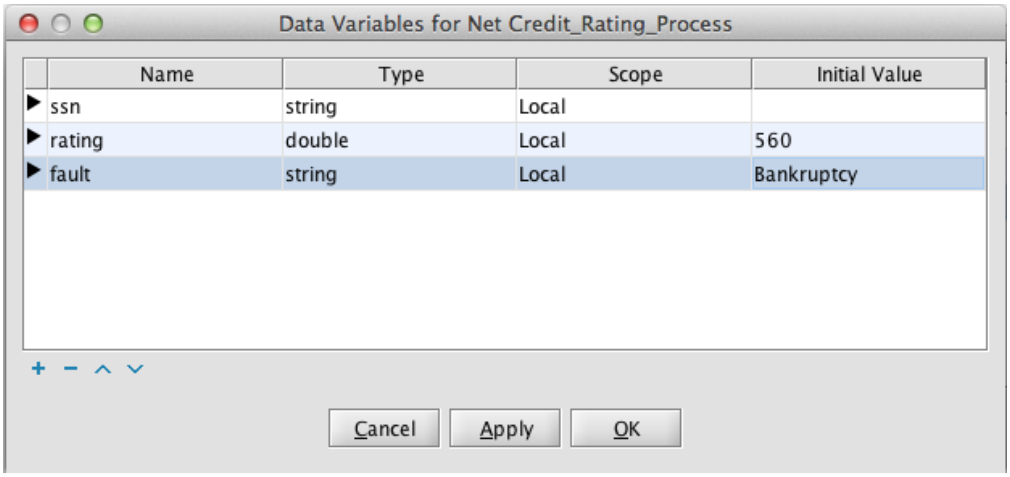


Figure 5.6: Net variable definitions

Figure 5.7 shows the parameter definition for the output-only variable *ssn* of task ReceiveSSN. This variable requests a social security number from the client and, as specified in the output parameter definition, it then passes the data to net variable of the same name via the XQuery "{/ReceiveSSN/ssn/text()}". The Editor automatically generates this query when the net variable is dragged to the decomposition level variable list.

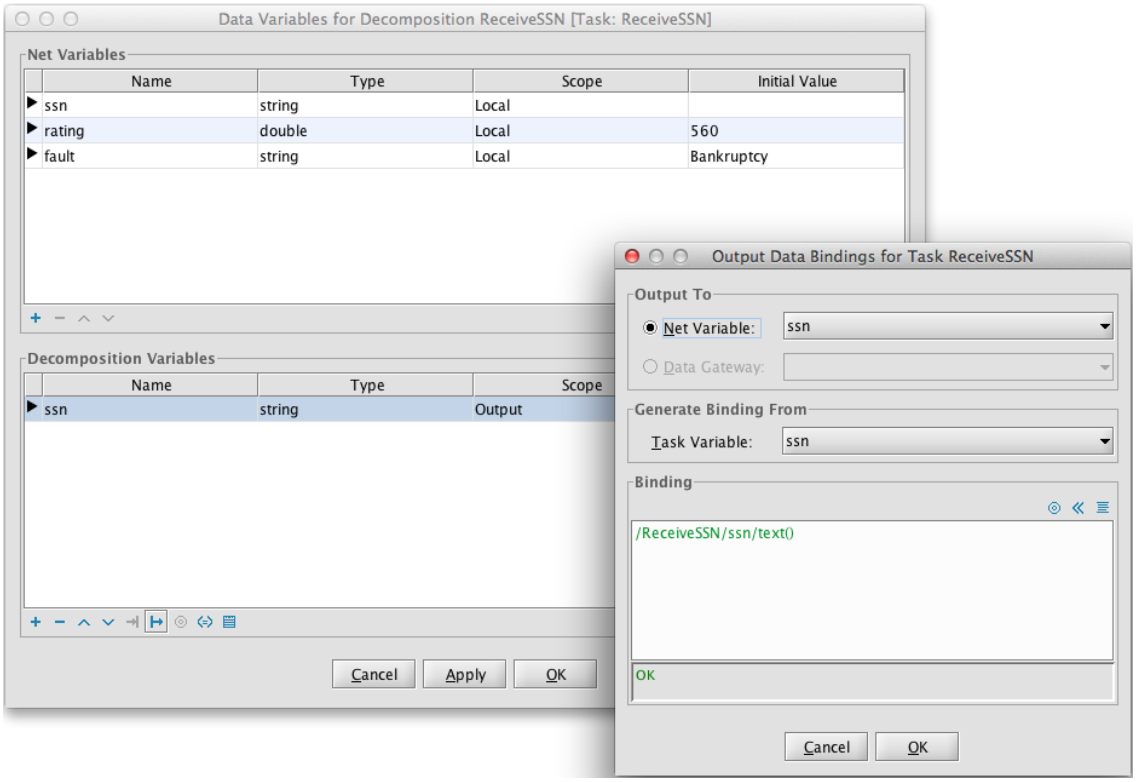


Figure 5.7: Parameter definition for task ReceiveSSN

Figure 5.8 shows the flow definition at task ReceiveSSN. The predicate for the flow leading to task Re-

portFault is "starts-with(/Credit_Rating_Process/ssn/text(),'0')"[5]. It returns true if the string assigned to variable *ssn* starts with zero. Otherwise, the flow leading to task DecideRating will be taken. Since the flow evaluation is performed from the top-most flow to bottom-most flow at run time, the bottom-most flow will be used as the default. The query "/Credit_Rating_Process/ssn/text()" can be generated by clicking the Generate button in the *Update Predicate for Flow* dialog.

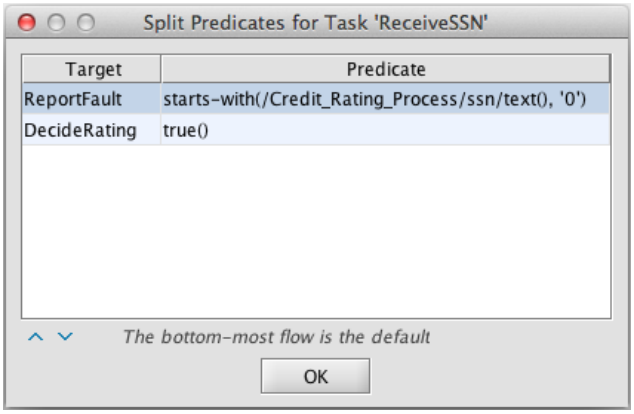


Figure 5.8: Split Predicates for task ReceiveSSN

Figure 5.9 shows the parameter definition for task ReportFault. The variable *fault* is used to carry the fault information ("Bankruptcy"). It is defined as input only because the fault information is only used for client notification upon execution of the task.

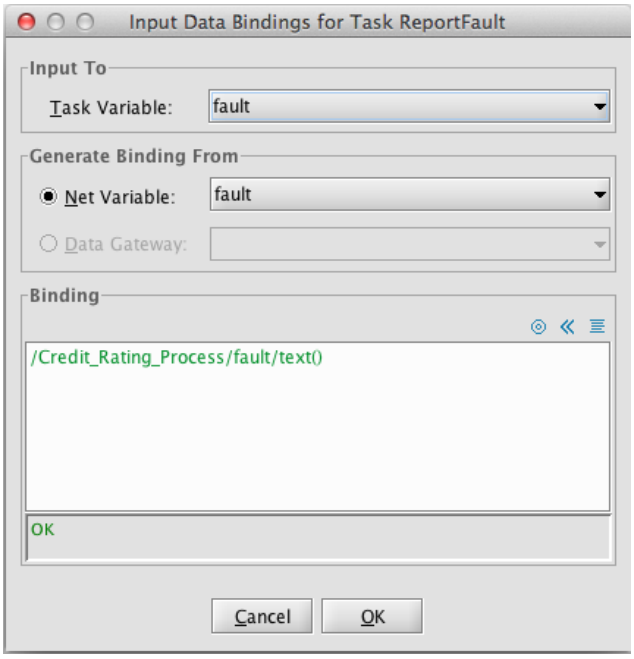


Figure 5.9: Input Parameter definition for task ReportFault

Similarly, task DecideRating has an input only variable *rating* defined to convey the credit rating (560) that is only used for client notification upon execution of the task.

[5] **starts-with**(*string*, *string*) is an inbuilt XPath boolean function. It returns true if the first argument string starts with the second argument string, and otherwise returns false.

Now we have completed the control flow and the data definition of the credit rating process. The specification will pass the syntax check in the YAWL editor and show the message "No problems reported". The last step before deployment is to specify the resource details for each of the manual tasks (i.e. ReceiveSSN, DecideRating, and ReportFault). Chapter 4, Section 4.8 illustrates how to allocate resources to a manual task. Since we are concentrating here on the data perspective, we will choose to not specify any resourcing details for those tasks; in doing so, each of those tasks will be assigned the default settings: the administrator will allocate the tasks to resources manually at run time.

Finally, we save the "Credit Rating Process" to file, which can be deployed (uploaded) in the engine and executed by launching a case.

Examples of Design-time/Run-time Errors

The above YAWL specification is both syntactically and semantically well formed, and can be executed without any problem in the Engine. Now we will introduce some errors into the above specification to see what we will encounter via design-time or run-time validation. These errors are considered to be common when designing YAWL specifications, especially as they become increasingly complex.

Error 1: Missing Data Assignment for Input Variable

Suppose that we forget to specify the mapping from net variable *fault* to task variable *fault* in the input parameter definition for task ReportFault. This results in two syntax errors after validating the specification in the Editor, as shown in Figure 5.10.

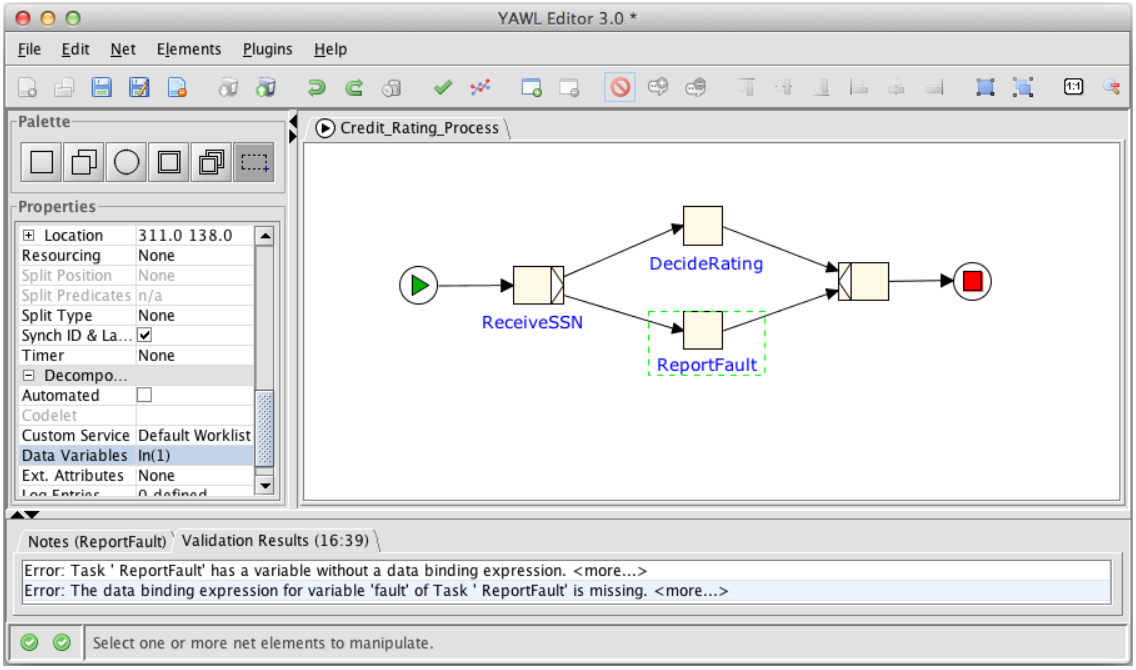


Figure 5.10: An invalid specification with missing data parameter for an input variable

Error 2: XQuery with Invalid Semantics

Suppose that we use XQuery "/ReceiveSSN/ssn" but not "/ReceiveSSN/ssn/text()" in the output parameter definition for task ReceiveSSN. This modified query is a valid XQuery but the mapping is incorrect in this case because it passes the entire XML element "<ssn> some social security number </ssn>" from

task variable *ssn* to net variable *ssn*, rather than only its content, as required. The mapping has a valid syntax (because the result of both expressions is a string) and thus passes syntax validation in the YAWL Editor. But at run time, after the user submits his/her social security number (via task ReceiveSSN), an "bad process definition" error page as shown in Figure 5.11 will appear, indicating a failure has occurred when passing the data extracted by the XQuery to the task variable. As a result, the executed credit rating process is halted.

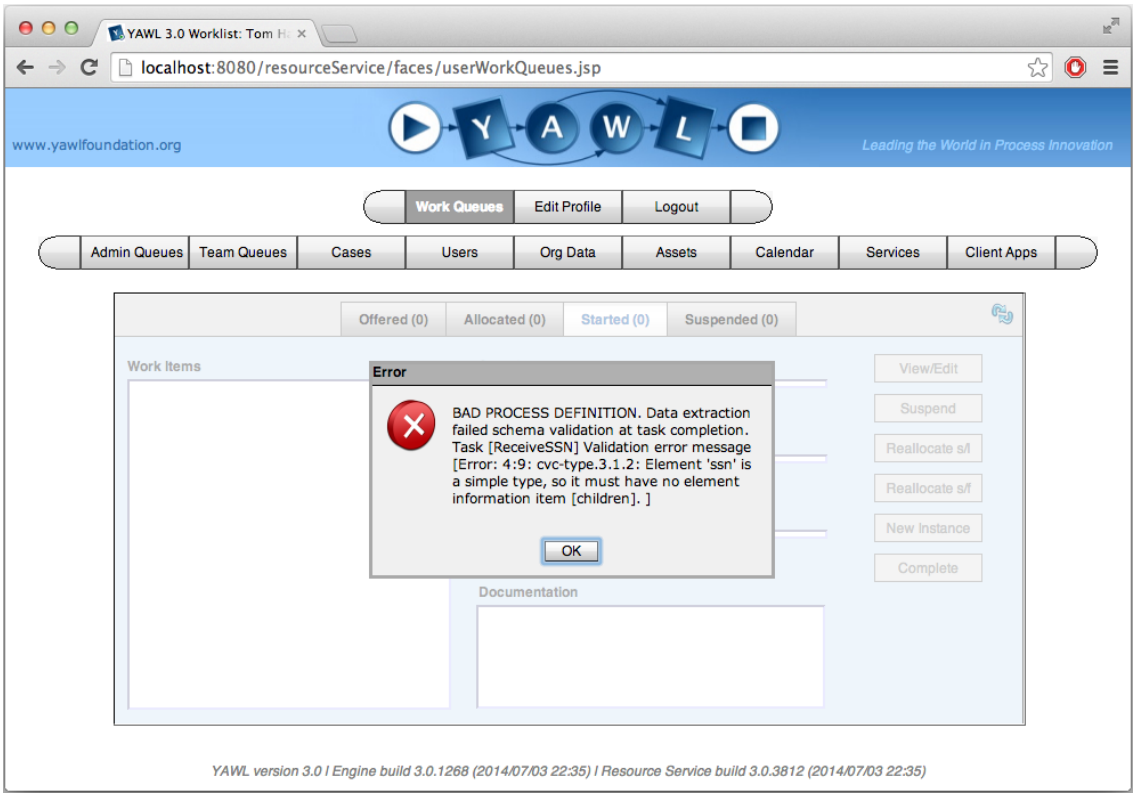


Figure 5.11: An error indicating invalid semantics of an XQuery used in the specification

5.5.2 Example 2: Simple Make Trip Process

This process revisits the example from Chapter 3, a trip booking service. When running this process, the user is first asked to register for the trip. The registration information includes: customer name, trip start date, trip end date, whether to include booking a flight, hotel, and/or car, and customer's payment account number. After the registration, the booking tasks requested by the user are executed in parallel, and the booking details are also provided to each task. After all the booking tasks complete, the user is then asked to make payment for the trip, and the process ends.

YAWL Specification

Figure 5.12 shows the YAWL net specifying the above simple make trip process. There are five labelled tasks: "register" for registering for the trip; "pay" for making payment; and the other three tasks for making the corresponding bookings. The task "register" has an OR split decorator, and the task "pay" has an OR join decorator.

Figure 5.13 shows the data type definitions in this process. There are two new data types: *tripRegistrationType* comprising information of trip start date (*startDate*), trip end date (*endDate*), whether to book a

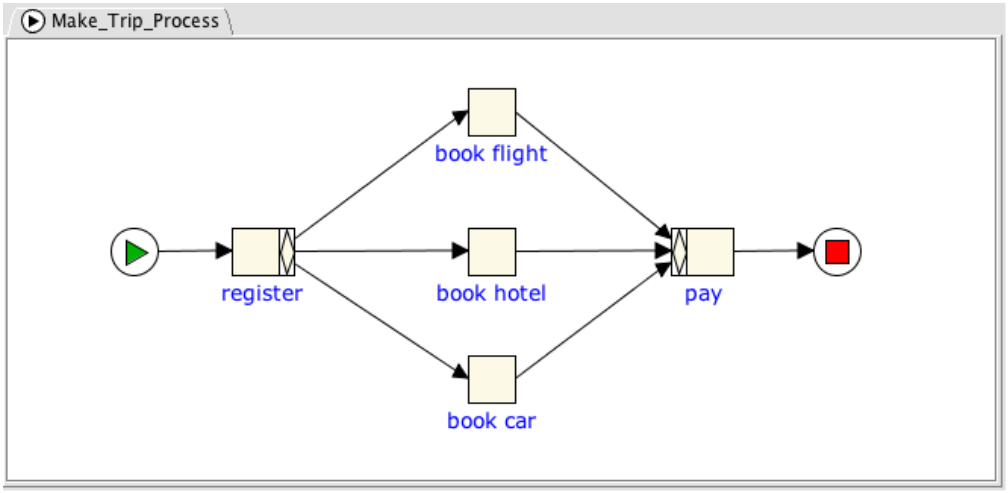


Figure 5.12: The simple "Make Trip Process" net

flight (*want_flight*), hotel (*want_hotel*) and/or car (*want_car*), the customer's payment account number (*payAccNumber*); and *dateType* comprising information of *year*, *month* and *day*, e.g., both *startDate* and *endDate* are of *dateType*.

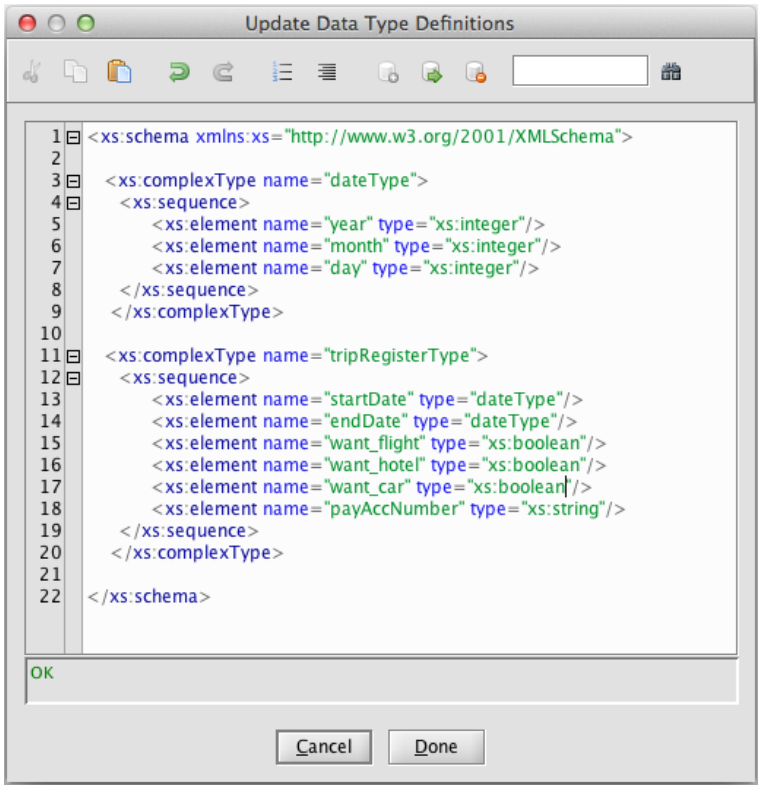


Figure 5.13: Data type definitions

Figure 5.14 shows the net level variable definitions for the process. All are local variables of string type except that "registrInfo" is an output only variable of *tripRegistrationType*. Also, the variable "customer" has an initial value of "Type name of customer" (as a prompt for the customer to enter his/her name).

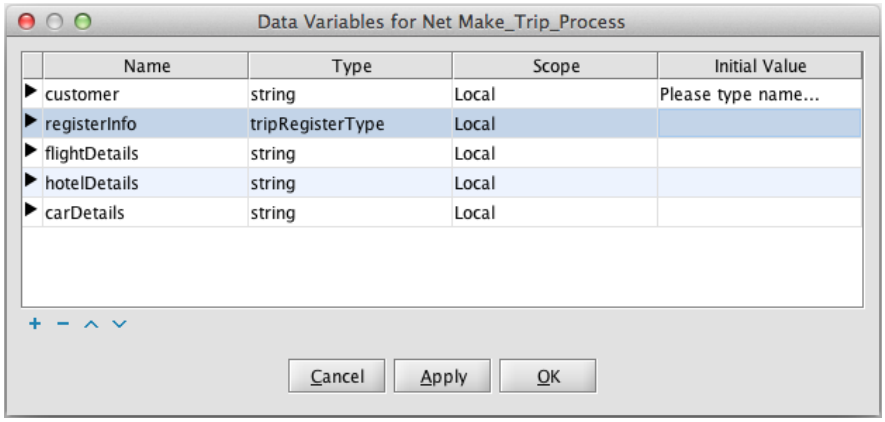


Figure 5.14: Net variable definitions

Figure 5.15 shows the parameter definition for the *registerInfo* variable of task "register". The variable requests registration information from the customer, and then passes the data to the corresponding net variable via the XQuery "{/register/registrInfo/*}". The task's other variable, input & output variable *customer*, gets the customer's name, and then passes it to the net.

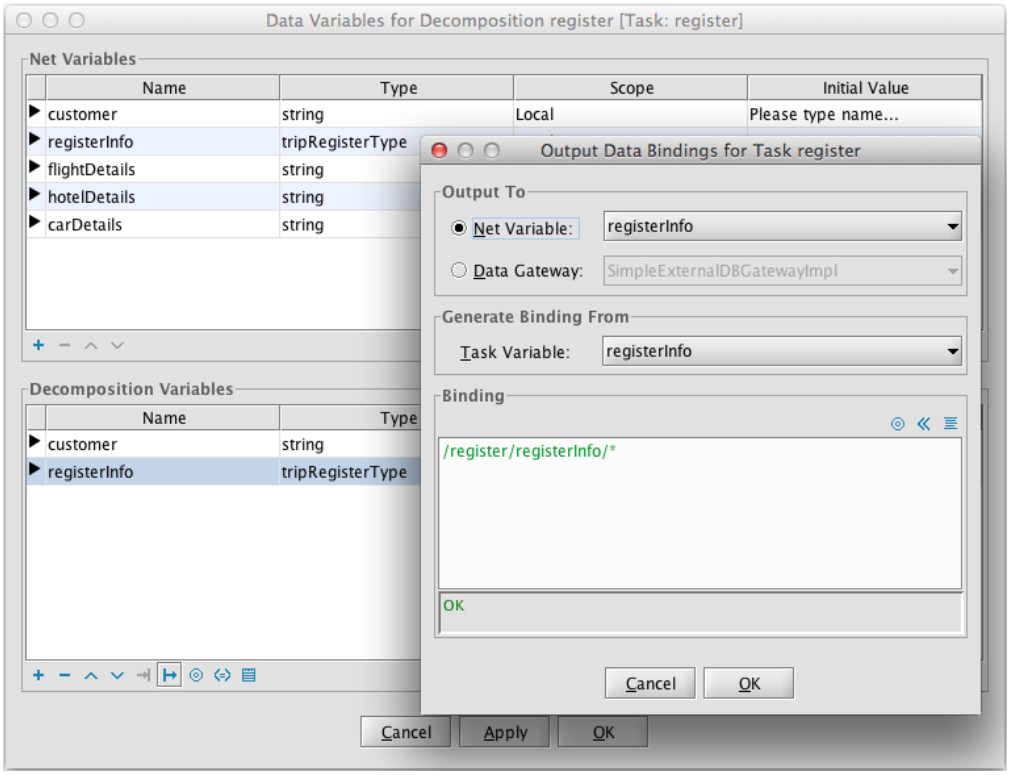


Figure 5.15: Parameter definition for task "register"

Figure 5.16 shows the Split Predicates for task "register". The predicate for the flow leading to task "book flight" is "/Make_Trip_Process/registrInfo/want_flight/text()='true'". The predicates associated with the flows leading to task "book hotel" and "book car" follow the same pattern. Each predicate is used to determine whether the variables included have a boolean value of "true" or "false". Note that these split predicates are defined for an OR split, so that one, two or three of the flows to booking tasks can be taken at runtime.

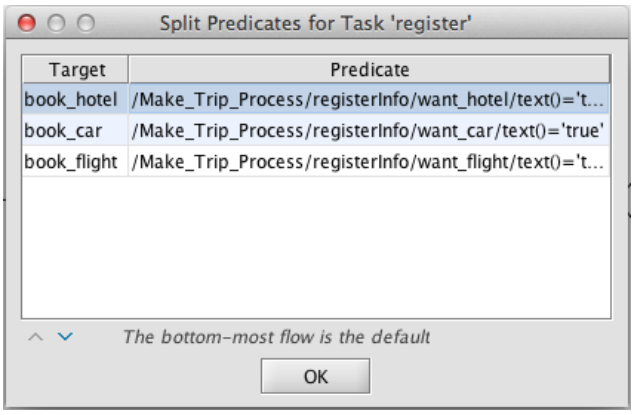


Figure 5.16: Flow definition at task "register"

Figure 5.17 shows the parameter definition of task "book flight" with four task variables. The first three are input only variables that get data from the net via appropriate XQueries. Both *startDate* and *endDate* get data from a part of the net variable *registrInfo*. There is one output only variable, *flightDetails*, which requires information from the customer, such as his/her preferable airlines or a flight number. The parameter definitions of tasks "book hotel" and "book car" are specified in a similar way, except that the output only variable in each case is *hotelDetails* or *carDetails* respectively.

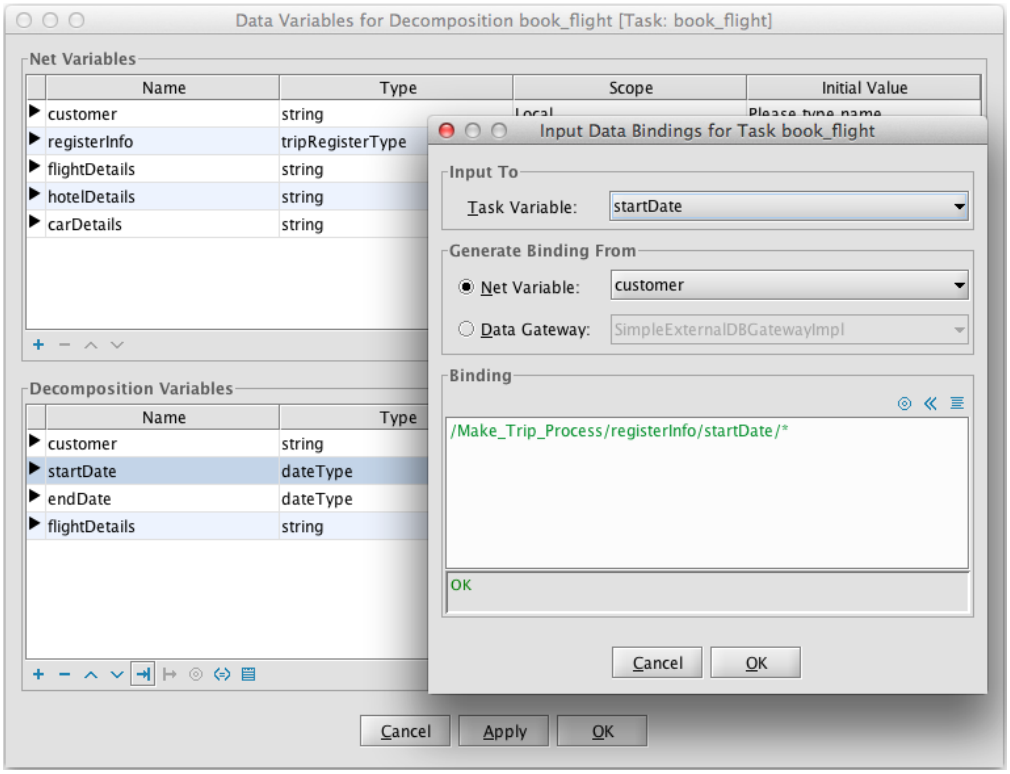


Figure 5.17: Parameter definition for task "book flight"

Figure 5.18 shows the parameter definition of task "pay" with five task variables. These are all input only variables and they get data correspondingly from those net variables with the same names.

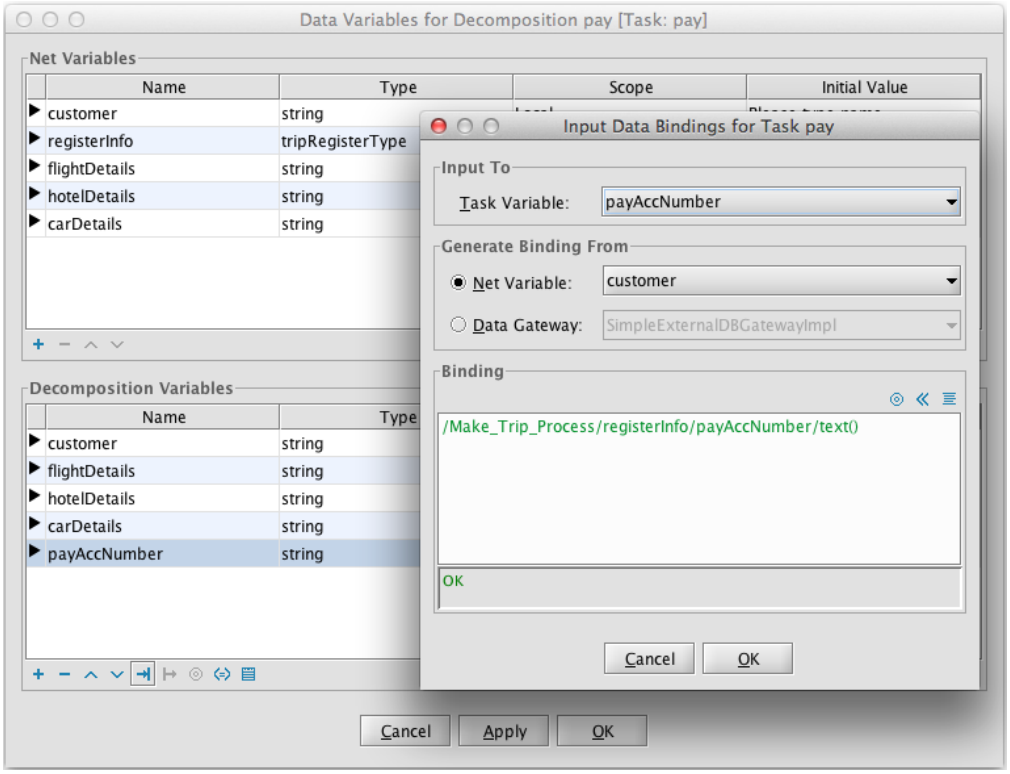


Figure 5.18: Parameter definition for task "pay"

5.5.3 Example 3: Make Trip Process with Multiple Instance Composite Tasks

This process provides a booking service for a trip that has several legs. For each leg, the same simple make trip process from the previous example will be executed as a sub-net of the overall process. As a result, we obtain a more complex Make Trip process by involving a multiple instance composite task for execution of the simple make trip process for each leg. Also, the customer is provided with a subtotal of the payment for each leg, and the subtotals for all the legs in the trip will be calculated into a total payment at the end of the process.

YAWL Specification

Figure 5.19 shows the YAWL nets that comprise the process. There are two nets: the root net "make trip" and a subnet that contains details for the composite task "do itinerary segment", which is basically the same "make trip" net in the previous example.

Figure 5.20 lists the data type definitions for the specification. There are five new (user-defined) data types. The *itineraryType* contains a list of itinerary segments of *itinerarySegType*, which each comprises information of *departure location*, *destination*, *startDate*, *endDate*, *flightDetails*, *hotelDetail*, *carDetails*, and *subtotal*. The *legsType* is a set of legs of *legType*, which provides the information of *departure location* and *destination*. Finally, the *serviceType* specifies whether to book a flight, hotel and/or car (as alternative to the previous example, we group these needs together this time).

Figure 5.21 shows the net-level variable definitions for the "make trip" net. There are three new variables, *itinerary*, *legs*, and *subTotal*, as compared to the previous simple "Make Trip Process" net in Section 5.5.2.

Figure 5.22 shows the parameter definition of task "register". There are three task variables, *customer*, *legs*, and *payAccNumber*, with appropriate input or output parameter definition.

Figure 5.23 shows the parameter definition of task "pay". There are four variables which are all input only

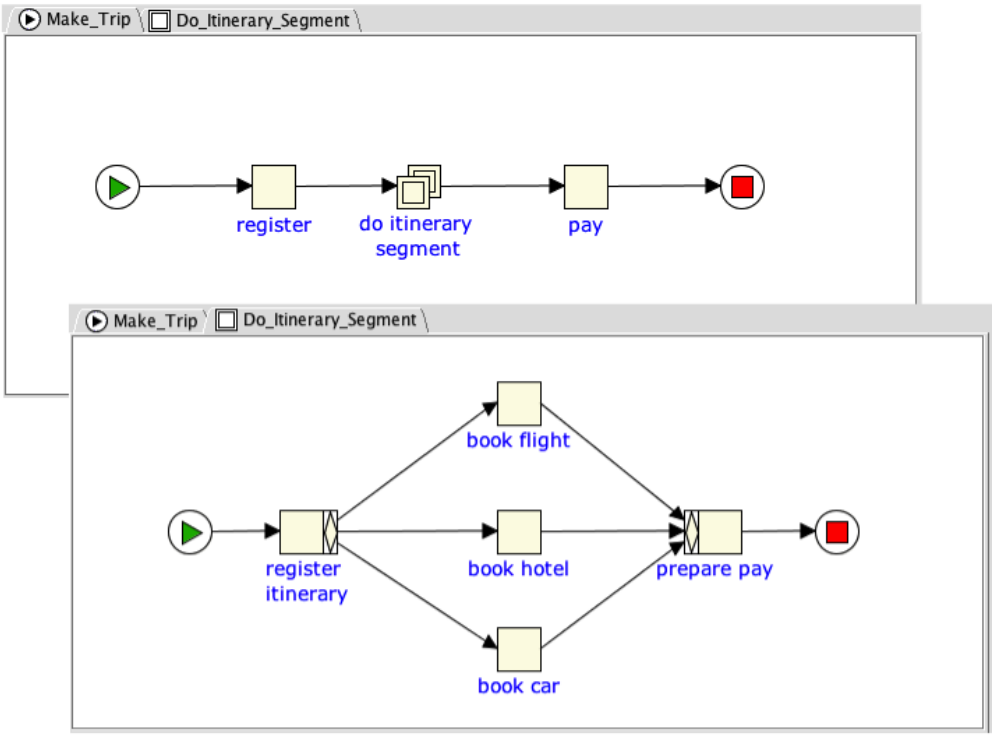


Figure 5.19: The "make trip" net with a multiple instance composite task "do itinerary segment" and the corresponding subnet

variables. Note the input parameter definition of the mapping from variable *subTotal* to variable *total*. The XQuery function *sum*() calculates a *total* sum of *subTotals* from each itinerary segment.

Figure 5.24 shows the parameter definition of the multiple instance composite task "do itinerary segment". There are nine task variables, which are also the net variables of the subnet of this task. All except *startDate*, *endDate*, and *serviceRequired* are input & output variables conveying data between root net and the subnet. The variable *serviceRequired* contains information only used within the subnet, and is therefore defined as a local variable. The *startDate* and *endDate* variables are output variables reporting the user input back to the corresponding net variables. Also, the input parameter definition for variable *legs_item* and the output parameter definition for variable *itinerary* are both determined by the task instance queries (see below).

Figure 5.25 shows the Input and Output binding dialogs for the multiple instance variable specific to individual execution instances of task "do itinerary segment" for each *leg* within one itinerary (i.e. a single process instance). There are two input and two output queries. Firstly, an *accessor query* manipulates the overall data carried by root net variable *legs* before the data is split out to each individual *legs_item* variables. This query determines the input parameter definition for variable *legs_item*. Secondly, a *splitter query* separates the unique values from the overall data carried by variable *legs*, and passes a unique value to variable *legs_item* associated with each instance. The data returned on completion of an instance is an XML document. Thirdly, an *instance query* transforms such an XML document to a form that is suitable for aggregation of data to the higher level, i.e. the root net "make trip". This query determines the output parameter definition for root net variable *itinerary*. Finally, an *aggregate Query* generates an overall result and passes the data to variable *itinerary* on completion of all instances of task "do itinerary segment" within a single itinerary.

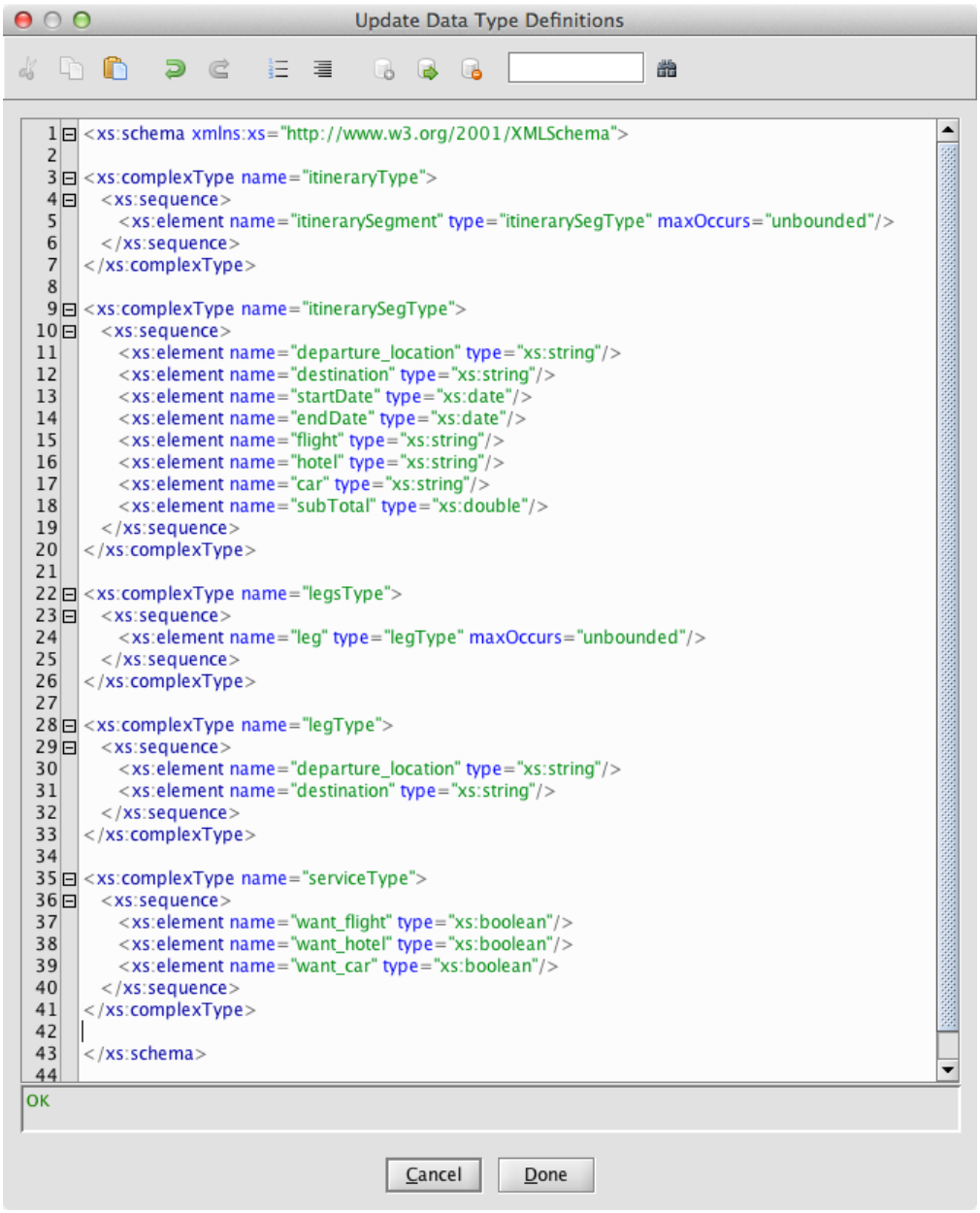


Figure 5.20: Updated data type definition for the "make trip" net

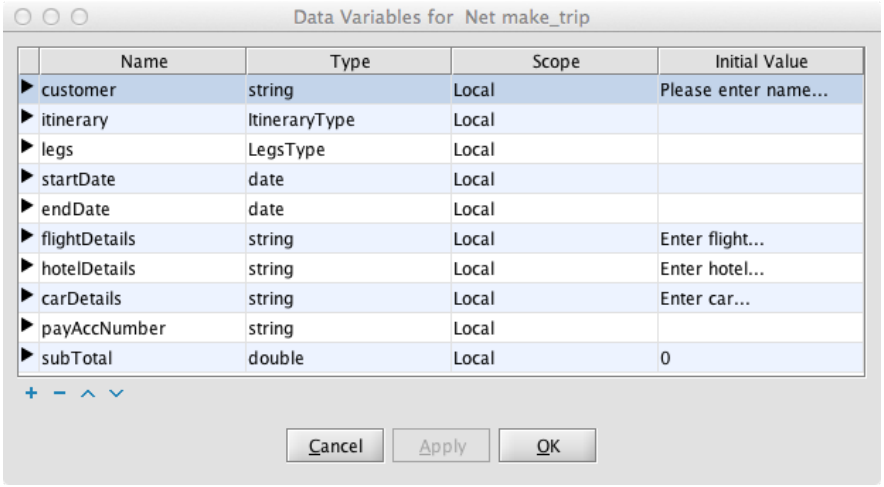


Figure 5.21: Net variable definitions for the "make trip" net

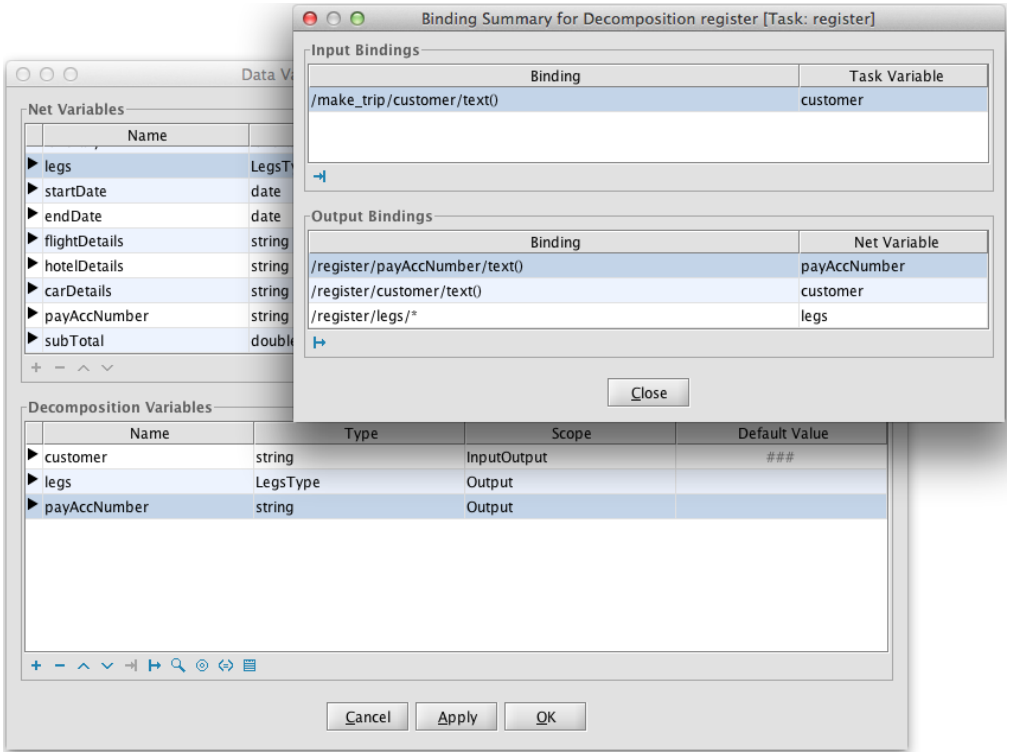


Figure 5.22: Parameter definition for task "register"

Here is the complete instance query (cf. Figure 5.25):

```
<itinerarySegment>
{/do_itinerary_segment/legs_Item/departure_location}
{/do_itinerary_segment/legs_Item/destination}
{/do_itinerary_segment/startDate}
{/do_itinerary_segment/endDate}
{/do_itinerary_segment/flightDetails}
{/do_itinerary_segment/hotelDetails}
```

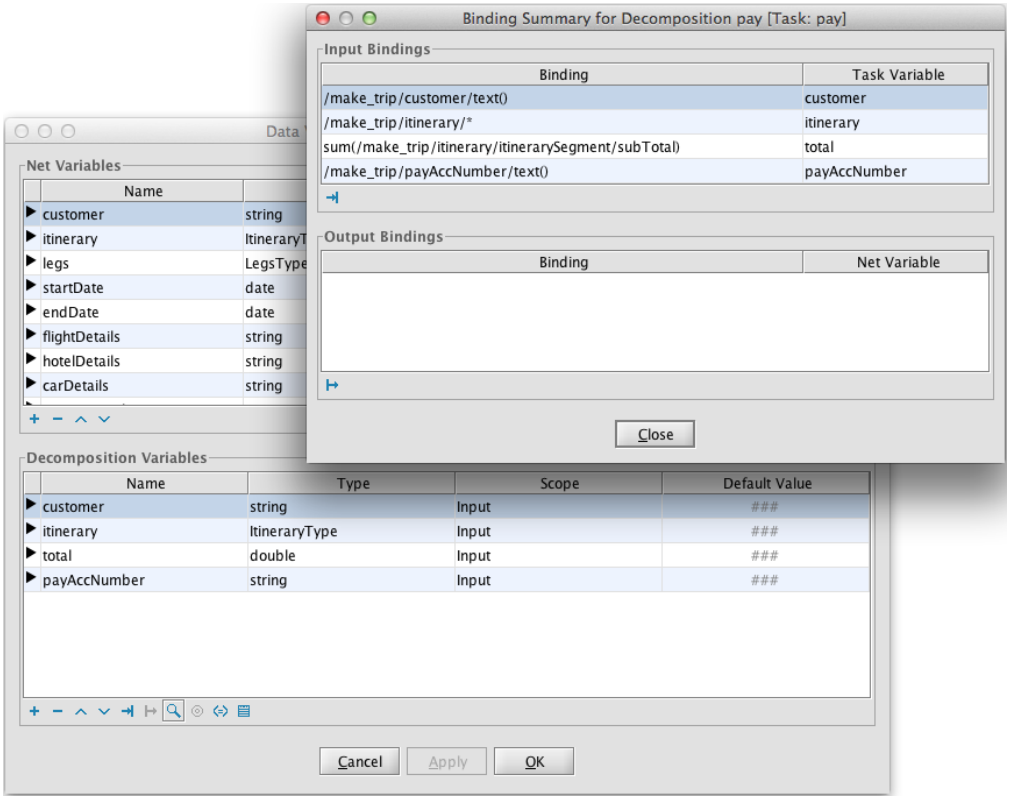


Figure 5.23: Parameter definition for task "pay"

```
{/do_itinerary_segment/carDetails}
{/do_itinerary_segment/subTotal}
</itinerarySegment>
```

This is an example of a more advanced use of multiple instance task variables, where the input variable (*legs*) is used to create individual task instances, based on the number of value elements its data contains, while the net-level output or target variable (*itinerary*) is a different variable whose data value is a composite of (most of) the other output variables of the task.

The subnet of task "do itinerary segment" specifies the simple "make trip" process. Thus, we do not go into every detail, but provide the parameter definitions for tasks "register itinerary", "book flight" and "prepare pay" in Figure 5.26 to Figure 5.28, respectively. The parameters for the other two tasks "book hotel" and "book car" are defined in a similar way to those of task "book flight".

Note Since the variable *subTotal* is of *double* type, the XQuery function *number*() is used to extract data from the variable.

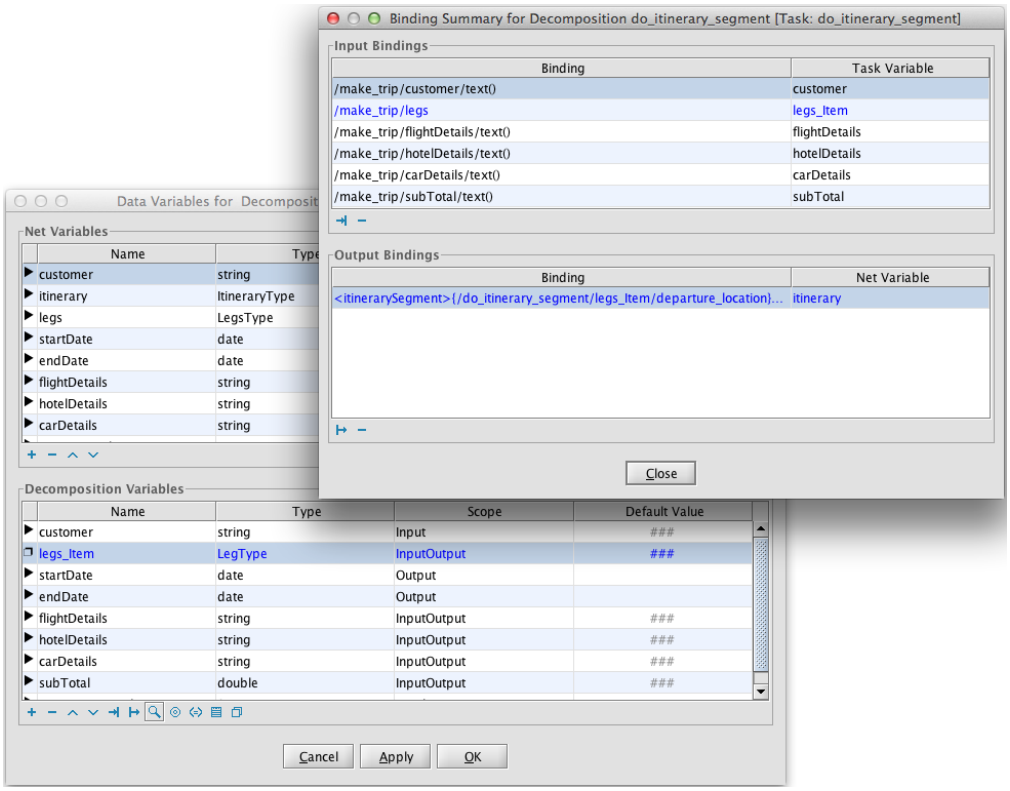


Figure 5.24: Parameter definition of task "do itinerary segment"

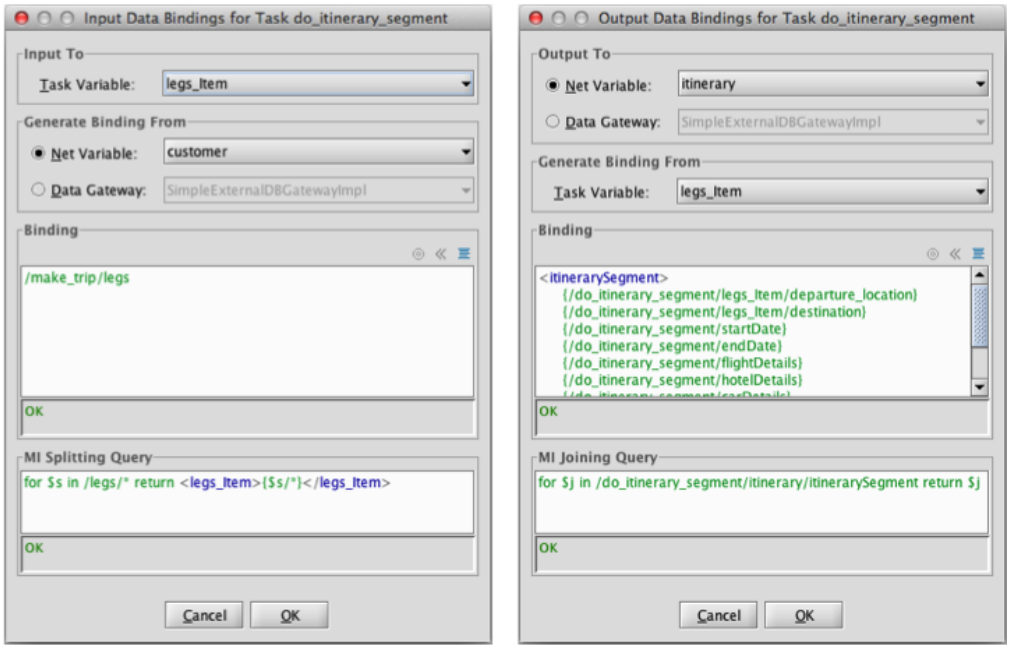


Figure 5.25: Input and Output dialogs for the multiple instance data of task "do itinerary segment"

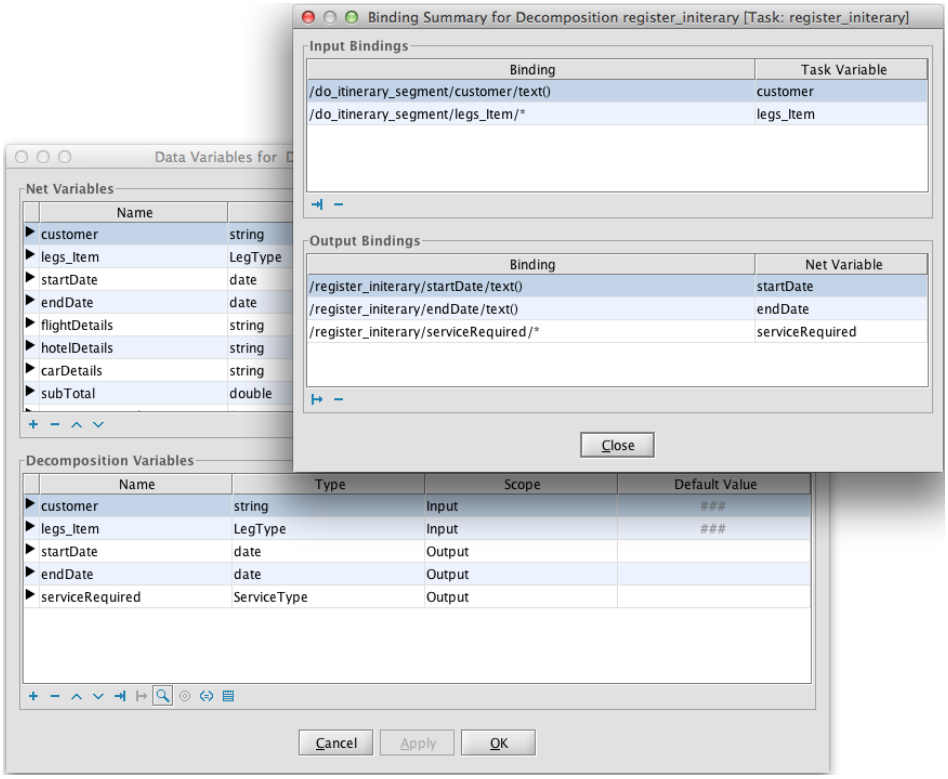


Figure 5.26: Parameter definition for task "register itinerary"

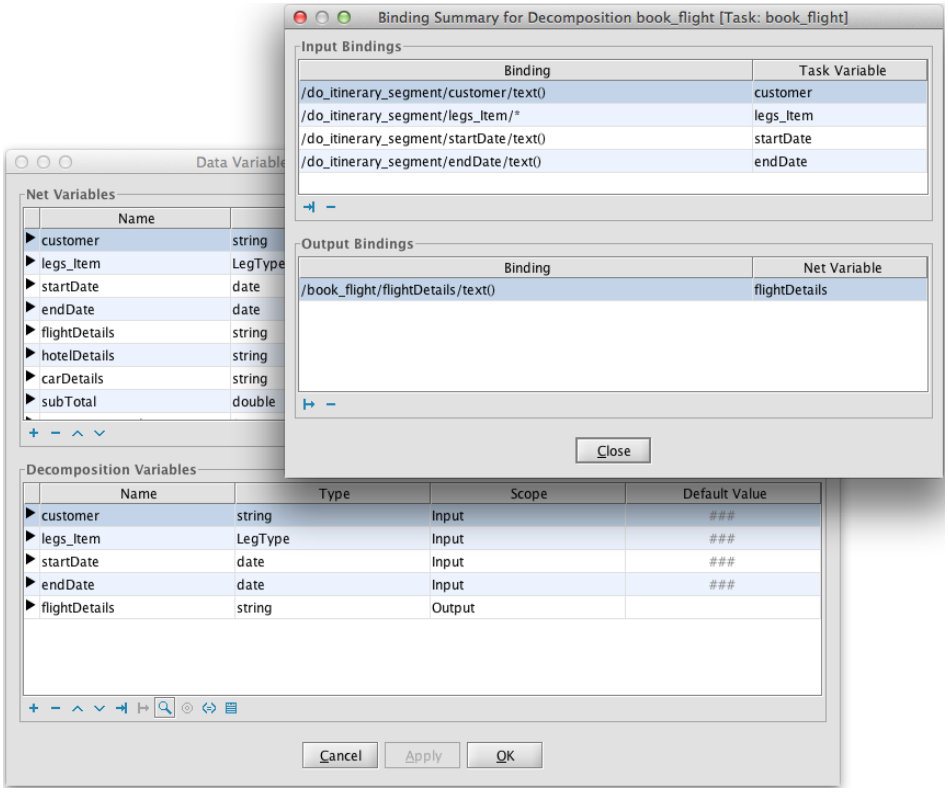


Figure 5.27: Parameter definition for task "book flight"

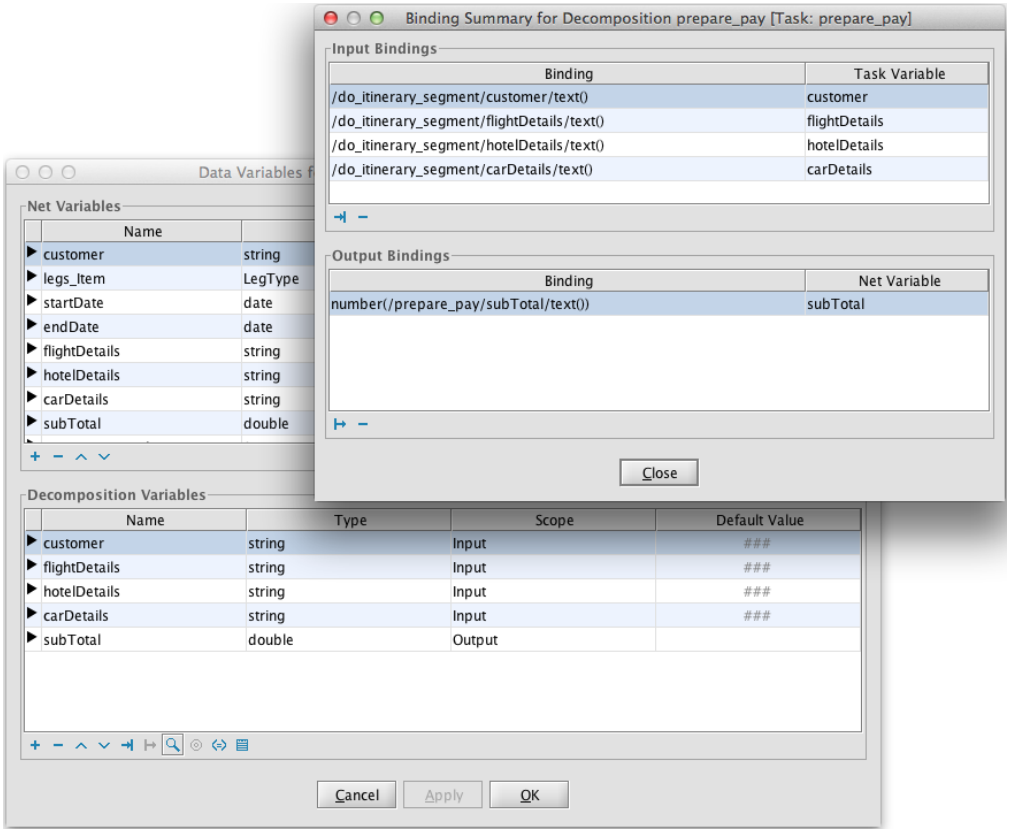


Figure 5.28: Parameter definition for task "prepare pay"

Chapter 6

The Runtime Environment

This chapter provides an overview of the runtime environment from a user perspective.

When a YAWL workflow specification has been completed in the Editor it can be saved to a file, the contents of which are in an XML format that can be interpreted by the YAWL Engine. The specification file contains descriptions of each of the three perspectives of a process: control-flow (task sequences, splits, joins etc.); data (variables, parameters, predicates etc.); and resourcing (participants, roles, allocators, filters etc.). However, the Engine is responsible only for the control-flow and data perspectives – it essentially ignores the descriptors for resourcing contained in a specification file, instead passing responsibility for the resource perspective to a dedicated custom service. In the core YAWL environment, a custom service, called the *Resource Service*, is supplied to provide comprehensive support for the resource perspective.

The resource perspective of Business Process Management (BPM) is concerned with the way work is distributed to resources. It is here that the link between the process model and the organisational model is formalised. This is a very important perspective in BPM and one that has not had as much attention as the control-flow perspective. In fact the state-of-the-art in BPM environments typically lacks sufficient support for the resource perspective (consult the Workflow Patterns Home Page at www.workflowpatterns.com for more details).

The realisation of the resource perspective in YAWL 2.0 is based on the analysis work reported in a technical report on newYAWL [27] and Nick Russell's PhD thesis [28][1].

The Resource Service is a large custom service that contains a number of components, primarily a *Resource Manager* that is responsible for the allocation of tasks to human users (referred to as 'participants'); a *Worklist* that comprise a series of web forms that provide a user interface to processes and process management; *Administration Tools* that comprise a series of web forms to administer the Engine, processes and organisational data; a *Dynamic Forms Generator*, which creates web forms on-the-fly for the presentation of work item data to participants so they can be performed and completed; and a *Codelet Coordinator* that manages the execution of codelets for automated tasks.

ASIDE: The Resource Service provides functionality to support the identified resource patterns [25] and associated activities. However, as a custom service, it is completely removed from the operation of the Engine. This means that developers are free to develop other custom services that communicate directly with the Engine (and thus bypassing the Resource Service), although support for the resource perspective would also be lost. Alternately, the Resource Service provides a number of interfaces that expose the full functionality of the service, which developers may exploit to 'override' service components. For example, other types of worklist services may be developed that leverage the resource management capabilities of the Resource Service but present work to users in different or novel ways; external organisational data sources may be used in place of the default internal data model supported by the Resource Service; Custom Forms may be defined to display work item data, rather than using the default, dynamically generated forms; and so on. Also, the Resource Service is *extendible* in many ways, for example developers may add new

[1] For a discussion of how the workflow resource patterns have been realised in newYAWL, see appendix A.3 starting on page 373 of [28].

allocation strategies, filters, constraints, codelets etc. at any time, which immediately become available for use in the service and the Editor. For more information, please consult the *YAWL Technical Manual*.

Since the Resource Service provides the default set of tools for user interaction with the YAWL system, this chapter describes the runtime environment by describing the use of the service. Configuration and Administration functionality is described first, followed by the various user interactions.

6.1 Engine Configuration Settings

The Engine has several configuration parameters that may be set in its *web.xml* file, which is located in the folder *\webapps\yawl\WEB-INF*. Below is the list of available parameters that are specific to the Engine, what they are used for and what values may be assigned to them.

- **EnablePersistence**: When set to true (default), allows the engine to persist (backup) current process data to a database, so that in the event of the Engine being restarted, the 'work-in-progress' can be restored. There is rarely any need to disable persistence, and in any case this setting should always match the 'EnablePersistence' settings of the Resource Service and the Worklet Service.

- **DefaultWorklist**: Each and every task that is enabled by the Engine during process execution is passed to a YAWL custom service for processing. Each task that is not explicitly mapped to a custom service at design time is dispatched to the *Default Worklist Handler*. That is, the default worklist handler is the fall-through service that handles all tasks that aren't explicitly associated with a chosen service. Therefore, the Engine requires a service to be nominated for this purpose. By default, the Resource Service, with its built-in worklist handler, serves this role, but any other service may be nominated as the default worklist by configuring this parameter. The value of the parameter is the URL of the worklist, and its password, separated by a hash character (#). The password must match the one the service will use to log onto the Engine.

- **EnableLogging**: The Engine records extensive data about running processes to its process log for later analysis and to keep an historical archive. Process logging is enabled by default when persistence is enabled. If persistence is enabled, setting this parameter to 'false' will disable process logging. If persistence is disabled, the value of this parameter has no effect.

- **AllowGenericAdminID**: This parameter allows for the generic 'admin' user account to be disabled. If the parameter's value is 'true', services and applications may connect to the YAWL engine using the generic administrator account 'admin', password 'YAWL' (or some other password if the password has been changed – see Section 6.4.4). If the parameter's value is 'false', each connecting service and application must connect using a unique account previously registered with the engine.

- **EnableHibernateStatisticsGathering**: When set to 'true', the hibernate database layer will collect statistics of all operations for later perusal. When set to 'false', the default, hibernate statistics collection is disabled. There is a slight performance overhead associated with statistics gathering.

- **InitialisationAnnouncementTimeout**: When the Engine completes its initialisation and is running, it sends an event announcement to all registered custom services to notify them that it is ready to execute processes and receive requests. This parameter allows you to set the maximum number of seconds the engine will spend trying to deliver the event to each registered service before giving up. The specified value should cover the period between the moment the Engine has fully initialised and the moment the server hosting a custom service is ready to accept HTTP connections. The value can be any positive integer, and defaults to 5 seconds if the value is missing or invalid.

- **InterfaceXListener**: The fully qualified URI of a custom service that wishes to receive event notifications on Interface X (the exception handling interface). Multiple URIs can be specified, separated by semi-colons ';'. Note that services can also register themselves as Interface X Listeners programmatically. See the *Technical Manual* for more details.

- **ObserverGateway**: The fully qualified class name of an ObserverGateway implementation. Multiple implementations may be specified, separated with semi-colons ';'. See the *Technical Manual* for more information about Observer Gateways.

6.2 Resource Service Configuration

The Resource Service has a number of configuration parameters that may be set in the service's *web.xml* file, which is located in the folder *\webapps\resourceService\WEB-INF*. Below is the list of available parameters that are specific to the Resource Service, what they are used for and what values may be assigned to them.

- **EnablePersistence**: When set to true (default), allows the service to persist (backup) current work queue data to a database, so that in the event of the Engine being restarted, the 'work-in-progress' can be restored. There is rarely any need to disable persistence, and in any case this setting should match the 'EnablePersistence' settings of the Engine and the Worklet Service.

- **OrgDataSource**: While the Resource Service offers an internal organisational database by default, it also supports organisational data being used that is stored in existing, external data sources. This is especially beneficial for sites that want to use org data already stored in HR Systems and so on. External data sources may be 'mapped' to YAWL by implementing a java class to take care of the necessary mappings (see the technical manual for details). This parameter allows for the specification of that mapping class name. The default setting is 'HibernateImpl', the internal Resource Service mapping class.

- **ExternalUserAuthentication**: When an external organisational data source is configured for use by the Resource Service, a choice can be made regarding the logon authentication of users, which specifies whether the Resource Service will take responsibility or if authentication will be handled by the mapping class (configured above). When this parameter is set to false (the default), user passwords are encrypted and stored within each user record, and authentication is handled within the Resource Service. When it is set to true, user authentication is deferred to the currently implemented external organisational data source, and passwords are sent to the data source as plain text (rather than the default encryption). It is up to the external data source to provide valid user authentication in this case. A setting of true is only relevant if an external data source is active; if the default YAWL org database is in use, this setting is ignored.

- **AllowExternalOrgDataMods**: By default, when an external organisational data source is configured for use by the Resource Service, that data is not allowed to be modified via the Resource Service's administration web forms (i.e. data from an external data source is considered read-only by default). When this parameter is set to true, modification of data from an external data source via the administration web forms is allowed. A setting of true is only relevant if an external data source is active; if the default YAWL org database is in use, this setting is ignored.

- **OrgDataRefreshRate**: This parameter provides for the setting of a regular time interval to refresh the organisational data caches in the Resource Service. This is particularly important when the org data is sourced externally, and that external source is 'live' (regularly updated through other systems). If an external data source is not being used, then the parameter's value should be left at -1 (the default, interpreted as 'never refresh'), since the internal data sources are only ever updated through the Resource Service. The parameter value specifies the number of minutes to wait between refreshes.

- **BlockOnUnavailableSecondaryResources**: A work item may have zero or more secondary resources (i.e. non-human resources, and human resources that aren't primarily responsible for the work item's performance) allocated to it. For each work item that has secondary resources allocated to it, setting this parameter to 'true' will prevent the work item from being started if any of its secondary resources are unavailable. When set to 'false' (the default), the missing resource(s) will be noted in the log only, but the work item will be allowed to start.

- **EnableLogging**: The Resource Service also records extensive data about running processes to its process log for later analysis and to keep an historical archive. Process logging is enabled by default when persistence is enabled. If persistence is enabled, setting this parameter to 'false' will disable process logging. If persistence is disabled, the value of this parameter has no effect.

- **LogOffers**: By default, all resourcing events (offer, allocation, start, reallocate, etc.) are written to a process log. If there are large numbers of participants in the organisational model, or work items are typically offered to a large number of potential participants, the logging of all offers may incur some processing overheads for little return (e.g. there may be instances where a work item is offered to several hundred participants – the members of that set can always be derived from design time resourcing parameter settings in any case). By setting this parameter to 'false', no offer events will be logged; logging of all other events will still occur.

- **DropTaskPilingOnLogoff**: A participant who has been granted the authority to 'pile' a certain task, may then explicitly and manually choose to do so, which means they will receive all instances of that task, across all current and future instances of the process that contains the task. When this parameter is set to true, piling of tasks for a participant will cease when the affected participant logs out. When this parameter is set to false (the default), piling of tasks for a participant will continue, whether the participant is logged on or not, until it is explicitly ceased by the affected user or an administrator. This setting is ignored (i.e. treated as 'true') if persistence is not enabled. This parameter applies globally to *all* piled task participants.

- **GenerateRandomOrgData**: This parameter allows you to quickly fill the organisational data base with randomly generated data (participants, roles, positions, capabilities and org groups), which is especially useful for testing purposes, or to examine the capabilities of YAWL without first having to manually populate the org database with real (or dummy) data. A parameter value of between 1 and 100 will generate that number of randomised participants (with associated membership of roles etc.); a value greater than 100 is treated as 100. A value of -1 (the default) turns off random org data generation. If you do make use of this feature, don't forget to reset the value to -1 after the generation is done.

- **ExternalPluginsDir**: The Resource Service supports the ability to extend its functionality through a number of 'pluggable' interfaces. These interfaces allow developers to provide their own pluggable classes (i.e. codelets, allocators, filters, constraints and so on). By default, these classes are inserted into the internal YAWL class packages, which becomes problematic whenever YAWL has a version upgrade. The 'ExternalPluginsDir' parameter sets an external base directory for third-party plug-in classes. The base directory must refer to a location on the local disk, for example: "C:\yawlplugins". Several base directories can be provided, separated by semi-colons, for example: "C:\yawlplugins;C:\some\other\place". The service will expect to find classes to be located in sub-directories of the specified external directories matching their own package structure, and ending in the 'type' of the plugin (i.e. 'codelets', 'allocators', 'constraints' or 'filters'). For example, if the 'ExternalPluginsDir' is set "C:\yawlplugins" and there is a codelet file called 'MyCodelet.class' and it has a package 'com.example.yawl.codelets' (since its a codelet, it must end in 'codelets') then its full file path is expected to be "C:\yawlplugins\com\example\yawl\codelets\MyCodelet.class". If there are no external plug-ins, then this parameter can be commented out or the param-value can be left blank.

- **EnableVisualizer**: When set to true, an extra button will appear on user work queues to show work items via the Visualizer applet (assumes the visualizer is available). The default setting is false.

- **VisualizerViewSize**: When the Visualizer applet is enabled, this parameter configures the size of the Visualizer applet's view window. The value must be two comma-separated positive integers (width,height). If no value is given, or the value is invalid, a default of 800,600 will be used. When the visualizer is disabled, this parameter is ignored.

- **InterfaceX_BackEnd**: This parameter is commented out by default. When the commenting is removed, the extensions to the work queues required for the Worklet Service are enabled. When enabled, the value of this parameter must equal the valid URI of the Worklet Service

- **InterfaceS_BackEnd**: This parameter is commented out by default. When the commenting is removed, scheduling event announcements to a scheduling service listening on Interface S are enabled. When enabled, the value of this parameter must equal the valid URI of a running Scheduling Service.
- **DocStore_BackEnd**: The URI of the DocumentStore service, which supports the passing of binary files as task data values (see Section 10.1 for more information). This parameter should be changed only when the document store is located remotely.

6.3 Logging On

To log on to YAWL:

1. Start the YAWL engine by choosing the option "Start Engine" from the YAWL program menu (or by starting Tomcat directly);
2. Navigate to the Resource Service's web UI either by choosing "YAWL Control Centre" or by pointing your browser at `http://localhost:8080/resourceService`.
3. Login with an existing userid and password. First time logons (i.e. where there are no participants defined in the organisational database) should use the generic userid "admin" and password "YAWL".

Note that a participant with administration privileges will have available the full menu of actions (Figure 6.4 shows an example); ordinary participants and participants with some extra privileges will see a subset of those actions when they log on. The "admin" userid is a 'special' logon, which can be used for administrative tasks only – but because it is not a formal participant (i.e. it is not a logon associated with a unique person), it has no access to an individual work queue, so the menu options for 'Work Queues', 'Team Queues' and 'Edit Profile' are not available for the "admin" logon.

6.4 Administration

A workflow administrator can load new workflow specifications, can start cases for them, can manage all active work items, can register or remove custom services and client applications, and add, manage and remove participants, roles, positions, non-human resources and organisational groupings. In this section we will explore how these functions are achieved.

6.4.1 Case Management

To upload a new workflow specification, first click *Case Mgt* in the top menu, which displays in the screen shown in Figure 6.1. Upload the specification by browsing to the particular file and clicking the *Upload File* button in the *Upload Specification* panel. Only valid specification files with a .yawl or .xml extension can be uploaded.

When a specification is uploaded, it is validated against the YAWL specification schema for validity. If there is a problem with the upload, an appropriate error message is displayed in a popup dialog.

Cases can be launched for a specification by selecting it from the list of loaded specifications, then clicking the *Launch Case* button in the *Loaded Specifications* panel. If the specification has input parameters a form will appear asking for values for these parameters to be provided before the case is launched.

Note that attempting to upload a specification that has already been uploaded (i.e. same specification id and version) is not possible and will result in an error message to this effect being displayed. It is possible to have different versions of the same specification loaded at any one time (e.g. if a specification has been updated, but there are still cases running against the older version), but new cases may only be launched for the latest version uploaded.

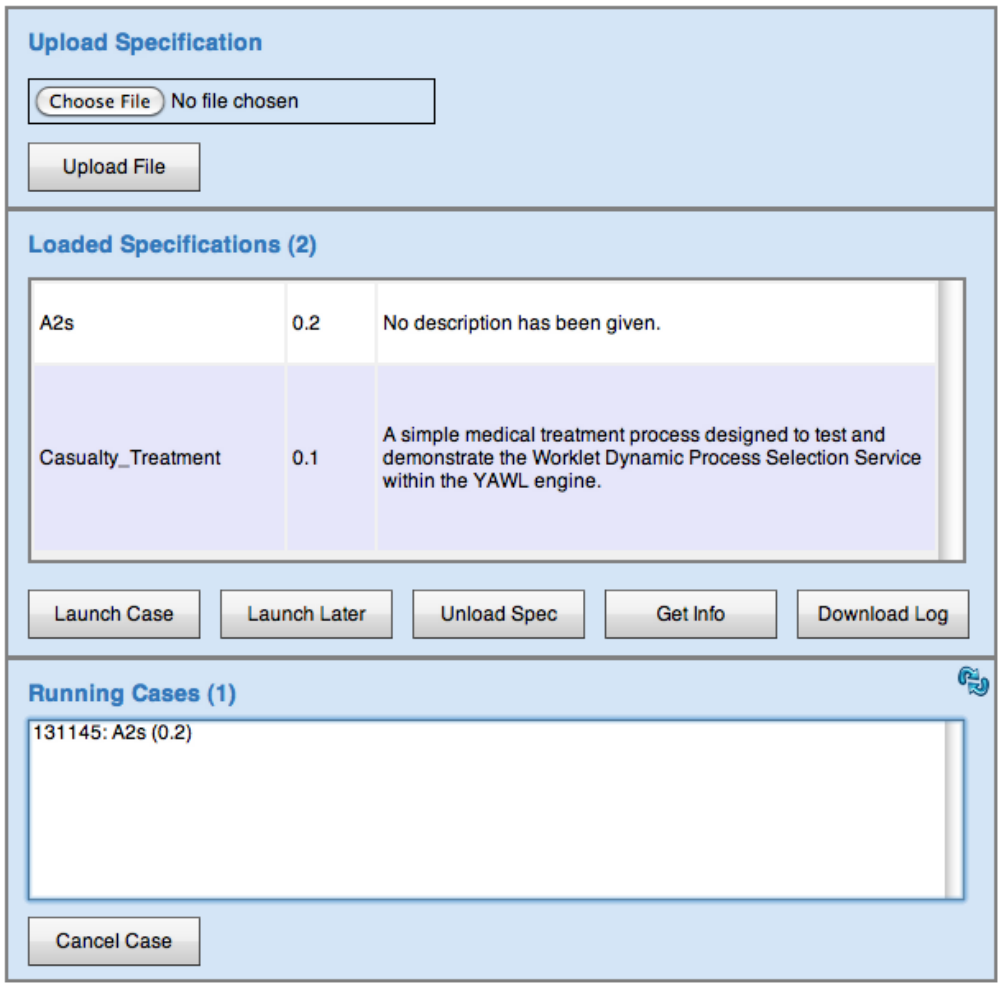


Figure 6.1: Case Management

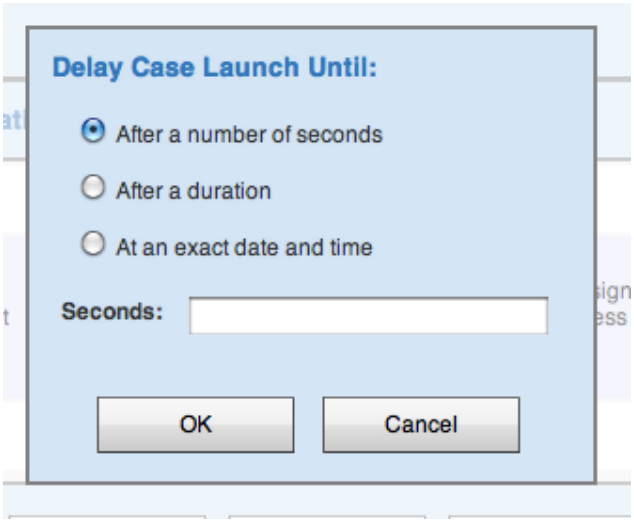


Figure 6.2: Delayed Launch dialog

The launching of a new case can be queued for a period by clicking the *Launch Later* button, which displays the popup dialog shown in Figure 6.2. The dialog provides three ways to specify a delay:

1. **A number of seconds:** allows for the entry of a whole number of seconds. When the dialog is closed with the *OK* button, the case will launch when the specified number of seconds has elapsed.

2. **A Duration:** allows for the entry of an xsd:duration value (see Section 4.13 for details on specifying Duration type values). When the dialog is closed with the *OK* button, the case will launch when the specified duration has elapsed.

3. **An exact date and time:** allows for the entry of a specific future moment as an xsd:dateTime value (i.e. of the form `CCYY-MM-DDThh:mm:ss`). When the dialog is closed with the *OK* button, the case will launch when the specified moment arrives.

To unload a specification from the Engine, select it and click the *Unload Spec* button in the *Loaded Specifications* panel. Note that an attempt to unload a specification will fail if there are any cases still executing against it.

The *Get Info* button in the *Loaded Specifications* panel will display some meta data about the selected specification – an example can be seen in Figure 6.3.

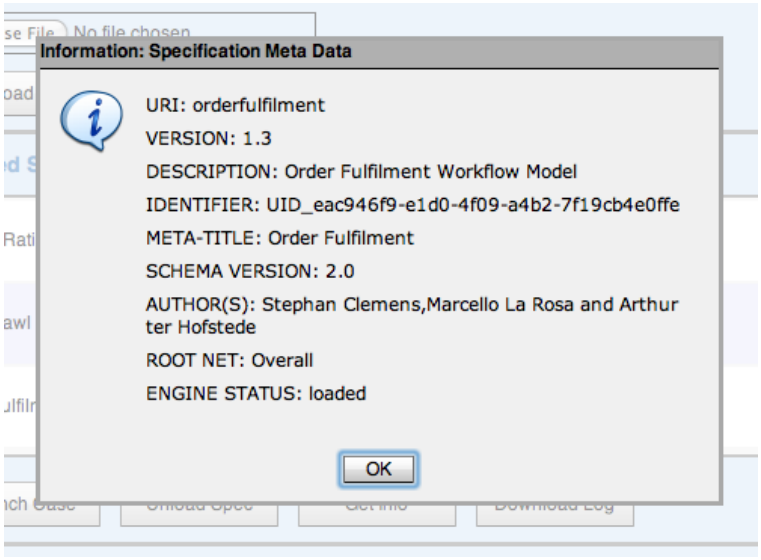


Figure 6.3: Specification meta data dialog

The fifth button in the *Loaded Specifications* panel is the *Download Log* button, which will generate and download a file containing the complete process log for the selected specification, in OpenXES format. The log file produced can be read directly into the process mining tool ProM[2]. Note that the generation of a process log file may take some time, especially if there has been a large number of cases executed for the selected specification.

The *Running Cases* panel shows a list of all the cases currently executing in the Engine, in the form "case number: specification name (version)". While the list is refreshed whenever the page is loaded, you may also refresh the list contents by clicking on the *refresh* icon located at the top right of the *Running Cases* panel. A case can be cancelled at any time by selecting it from the list of running cases and clicking the *Cancel Case* button.

[2] http://processmining.org

6.4.2 Admin Queues

An administrator can view all of the various work items that are currently active and their statuses in the Admin Queues Screen (see Figure 6.4). There are two Admin queues, each of which can be selected by clicking on the appropriate tab: *Worklisted*, which lists all the work items on participants' work queues, and *Unoffered*, which lists the active work items that do not currently reside on any participant's queues. From the *Unoffered* queue, an administrator can assign unoffered work items to the offered, allocated or started work lists of selected participant(s) via the buttons *Offer*, *Allocate* and *Start* respectively.

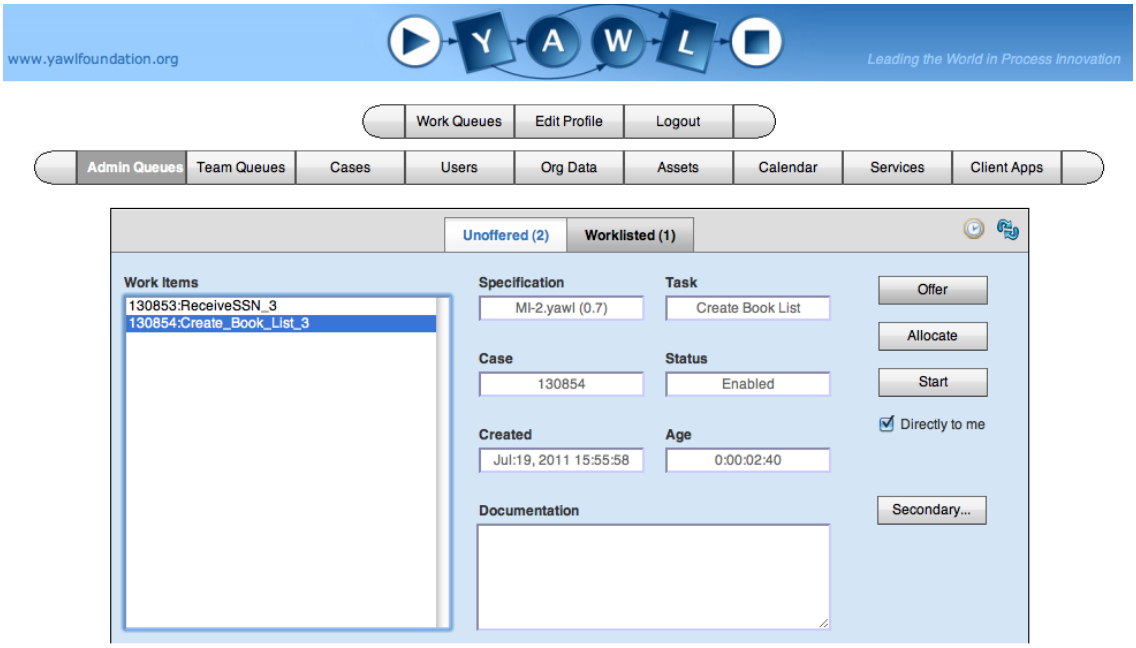


Figure 6.4: Administration Queues

From the *Worklisted* queue, an administrator can change the participant and/or the status of the work item through the *Reoffer*, *Reallocate* and *Restart* buttons. Reverting to a previous state is allowed using these buttons:

- If the current resource status is *Offered*, the work item may be *Reoffered* to one or more participants.
- If the current resource status is *Allocated*, the work item may be *Reoffered* to one or more participants, or *Reallocated* to a single participant.
- If the current resource status is *Started*, the work item may be *Reoffered* to one or more participants, *Reallocated* to a single participant, or *Restarted* to a single participant.

While the queues are refreshed whenever the page is loaded, you may also refresh the queue contents by clicking on the *refresh* icon located at the top right of the work queues panel.

If the *Directly to me* checkbox is unchecked, when one of these buttons is clicked, a list of all participants will be displayed, from which selections can be made. If the checkbox is checked, the action triggered by the button click will occur as if the currently logged on participant had selected themselves from the list (thus bypassing the list display). The *Directly to me* checkbox is available only to participants with administrator privileges, but not to the generic "admin" userid (since "admin" is not a participant, it cannot have work items assigned to it).

The Resource Service maintains a local cache of active work items. On rare occasions, this cache may become out-of-synch with the list of active work items maintained by the Engine (for example, where another

custom service has modified the status of a work item). The *Synchronise* icon (to the left of the refresh icon) allows administrators to re-synchronise the local cache with that of the Engine. The results of any changes brought about by re-synchronisation can be noted in the Tomcat log files. There are some system overheads involved with re-synchronisation, and it is rarely necessary, but may occasionally be of some benefit to long-running instances.

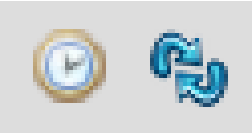


Figure 6.5: Admin Queue Tool Buttons (detail): Synch (left), Refresh (right)

Secondary Resources Administration

The *Secondary...* button on the *Admin Queues* form provides administrators with the ability to dynamically alter the set of secondary resources allocated to a work item, if the work item has not yet been started. This becomes especially important if the service has been configured to block the starting of work items on unavailable secondary resources (cf. Section 6.2). It is also possible to add secondary resources to a work item at runtime (before it starts) even if none were allocated to it at design time.

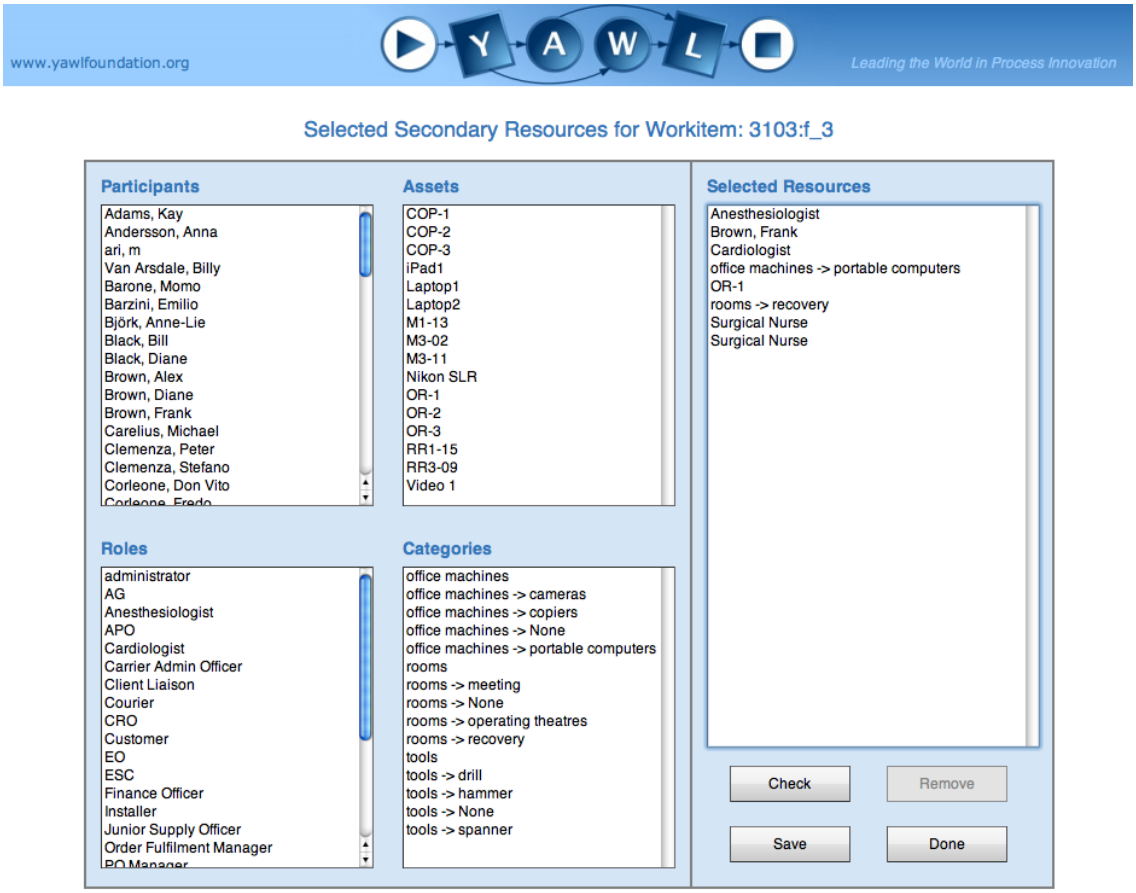


Figure 6.6: Secondary Resources dialog

Clicking on the *Secondary...* button opens the *Secondary Resource Administration* dialog shown in Figure 6.6.

This dialog has a similar layout to the 'Secondary Resources' panel of the Editor's Resourcing Dialog (cf. Section 4.8.2). On the left are lists of individual participants and their Roles, and individual 'assets' (i.e. non-human resources) and their categories. On the right is the list of selected secondary resources for the chosen work item. Selecting an item from a list on the left will move it to the list of selected resources on the right. Individual participants and assets can only be added once, but roles and categories may be added several times (as in the example of Figure 6.6, where the role "Surgical Nurse" has been chosen twice because two nurses are required for the work item).

ASIDE: Every non-human resource, or asset, belongs to a category. For example, a photocopier, a printer, a portable computer and a camera may all belong to the category "office machines". Categories may be further divided into a set of sub-categories, so for example, a photocopier may belong to the "copiers" sub-category, and a camera may belong to the "cameras" sub-category of the "office machines" category. Each category has a default sub-category called "None", into which all of the assets of the category that have not been explicitly sub-categorised are placed.

There are four buttons on the *Secondary Resource Administration* dialog:

- **Check**: This button will check the availability of each resource listed as a selected resource. A popup message dialog will list those resources that are currently unavailable, or a "success" message if all the selected resources are currently available.
- **Remove**: This button will remove the currently selected resource from the list of selected resources.
- **Save**: This button will save any changes made.
- **Done**: Will close the dialog.

See Section 6.5 for more information about the management of non-human resources.

6.4.3 Service Management

The Service Management Screen can be used to add and to remove registered custom services. In the example in Figure 6.7, three services are already registered, the *SMS Service*, the *Worklet Service* and the *Web Service Invoker Service*.

A new service can be added by providing a name, a password and confirmation, a URI and a Description. The password and confirmation password must match each other, and name and password must also exactly match the credentials that will be used by the service to log onto the Engine (see Table 6.1 for a list of credentials for each of the standard YAWL custom services). The URI is validated by contacting it and waiting for an appropriate response, so care should be taken that the URI provided exactly matches that of the specified service.

Name	Password	URI
workletService	yWorklet	http://localhost:8080/workletService/ib
wsInvokerService	yWSInvoker	http://localhost:8080/yawlWSInvoker
smsService	ySMS	http://localhost:8080/yawlSMSInvoker
mailService	yMail	http://localhost:8080/mailService/ib
twitterService	yTwitter	http://localhost:8080/twitterService/ib
digitalSignatureService	yDigitalSignature	http://localhost:8080/digitalSignature/ib

Table 6.1: Logon Credentials and URIs for the Standard Custom Services

Although the Resource Service itself is a custom service, it is *not* registered using this form, because it is pre-registered with the Engine as the *Default Worklist Handler* via a configuration setting.

The Editor, when connected to the Engine, will retrieve the set of registered services, and make them available for assigning to tasks by displaying the description entered for each service in the drop down list of the Task Decomposition dialog (see Section 4.10.2).

To deregister a service from the Engine, select it in the list, then click the *Remove* button.

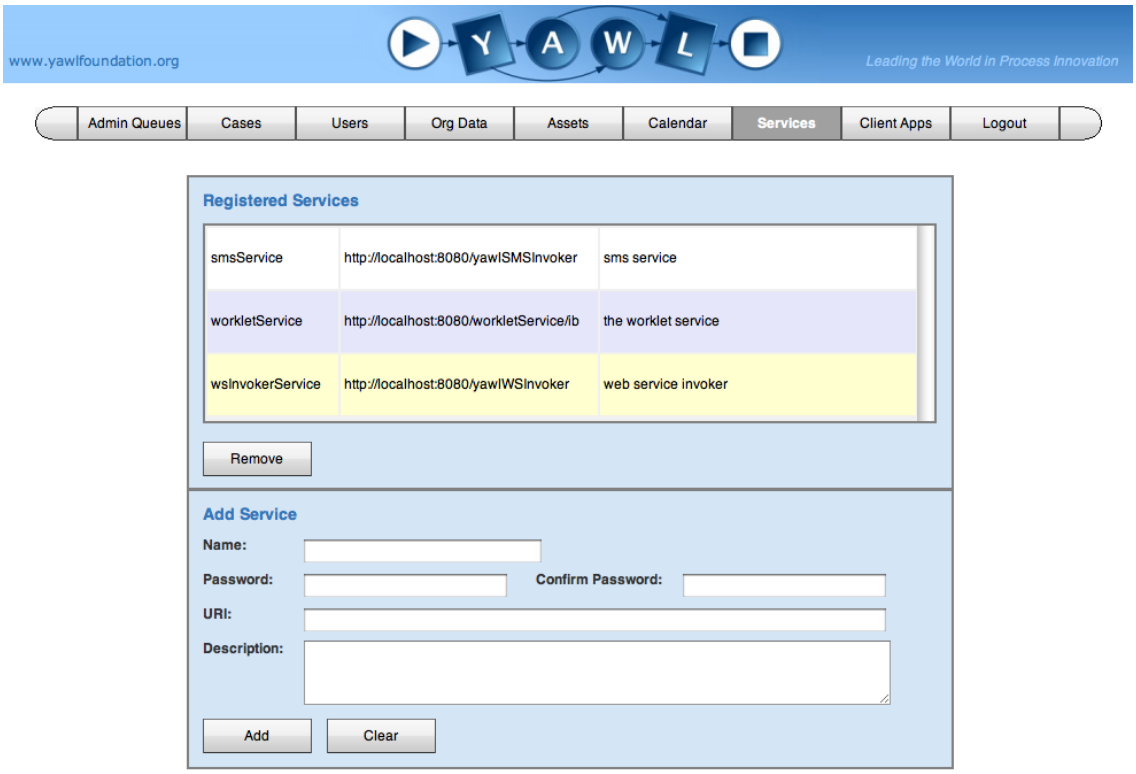


Figure 6.7: Service Management

6.4.4 Client Application Management

Similar to the Service Management Screen, the Client Application Management Screen provides for the registration of logon credentials for external applications and services (i.e. non-custom services) that wish to connect to the Engine via its interfaces. Figure 6.8 shows and example of the Client Management Screen.

To add a new client application, provide a name, a password and confirmation, and a Description. The password and confirmation password must match each other, and name and password must also exactly match the credentials that will be used by the application to log onto the Engine. The credentials for the Editor are editor (username) and yEditor (password), and for the monitor service they are monitorService (username) and yMonitor (password).

Note that in Figure 6.8 the generic admin user id is also listed in the registered accounts. Its listing here allows administrators to change the password on the generic admin if desired. To change the password or description of a registered client application account, select it from the list then click the *Edit* button – the details of that account will populate fields on the lower half of the form, allowing you to modify and save them.

To remove an account from the Engine, select it in the list, then click the *Remove* button. Note that the generic admin account cannot be removed in this way, but may be disabled via an Engine configuration setting (see Section 6.1).

ASIDE: The reason why the monitor service is listed as a client application and not as a custom service is simply because the monitor service does not qualify as a custom service. To qualify as a custom service, a service must be able to take responsibility for the execution of a task. That is, the service must be able to be assignable to a task at design time via the Editor's Task Decomposition dialog (as are all those services mentioned in Section 6.4.3). The monitor service can't be assigned tasks for execution, since it is designed to display information about current processes, and so rightly belongs in the list of client applications, as do all such services.

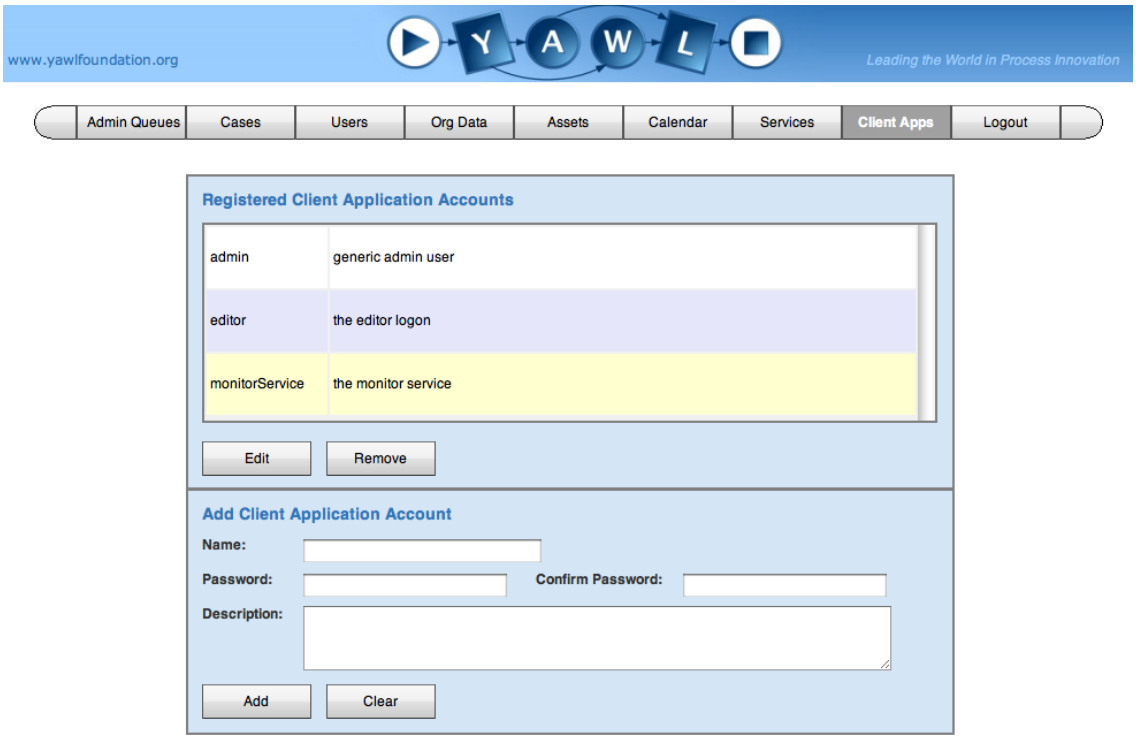


Figure 6.8: Client Application Management

6.4.5 Managing Organisational Data

Roles, capabilities, positions and organisational groupings can be defined through the Org Data Mgt Screen, see Figure 6.9:

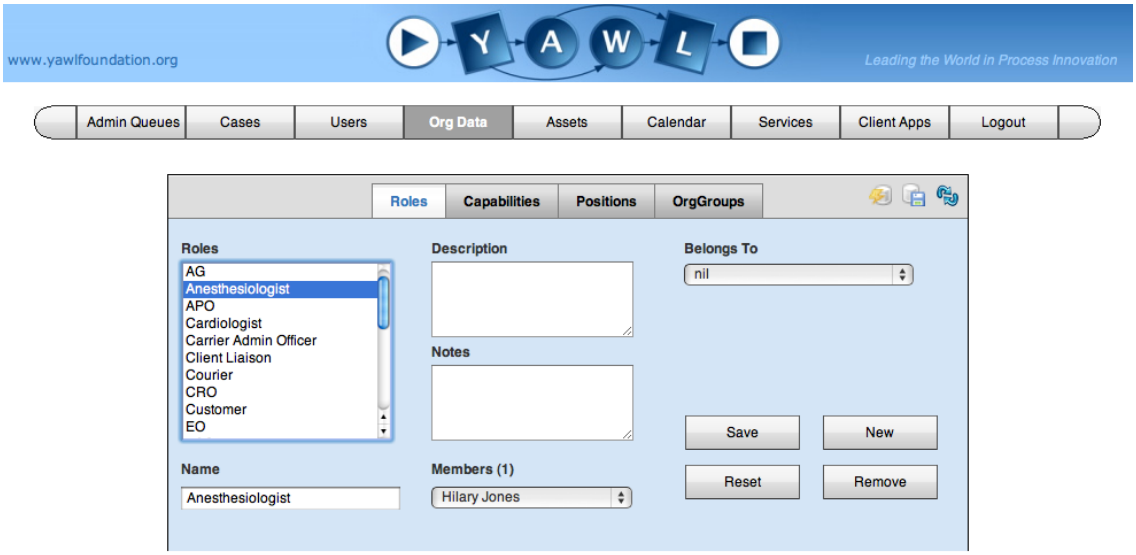


Figure 6.9: Organisational Data Management

- **Role**: Generally, a role is a duty or set of duties that are performed by one or more participants. For ex-

ample, bank teller, police constable, credit officer, auditor, properties manager and junior programmer are all examples of roles that may be carried out by one or more participants within an organisation. There may be several participants performing the same role (for example, a bank may have a number of tellers), so a typical role in an organisational model may contain a number of participants. Conversely, a certain participant may perform multiple roles. Further, a role may *belong to* a larger, more general role (for example, the roles junior teller and senior teller may both belong to a more general role called 'teller'). A role may be included in the distribution set for a task at design time, meaning that all of the participants performing that role (or any of its sub-roles) are to be considered as potential recipients of a work item created from the task at runtime.

- **Capability**: A capability is some desired skill or ability that a participant may possess. For example, first aid skills, health and safety training, a forklift license or a second language may all be considered as capabilities that a participant may possess that may be useful to an organisation. There may be several participants within an organisation possessing the same capability, and a certain participant may possess a number of capabilities. A capability (or capabilities) may be included in a filter defined at design time that is run over the distribution set for a task at runtime, so that only those participants within the distribution set that possess the specified capability or capabilities are potential recipients of a work item created from the task.

- **Position**: A position typically refers to a unique job within an organisation for the purposes of defining lines-of-reporting within the organisational model. Examples might include CEO or Bank Manager, or may be internal job codes (such as 'TEL0123'). A position may report to zero or one other positions (for example, bank teller 'TEL0123' may *report to* the Bank Manager), and may *belong to* zero or one Org Groups (see below). Like capabilities, a position (or positions) may be included in a filter defined at design time that is run over the distribution set for a task at runtime. Positions are also used at runtime to enable resource patterns such as delegation, reallocation and viewing of team work queues (see Section 6.7 for more details).

- **Org Group**: An organisational group (org group) is a functional grouping of positions. Common examples might include Marketing, Sales, Human Resources and so on, but may be any grouping relevant to an organisation. In the YAWL model, each position may *belong to* zero or one org groups. Further, like roles, an org group may *belong to* a larger, more general org group (for example, the groups Marketing and Sales may each belong to the more general Production group). Org groups are often also based on location. Like positions, org groups may be included in a filter defined at design time that is run over the distribution set for a task at runtime.

While the descriptions of the various entities in the YAWL model above discuss the typical uses of each, it should be clear that they represent, at the most basic level, merely various ways to group participants. The main point of distinction between them is that only roles can be used to populate a distribution set in the Editor, the other three may be used to perform filtering over the set.

The Org Data Screen contains four tab-pages, one for each of org entities listed above. The methods used to maintain the data for each entity is similar on each of these four pages:

- To **Add** a new entity, click the *New* button, which will display an 'Add New' input field. Enter a name for the entity, and optionally a description and note, choose the entity it belongs to and/or reports to as required (see below), then click the *Add* button. You may cancel the addition at any time by clicking the *Reset* button instead of the *Add* button.

- To **Modify** an entity, select it from the list, then add or change its name, description, note, belongs to and/or reports to values, then click the *Save* button.

- To **Delete** an entity, select it from the list and click the *Remove* button.

- To view the list of participants that 'own' an entity, select it from the list. The *Members* dropdown shows all of the participants associated with the selected entity.

With regards to the *belongs to* and *reports to* relations:

- A role may belong to another role – you may set this relation using the *Belongs To* dropdown on the Roles tab-page. This allows a hierarchy of roles to be created, so that when a task is assigned to a role in the Editor, and that role has other roles belonging to it, those roles are also implicitly included (by inheritance). You will not be allowed to have a role belong it itself, either directly or as part of a cycle back to itself, for example if role A belongs to role B which in turn belongs to role C, you will not be allowed to have role C belonging to role A (or B).
- An org group can belong to another org group, similarly to a role. You may also set an *Group Type* for an org group via a dropdown; select the type of group from the list then click the *Save* button.
- A position may report to another position, allowing a lines-of-reporting hierarchy of positions to be created. Again, this is done in a similar fashion to setting a role *belongs to* hierarchy, and the same cyclical constraints apply. A position may also belong to an Org Group, which can be chosen via the Org Group dropdown on the Positions tab-page (Figure 6.10).

Figure 6.10: The Positions tab-page

To enable easy backup and recovery of organisational data, two tool buttons are provided on the top right of the tab panel. They may be seen in detail in Figure 6.11.

Figure 6.11: Org Data Form Tool Buttons

The button on the left is the *Import Org Data from File* button, and the button in the centre is the *Export Org Data to File* button (the third button refreshes the form in a similar way to the refresh button on other pages, as mentioned previously). To export your current set of org data, click on the *Export* button – a file called 'YAWLOrgDataBackup.ybkp' will be created and downloaded via your browser. The file will contain your entire org database, including participants and non-human resources, in XML format (passwords are encrypted).

At any time, backed up org data can be re-imported by clicking on the *Import* button. You will be prompted for the file to import, via an *Import File* panel that will appear on the bottom of the form. Browse to the file, then click the *Import* button. Existing data is not removed – importing data will append new data and update existing data. A message describing the effects of the import will be displayed on completion.

6.4.6 Managing Users

Though the User Mgt Screen (see Figure 6.12), an administrator can add participants and change details and privileges for existing participants.

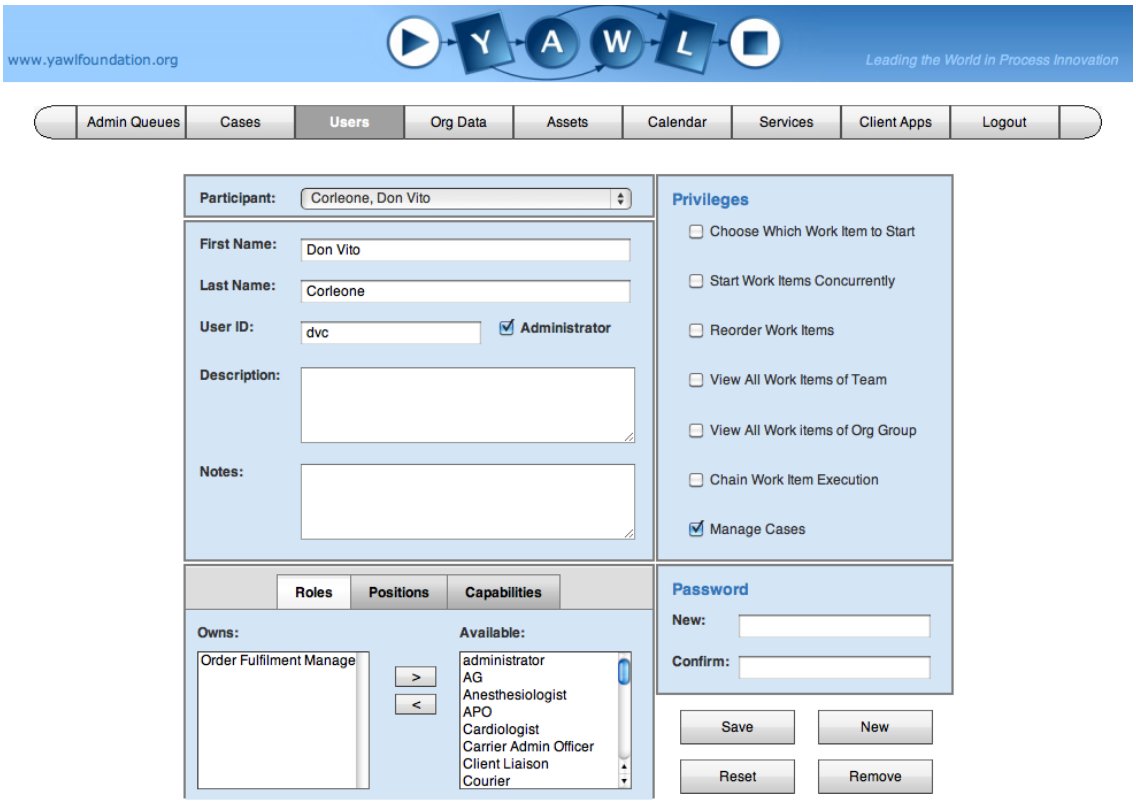


Figure 6.12: User Management

The User Mgt Screen consists of four panels: top-left shows the participant's personal details (name, password, userid and so on); top-right allows the setting of user privileges (see below); bottom-left allows the assigning of the participant to various roles, positions and capabilities; and bottom-right is where the participant's password can be reset. There are many similarities between the User Mgt and Org Data Mgt Screens regarding the addition, modification and removal of items:

- To **View** an existing participant's details, select the participant's name from the *Participant* dropdowm list.
- To **Add** a new participant, click the *New* button, which will disable the dropdown and activate all other fields. For a new participant, entries for first name, last name, userid and password (new and confirm) are required. Userids must begin with a character and may contain the letters, digits and underscores. Passwords must be at least 4 characters in length. The description, note and administrator fields are optional, as are privilege settings (by default all are unselected) and role/position/capability memberships. When you have finished adding participant information, click the *Add* button. You may cancel the addition at any time by clicking the *Reset* button instead of the *Add* button.

- To **Modify** a participant, select them from the list, then add or change the desired fields, then click the *Save* button.
- To **Delete** an participant, select them from the list and click the *Remove* button.

User Privileges

Primarily, each participant may be designated a 'user' (the default) or an 'administrator'. To grant administrator privileges for a participant, select the participant from the dropdown list, tick the *Administrator* checkbox, then click the *Save* button. Administrator privilege overrides all other user privileges. Participant's without administrator privileges may be granted specific privileges by selecting the participant from the dropdown list, then ticking the desired privileges, then clicking the *Save* button. The privileges that may be assigned to participants on an individual basis are:

- **Choose Which Work Item to Start**: When granted, this privilege allows a participant to choose any work item listed on their allocated queue to start. When denied (the default) only the first listed work item may be chosen. Work items are listed in order of age, with the oldest work item at the top of the list.
- **Start Work Items Concurrently**: When granted, this privilege allows a participant to have a number of work items executing concurrently on their started queue (or, more accurately, may choose to start additional work items from their allocated queue while other previously started work items have not yet completed). When denied (the default), a work item on the participant's allocated queue may not be started while there is a previously started work item on their start queue (i.e. one that has not yet completed).
- **Reorder Work Items**: When granted, the participant may choose a work item to start from anywhere in the list of allocated work items. When denied, only the first listed work item may be chosen. In the YAWL environment, there is essentially no difference between this privilege and *Choose Which Work Item to Start*.
- **View All Work Items of Team**: When granted, this privilege gives a participant access to the *Team Queues* form, and displays on that form a consolidated list of all work items on all work queues of all participants subordinate to the participant who has been granted the privilege (that is, participants holding positions that report to a position held by the granted participant, either directly or through a hierarchy of positions). When denied (the default), the *Team Queues* form is not available to the participant.
- **View All Work Items of Org Group**: When granted, this privilege gives a participant access to the *Team Queues* form, and displays on that form a consolidated list of all work items on all work queues of all participants in the same Org Group as the granted participant. When denied (the default), the *Team Queues* form is not available to the participant.
- **Chain Work Item Execution**: When granted, this privilege allows a participant to chain work items for a case. When denied (the default), the participant may not chain cases (see Section 6.7 for details regarding the chaining of tasks).
- **Manage Cases**: When granted, this privilege gives a participant access to the *Case Mgt* form, providing the ability to load process specifications, and start and cancel case instances. When denied (the default), the *Case Mgt* form is not available to the participant.

A participant with default user privileges (i.e. all unselected) have access to their own work queues, and may view/edit their own profile. A participant with *Manage Cases* privilege can also access the Case Mgt screen. A participant with *View All Work Items of Team* or *View All Work Items of Org Group* privilege can also access the Team Queues screen. All other screens can only be accessed by participants with administrator access.

6.4.7 Task Privileges

Task privileges (or, more precisely, *User-Task* privileges), unlike the *User* privileges described above, are set at design time via the Editor (see Chapter 4, Section 4.8.3) *on an individual task basis*. The relevant dialog tab is re-shown in Figure 6.13, and a description of each task privilege is included here for completeness.

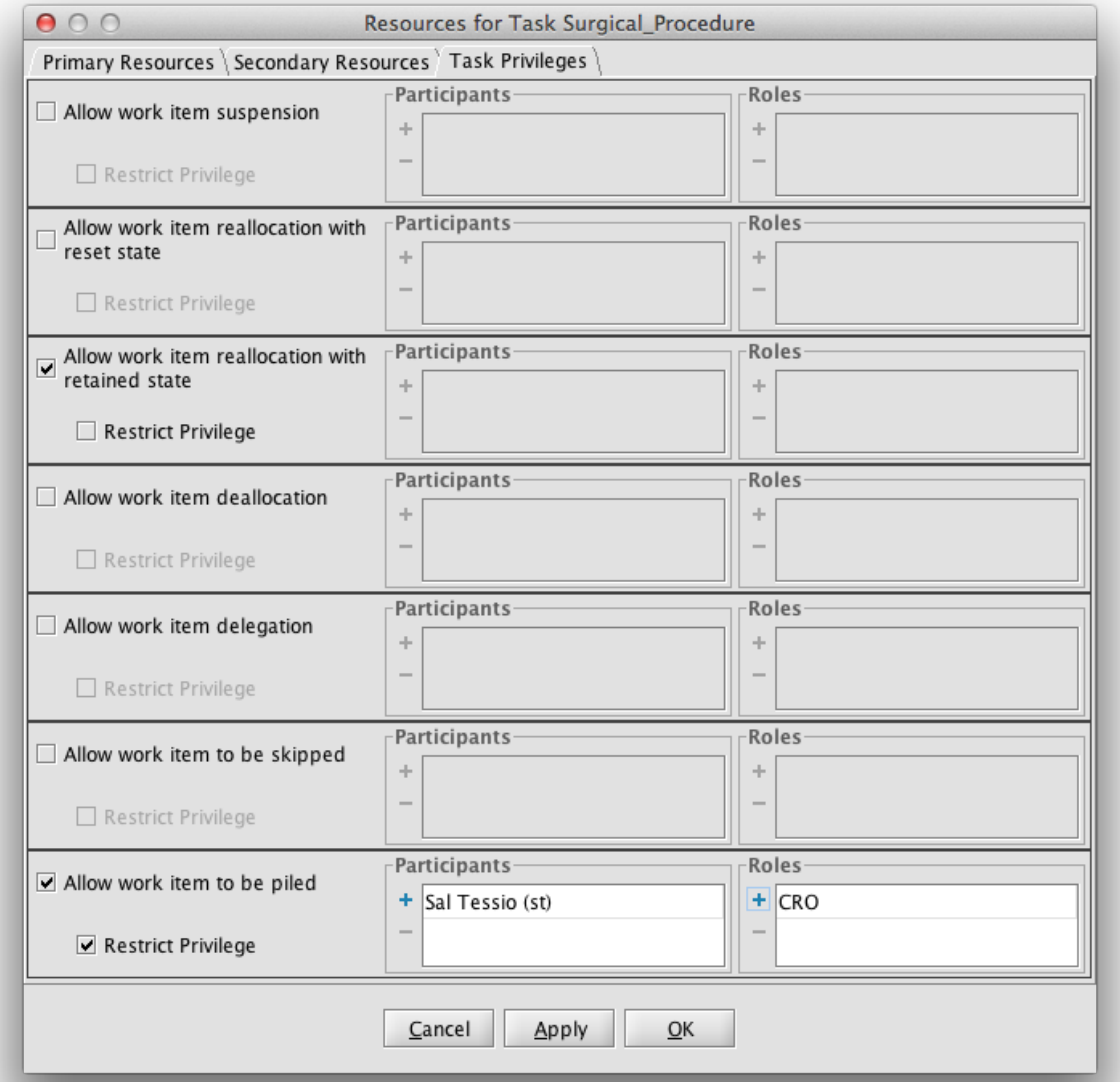


Figure 6.13: The Task Privileges tab of the Resourcing Dialog

Broadly speaking, task privileges grant or deny the ability to affect in various ways how work items are resourced after initial distribution has completed. There are seven task privileges:

- **Allow work item suspension**: When granted, allows a participant to suspend the execution of a work item after it has been started.
- **Allow work item reallocation with reset state**: When granted, allows a participant to transfer re-

sponsibility for the execution of a work item from themselves to another participant, with the data parameters of the work item reset to the values held when the work item was first started.

- **Allow work item reallocation with retained state**: When granted, allows a participant to transfer responsibility for the execution of a work item from themselves to another participant, with the data parameters of the work item having their current values maintained.
- **Allow work item deallocation**: When granted, allows a participant to reject or rollback the allocation of a work item to their allocated queue. The work item is redistributed using the original resourcing specification, but with the participant removed from the distribution set.
- **Allow work item delegation**: When granted, allows a participant to delegate the responsibility for the execution of a work item to a subordinate member of their work team, as defined by the organisational model.
- **Allow work item to be skipped**: When granted, allows a participant to have the execution of a work item skipped – that is, immediately completed without performing its work.
- **Allow work item to be piled**: When granted, allows a participant to demand that all future instances of work items derived from this task, in all future instances of the specification of which the task is a member, are immediately directly routed to the participant and started.

All task privileges are denied by default, and so must be set explicitly for each task as required. Each privilege may be set to allow all participants and roles, or restricted to allow only those participants and/or roles explicitly specified.

6.5 Managing Non-Human Resources

Similarly to the *Org Data Management* form, the *Asset Management* form allows administrators to add, modify and remove non-human resources and their categories. The form consists of two tabs, *Resources* and *Categories*, as can be seen in Figure 6.14.

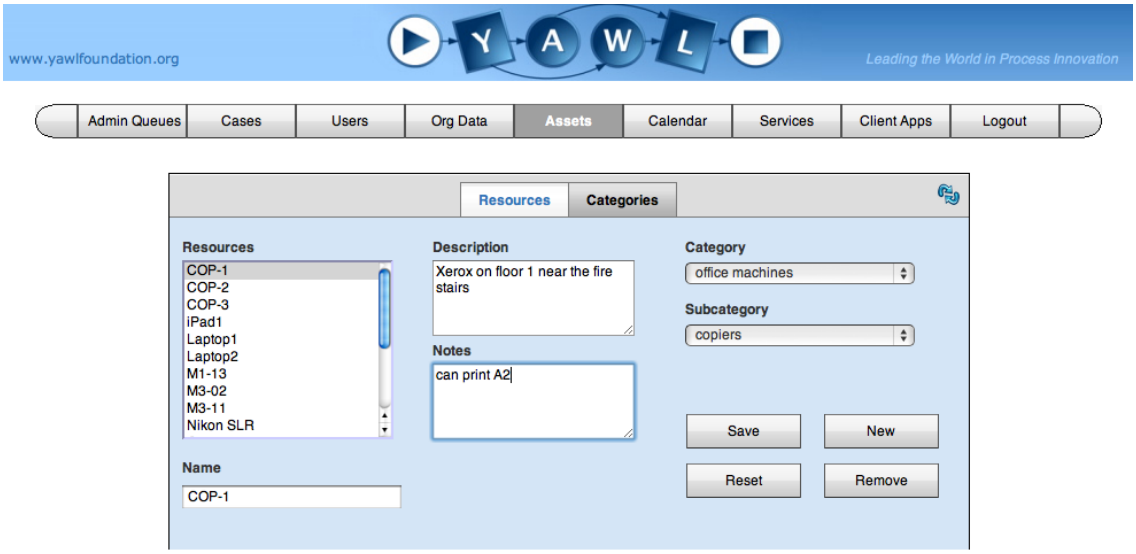


Figure 6.14: Non-Human Resource Management (Resources tab)

- **Resources**: A non-human resource (or *asset*) is any organisational resource that isn't a person. Examples may include vehicles, meeting rooms, tools, raw materials, computers and other office equipment, and so on. Zero or more non-human resources may be allocated to a task, at design time and/or at runtime before the task is started, as so-called *secondary resources*. The set of allocated non-human resources represents those additional organisational resources that are required to perform the work of the work item. However, unlike the *primary resource*, they do not directly interact with the work list that the work item appears on. For example, in a hospital surgery setting, the primary resource for a *Schedule Surgery* task may be a participant that belongs to the *Schedulers* role (only a participant can be a primary resource, and a task can have exactly one primary resource), while secondary resources for the task may include an operating theatre, a recovery room, sets of surgical instruments, a number of surgeons and surgical nurses (participants can be secondary resources, too) and so on.
- **Categories** A category is a grouping of non-human resources that relate to each other in some way. For example, a category called *Rooms* may include all of the usable rooms of an organisation that can be used in some way to perform an activity. A category may be further split into a number of sub-categories, for example the *Rooms* category in a hospital setting may include sub-categories such as *Meeting Rooms*, *Operating Theatres*, *Recovery Rooms*, *Lunch Rooms* and so on. A resource that is placed in a particular sub-category is also considered to be a member of its parent category (a room called 'M-123' in sub-category *Meeting Rooms* is also a member of category *Rooms*). Each resource therefore belongs to a single category and optionally a single sub-category within that category. Each category contains a 'special' sub-category called *None* which implicitly contains all the resources of the category that have not been explicitly placed into one of its sub-categories (for example, if the organisation has a single Conference room, there is little benefit creating a *Conference Rooms* sub-category for that one room).

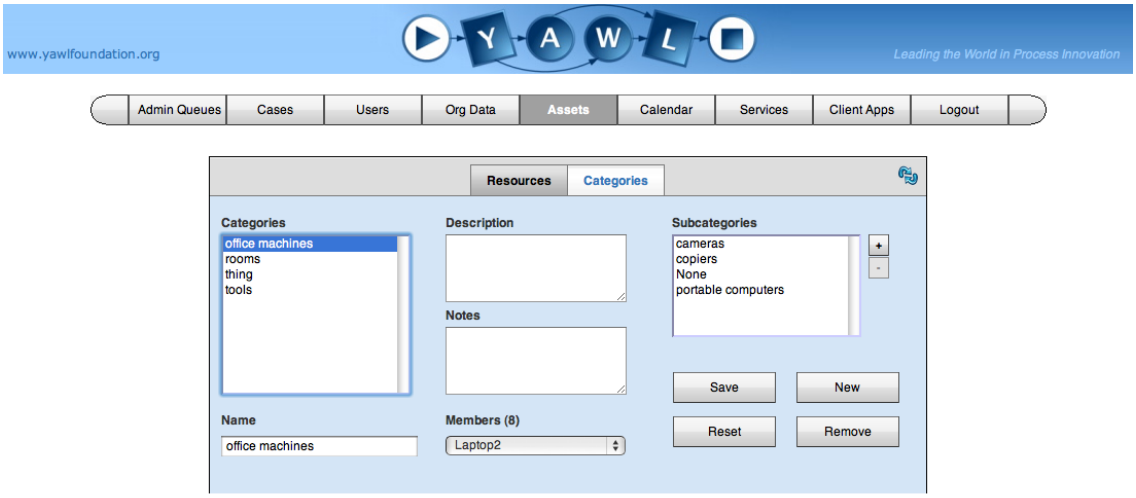


Figure 6.15: Non-Human Resource Management (Categories tab)

Of course, an organisation would include in its organisational model only those non-human resources that are limited in some way and are required exclusively to complete some activity. Also, how non-human resources are categorised is entirely subjective.

The **Resources** tab of the *Non-human Resources Management* form (Figure 6.14) is where individual non-human resources can be added, modified or removed.

- To **Add** a new non-human resource, first click the *New* button. Enter a name for the resource, and optionally a description and note, then choose the category it belongs to and optionally its sub-category, and finally click the *Add* button. You may cancel the addition at any time by clicking the *Reset* button instead of the *Add* button. Changing the selected category also changes the list of related sub-categories. If the resource does not have a sub-category, leave it set as 'None'.

- To **Modify** an non-human resource, select it from the list, then add or change its name, description, note, category and/or sub-category values, then click the *Save* button.
- To **Delete** an entity, select it from the list and click the *Remove* button.

On the **Categories** tab of the *Non-human Resources Management* form (Figure 6.15), categories and sub-categories can be added or removed, and member resources of each category can be viewed.

Categories can be added, updated and removed in the same way as non-human resources (on the *Resources* tab). To **add a sub-category** for the selected category, first click on the *Add Sub-category* button (the '+' button to the right of the sub-categories list), enter the name of the new sub-category in the input field that appears at the bottom of the list, then click the accept (tick) button to save it to the list, or the cancel (cross) button to cancel the addition (Figure 6.16). To **remove a sub-category** for the selected category, first select it then click on the *Remove Sub-category* button (the '−' button to the right of the sub-categories list). Note that the sub-category 'None' cannot be removed.

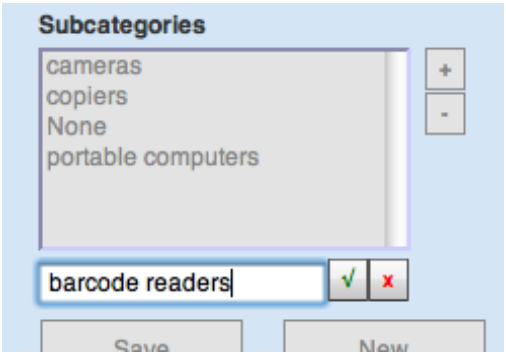


Figure 6.16: Non-Human Resource Management (Categories tab - detail)

6.6 Resource Calendar Management

As mentioned in the previous section, all of the resources (human and non-human, primary and secondary) allocated to a work item are marked as *in use* while the work item is executing, and are *released* when it completes. This means that while a resource is in use, it is unavailable to other work items that it may also have been allocated to. However, there may be other reasons why a resource is not available, even if it is not currently engaged in the performance of a work item. For example, a participant may be on annual leave, or away sick, or a particular machine may be offline for maintenance, and so on. To allow these periods of unavailability to be recorded and taken into account, a resource calendar is maintained by the Resource Service. The *Resource Calendar Management* form is shown in Figure 6.17. On this form, calendar entries for resources may be viewed, added, modified and removed. An entry in the calendar denotes that the specified resource is *unavailable* for the specified period.

At the top left of the form there is a date selector; choose a particular date to view entries for that date. Clicking the calendar button will display a calendar component from which a particular date may be selected. Clicking the left arrow button will go to the previous day, while the right arrow button will go to the next day.

On the top right, there is a dropdown list called *Filter*, with the following selections:

- **Unfiltered**: All entries for all resources are listed for the selected date (as in Figure 6.17). Entries cannot be added in this mode (see below).
- **All Resources**: Allows you to add entries that apply to all resources, both human and non-human. Lists all the *All Resources* entries for the selected date.

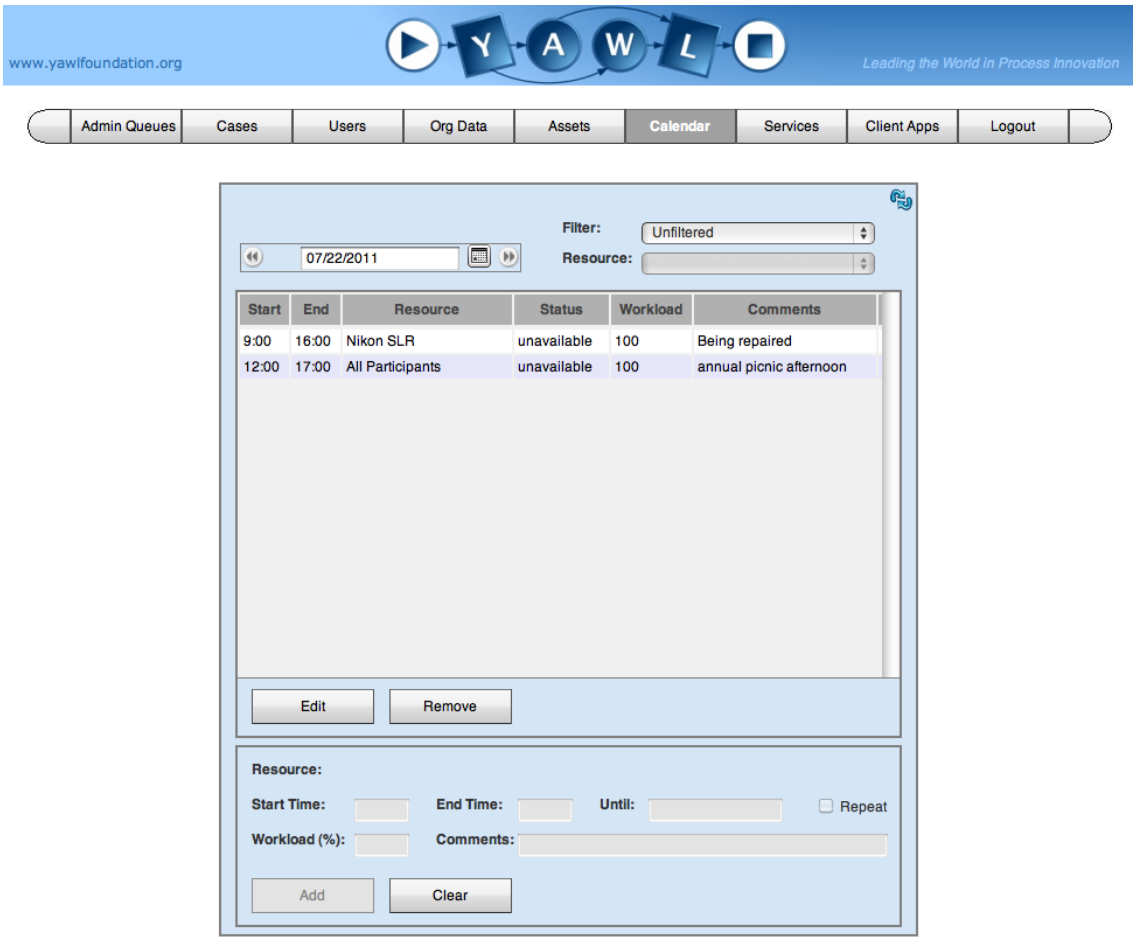


Figure 6.17: Resource Calendar Management Form

- **All Participants**: Allows you to add entries that apply to all participants. Lists all the *All Participants* entries for the selected date.
- **All Assets**: Allows you to add entries that apply to all assets (i.e. non-human resources). Lists all the *All Assets* entries for the selected date.
- **Selected Participant**: Enables the *Resource* dropdown list beneath it, and allows you to select an individual participant from that list. Once selected, all the existing entries for the participant for the selected date are listed, and allows new entries to be added for the participant.
- **Selected Asset**: Enables the *Resource* dropdown list beneath it, and allows you to select an individual asset from that list. Once selected, all the existing entries for the asset for the selected date are listed, and allows new entries to be added for the asset.

As mentioned above, calendar entries may be added that apply to All Resources (as a group), All Participants (as a group), All Assets (as a group), an individual participant or an individual asset. To **add** a new entry:

- Select the group or individual resource using the *Filter* and, if required, the *Resource* dropdowns. The name of the selected group or resource will be displayed on the bottom panel (for example, see Figure 6.18)
- Add the *Start Time* and *End Time* that the group or resource will be unavailable for. Times are entered in 24 hour format (h:mm), for example 9:00 or 16:45. If a time has zero minutes, the minutes part can

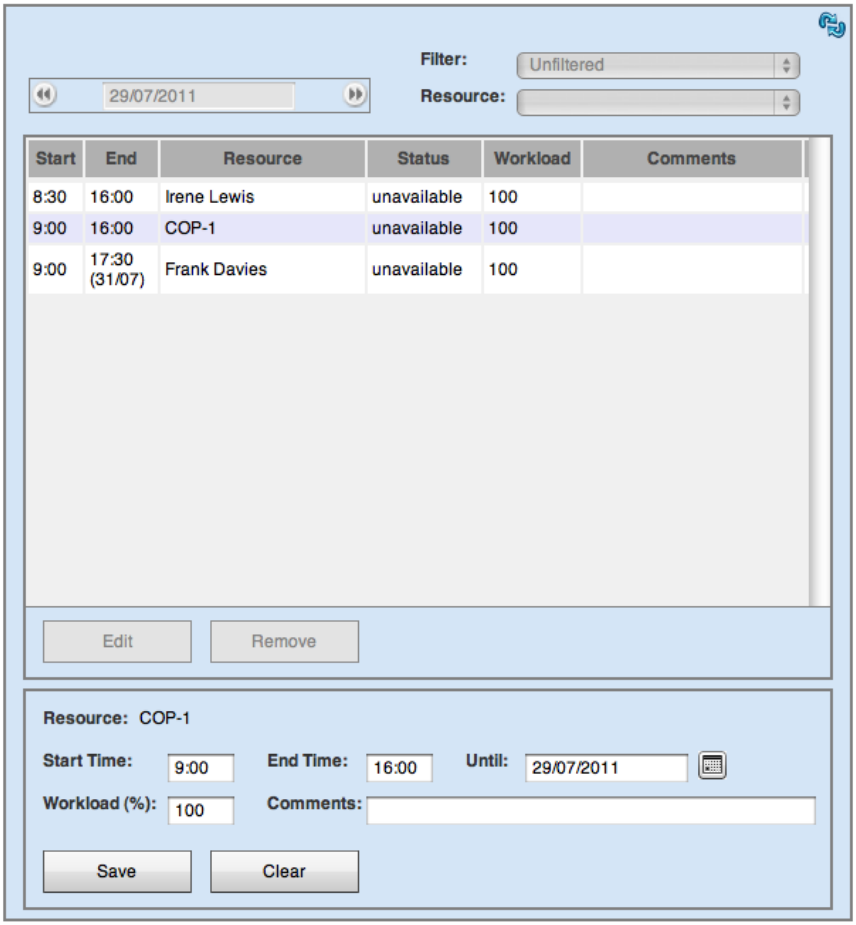


Figure 6.18: Resource Calendar Management – Editing an entry

be omitted (e.g. 9:00 can be entered as 9). Note that the start and end time fields are the only fields that are mandatory.

- (Optional) In the *Until* field, add the end date that the group or resource will be unavailable for. An entry in this field denotes that the group or resource will be unavailable from the date selected on the top left of the form *until* the date selected in the *Until* field. Entries spanning more than a single day will show the start and end date in addition to the start and end time. If this field is left blank, the entry will apply only for the selected date.

- (Optional) Check the *Repeat* checkbox to denote a certain block of time each day across a range of dates. If it is unchecked, the entry will span a continuous block from the start date (selected at top-left of the form) and start time, until the end date (specified in the *Until* field)) and end time. If it is checked, a number of entries will be added, each one spanning from the specified start time to the specified end time, one entry for each date in the range of dates specified. For example, in Figure 6.18, the entry for Irene Lewis was added with an 'until' date entered and 'repeat' checked, and so she is marked as unavailable from 8:30–16:00 for each day in the date ranged entered, while the entry for Frank Davies was added without 'repeat' checked, which means he is unavailable for one continuous period from 29/07/2011 at 9:00 until 31/07/2011 at 17:30. Whether *Repeat* is checked or not is only relevant if an *Until* date has also been specified.

- (Optional) Enter a percentage workload amount (between 1-100) in the *Workload* field. A workload of less than 100 means the resource will only be partially unavailable for the specified period. For

example, a value of 75 means that the resource may also be allocated as a secondary resource to a task at the same time, if that task only requires 25% of the output of that resource — that is, the resource can be shared across two or more tasks during the same period (an example may be a part-time worker, or a surgical nurse who can work between two operating theatres at the same time). If this field is leaft blank, it will default to 100%.

- (Optional) Enter a comment which explains why the group or resource is unavailable for the period.

To **edit** an entry, select it from those listed and click the *Edit* button. The entry's values will be copied to the relevant fields in the bottom panel. Edit the values as desired, then click the *Save* button the save the changes, or the *Clear* button to reject the changes and return to view mode. If an entry spans more than one day, you must select the first day of the span before you can edit it (a warning message will be displayed if there is an attempt to edit a multi-date entry that is not positioned on its start date).

Finally, to **remove** an entry, select it from the list then click the *Remove* button.

6.7 Work Queues

Work items have an associated life cycle and when interacting with the Resource Service it is important to understand the various stages that a work item can go through. An overview (not complete, but sufficient for our purposes) of the life-cycle of a work item is shown in Figure 6.19. The labels of the arcs correspond to the names of buttons that users of the Resource Service can click on to effect the state change.

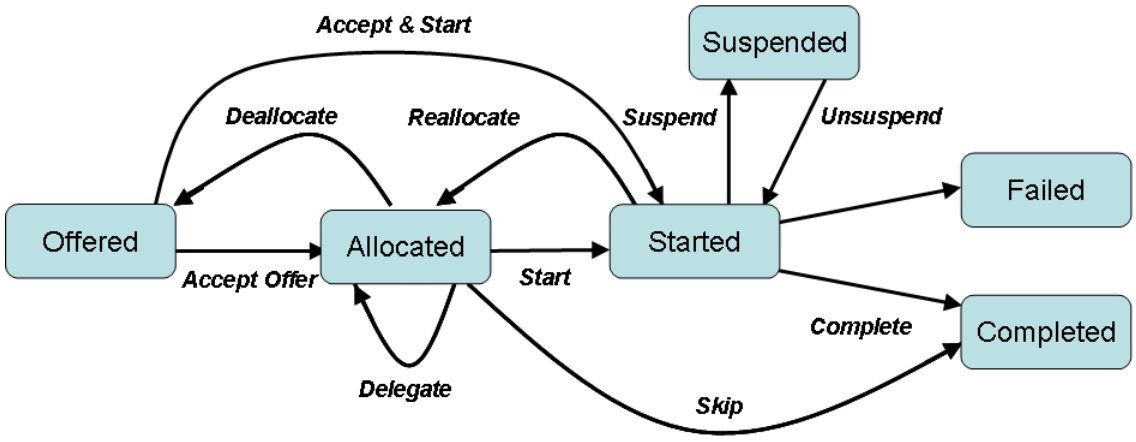


Figure 6.19: Part of the Life-cycle of a Work Item

Each participant has access to their own work queues, which are collectively known as a *worklist* – a graphical representation of their work queues via a series of web forms. Each worklist consists of four work queues: *Offered*, *Allocated*, *Started* and *Suspended*. Depending on a participant's privileges, there are a number of actions that can be performed on a work item in each queue. Some are concerned with processing the work item, while others provide for changes to the work item's resourcing.

The layout of each work queue is similar (see for example Figure 6.20).

- On the left is a list of the work items currently held in that queue.
- In the centre are some fields that describe the currently selected work item. The *Age* field, which shows how long it is since the work item was first created, changes to an *Expires In* field for timed work items (i.e. how long the work item has before its timer expires).
- On the right are a set of buttons representing the actions that may or may not be taken (depending on privileges) on that queue for the currently selected work item.

- At the bottom of each work queue is a *Documentation* field. If documentation was assigned to the task at design time, it will appear here at runtime. Additionally, the field is editable, so that documentation may be added at runtime. The following macros may be embedded in the documentation:
  - now - insert the current date and time
  - $date$ - insert the current date
  - $time$ - insert the current time

  An administrator may add documentation for a work item via the admin work queues and it will be immediately viewable on the queues of those participants who have that work item listed, and vice versa.

Each work queue may be selected by clicking on the appropriate tab.

6.7.1 The Offered Queue

The Offered queue lists the work items that have been offered to a participant. Each work item in an offered queue may have potentially been offered to a number of participants, which means there is no implied obligation to accept the offer, rather it is understood that the participant is one of a group, any one of who *may* choose to perform the work item.

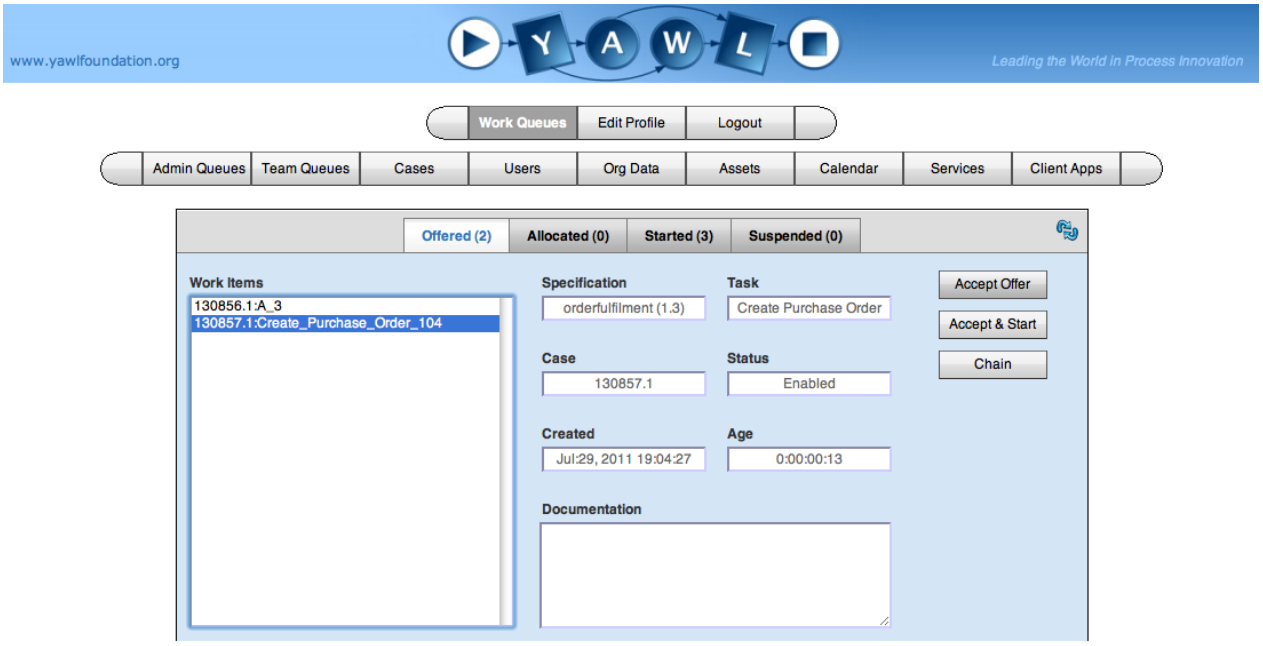


Figure 6.20: The Offered Work Queue

A participant may take the following actions on a work item in an offered queue:

- **Accept Offer**: By accepting an offer, a participant takes responsibility for the execution of the work item. The work item is moved from the offered queue and, if the start interaction is user-initiated, placed on the participant's allocated queue, or if the start interaction is system-initiated, the work item is immediately started and placed on the participant's started queue. This action removes the work item from the offered queues of all other participants that had been previously offered the work item.
- **Accept & Start**: This action works similarly to *Accept Offer*, except that if the work item's start interaction is user-initiated, the work item will instead be immediately started and placed on the participant's

started queue. Effectively, this concatenates two user actions into one, simply as a convenience for the user.

- **Chain**: This action will chain all the eligible work items of the case of which the work item is a member to this participant. Chaining means that, when a participant chooses to enact it, each remaining work item for the case is routed to the participant and immediately started, but *only* if the participant is a member of the distribution set for the work item. Chaining is effectively a short-circuiting of a resource specification for a task, where the participant chooses to automatically and immediately allocate and start any work item offered to him/her within the chosen case. Chaining of work items for a case continues until the case completes, or the participant turns off chaining via the *View Profile* form. A participant must have the "Chain Work Item Execution" user privilege to enable chaining.

6.7.2 The Allocated Queue

The Allocated queue lists the work items that have been allocated to a participant. Unlike an offer, a work item on an allocated queue means that it has been allocated to that participant alone, and comes with the understanding that the participant will at some time start the work item and perform its work.

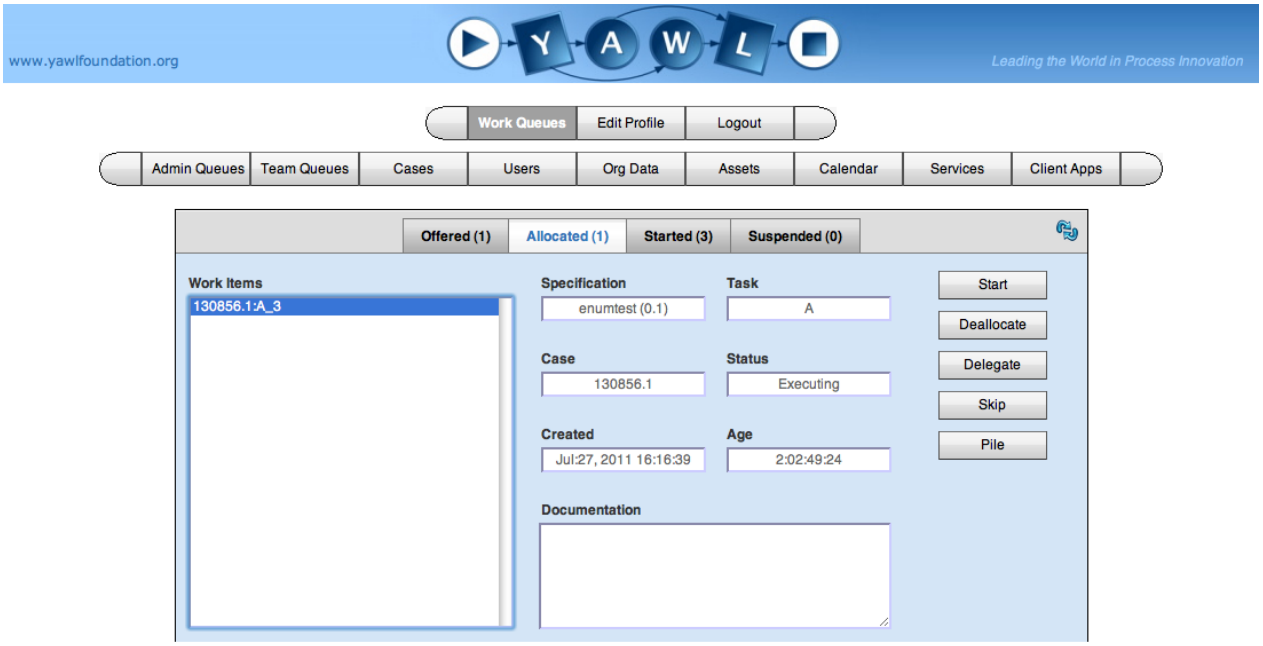


Figure 6.21: The Allocated Work Queue

A participant may take the following actions on a work item in an allocated queue:

- **Start**: The work item is started (i.e. begins executing), and moved to the participant's started queue.
- **Deallocate**: This action provides an authorised participant with a means of rejecting a work item that has been allocated to them. The work item is removed from the participant's allocated queue, the participant is removed from the original distribution set and the work item is redistributed as per the resourcing specification for the task. A participant must have the task privilege "Can Deallocate" to enable deallocation.
- **Delegate**: This action allows a participant to delegate responsibility for a work-item to another participant. The receiving participant must be subordinate to the delegating participant by Position. The work item is moved from the allocated queue of the delegator to the allocated queue of the receiver.

A participant must have the task privilege "Can Delegate" and have subordinate staff to successfully deallocate a work item.

- **Skip**: This action skips the execution of the work item – that is, the work item is immediately started and then completed, allowing the process to continue according to its subsequent control-flow. A participant must have the task privilege "Can Skip" to enable the skipping of a work item.
- **Pile**: When a work item is piled, the work item is immediately started and placed in the participant's started queue. Furthermore, *each and every future instance of the work item across all cases of the same specification* is automatically allocated to the participant and started, completely ignoring any resourcing specification for the task from which the work item is created. To put it another way, by piling a work item, a participant is entering into a contract with the Resource Service, asking that this work item, and all future occurrences of such work items created from the same task description as the original work item was created from, be immediately allocated and started to him/her. Piling of such work items continues until the participant turns off piling for the task via the *View Profile* form, or the participant logs out (if so configured). A participant must have the "Can Pile' task privilege to enable piling.

6.7.3 The Started Queue

The Started queue lists the work items that have been started by or for a participant. Each work item on a started queue has begun execution in a system sense, but may or may not have had any actual work begun for it by the participant—such work is performed by the participant viewing, editing and finally completing the work item.

Figure 6.22: The Started Work Queue

A participant may take the following actions on a work item in an started queue:

- **View/Edit**: This action will display the data parameters and their current values for the selected work item, either on a dynamically generated form (see Figure 6.23) or, if specified, a custom form, allowing the participant to view and/or edit the form's values. Any modified values are stored so that this action can be repeated for a particular work item a number of times before completion, allowing the

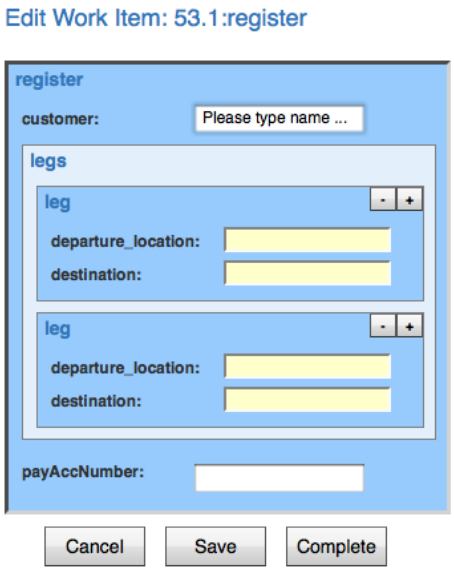


Figure 6.23: An Example of a Dynamically Generated Form

work item to be processed by the participant in a progressive manner, if required. This action is disabled if the work item has no data parameters to display or gather values for.

- **Suspend**: This action suspends the selected work item. The work item is removed from the started queue and placed on the participant's suspended queue. A participant must have the task privilege "Can Suspend" to successfully suspend a work item.
- **Reallocate Stateless**: This action allows a participant to reallocate a work item to another participant. The receiving participant must be subordinate to the reallocating participant by Position. The work item's data values are reset to the values that existed when the work item was first started (i.e. stateless reallocation), and it is moved from the started queue of the reallocator to the *started* queue of the receiver. A participant must have the task privilege "Can Reallocate Stateless" and have subordinate staff to successfully reallocate a work item.
- **Reallocate Stateful**: Similar to "Reallocate Stateless", except that any modified data values are maintained when the work item is reallocated. A participant must have the task privilege "Can Reallocate Stateful" and have subordinate staff to successfully reallocate a work item.
- **New Instance**: This action allows for the creation of a new instance of the selected work item; it is enabled only for a work item of a multiple instance atomic task that allows dynamic creation of additional work item instances.
- **Complete**: Completes the selected work item. The work item is posted back to the engine, which then progresses the case according to its control-flow. This action is initially disabled if the work item contains mandatory editable data variables (as in Figure 6.22), and becomes enabled after the first view/edit of the work item.

6.7.4 The Suspended Queue

The Suspended queue lists executing work items that have been suspended via the *Suspend* button on the Started queue. Note that suspended work items must have already been started and not yet completed, and so this queue may be seen as an extension of the started queue. This queue contains one action, **Unsuspend**, which resumes the work item, removing it from the suspended queue and returning it to the started queue.

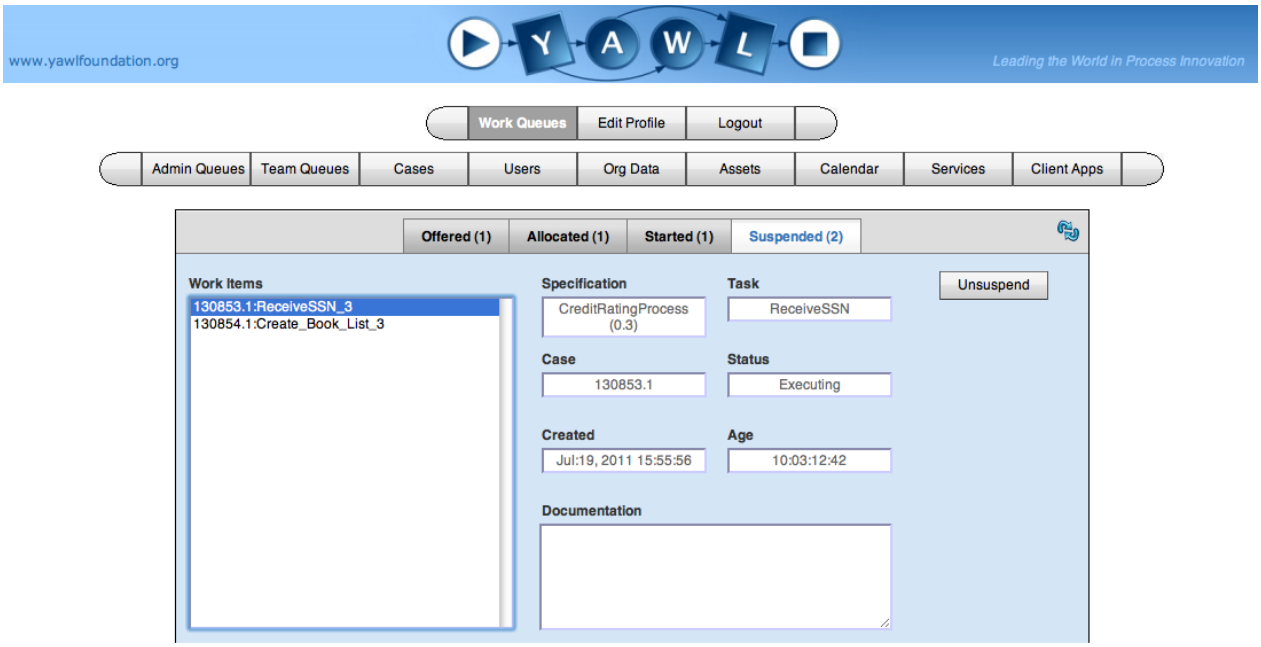


Figure 6.24: The Suspended Work Queue

6.8 User Profiles

The *Edit Profile* screen consists of four panels (Figure 6.25):

- A *Personal Information* panel, where a participant can see their name, userid, lists of their roles, positions and capabilities, and whether he/she has administrator privileges. All of this information is read-only.
- A *Change Password* panel, where the participant may change their password. Passwords must be at least 4 characters in length, and the 'New' and 'Confirm' entries must match.
- A *Piled Tasks* panel, where all of the tasks that are currently piled to the participant are listed. The participant may choose to cease the piling of a task to them by selecting it from the list and clicking the 'Unpile' button.
- A *Chained Cases* panel, where all of the cases currently chained to the participant are listed. The participant may choose to cease the chaining of a case to them by selecting it from the list and clicking the 'Unchain' button.

6.9 Team Queues

The *Team Queues* screen shows groups of active work items in a single list. There are two types of groupings possible:

- A participant who has been granted the user privilege 'View All Work Items of Team' may view a list of all the active work items that appear on the work queues of all the participants who are subordinate to them by Position. For example, in Figure 6.26 the participant listed holds the position 'CD clerk', which reports to the position of the currently logged on participant, 'Head of CD'.

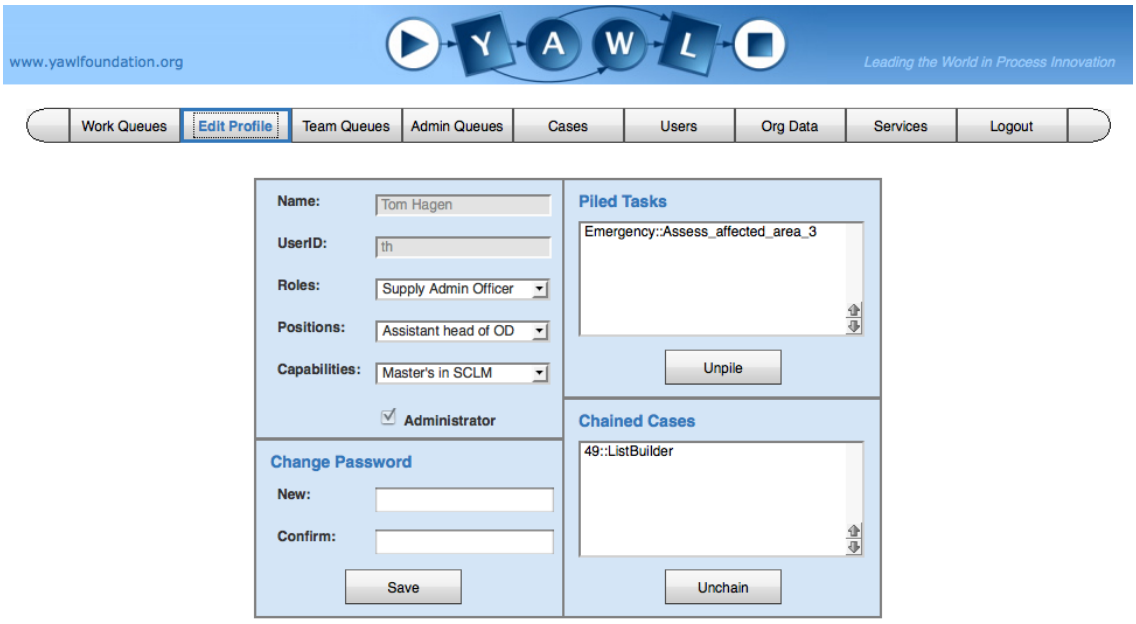


Figure 6.25: The Edit Profile Screen

Figure 6.26: The Team Queues Screen

- A participant who has been granted the user privilege 'View All Work Items of Org Group' may view a list of all the active work items that appear on the work queues of all the participants that are members of the same Org Group as the logged on participant.

All of the information displayed on this screen is read-only. A participant who has been granted both privileges may switch between views using the radio buttons to the right of the form; if they have been granted one of the two privileges, the other choice is disabled.

Chapter 7

The Monitor Service

The Monitor Service is a basic service that provides a summary view of all currently active cases within the YAWL engine. It consists of three screens that are viewed hierarchically – the case level, the workitem level and the parameter level.

7.1 Installation and Logging On

To install the Monitor Service:

1. Copy the file *monitorService.war* to your tomcat/webapps directory, and wait for it to fully unpack.
2. Go to the Client Applications web form, and register an account for the Monitor Service with these credentials:
   - **username**: monitorService
   - **password**: yMonitor
   - **description**: The YAWL Monitor Service

To log onto the Monitor Service, browse to: `http://localhost:8080/monitorService` (see Figure 7.1). The logon form has a green background to differentiate it from the Resource Service logon form. Any participant registered via the resource service that has administration privileges, or the generic 'admin' account (if enabled), can be used to logon to the Monitor Service.

7.2 Active Cases

The Active Cases form lists all of the currently executing case instances. Figure 7.2 shows an example.

In common with the other Monitor Service forms, the Active Cases form consists of a header, an information bar and a table. Inside the header are buttons (reading from the right) to refresh the page contents and to immediately logoff. In the table, clicking on a column heading will sort the contents on that column, alternating between ascending and descending order. The information bar for the Active Cases form shows a date and time of the last Engine (re)start.

The Active Cases table has columns for the case id, specification name, specification version and the data/-time the case began. A single click on any Active Case table row will show the work item detail for that case instance on the following form.

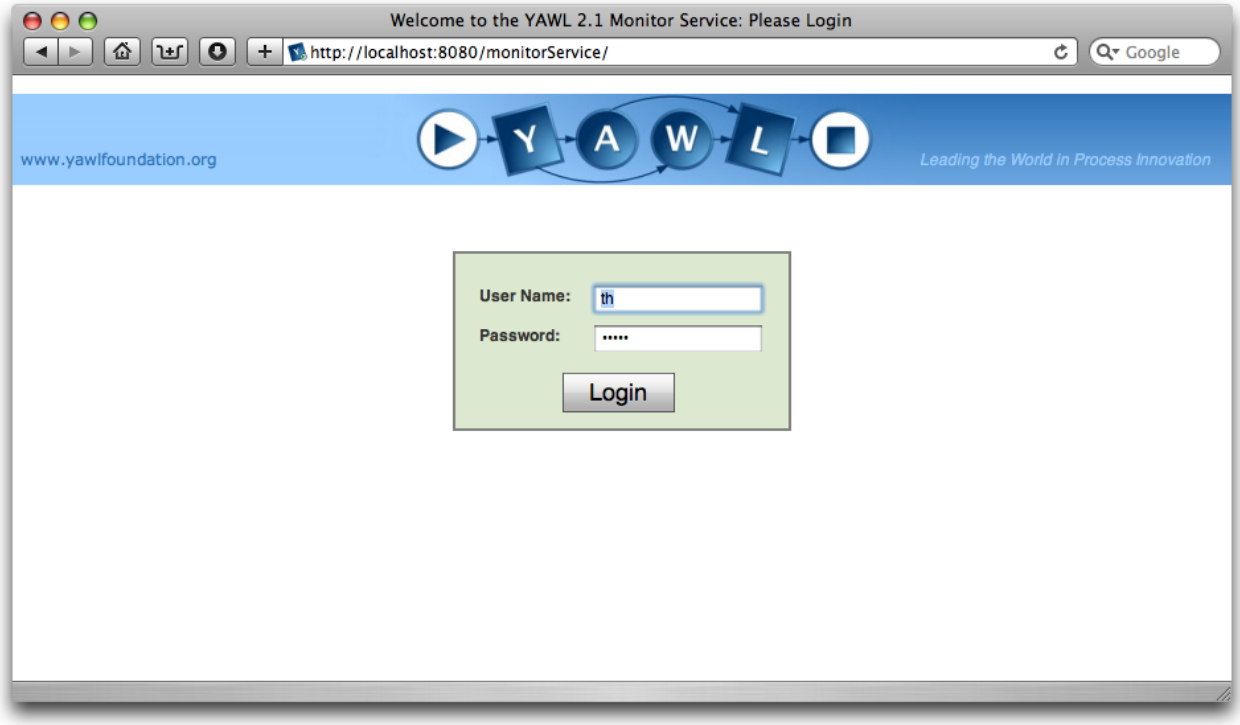
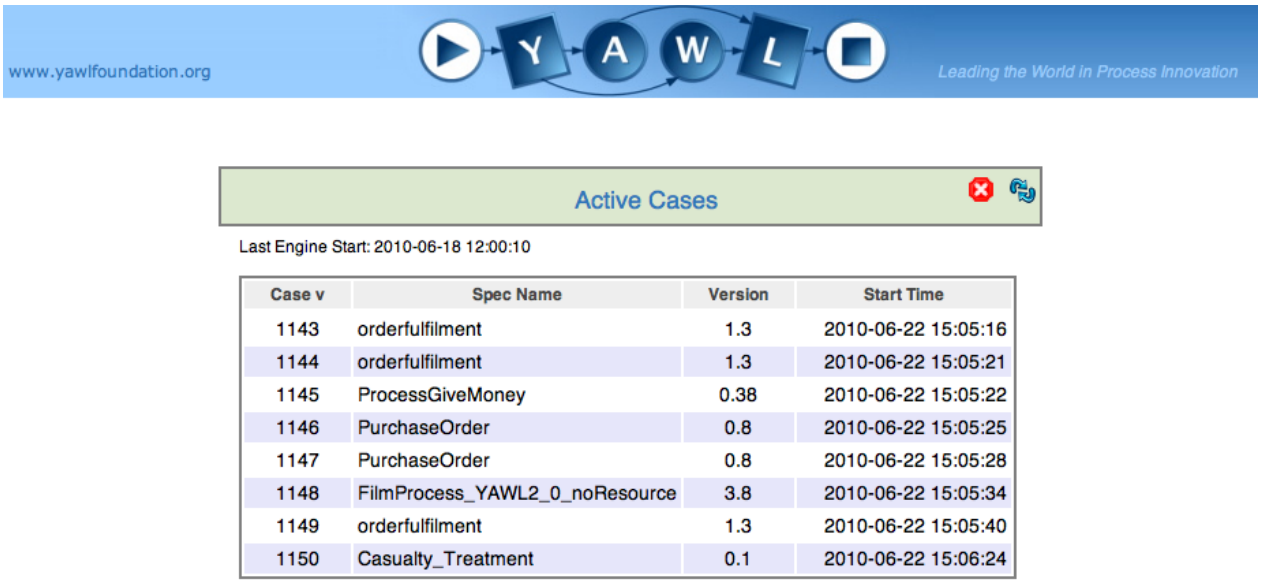


Figure 7.1: The Monitor Service logon screen



Case v	Spec Name	Version	Start Time
1143	orderfulfilment	1.3	2010-06-22 15:05:16
1144	orderfulfilment	1.3	2010-06-22 15:05:21
1145	ProcessGiveMoney	0.38	2010-06-22 15:05:22
1146	PurchaseOrder	0.8	2010-06-22 15:05:25
1147	PurchaseOrder	0.8	2010-06-22 15:05:28
1148	FilmProcess_YAWL2_0_noResource	3.8	2010-06-22 15:05:34
1149	orderfulfilment	1.3	2010-06-22 15:05:40
1150	Casualty_Treatment	0.1	2010-06-22 15:06:24

Figure 7.2: The Monitor Service Active Cases form

7.3 Work Items

Figure 7.3 shows an example of the Work Items of Selected Case form. The structure is similar to the Active Cases Form, but with the additional detail:

- a button on the left of the header bar for returning to the Active Cases form

- the information bar, containing the data/time of the case start, the service that started the workitem (in this example, the 'DefaultWorklist' – that is, the Resource Service), the participant who started the workitem, and the current case-level data (as XML).
- a table of workitems created by the case instance, with columns case id, task id, current status, starting service, enabled time, started time, completed time, timer status (if any) and timer expiry time (if applicable).

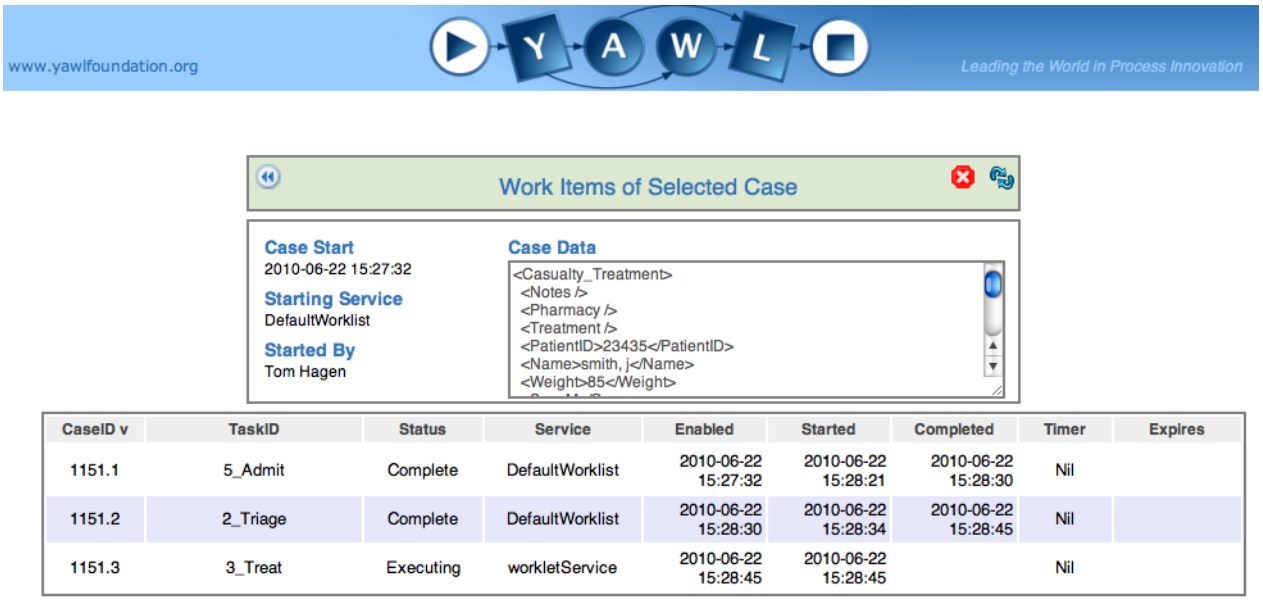


Figure 7.3: The Monitor Service Work Items of Selected Case form

Note that *all* work items for the case are listed, both currently active and those that have already completed. Also notice in this example that the first two tasks were started by the Resource Service, but the third by the Worklet Service (since that task is mapped to the worklet service at design time. A single click on a table row will show the associated parameters for that workitem.

7.4 Parameters

Figure 7.4 shows an example of the Parameters od Selected Work Item form:

- the information bar contains a list of logged events associated with this workitem from both the engine and the resource service (where applicable)
- the table shows all the parameters defined for the workitem, with columns name, data type, data schema (complex type definitions are listed here when used), the parameter's usage type (inputOnly, inputOutput, outputOnly), input and output mapping expressions (as applicable) and the input, default and last values.

Note that the current version of the monitor service does not support data persistence. That is, any information stored about completed items is lost if the Engine is restarted. This represents the initial release of the monitor service and it is envisaged that its functionality will grow in future versions.

Parameters of Selected Work Item

Engine Log Events

Resource Log Events

2010-06-22T15:28:21.563+1000	allocate	Tom Hagen
2010-06-22T15:28:22.261+1000	start	Tom Hagen
2010-06-22T15:28:30.200+1000	complete	Tom Hagen

Name v	DataType	DataSchema	Usage	Input Pred.	Output Pred.	Orig. Value	Def. Value	Value
Age	long	long	inputOutput	{number(/Casualty_Treatment/Age/text())}	{number(/Admit/Age/text())}	21		21
DiastolicBP	long	long	inputOutput	{number(/Casualty_Treatment/DiastolicBP/text())}	{number(/Admit/DiastolicBP/text())}	80		80
HeartRate	long	long	inputOutput	{number(/Casualty_Treatment/HeartRate/text())}	{number(/Admit/HeartRate/text())}	72		72
Height	double	double	inputOutput	{number(/Casualty_Treatment/Height/text())}	{number(/Admit/Height/text())}	1.8		1.8
Name	string	string	inputOnly	{/Casualty_Treatment/Name/text()}		smith, j		smith, j
PatientID	string	string	inputOnly	{/Casualty_Treatment/PatientID/text()}		23435		23435
Sex	string	string	inputOutput	{/Casualty_Treatment/Sex/text()}	{/Admit/Sex/text()}	M		M
SystolicBP	long	long	inputOutput	{number(/Casualty_Treatment/SystolicBP/text())}	{number(/Admit/SystolicBP/text())}	120		120
Weight	long	long	inputOutput	{number(/Casualty_Treatment/Weight/text())}	{number(/Admit/Weight/text())}	85		85

Figure 7.4: The Monitor Service Parameters of Selected Work Items form

Chapter 8

The Worklet Service

This chapter contains instructions for installing and using the *Worklet Dynamic Process Selection & Exception Handling Custom Service for YAWL*.

Each section describes one part in the process of setting up and using the Worklet Service. It is probably best to work through the chapter from start to finish the first time it is read. This chapter focuses on the practical use of the Worklet Service. For those interested, a more technical description of the inner operations of worklets and the rule sets that support them can be found in the YAWL Technical Manual. A concise version of the worklets approach can be found in [14], and of the exlets approach in [13]. The ultimate reference on worklets and exlets is Michael Adams' PhD thesis [12]. All these publications can be downloaded from `www.yawlfoundation.org`.

All of the example parent specifications, worklet specifications and rule sets referred to in this chapter can be found in the "samples" directory distributed with the service as part of the YAWL 4.0 (or later) release.

This icon indicates a hands-on method or instruction.

8.1 What is the Worklet Service?

An important point of extensibility of the YAWL system is its support for interconnecting external applications and services with the workflow execution engine, using a service-oriented approach. This enables running workflow instances and external applications to interact with each other in order to delegate work, to signal the creation of process instances and workitems, or to notify a certain event or a change of status of existing workitems.

Custom YAWL services are external applications that interact with the YAWL engine through XML/HTTP messages via certain endpoints, some located on the YAWL engine side and others on the service side. Custom YAWL services are registered with the YAWL engine by specifying their location, in the form of a "base URL". Once registered, a custom service may send and receive XML messages to and from the engine. More specifically, Custom YAWL services are able to check-out and check-in workitems from the YAWL engine. They receive a message when an item is enabled, and therefore can be checked out. When the Custom YAWL service is finished with the item it can check it back in, in which case the engine will set the work item as completed, and proceed with the execution.

The *Worklet Dynamic Process Selection & Exception Handling Service* for YAWL comprises two distinct but complementary services: a Selection Service, which enables dynamic flexibility for YAWL process instances; and an Exception Handling Service, which provides facilities to handle both expected and unexpected process exceptions (i.e. events and occurrences that may happen during the life of a process instance that are not explicitly modelled within the process) at runtime. A brief introduction to each Service follows.

8.1.1 The Selection Service

The Worklet Dynamic Process Selection Service (or *Selection Service*) enables flexibility by providing a process designer with the ability to designate a workitem in a YAWL process to be substituted at runtime with a dynamically selected "worklet" – a discrete YAWL process that acts as a sub-net in place of the workitem and so handles one specific task in a larger, composite process activity. The worklet is dynamically selected and invoked, and may be created at any time, unlike a static sub-process that must be defined at the same time as, and remains a static part of, the main process model.

An extensible repertoire (or catalogue) of worklets is maintained by the Service. Each time the Service is invoked for a workitem, a choice is made from the repertoire based on the contextual data values within the workitem, using an extensible set of rules to determine the most appropriate substitution.

The workitem is checked out of the YAWL engine, and then the selected worklet is launched as a separate case. The data inputs of the original workitem are mapped to the inputs of the worklet. When the worklet has completed, its output data is mapped back to the original workitem, which is then checked back into the engine, allowing the original process to continue. Worklets can be substituted for atomic tasks and multiple-instance atomic tasks. In the case of multiple-instance tasks, a worklet is launched for each child workitem. Because each child workitem may contain different data, the worklets that substitute for them are individually selected, and so may all be different.

The repertoire of worklets can be added to at any time, as can the rules set used for the selection process. Thus the service provides for dynamic ad-hoc change and process evolution, without having to resort to off-system intervention and/or system downtime, or modification of the original process specification.

8.1.2 The Exception Service

During almost every instance of a workflow process, certain things happen "off-plan". That is, regardless of how much detail has been built into the process model, certain events occur during execution that affect the work being carried out, but were not defined as part of the process model. Typically, these events are handled "off-system" so that processing may continue. In some cases, the process model will be modified to capture this unforeseen event in future instances, which involves an organisational cost (downtime, remodelling, testing and so on).

The Worklet Dynamic Exception Handling Service (or *Exception Service*) provides the ability to handle these events in a number of ways and have the process continue unhindered. Additionally, once an unexpected exception is handled a certain way, that method automatically becomes an implicit part of the process specification for all future instances of the process, which provides for continuous evolution of the process, while avoiding the need to modify the original process definition.

The Exception Service uses the same repertoire of worklets and dynamic rules approach as the Selection Service. The difference is that, while the Selection Service is invoked for certain tasks in a YAWL process, the Exception Service, when enabled, is invoked for every case and task executed by the YAWL engine, and will detect and handle up to ten different kinds of process exception. As part of the handling process, a process designer may choose from various actions (such as cancelling, suspending, restarting and so on) and apply them at a workitem, case and/or specification level.

The Exception Service is extremely flexible and multi-faceted, and allows a designer to provide tailor-made solutions to runtime process exceptions, as described in the following pages.

8.2 Installation

8.2.1 Worklet Installation Package

The Worklet Service is distributed as a default component of the YAWL environment, and so is included in each of the various installer packages, and the **CoreWebServices.zip** file used for manual installations.

8.2.2 Configuring the Worklet Service

Manual Installs Only: The *workletService.war* file should be located in the *webapps* directory of your Tomcat installation (if necessary, refer to Chapter 2, Section 2.4 for more information). Then, the file needs to be extracted to its own directory under *webapps*. The easiest way to achieve this is to simply start Tomcat - it will automatically extract, install and start the Worklet Service.

Disabling Exception Handling

Exception handling is enabled by default. To disable it, a parameter has to be set in its *web.xml* file.

Open the Worklet Service's *web.xml* file (in folder *\webapps\workletService\WEB-INF*). Locate the parameter named *EnableExceptionHandling*. To disable exception handling, change the *param-value* to **false** (see Figure 8.1). Save and close *web.xml*.

```
<context-param>
    <param-name>EnableExceptionHandling</param-name>
    <param-value>false</param-value>
    <description>
        'true' to enable exception handling functionality
        'false' to disable
    </description>
</context-param>
```

Figure 8.1: The Worklet Service's *web.xml* file (detail)

Manual Installs Only: If the Resource and Worklet Services are installed on different servers, you will also need to amend the *ResourceServiceURL* context parameter in the Worklet Service's *web.xml* to refer to the correct location.

Enabling the Worklist Extensions (optional)

The Worklet Service uses extensions (or 'hooks') in the YAWL default worklist handler (a component of the Resource Service) to provide some web pages, which allow an administrator to raise 'external' exceptions and to reject a worklet selection (see later in this chapter for more details).

To enable those extensions, locate and open the Resource Service's *web.xml* file, which is located in the folder *\webapps\resourceService\WEB-INF*.

Locate the context parameter named *InterfaceX_BackEnd*. By default, the entire parameter block is commented out. Simply remove the comment tags (the `<!-- and -->` surrounding the *context-param* block – see Figure 8.2). Save and close *web.xml*.

8.3 The Worklet Service and Dynamic Flexibility

Fundamentally, a worklet is nothing more than a YAWL process specification that has been designed to perform one part of a larger, or 'parent', specification. However, it differs from a decomposition or sub-net in that it is dynamically assigned to perform a particular task at runtime, while sub-nets are statically assigned at design time. Also, worklets can be added to the repertoire at any time during the life of a specification, even while instances are running. So, rather than being forced to define all possible "branches" in a specification when it is first defined, the Worklet Service allows you to define a much simpler specification that will

```
<!-- This param, when available, enables the worklet exception
     service add-ins to the worklist. If the exception service
     is enabled in the engine, then this param should also be
     made available. If it is disabled in the engine, the
     entire param should be commented out. -->
<!--
<context-param>
    <param-name>InterfaceX_BackEnd</param-name>
    <param-value>http://localhost:8080/workletService</param-value>
    <description>
        The URL location of the worklet exception service.
    </description>
</context-param>
-->
```

Figure 8.2: The Resource Service's *web.xml* file (detail)

evolve dynamically as more worklets are added to the repertoire for a particular task as different contexts arise.

The first thing you need to do to make use of the service is to create a number of YAWL specifications – one that will represent the parent (top-level) specification, and one or more worklets that will be dynamically substituted for particular parent tasks at runtime.

The YAWL Editor is used to create both parent and worklet specifications. A knowledge of creating and editing YAWL specifications, and the definition of data variables and parameters for tasks and specifications, is assumed. For more information on how to use the YAWL Editor, see Chapter 4.

Before opening the YAWL Editor, make sure that the Worklet Service is correctly installed and that Tomcat is running (see Section 8.2 of this chapter and/or Chapter 2 for more information).

8.3.1 Designing a Worklet-enabled Parent Specification

To define a parent specification, open the YAWL Editor and create a process specification in the usual manner. Choose one or more tasks in the specification that you want to have replaced with a worklet at runtime. Each of those tasks needs to be associated via the YAWL Editor with the Worklet Service.

Worklets may be associated with any atomic task, or any multiple-instance atomic task. Any number of worklets can be associated with (i.e. comprise the repertoire of) an individual task, and a particular worklet may be a member of any number of repertoires. Any number of tasks in a particular specification can be associated with the Worklet Service.

For example, Figure 8.3 shows a simple specification for a *Casualty Treatment* process.[1] In this process, we want the *Treat* task to be substituted at runtime with the appropriate worklet based on the patient data collected via the *Admit* and *Triage* tasks. That is, depending on each patient's actual physical data and reported symptoms, we would like to run the worklet that best handles the patient's condition.

Here, we want to associate the *Treat* task with the Worklet Service. To do so, in the Editor choose the *Custom Service* property in the Decomposition section of the Properties Pane, then select *Worklet Service* from the dropdown list. That's all that is required to make the top-level specification worklet-enabled.

The list of task-level variables for the *Treat* task, from a section of the Data Variables dialog in the Editor, is shown (Figure 8.4). Each task-level variable maps to a net-level variable, so that in this example all of the data collected from a patient in the first two tasks are made available to the *Treat* task. The result is that

[1] This and all other specifications referred to in this chapter are available in the *samples* folder of the Worklet Service installation.

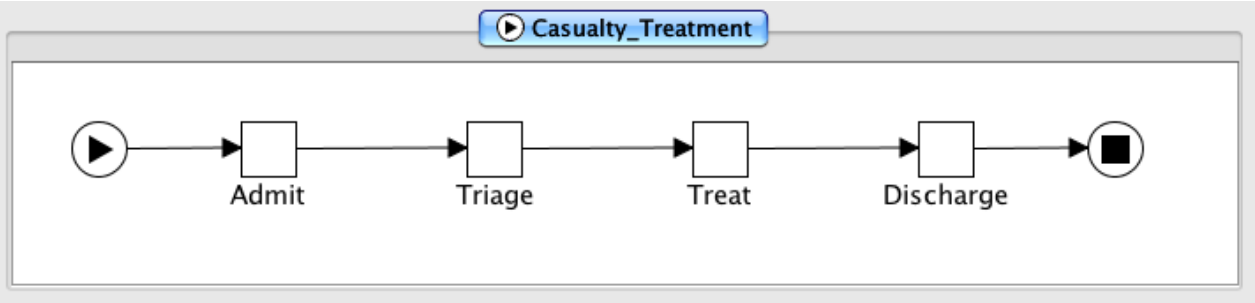


Figure 8.3: Example Top-level Specification

all of the relevant current case data for this process instance can be used by the Worklet Service to enable a contextual decision to be made. Note that it is not necessary to map all available case data to a worklet enabled task, only that data required by the Service to make an appropriate decision. How this data is used will be discussed later in this chapter.

Decomposition Variables

Name	Type	Scope	Default Value
PatientID	string	Input	###
Sex	string	Input	###
DiastolicBP	long	Input	###
Notes	string	InputOutput	###
Height	double	Input	###
HeartRate	long	Input	###
Pharmacy	string	InputOutput	###
SystolicBP	long	Input	###
Fracture	boolean	Input	###
Age	long	Input	###
Treatment	string	InputOutput	###
Weight	double	Input	###
Fever	boolean	Input	###
Rash	boolean	Input	###
Wound	boolean	Input	###
Name	string	Input	###
AbdominalPain	boolean	Input	###

Figure 8.4: Decomposition variables for the 'Treat' task

The list of task variables in Figure 8.4 also show that most variables are defined as 'Input Only' – this is because those values will not be changed by any of the worklets that may be executed for this task; they will only be used in the selection process. Three variables are defined as 'InputOutput' so that the worklet can "return", or map back to these variables, data values that are captured during the worklet's execution.

Next, we need to create one or more worklet specifications to execute as substitutes for the worklet-enabled task.

8.3.2 Designing Worklet Specifications

When the parent *Casualty Treatment* specification is executed, the YAWL Engine will notify the Worklet Service when the worklet-enabled *Treat* task becomes ready for execution. The Worklet Service will then examine the data in the task and use it to determine which worklet to execute as a substitute for the task. Any or all of the data in the task may also be mapped to the selected worklet case as input data. Once the worklet instance has completed, any or all of the available output data of the worklet case may be mapped back to the Treat task to become its output data, and the parent process will continue.

A worklet specification is a standard YAWL process specification, and as such is created in the YAWL Editor in the usual manner. Figure 8.5 shows a simple example worklet to be substituted for the *Treat* top-level task when a patient complains of a fever.

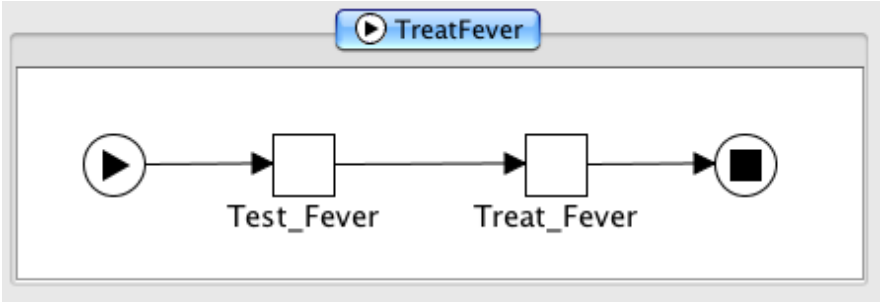


Figure 8.5: The *TreatFever* Worklet

In itself, there is nothing special about the *TreatFever* specification. Even though it will be considered by the Worklet Service as a member of the worklet repertoire and may thus be considered a "worklet", it remains a standard YAWL specification and as such may be executed independently directly by the YAWL engine without any reference to the Worklet Service.

The data variables that are required to be passed from the parent task to the worklet specification need to be defined as net-level input variables in the worklet specification. Figure 8.6 shows the net-level variables for the *TreatFever* worklet specification.

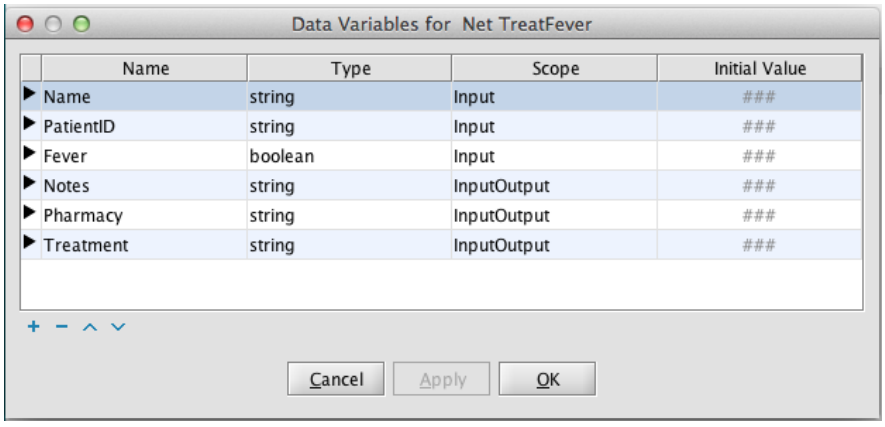


Figure 8.6: Net-level Variables for the *TreatFever* Specification

Note the following:

- Only a sub-set of the variables defined in the parent *Treat* task (see Figure 8.4) are defined here. It is only necessary to map from the parent task those variables that contain values to be displayed to the user, and/or those variables that the user will supply values for to be passed back to the parent task when the worklet completes.

- The definition of variables is not restricted to those defined in the parent task. Any additional variables required for the operation of the worklet may also be defined here.
- Only those variables that have been defined with an identical name and data type to variables in the parent task and with a *Scope* of 'Input Only' or 'Input & Output' will have data passed into them from the parent task when the worklet is launched.
- Only those variables that have been defined with an identical name and data type to variables in the parent task and with a *Scope* of 'Output Only' or 'Input & Output' will pass their data values back to the parent task when the worklet completes.

In Figure 8.6, it can be seen that the values for the *PatientID*, *Name* and *Fever* variables will be used by the *TreatFever* worklet as display-only values, while the *Notes*, *Pharmacy* and *Treatment* variables will receive values during the execution of the worklet and will map those values back to the parent *Treat* task when the worklet completes.

The association of tasks with the Worklet Service is not restricted to parent specifications. Worklet specifications also may contain tasks that are associated with the Worklet Service and so may have worklets substituted for them, so that a hierarchy of executing worklets may sometimes exist. It is also possible to recursively define worklet substitutions - that is, a worklet may contain a task that, while certain conditions hold true, is substituted by another instance of the same worklet specification that contains the task.

As mentioned previously, any number of worklets can be created for a particular task. For the *Casualty Treatment* example, there are (initially) five worklets in the repertoire for the *Treat* task, one for each of the five primary conditions that a patient may present with in the *Triage* task: Fever, Rash, Fracture, Wound and Abdominal Pain. Which worklet is chosen for the *Treat* task depends on which of the five is given a value of *true* in the *Triage* task.

How the Worklet Service uses case data to determine the appropriate worklet to execute is described in Section 8.6.

8.4 The Worklet Service and Exception Handling

In the previous section, we saw how the Worklet Service adds dynamic flexibility to a usually static YAWL specification by substituting tasks with contextually chosen worklets at runtime. The Worklet Service uses the same framework to also provide support for the myriad exceptions that may occur during the execution of any process instance.

Almost every process instance, no matter how rigidly structured the business process, will experience some kind of exception during its execution. While the word 'exception' conjures up ideas of errors or problems occurring within the executing process instance, the meaning in terms of workflow processes is much broader: exceptions are merely events or occurrences that, for one reason or another, were not defined in the process model. It may be that these events are known to occur in a small number of cases, but not enough to warrant their inclusion in the process model, or they may be things that were never expected to occur (or maybe never even imagined could occur). In any case, when they do happen, if they are not part of the process model, without a system that supports exception handling, they must either be handled "off-line" before the process continues (and the way they are handled is rarely recorded) or in some instances the entire process must be cancelled.

Alternately, an attempt might be made to include every possible twist and turn into the process model so that when such events occur, there is a branch in the process to take care of it. This approach may lead to very complex models where much of the original business logic is obscured, and doesn't avoid the same problems occurring when the next unexpected exception occurs.

The Worklet Service addresses these problems by allowing you to define exception handling processes, which may include worklets as compensation handlers, for workflow instances when certain events occur. Rules are defined in much the same way as for dynamic flexibility, but with added features that enable you to pause, resume, cancel or restart the task, case, or all cases of a specification, that triggered the exception.

Because the service allows you to define exception handlers for all exception events, and even to add new handlers at runtime, all exception events are able to be captured "on-system", so that the handlers are available to all future occurrences of a particular event for the same context. And, since the handlers are worklets, the original parent process model only needs to contain the actual business logic for the process, while the repertoire of handlers grows as new exceptions arise or different ways of handling exceptions are formulated.

IMPORTANT: While for dynamic flexibility the Worklet Service is linked explicitly to a task via the YAWL Editor, and thus available whenever a worklet-enabled task is executed, exception handling is either enabled (the default) or disabled (see Section 8.2). When it is enabled, it manages exception handling for **all** process instances executed by the engine – explicitly linking a task or process to the service is not required.

8.4.1 Exception Types

This section introduces the ten different types of exception that are supported by the Worklet Service. Some are related, while others are more discrete. Later sections will show examples of each of these.

When exception handling is enabled, the Worklet Service is notified whenever any of these exception types occur for every process instance executed by the YAWL Engine. The Exception Service maintains a set of rules (described in detail in Section 8.6) that are used to determine which exception handling process, if any, to invoke. If there are no rules defined for a certain exception type for a specification, the exception event is simply ignored by the service. Thus you only need to define rules for those exception events that you actually want to handle for a particular specification.

Constraint Types

Constraints are rules which are applied to a workitem or case immediately before and after their execution. Thus, there are four types of constraint exception:

- **CasePreConstraint** - case-level pre-constraint rules are checked when each case (i.e. process instance) begins execution;
- **ItemPreConstraint** - item-level pre-constraint rules are checked when each workitem in a case becomes enabled (i.e. ready to be started);
- **ItemPostConstraint** - item-level post-constraint rules are checked when each workitem moves to a completed status; and
- **CasePostConstraint** - case-level post constraint rules are checked when a case completes.

The Worklet Service receives notification from the YAWL Engine when each of these events occur, then checks the rule set for the specification to determine, firstly, if there are any rules of that type defined for the case, and if so, if any of the rules evaluate to true using the contextual data of the case or workitem. If the rule set finds a matching rule for the exception type and data context, an exception handling process is invoked.

Note that for each of the constraint events, an exception process is invoked for a rule when that rule's condition evaluates to **true**. So, for example, if the condition of an *ItemPreConstraint* rule for a *Triage* task was "PrivateInsurance=true", and that value of that variable in the workitem was also true, then the exception process defined for that rule would be invoked.

Externally Triggered Types

Externally triggered exceptions occur, not through the case's data values, but because something has happened outside of the process execution that has an affect on the continuing execution of the process. Thus, these events are triggered by a user; depending on the actual event, a particular handler will be invoked.

There are two types of external exceptions, **CaseExternalTrigger** (for case-level events) and **ItemExternalTrigger** (for item-level events). See later in this section for examples of each and how they are invoked.

TimeOut

A timeout event occurs when a workitem has an associated timer and the deadline set for that workitem is reached. In this case, the Engine notifies the Worklet Service of the timeout event, and passes to the service a reference to the timed-out workitem and each of the other workitems that were running in parallel with the timed-out workitem. Therefore, timeout rules may be defined for each of the workitems affected by the timeout (including the actual time-expired workitem itself).

ResourceUnavailable

This event occurs when an attempt has been made to allocate a workitem to a resource and the Resource Service reports that the resource is unavailable to accept the allocation.

ItemAbort

An ItemAbort event occurs when a workitem being handled by an external service or application (as opposed to a human user) reports that the program has aborted before completion of the workitem. Thus, these events are triggered externally by the delegated service or application.

ConstraintViolation

This event occurs when a data constraint has been violated for a workitem during execution (as opposed to pre or post execution). These events are triggered externally by the service or application that has been delegated execution of the workitem.

8.4.2 Exception Handling Primitives

For any exception event that occurs, a handling process may be invoked. Each handling process, also called an *exlet*, contains a number of steps, or *primitives*, in sequence, and may be defined graphically, or via directed choices, using the Worklet Editor plugin (see Section 8.6). Each of the handling primitives is introduced below.

 **Suspend WorkItem** - suspends (or pauses) execution of a workitem, until it is continued, restarted, cancelled, failed or completed, or its parent case is cancelled or completed.

 **Suspend Case** - suspends all "live" workitems in the current case instance (a live workitem has a status of fired, enabled or executing), effectively suspending execution of the case.

 **Suspend All Cases** - suspends all "live" workitems in all of the currently executing instances of the specification in which the workitem is defined, effectively suspending all running cases of the specification.

 **Continue Workitem** - continues (i.e. un-suspends) execution of the previously suspended workitem.

 **Continue Case** - un-suspends execution of all previously suspended workitems for the case, effectively continuing case execution.

Continue All Cases - un-suspends execution of all previously suspended workitems for all cases of the specification in which the workitem is defined or of which the case is an instance, effectively continuing all running cases of the specification.

Remove Workitem - removes (or cancels) the workitem; execution ends, and the workitem is marked with a status of cancelled. No further execution occurs on the process path that contains the workitem.

Remove Case - removes (cancels) the case. Case execution ends.

Remove All Cases - removes (cancels) all case instances for the specification in which the workitem is defined, or of which the case is an instance.

Restart Workitem - rewinds an executing workitem back to its initial state. That is, it resets the workitem's data values to those it had when it began execution.

Force Complete WorkItem - completes a "live" workitem. Execution of the workitem ends, and the workitem is marked with a status of ForcedComplete, which is regarded as a successful completion, rather than a cancellation or failure. Execution proceeds to the next workitem on the process path.

Force Fail Workitem - fails a "live" workitem. Execution of the workitem ends, and the workitem is marked with a status of Failed, which is regarded as an unsuccessful completion, but not a cancellation - execution proceeds to the next workitem on the process path.

Compensate - run a compensatory process (i.e. a worklet). Depending on previous primitives, the worklet may execute simultaneously to the parent case, or execute while the parent is suspended (or even removed).

Figure 8.7 shows an example of the graphical definition of an exception handing process. When invoked, this handler will suspend the current case, then run a compensating worklet, then continue execution of the case.

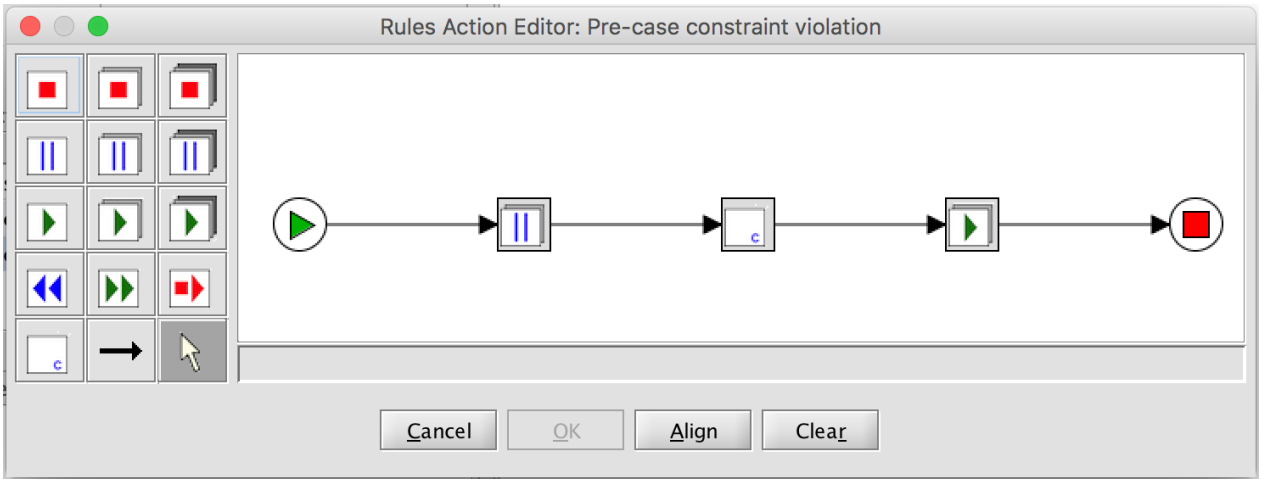


Figure 8.7: Example Exlet in the Worklet Rules Editor

Notes regarding exlet definitions:

- The primitives *Suspend All Cases*, *Continue All Cases* and *Remove All Cases* may be edited so that their action is restricted to ancestor cases only. Ancestor cases are those in a hierarchy of worklets back to the parent case (that is, where a case invokes a worklet which invokes another worklet and so on), including the parent case itself.
- the workitem-level primitives cannot be used for a case-level exception handler (*CasePreConstraint*, *CasePostConstraint*, *CaseExternalTrigger*).
- The *Remove*, *Restart*, *Force Complete*, and *Force Fail* primitives cannot be applied to a post-constraint trigger, because such triggers only occur when a case or work item has completed, and therefore no longer exist. Thus, for post-constraint triggers, only the *Compensate* primitive is applicable for case-level exceptions, while only *Compensate*, and *Suspend* and *Continue* for the case or all cases, are applicable for workitem level exceptions.

In the same manner as the Selection Service, the Exception Service also supports data mapping from a case to a compensatory worklet and back again. For example, if a certain variable has a value that prevents a case instance from continuing, a worklet can be run as a compensation, during which a new value can be assigned to the variable and that new value mapped back to the parent case, so that it may continue execution.

The full capabilities of the Exception Service are better described in the walkthroughs in Section 8.7. But before we consider the walkthroughs, we must first look at exactly how the rule sets are formed and how they operate, and then how to use the Worklet Rules Editor to manage rule sets for specifications. These topics are discussed in the next sections.

8.5 Worklet Rule Sets

A 'plugin' for the YAWL Editor, called the *Worklet Management* plugin, but informally referred to as the *Rules Editor*, manages the creation and modification of worklet rule sets for specifications. A worklet rule set is basically a set of linked rules that are evaluated by the Worklet Service to determine when a worklet should be launched, or an exception raised.

This section describes the structure and operation of worklet rule sets, while the next section shows how to use the Rules Editor to display and manipulate them.

It is not necessary to have a full understanding of the structure of a rule set, nor the search and evaluation algorithms used on them. The information presented here is intended more as a way of better understanding the operations of the Rules Editor.

Again, the Worklet Selection and Exception Services work in very similar ways, but with some necessary differences. In this section, the discussion of rule sets applies to both services, except where specified.

* * *

Any YAWL specification may have an associated rule set maintained for it by the Worklet Service. A rule set for a specification consists of a collection of *rule trees*. Each rule tree maintains a rule node hierarchy in a binary-tree structure. When a rule tree is queried, it is traversed from the root node of the tree along the branches, each node having its condition evaluated along the way. If a node's condition evaluates to *true*, and it has a true child, then that child node's condition is also evaluated. If a node's condition evaluates to *false*, and there is a false child, then that child node's condition is evaluated. When a terminal node is reached (i.e. a node without any child nodes), if its condition evaluates to *true*, then that conclusion is returned as the result of the tree traversal; if it evaluates to *false*, then the last node in the traversal that evaluated to *true* is returned as the result. The root node of the tree is always a default node with a default *true* condition, and so can only have a *true* branch.

Effectively, each rule node on the true branch of its parent node is an exception rule to the more general one of its parent (that is, a *refinement* of the parent rule), while each rule node on the false branch of its

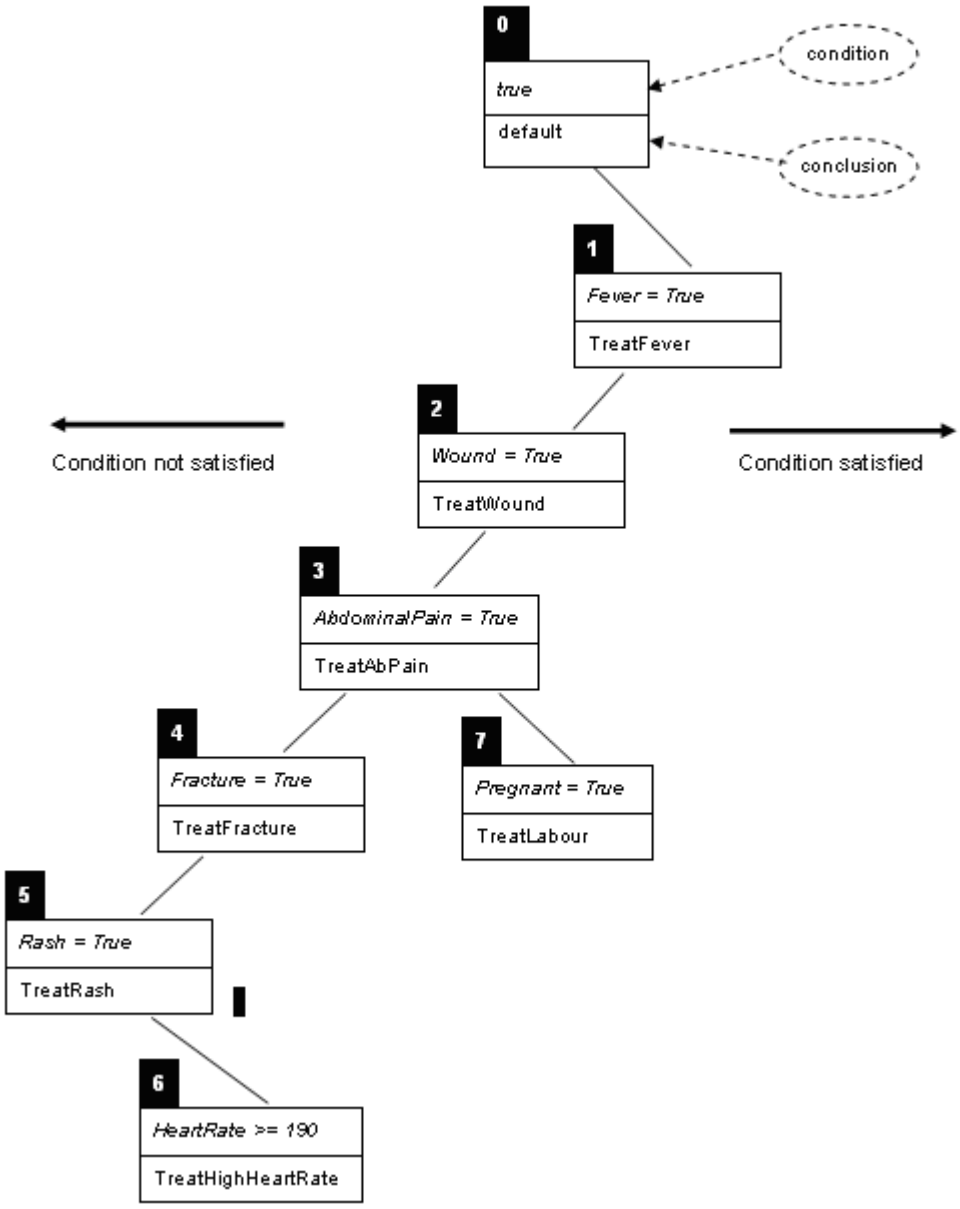


Figure 8.8: Example Rule Tree (Casualty Treatment spec)

parent node is an "else" rule to its parent (or an *alternate* to the parent rule). Referring to the selection rule tree for the *Casualty Treatment* specification (Figure 8.8) as an example, the condition part is the rule that is evaluated, and the conclusion is the name of the worklet selected by that rule if the condition evaluates to true. For example, if the condition "Fever = true" evaluates to true, then the *TreatFever* worklet is selected (via node 1); if it is false, then the next false node is tested (node 2). If node 2 is also false, then node 3 is tested. If node 3 evaluates to true, then the *TreatAbPain* worklet is selected, ***except if*** the condition in its next true node (node 7) also evaluates to true, in which case the *TreatLabour* worklet is selected.

Each rule set is associated with one specification, and may contain up to eleven sets of rule trees (or *tree sets*), one for selection rules (i.e. for dynamic flexibility) and one for each of the ten exception types. Three of the eleven relate to case-level exceptions (i.e. *CasePreConstraint*, *CasePostConstraint* and *CaseExternalTrigger*) and

so each of these will have only one rule tree in the tree set. The other eight tree sets relate to the workitem-level (seven exception types plus selection), and so may have one rule tree for each task in the specification - that is, the tree sets for these eight rule types may consist of a number of rule trees.

It is not necessary to define rules for all eleven rule types for each specification. You only need to define rules for those types that you want to handle - any exception types that aren't defined in the rule set file are simply ignored. So, for example, if you are only interested in capturing pre and post constraints at the workitem level, then only the *ItemPreConstraint* and *ItemPostConstraint* tree sets need to be defined (i.e. rules defined within those tree sets). In this example, any *Timeout* exception events that occur during the execution of the specification would be ignored by the Exception Service. Of course, rules for a *Timeout* event could be added later if required (as could any of the other types not yet defined in the rule set).

To summarise the hierarchy of a rule set (from the bottom up):

- **Rule Node:** contains the details (condition, conclusion, id, parent and so on) of one discrete "ripple-down" rule.
- **Rule Tree:** consists of a number of rule nodes in a binary tree structure.
- **Tree Set:** a set of one or more rule trees. Each tree set is specific to a particular rule type (timeout, selection, etc.). The tree set of a case-level exception rule type will contain exactly one rule tree. The tree set of an item-level rule type will contain one rule tree for each task of the specification that has rules defined for it (not all tasks in the specification need to have a rule tree defined).
- **Rule Set:** a set of one or more tree sets representing the entire set of rules defined for a specification. Each rule set is specific to a particular specification (regardless of version). A rule set will contain one or more tree sets – one for each rule type for which rules have been defined.

This background information provides an understanding of the structure or rule sets, and sits as a lead-in to the next section. For those interested, more detailed information is available via the references provided at the beginning of this chapter.

8.6 The Worklet Management Plugin (or Rules Editor)

The Worklet Management Plugin for the YAWL Editor provides facilities for:

- Saving a worklet specification displayed in the YAWL Editor directly to the repertoire of worklets available for use by parent specifications.
- Retrieving a worklet specification from the Worklet Service directly into the YAWL Editor for editing.
- Creating, browsing, exporting and removing rule sets and their rule nodes.
- Rejecting and replacing launched worklets with others following a rule tree update.
- Bulk uploading of worklet specifications and legacy rule sets stored as XML representations in files.

The plugin can be accessed via the YAWL Editor's Plugins menu (Figure 8.9). There is also a toolbar (Figure 8.10) that performs the menu actions, which can be displayed via the *Plugins* → *Toolbars* menu.

Configuration The plugin has configuration settings where the location of the Worklet Service, and the credentials for logging on to the service, can be amended (Figure 8.11). These settings are similar to those required by the Editor for connecting to the Engine and to the Resource Service (see Section 4.10 of Chapter 4 for more details). In most cases the default settings need not be changed.

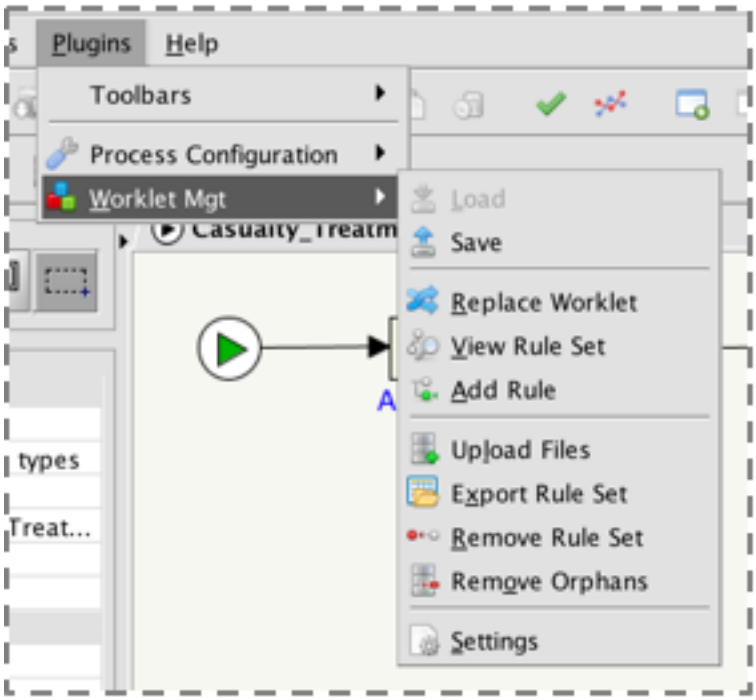


Figure 8.9: Plugin Menu (detail)

Figure 8.10: Plugin Toolbar

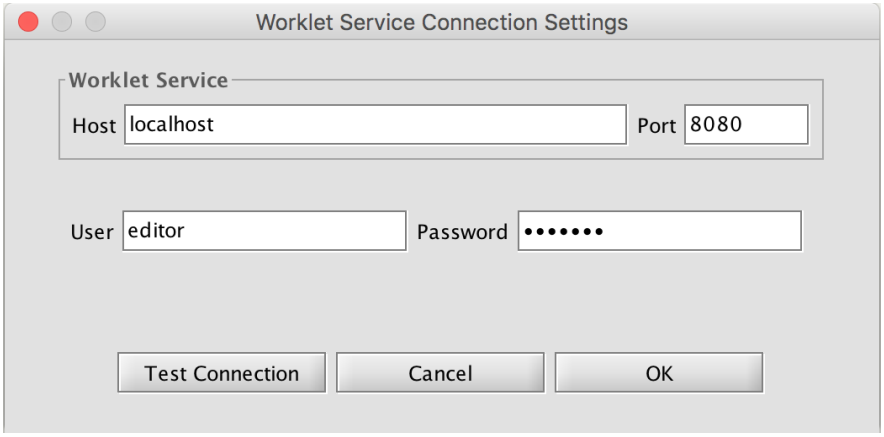


Figure 8.11: The Rules Editor Settings Dialog

8.6.1 Loading and Saving Worklets

The Worklet Service maintains a set of worklets, which it provides for inclusion in rules (as selections or compensations), and invokes at runtime as needed.

To **save** a worklet to the set, first create it or load it from disk into the Editor as you would any YAWL specification. Ensure the Worklet Service is running at the configured location, then either click the *Plugins*

→ *Worklet Mgt* → *Save* menu item, or click the *Save* button on the worket management toolbar.

If there is an earlier version of the same worklet specification already existing in the set, it will be overwritten with the newer version.

To **load** a worklet from the set into the Editor, first ensure there is no specification currently loaded in the Editor. Ensure the Worklet Service is running at the configured location, then either click the *Plugins* → *Worklet Mgt* → *Load* menu item, or click the *Load* button on the worket management toolbar.

The worklet specification will be loaded into the Editor (but is not removed from the Worklet Service). It can then be edited and re-saved to the worklet set (as above), as required.

8.6.2 Creating a New Rule Set and/or Adding a New Rule

As mentioned previously, it is not necessary to create tree sets for all of the rule types, nor a rule tree for an item-level rule type for every task in a specification. Most typically, rule sets may have rules defined for a few rule types, with some rules and/or tasks left undefined (remember that any events that don't have associated rules for that type of event are simply ignored).

It follows that there will be occasions when you want to add a new tree set to a rule set for a previously undefined rule type, or add a new tree for a previously undefined task to an existing tree set, or add a new rule to an existing rule tree. Also, when a new specification has been created, you may also want to create a corresponding base rule set (if you want to handle selections and exceptions for the new specification).

For each of these situations, the Rules Editor provides a *Add Rule* dialog, which allows for:

- the definition of new rule nodes for existing rule trees;
- the definition of new rule trees (with any number of rule nodes) for existing tree sets (where there is a task of the specification that has not yet had a tree defined for it within the tree set);
- the definition of new tree sets for specifications that have not yet had a tree set defined for a particular rule type; and
- entirely new rule sets for specifications that have not yet had a rule set created for them.

IMPORTANT All added rules will be associated with the (parent) specification currently loaded in the Editor.

To begin adding a new rule for the current specification, first ensure the Worklet Service is running at the configured location, then either click the *Plugins* → *Worklet Mgt* → *Add Rule* menu item, or click the *Add Rule* toolbar button. The *Add Rule* dialog will be displayed (Figure 8.12).

On the *Add Rule* dialog:

- The *Rule Type* drop-down list allows you to select the desired rule type (selection and the ten exception types) to add the new rule for.
- The *Task Name* drop-down list shows the names of the all atomic tasks for the currently loaded specification, from which the desired task can be selected. The *Task Name* list is disabled for case-level rule types.
- The *Condition* field is where the new rule's condition is added (see Section 8.6.3 below for details on rule conditions).
- The *Actions* table is where action 'primitives' can be defined (see Section 8.6.4 below for details on rule actions).

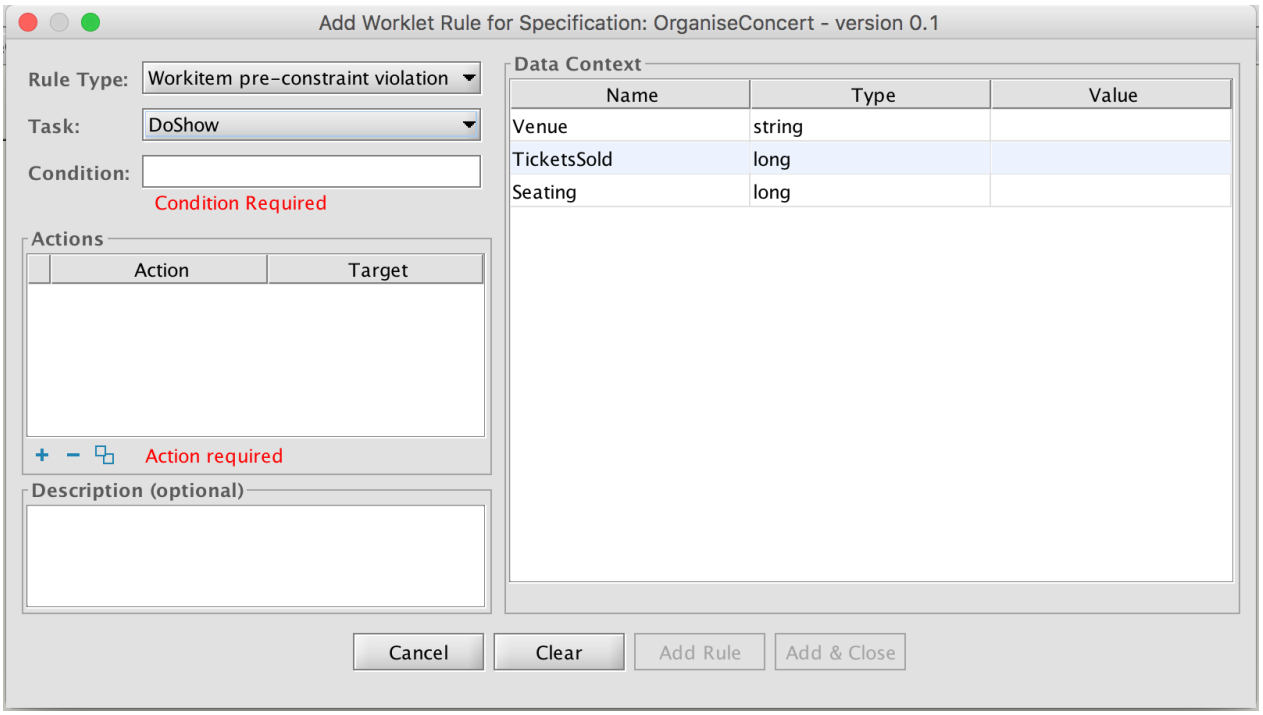


Figure 8.12: The Add Rule Dialog

- The *Data Context* panel shows the set of data variables defined for the task, or for case-level rule types, the case-level input variables. You may add values for those variables to define a data context that will allow the rule's condition to evaluate to true (see Section 8.6.3 below for more). Formally, the new rule's data context becomes the *cornerstone data* for the new rule, and is required for determining the correct location in the rule tree to insert the new rule.
- The *Description* area is where comments regarding the rule can be entered. Note the description is provided simply as a place for further information, but has no bearing on the operation of the rule, and so may be left empty if desired.

To add a new rule, select the rule type, and task (for item-level rule types), add a condition and one or more actions applicable for the selected rule type, some appropriate values for the data context, and optionally some descriptive text. Then, click the *Add Rule* button to add the rule and continue (if you'd like to add further rules), or the *Add & Close* button to add the rule and close the dialog. Both these buttons are enabled only when the entered information for the new rule is valid. You may clear entered input at any time before a rule is added using the *Clear* button, and the *Cancel* button will close the dialog, ignoring any unsaved information that has been entered.

When a rule is added via the *Add Rule* dialog, the following sequence occurs:

1. If a rule set for the specification does not currently exist, it is created.
2. If a tree set for the selected rule type does not currently exist in the rule set, it is created.
3. If a rule tree does not currently exist in the tree set for the rule type and task, it is created.
4. The new rule is added to the appropriate location in the rule tree (or added as its first child node if the tree was created in step 3).

In other words, the appropriate rule framework is created as necessary on the creation of a new rule, transparently from a user perspective.

8.6.3 Defining a Rule Condition

Every rule has a condition that is evaluated at various times during the execution of an instance of the specification the rule was defined for. All conditions must finally evaluate to a simple Boolean value (i.e. *true* or *false*).

Conditions are expressed as strings of operands and operators of any complexity, and sub-expressions may be parenthesised. The supported operators are shown in Figure 8.13. Conditions may also be defined using XPath and XQuery expressions. The variables used in a condition refer to the variables defined in the rule's data context, that is the variables defined for the relevant task or case.

Precedence	Operators	Type
1	* /	Arithmetic
2	+ –	
3	= > < != >= <=	Comparison
4	&	Logical AND
	|	Logical OR
	!	Logical NOT

Figure 8.13: Operators Supported

It is a requirement that a condition added to a rule must evaluate to true, based on its data context. For example, a condition 'Seating >10000' will be deemed valid only if the value for the 'Seating' variable in the data context pane exceeds 10,000. When adding a new rule via the *Add Rule* dialog, this means that you must enter values for relevant variables, as required, to ensure the condition entered evaluates to true. If it does not, an error will display under the condition, and the rule will not be able to be saved.

To assist in condition creation for new rules, when a value is entered in the data context, a simple condition is automatically generated based on the value entered. The operator(s) in the rule can then be edited to suit, as required. If a compound expression is required, place the appropriate logical operator at the end of the condition – on adding a value to a different variable, the sub-condition will be appended to the existing one (that is, if the current condition ends with a logical operator, newly generated sub-expressions will be appended, if there is no logical operator, the entire condition is replaced with the new condition). Of course, conditions may also be entered and/or edited directly in the *Condition* field.

Once the condition is valid for its data context (i.e. it evaluates to true), no error will appear under the condition field.

When adding a rule to extend a rule tree at runtime (via the *Replace Worklet* dialog; see Section 8.6.7), the data context displayed will reflect the actual data of the case or task itself, and thus may not be edited. However, clicking on rows in the data context will auto-generate conditions in a similar manner to that described above, using the actual data values rather than user-inserted values.

IMPORTANT To get the greatest benefit from the worklet rules framework, the recommended approach is to initially define as few rules as possible for a (parent) specification, prior to the execution of specification instances. That is, rules best reflect actual work practices when they are added during instance execution, as a result of differences in the data contexts of each instance. When rules are added outside of instance execution, assumptions must be made about the data values that will trigger selection of that rule at some later runtime, and so may not reflect actual practices. So, add a few rules for selected rule types as desired, then allow the rule tree to grow over time through instance executions. See Section 8.6.7 for details of adding rules at runtime.

8.6.4 Defining an Action Set

An action set defines the actions that are to be taken when a rule is returned from a rule tree search, having been evaluated as meeting the condition of the rule based on the current data context.

Each action set consists of one or more *primitives*, each primitive being an Action–Target pair. Every rule requires the definition of an action set in order to be valid. For a *Selection* rule type, an action set will consist of one or more primitives, each having an action of 'select' and a target of one or more worklets to execute. An action set for a rule type other than *Selection* (i.e. an exception rule type), will consist of a number of primitives forming an exception handling sequence (or *Exlet*) that will manage the handling process invoked by the rule. Section 8.4.1 detailed the various actions that make up the available set of exception handling primitives that may be sequenced to form an entire handling process. Depending on the rule type selected, particular primitives may or may not be valid.

There are two methods for creating an action set: by adding rows and making choices in the *Actions* table, or drawing a graphical representation of the exlet. Each method is described below.

The Actions Table

The *Actions* table allows you to build an action set by adding rows and making selections from the choices that appear on each row. Each row represents one primitive (i.e. an Action–Target pair).

To add a row, click the add button + on the *Actions* table toolbar. On each added row, click in the *Action* cell to drop down a list of available actions for the currently selected rule type (only the valid actions for the selected rule type are displayed).

Once an action has been chosen, click in the *Target* cell to make a choice from the available targets for the chosen action. If a *Selection* or *Compensate* action has been chosen, a dialog will appear listing all available worklets, from which one or more may be chosen to represent the worklets to be executed. If more than one worklet is selected, they are executed concurrently. To execute a number of worklets sequentially, add a sequence of *Selection* or *Compensate* actions. If the current rule type is *Selection*, only selection actions can be added. Figure 8.14 shows the equivalent action set from the graphical representation of Figure 8.7 shown in the *Actions* table of the *Add Rule* dialog.

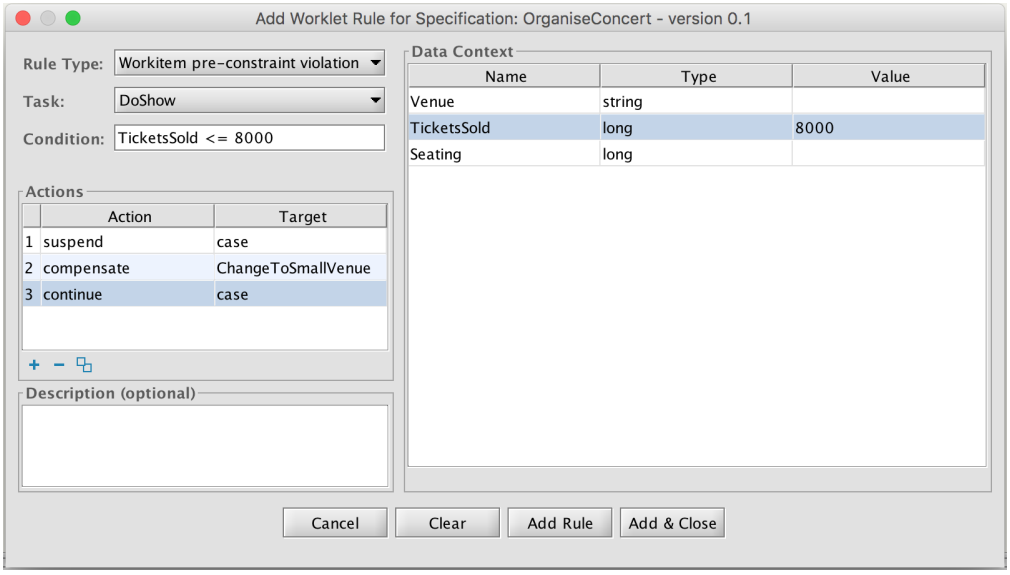


Figure 8.14: Example Action Set in the Add Rule dialog

Each added primitive is auto-validated to ensure it is valid within itself (i.e. the action can be performed

on the target), valid for the current rule type, and that the entire sequence forms a valid action set. For example, an action set will be deemed invalid if a *Continue–Case* primitive is not preceded by a *Suspend-Case* primitive. If an action set fails validation, the affected table row(s) will be coloured red and an explanatory message will be displayed under the table (Figure 8.15).

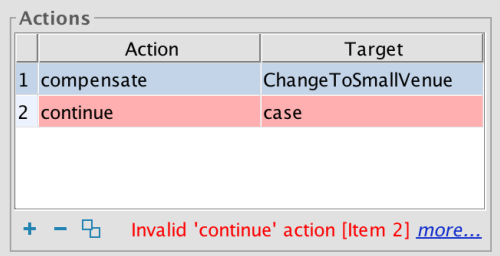


Figure 8.15: Example of an invalid Action Set (detail)

A row can be removed at any time by selecting it and clicking the remove button − on the *Actions* table toolbar.

Drawing Exlets Graphically

To open the graphical actions editor, click the editor button on the *Actions* table toolbar. The graphical actions editor will open in a separate dialog. Note that the actions editor is disabled when the *Selection* rule type is selected, since only one type of action is valid for that rule type.

The *Actions Editor* dialog (cf. Figure 8.7 on page 190) makes the process of defining an exception handling sequence easier by allowing you to create the sequence graphically:

- To place a primitive on the drawing canvas, select the appropriate primitive from the toolbox on the left, and then click on the canvas (**Tip**: mouse-over each primitive to see its name).
- The *Compensate* primitive will, when invoked at runtime, execute a worklet as a compensation process as part of the overall exception handling process. When you place a *Compensate* primitive on the canvas, a dialog will display listing all available worklets known to the Worklet Service (see Section 8.6.1 for details on adding a worklet to a repertoire). Select one or more worklets from the list, then click OK to complete the addition of the *Compensate* primitive to the canvas. You can also double click on a *Compensate* primitive to view and/or change the selected worklets for that primitive.
- The primitives *SuspendAllCases*, *RemoveAllCases* and *ContinueAllCases* may be limited to ancestor cases only by right-clicking on primitives of those kinds and selecting *Ancestor Cases Only* from the popup menu. Ancestor hierarchies occur where a worklet is invoked for a case, which in turn invokes a worklet, and so on. When a primitive is limited to ancestor cases, it applies the primitive's action to all cases in the hierarchy from the current case back to the original parent case, rather than all running cases of the specification.
- Use the *Arc Tool* to define the sequence order. First, select the *Arc Tool* in the toolbox, then click and hold on the first node, drag the mouse pointer until it is over the next node in the sequence, then release the mouse. For an action set to be valid (and thus allowed to be saved) there must be a direct, unbroken path from the start node to the end node (the start and end nodes are always displayed on the canvas). Also, the action set will be considered invalid if there are any nodes on the canvas that are not attached to the sequence. Finally, certain sequences of primitives may not be valid – please refer to the status bar under the canvas for validation messages; a green 'OK' will appear when the sequence is valid.
- Use the *Select Tool* to move placed primitives around the canvas. First, select the *Select Tool* in the toolbox, then click and drag a primitive to a new location.

- The *Align* button will immediately align the nodes horizontally and equidistantly between the start and end nodes (as in Figure 8.7).
- The *Clear* button will remove all added nodes to allow a restart of the drawing process.
- The *Cancel* button discards all work and returns to the previous dialog.
- The *OK* button will save the exlet and return to the previous dialog (the button will be disabled if the exlet is invalid).
- To delete a primitive from the canvas, select the primitive and use the *Delete* or *Backspace* key.

When a valid sequence is saved, you will be returned to the previous dialog, where the action set will be displayed textually in the *Actions* table. Changes made to the action set in the table will be reflected in its graphical representation, and vice versa.

8.6.5 Browsing an Existing Rule Set

To browse an existing rule set, first ensure the Worklet Service is running at the configured location. Then, either click the *Plugins* → *Worklet Mgt* → *View Rule Set* menu item, or click the *View Rule Set* toolbar button. The *Rule Browser* dialog will be displayed (Figure 8.16).

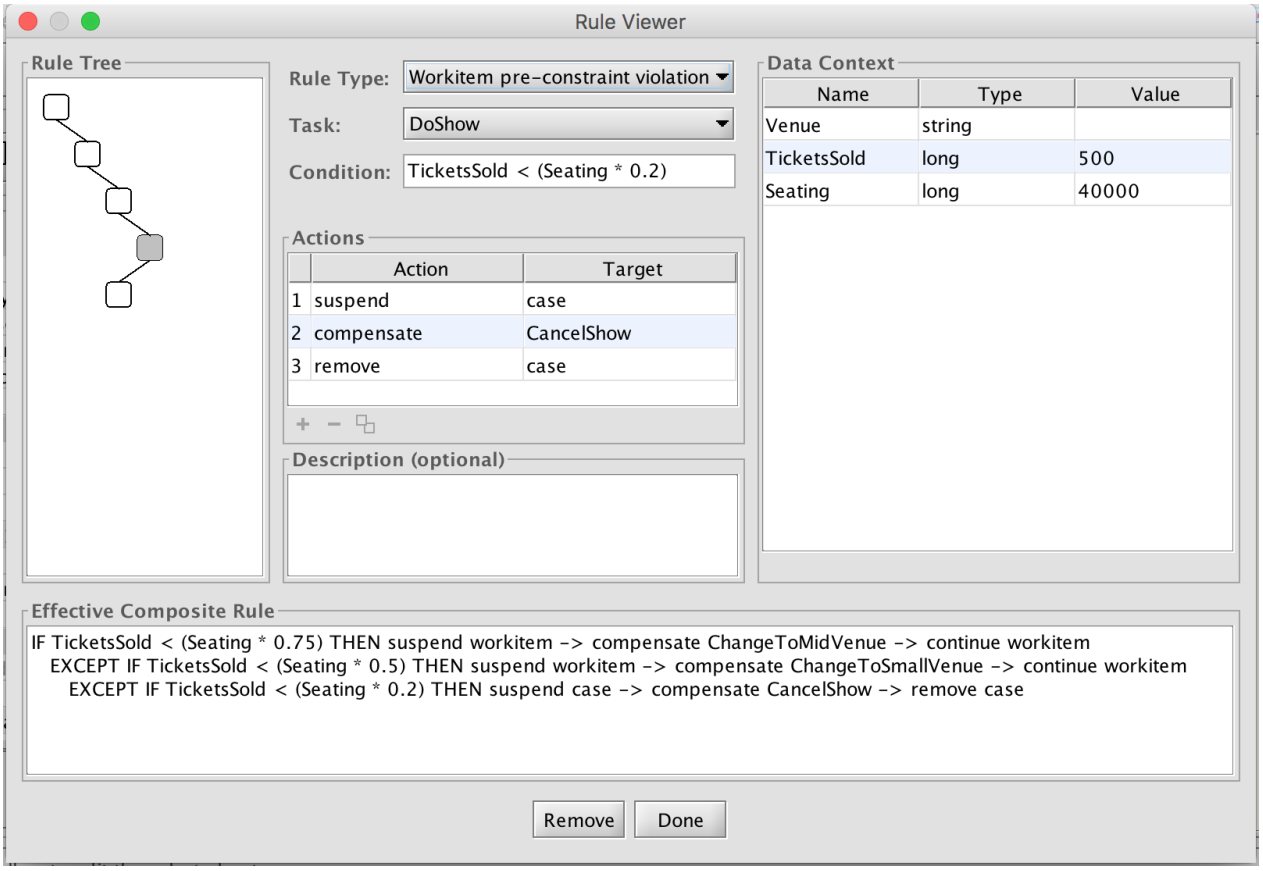


Figure 8.16: The Rule Browser dialog

The *Rule Browser* dialog provides for the viewing of all existing rules for the currently opened specification, and for the optional removal of selected rules. Most of the components are the equivalents of those found

in the *Add Rule* dialog, with the addition of a graphical rule tree viewer on the left, from which rules may be selected, and a text area at the bottom of the dialog that displays the "Effective Composite Rule" (more below).

Choose a rule type from the selections available in the drop-down list to view rules for that type. If a case-level rule type is chosen, its rule tree will be graphically displayed in the left of the dialog. For item-level rule types, first choose a task from the selections available to show the rule tree for that task. Then, click on a particular node on the rule tree to display the (read-only) content of that rule in the dialog. Note that only those rule types and tasks that have rules already defined for them will appear as available selections in the drop-down lists.

The "Effective Composite Rule" text area shows a representation of the conjunction of each rule's condition from the top of the rule tree to the selected rule. In other words, it shows the effective rule that will be satisfied when that rule is reached, each part of the effective rule being drawn from the condition of each individual rule along the path traversed through the tree.

To remove a rule, select it on the tree, click the Remove button, and confirm the removal in the dialog that appears. If you remove a non-leaf rule node, that is a node that has a 'child' node, the rule tree will be automatically reconstructed to allow for the removal (since the meaning of a 'child' condition depends on its parent), and so may appear with a slightly different structure after the removal.

8.6.6 Exporting a Rule Set to File

A rule set may be exported in XML format to disk file, primarily to allow uploading of rule sets to Worklet Service deployments in other locations.

To export a rule set, first ensure the Worklet Service is running at the configured location, then, either click the *Plugins* → *Worklet Mgt* → *Export Rule Set* menu item, or click the *Export Rule Set* toolbar button. The *Export Selected Rule Sets* dialog will be displayed, which lists all of the specifications that have rule sets currently stored by the Worklet Service. Select one or more specifications, then click OK to export the selected rule sets. Choose a directory to save to from the *File . . . Save* dialog that appears, then click *Save* to save the file to disk. If you have chosen to export multiple rule sets, they will be saved to a compressed (zip) file.

8.6.7 Rejecting a Selected Worklet (Extending a Rule Tree During Execution)

There are occasions when the worklet launched for a particular case, while the correct choice based on the current rule set, turns out to be an inappropriate choice for the context of that instance. For example, suppose a patient in a *Casualty Treatment* case presents with a rash and a heart rate of 190. While the current rule set correctly returns the *TreatRash* worklet (via the rule with condition 'Rash=true'), it may be desirable to first treat the racing heart before the rash is attended to. In such a case, while the Worklet Service begins execution of an instance of the *TreatRash* process, it is clear that a new rule needs to be added to the rule set so that cases that have such a data context (i.e. a rash **and** a high heart rate) will be handled correctly, both for the current instance and into the future.

IMPORTANT This is the preferred method for extending a rule set, since here we are dealing with the context of an actual case driving the requirement of a new rule, rather than having to invent a data context when adding rules via the *Add Rule* dialog.

To amend a rule set by adding a new rule following a worklet choice that is deemed inappropriate for its data context, first ensure the Worklet Service is running at the configured location, then, either click the

Plugins → *Worklet Mgt* → *Replace Worklet* menu item, or click the *Replace Worklet* toolbar button. The *Replace Worklet* dialog will be displayed (Figure 8.17).

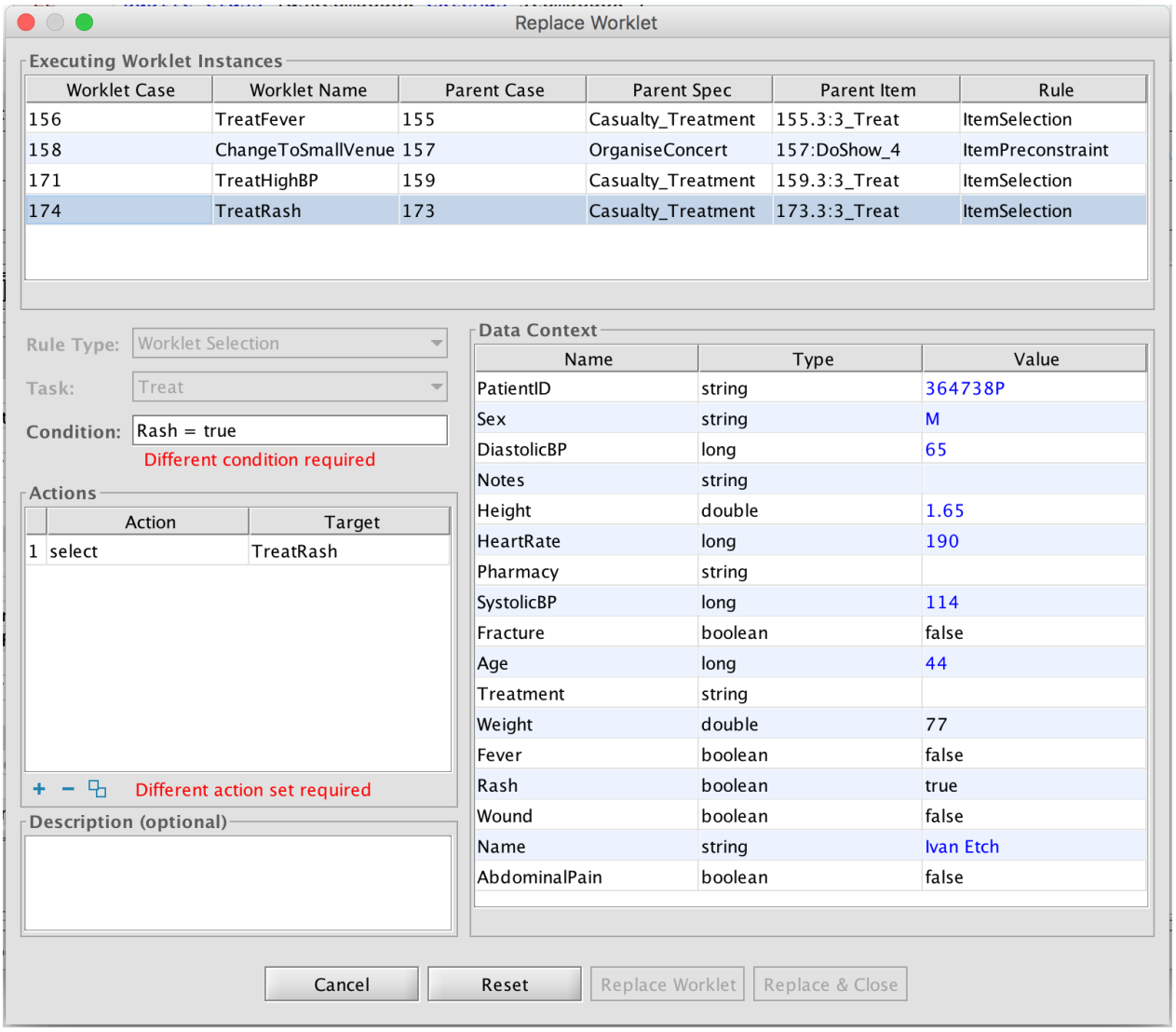


Figure 8.17: The Replace Worklet dialog

Again, most of the components are the equivalents of those found in the *Add Rule* dialog, with the inclusion of a list of all currently executing worklets at the top of the dialog. Selecting an executing worklet will display the details of the rule that selected the worklet in the dialog, below the executing worklets list. The *Rule Type* and *Task* drop down lists are disabled, because the new rule will be added to the actual rule tree for the rule type (and task) that caused the original worklet selection.

To add a new rule to accommodate the current case context, you are required to enter a different condition (since the same condition will return the original worklet) and a different action set (since, if the same action set is desired, there is no need to add a new rule). To aid in the creation of a new condition, the data context table shows the actual values of the case or task variables of the selected parent instance that were used to evaluate the rule set that derived the original selection. The data values that differ from those of the cornerstone case (i.e. the data values that existed when the original rule was added) are shown in blue; mouse over those values to see the cornerstone values in a popup. Values may be selected to form a new condition (as for the *Add Rule* dialog), but may not be changed, since they represent the actual data context of the current case.

IMPORTANT Since we have the case data for the original rule, and the case data for the new rule, to define a condition for the new rule it is only necessary to determine what it is about the current case that makes it necessary for the new rule to be added. That is, it is only where the case data items *differ* that distinguish one case from the other, and further, only a subset of that differing data is relevant to the reason why the original selection was inappropriate.

For example, there are many data items that differ between the two case data sets shown in Figure 8.17, namely *PatientID*, *Name*, *Sex*, *HeartRate*, *Blood Pressure readings*, *Height* and *Age*. However, the only differing data item of relevance here is *HeartRate* - that is the only data item that, in this case, makes the selection of the *TreatRash* worklet inappropriate.

Clicking on the line "HeartRate" in the *Data Context* table copies that line to the *Condition* input. Thus, a condition for the new rule has been easily created, based on the differing data attribute and value that has caused the original worklet selection to be invalid for this case.

Note that it is not necessary to define the rule as "Rash = True & HeartRate = 190", as might first be expected, since this new rule will be added to the true (***except***) branch of the *TreatRash* node. Consequently, it will only be evaluated if the condition of its parent, "Rash = True", first evaluates to True. Therefore, any rule nodes added to the true branch of a parent become **exception** rules of the parent. In other words, this particular tree traversal can be interpreted as: "if Rash is True then return TreatRash **except** if HeartRate is also 190 then return ???" (where ??? = whatever worklet we decide to select for this rule - see more below).

Now, the new rule is fine if, in future cases, a patients heart rate will be exactly 190, but what if it is 191, or 189, or 250? Clearly, the rule needs to be amended to capture all cases where the heart rate exceeds a certain limit; say 165.

To make the condition for the new rule more appropriate, the condition "HeartRate = 190" should be edited to read "HeartRate $>$ 165".

After defining a condition for the new rule, the worklet to be launched when this condition evaluates to true must be chosen as the target for the 'select' action in the *Actions* table, by clicking on the target and choosing the appropriate worklet from the list.

TIP If a new worklet is required, create it first in the normal manner, then upload it to the Worklet Service (cf. Section 8.6.1). It will then appear in the list of available worklets for addition to the action set.

Once all the fields for the new rule are complete and valid, click the *Replace Worklet* or the *Replace & Close* button to:

- Add the newly created rule to the rule tree.
- Remove the selected executing worklet from the YAWL Engine
- Perform a new selection for the case/task, taking into account the new rule.

A message dialog will be shown soon after with the results of the replacement process sent from the Worklet Service back to the Rules Editor, similar to Figure 8.18, with the case ID for the newly launched worklet. After you click OK, if you chose the *Replace Worklet* button (instead of the *Replace & Close* button) you will see the new worklet in the list of executing worklets, and the original worklet will be no longer listed.

8.6.8 Rules Maintenance

Over time, the Worklet Service's repository may become bloated with rule sets that are no longer used, or worklets that are not referenced by any rule. To maintain an efficient repository, cleanups should be periodically run, as follows.

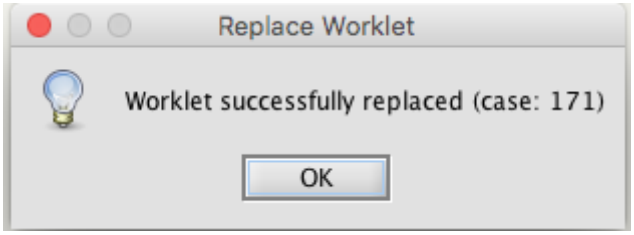


Figure 8.18: Information Message after Worklet Replacement

Removing a Rule Set It may be desirable to remove a complete rule set from the Worklet Service for a specification that is no longer in use. To remove a rule set, first ensure the Worklet Service is running at the configured location, then, either click the *Plugins* → *Worklet Mgt* → *Remove Rule Set* menu item, or click the *Remove Rule Set* toolbar button. The *Remove Selected Rule Sets* dialog will be displayed, which lists all of the specifications that have rule sets currently stored by the Worklet Service. Select one or more specifications, then click OK to remove the selected rule sets.

Removing 'Orphaned' Worklets Over time, the set of worklets stored in the Worklet Service repository may contain worklets that are no longer referenced by any rule set. To remove them, first ensure the Worklet Service is running at the configured location, then, either click the *Plugins* → *Worklet Mgt* → *Remove Orphans* menu item, or click the *Remove Orphans* toolbar button. The *Remove Selected Unreferenced Worklets* dialog will be displayed, which lists all of the worklets currently stored by the Worklet Service that are not referenced by any rule set. Select one or more worklets, then click OK to remove them from the repository.

8.6.9 Bulk Uploading of Worklets and Rule Sets

This section will be of particular interest to those who have used or are using previous versions of the Worklet Service and/or the Worklet Rules Editor. With this version of Worklet Service, the old file-based repository is no longer supported. Rather, all worklets and rule sets are be stored in database tables supported by the Worklet Service.

If you have a number of worklets, and/or you have a number of rule sets stored as XML files, you may upload them in bulk using the *Upload Files* functionality of the Worklet Management plugin.

To upload existing worklet or rules files, first ensure the Worklet Service is running at the configured location, then, either click the *Plugins* → *Worklet Mgt* → *Upload Files* menu item, or click the *Upload Files* toolbar button. A directory chooser dialog will display, allowing you to select the file location where the worklet and/or rules files are stored. Choose the desired directory, then click OK. Any file with a *.yawl* extension in that directory, or any of its subdirectories, will be added as a worklet to the service's worklet repository, if it contains a valid YAWL specification. Any file with a *.xrs* extension in that directory, or any of its subdirectories, will be added as a rule set to the service's rules repository, if it contains a valid worklet rule set (as XML).

CRITICAL For a rule set to validate and operate successfully, it **MUST** contain the identifiers of the parent specification it represents as attributes of its *spec* XML element, and the identifier value for each worklet target of a 'select' or 'compensate' action. For each rule set file (.xrs) to be imported, first open it in a text editor to ensure the specification identifier attributes are included. If they do not appear, open the specification file (.yawl) in a text editor, then copy the required values from the locations indicated in the example in Figure 8.19. Then, for each worklet named as a target in the rule conclusion, replace the name with the identifier value (cf. Figure 8.20). Save the updated rules file. Once all rules files are verified to have a valid structure, they can be bulk uploaded into the service.

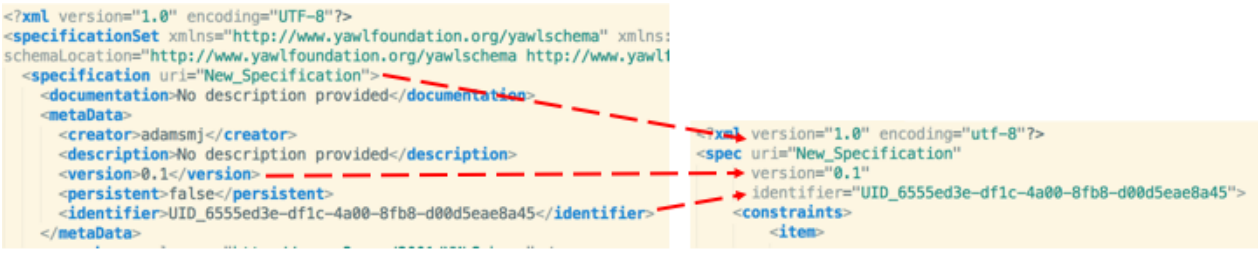


Figure 8.19: Assigned parent specification identifiers to rule file attributes

Figure 8.20: Assigned worklet specification identifier to rule's compensate target

8.7 Walkthrough – Using the Worklet Service

The samples that come with the Worklet Service deployment (in the 'samples' directory) contains a number of example specifications with worklet-enabled tasks, each with an associated rule set and a number of associated worklets. This section will step through the execution of several of these examples. The first two examples feature the Selection Service; the remainder the Exception Service. Knowledge of how to use the YAWL system is assumed. Before we begin, make sure the Worklet Service is correctly installed and operational, then use the Editor plugin described in the previous section to load the sample worklets and rules into the Worklet Service's repository (from the *samples/worklets* directory, and finally log into the YAWL system.

A. Selection: Worklet-Enabled Atomic Task Example

The *Casualty Treatment* specification used in the previous sections of this manual is an example of a specification that contains an atomic task (called *Treat*) that is worklet selection-enabled. We'll run through a complete instance of the example specification to see how worklet selection operates.

Log on to YAWL with a user that has administrator or 'can manage cases' privileges. Navigate to the *Case Mgt* page and upload the *Casualty Treatment* specification from the *samples/parents* directory of the service. Then, launch a *Casualty Treatment* case from the same page.

The case begins by requesting a patient id and name - just enter some data into each field then click *Start* (Figure 8.21).

Go to the *Work Queues* page, and the first task in the case (*Admit*) will be listed as an offered workitem. Make a note of the case number. *Accept & Start* the *Admit* workitem.

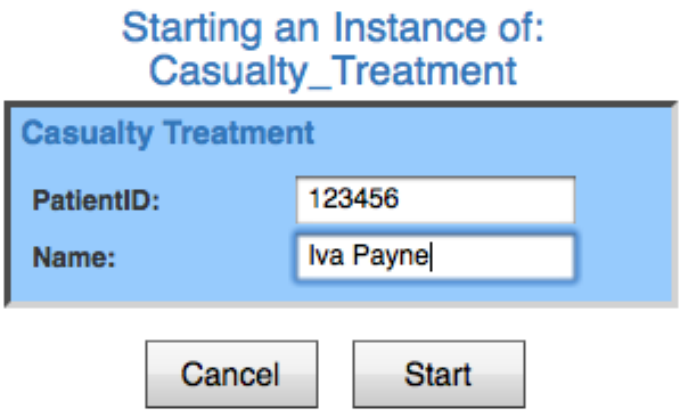


Figure 8.21: Launching a Casualty Treatment Case (detail)

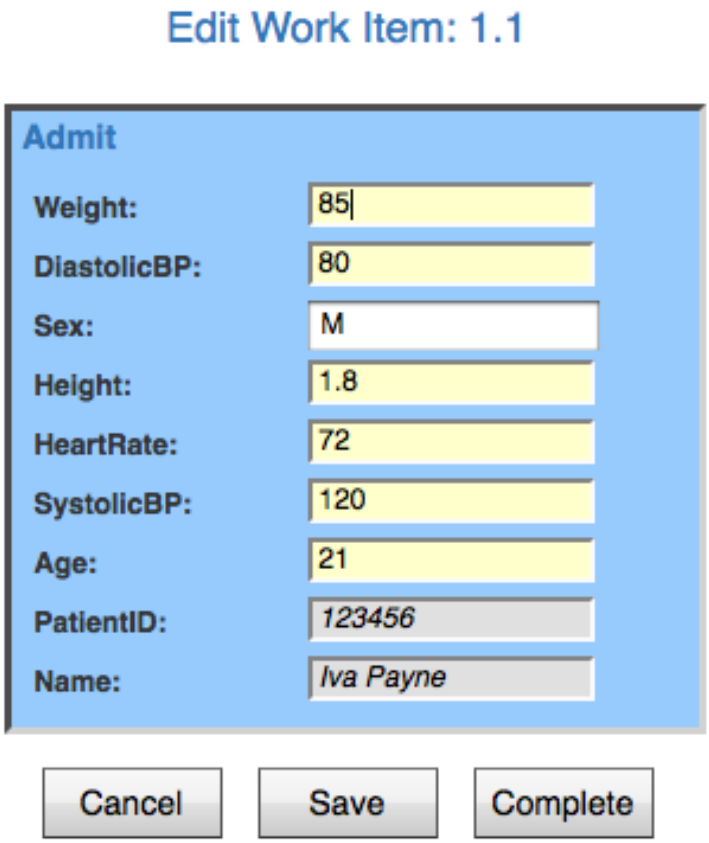


Figure 8.22: Editing the *Admit* Workitem (detail)

The *Admit* workitem simulates an admission to the Casualty department of a hospital, where various initial checks are made of the patient. You'll see that, in addition to the patient name and id specified when the case started, there are a number of fields containing some medical data about the patient. Each field has some default data (to save time), but you may edit any fields as you wish (Figure 8.22). When done, click the *Complete* button.

Go back to the *Work Queues* page and start the next workitem, *Triage*. The *Triage* task simulates that part of the process where a medical practitioner asks a patient to nominate their symptoms. You'll see that the patient's name and id have again been displayed for identification purposes, in addition to five fields which approximate the medical problem. One field should be set to *true* (checked), the others to *false* (unchecked).

Lets assume the patient has a fever. Check the *Fever* field, leave the rest unchecked, and then click the *Complete* button (Figure 8.23).

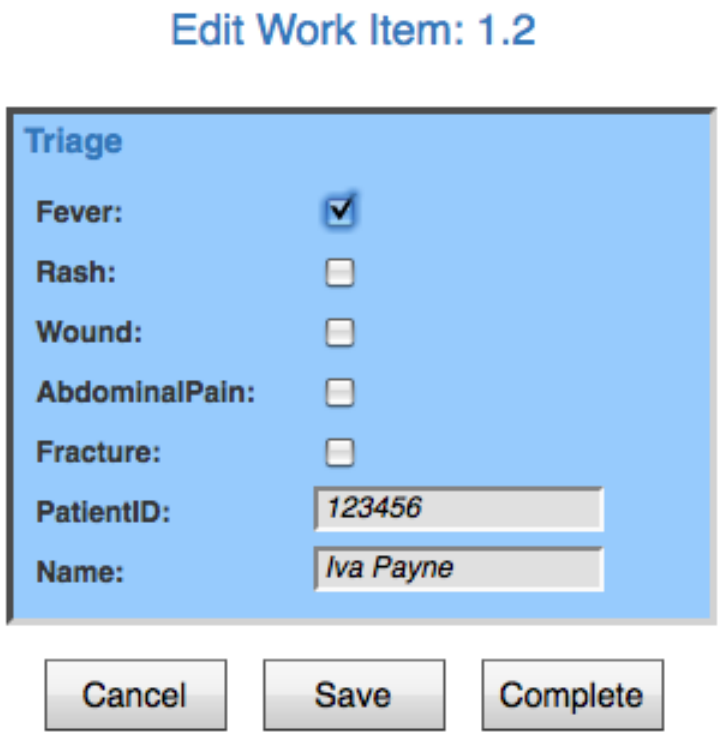


Figure 8.23: Editing the *Triage* Workitem (detail)

There is nothing special about the first two tasks in the process; they are standard YAWL tasks and operate as expected. However, the next task, *Treat*, has been associated (using the YAWL Editor) with the Worklet Service. The *Treat* task simulates that part of the process that follows the collection of patient data and actually treats the patient's problem.

Of course, there are many medical problems a patient may present with, and so there are just as many treatments, and some treatment methods are vastly different to others. In a typical workflow process, this is the part of the process where things could get very complicated, particularly if we tried to build every possible treatment for every possible medical problem as a conditional branch directly into the process model.

The Worklet Service greatly simplifies this problem, by providing an extensible repertoire of discrete workflow processes (worklets) which, in this example, each handle the treatment of a particular medical problem. By examining the case data collected in the earlier tasks, the Worklet Service can launch, as a separate case, the particular treatment process for each case.

This method allows for a simple expression of the task in the 'parent' process (i.e. a single atomic *Treat* task signifies the treatment of a patient, whatever the eventual treatment process may be) as well as the ability to add to the repertoire of worklets at any time as new treatments become available, without having to modify the original process.

When the *Triage* workitem is completed, the next task in the process, *Treat*, becomes enabled. Because it is

worklet-enabled, the Worklet Service is notified. The Service checks to see if there is a set of rules for *selection* associated with this workitem, and if so the service checks out the workitem.

When this occurs, the YAWL Engine marks the workitem as executing (externally to the Engine) and waits for the workitem to be checked back in. In the meantime, the Worklet Service uploads the relevant specification for the worklet chosen as a substitute for the workitem and launches a new case for the specification. When the worklet case completes, the Worklet Service is notified of the cases completion, and the service then checks the original workitem back into the Engine, allowing the original process to continue.

We have completed editing the *Triage* workitem and clicked the *Complete* button. Go to the *Work Queues* page. Instead of seeing the next workitem listed (i.e. *Treat*), we see that *Test Fever*, the first workitem in the *TreatFever* process, is listed in its place (Figure 8.24). The *TreatFever* process has been chosen by the Worklet Service to replace the *Treat* workitem based on the data passed to the service.

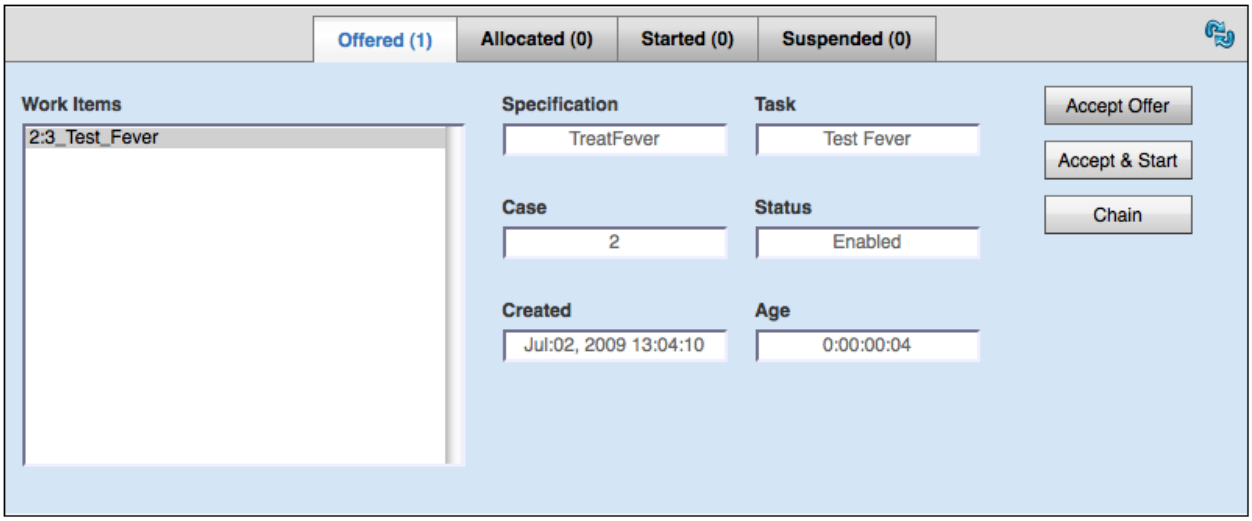


Figure 8.24: New Case Launched by the Worklet Service

Note that the case id for the Test Fever workitem is different to the case id of the parent process. Worklets run as completely different cases to the parent process, but the Worklet Service keeps track of which worklets are running for which parent cases. Go to the *Case Mgt* page to see that a *Casualty Treatment* case is still running, and that the *TreatFever* specification has been loaded and it also has a case running (Figure 8.25).

Go back to the *Work Queues* page and start the *Test Fever* workitem. The *Test Fever* workitem has mapped the patient name and id values, and the particular symptom - fever - from the *Treat* workitem checked out by the Worklet Service. In addition, it has a *Notes* field where a medical practitioner can enter observations about the patient's condition (Figure 8.26). Enter some information into the *Notes* field, and then complete the workitem.

Start the next workitem, *Treat Fever*, and then edit it. This workitem has two additional fields, Treatment and Pharmacy, where details about how to treat the condition can be entered (Figure 8.27). Enter some data here, and then complete the workitem.

When the *Treat Fever* workitem is submitted, the worklet case is completed. The Worklet Service maps the output data from the worklet case to the matching variables of the original *Treat* workitem, then checks that workitem back in, effectively completing it and allowing the next workitem in the *Casualty Treatment* process, *Discharge*, to execute.

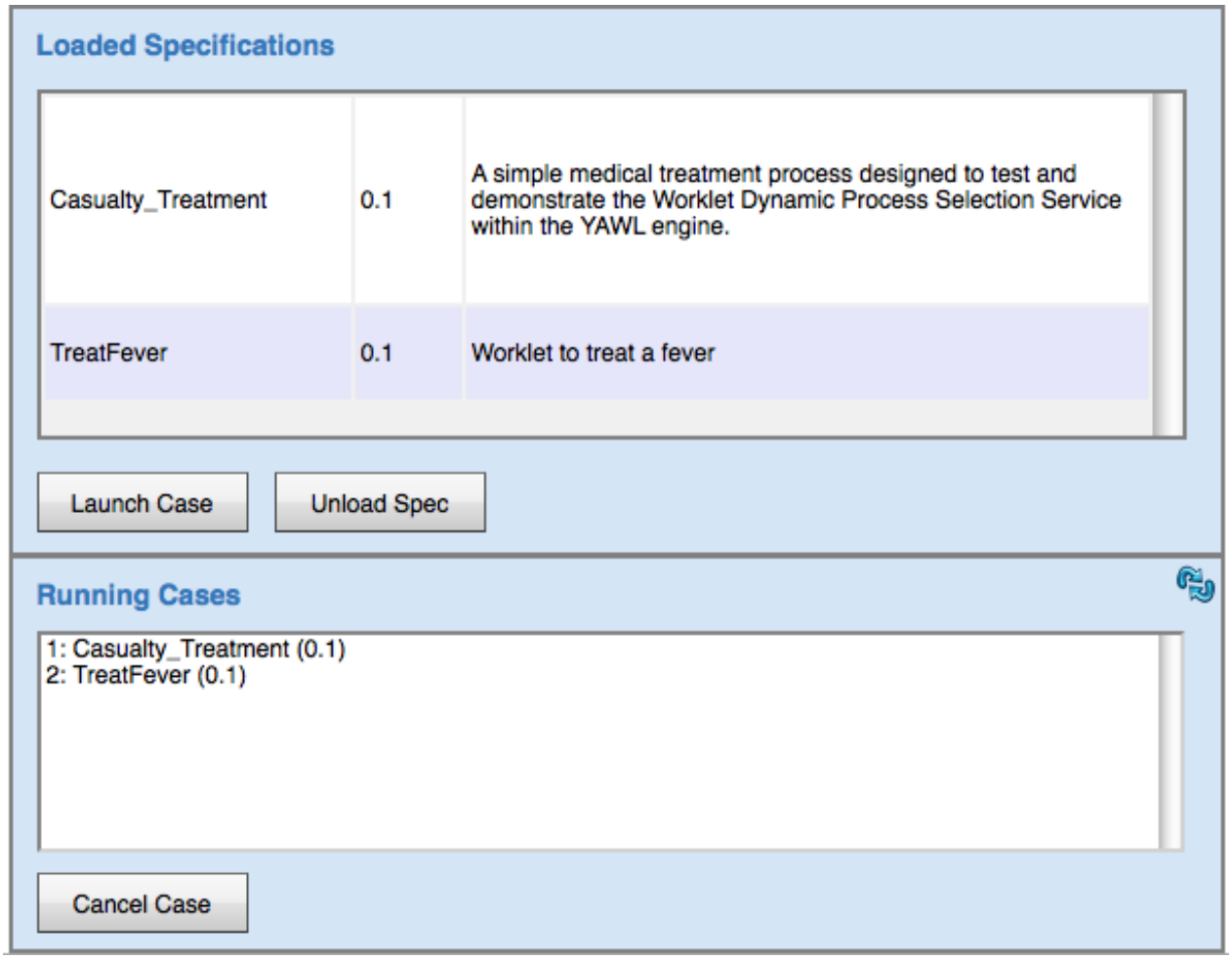


Figure 8.25: TreatFever Specification Uploaded and Launched

Go to the *Work Queues* page, and you'll see that the *Discharge* workitem is available (Figure 8.28). Edit it to see that the data collected by the *TreatFever* worklet has been mapped back to this workitem. Submit it to complete the case.

B. Selection: Worklet-Enabled Multiple Instance Atomic Task Example

This walkthrough takes the *List Maker* example from Chapter 4 (YAWL Editor) and worklet-enables the *Verify List* task to show how multiple instance atomic tasks are handled by the Worklet Selection Service.

The specification is called *wListMaker*. The only change made to the original List Maker specification was to associate the *Verify List* task with the Worklet Service using the YAWL Editor. Figure 8.29 shows the specification.

Go to the *Case Mgt* page and upload the *wListMaker* specification from the *samples/parents* directory. Then, launch an instance of *wListMaker*.

When the case begins, enter three values for the *Bob* variable, as shown in Figure 8.30 - you will have to click the *add* (+) button twice to get three input fields. Make sure you enter the values "one", "two" and "three" (without the quotes and in any order). Complete the form.

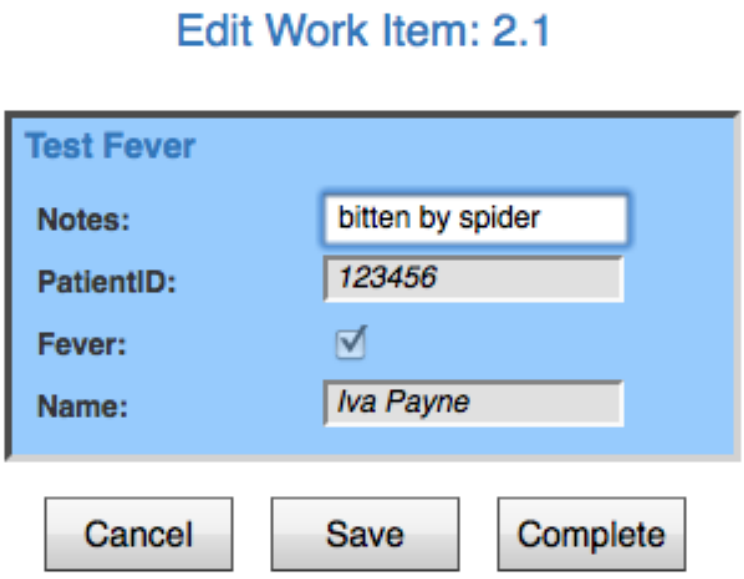


Figure 8.26: Test Fever Workitem (detail)

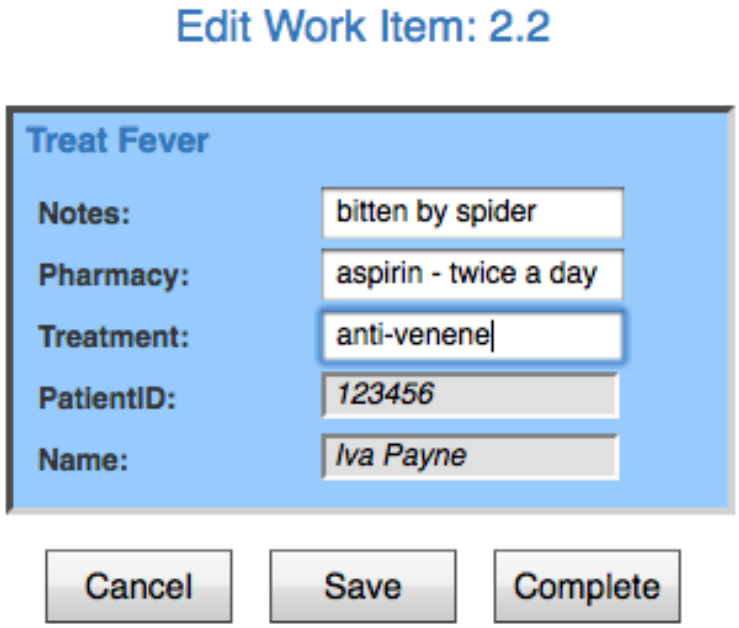


Figure 8.27: Treat Fever Workitem (detail)

Start and edit the *Create List Items* workitem. Since the values have already been entered there is no more to do here, so click the *Complete* button to continue.

The next task is *Verify List*, which has been associated with the Worklet Service. Since this task is a multiple-instance atomic task, three child workitem instances of the task are created, one for each of the *Bob* values entered previously. The Worklet Service will determine that it is a multiple instance atomic task and will treat each child workitem instance separately, and will launch the appropriate worklet for each based on the data contained in each. Since the data in each child instance is different in this example, the Worklet Service starts three different worklets, called *BobOne*, *BobTwo* and *BobThree*. Each of these worklets contains only

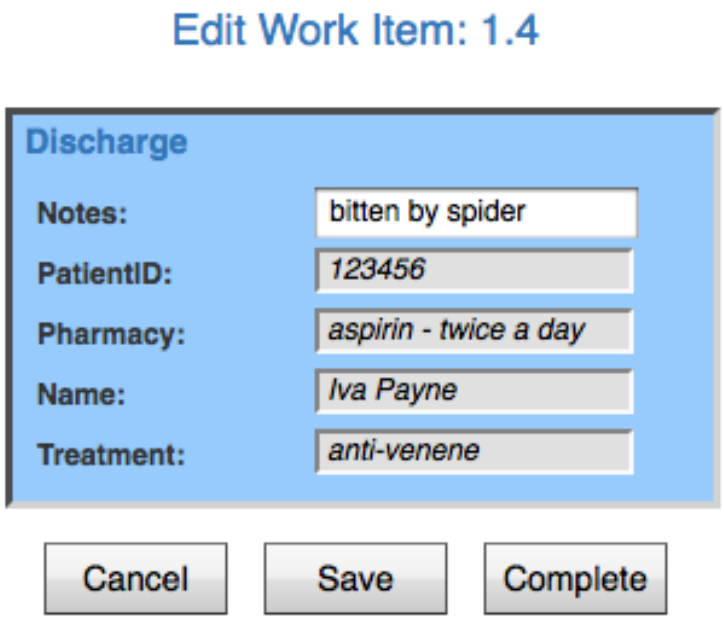


Figure 8.28: Discharge Workitem with Data Mapped from *TreatFever* Worklet

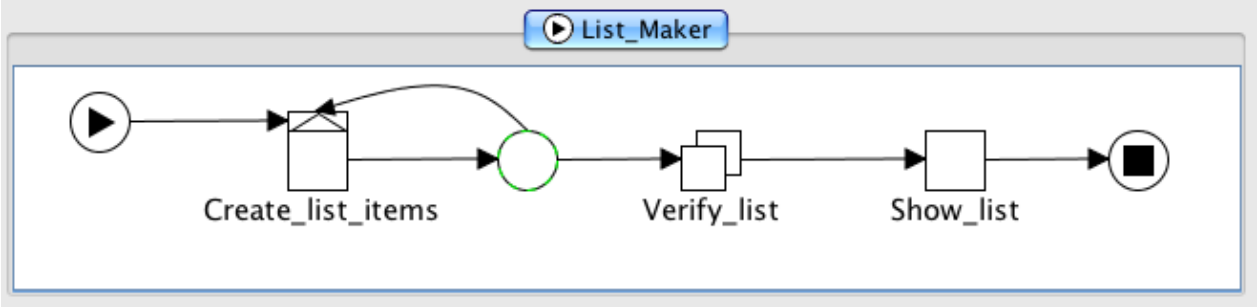


Figure 8.29: The *wListMaker* Specification

one task.

Go to the *Work Queues* page. There are three workitems listed, each one the first workitem of a separate case (see Figure 8.31).

Go to the *Case Mgt* page to see that the *BobOne*, *BobTwo* and *BobThree* specifications have been uploaded and launched by the Worklet Service as separate cases (Figure 8.32 – note the case numbers).

Go back to the *Work Queues* page and check out all three workitems. Edit each of the *Get_Bob* workitems, and modify the values as you wish - for this walkthrough, we'll change the values to "one - five", "two - six" and "three - seven" respectively.

As you edit and complete each *Get_Bob* workitem, the corresponding *Verify List* workitem from the parent instance is automatically checked in to the Engine by the Worklet Service. Since the *Bob* worklets contains only one task, editing and completing this workitem also completes the worklet case.

After the third workitem has been edited and completed, and so the third *Verify List* workitem is checked back into the Engine by the Worklet Service, the Engine determines that all the child items of the *Verify List*

Figure 8.30: Start of *wListMaker* Case with Three 'Bob' Values Entered (detail)

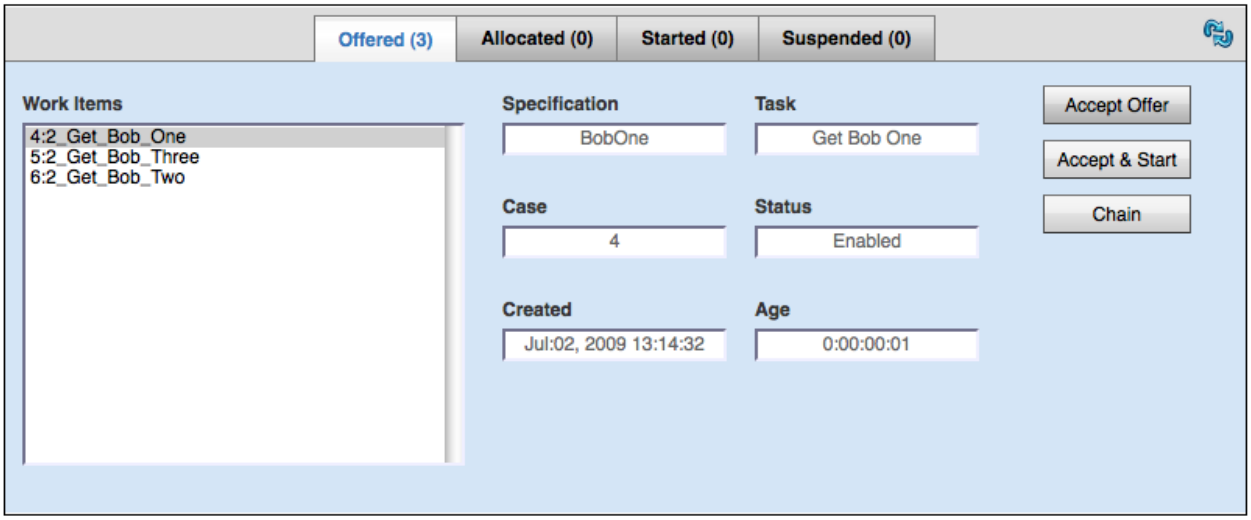


Figure 8.31: Workitems from each of the three Launched Worklet Cases

workitem have completed and so the original (parent) process continues to its final workitem, *Show List*. Start and edit the *Show List* workitem to show the changes made in each of the *Get_Bob* worklets have been mapped back to the original case (Figure 8.33).

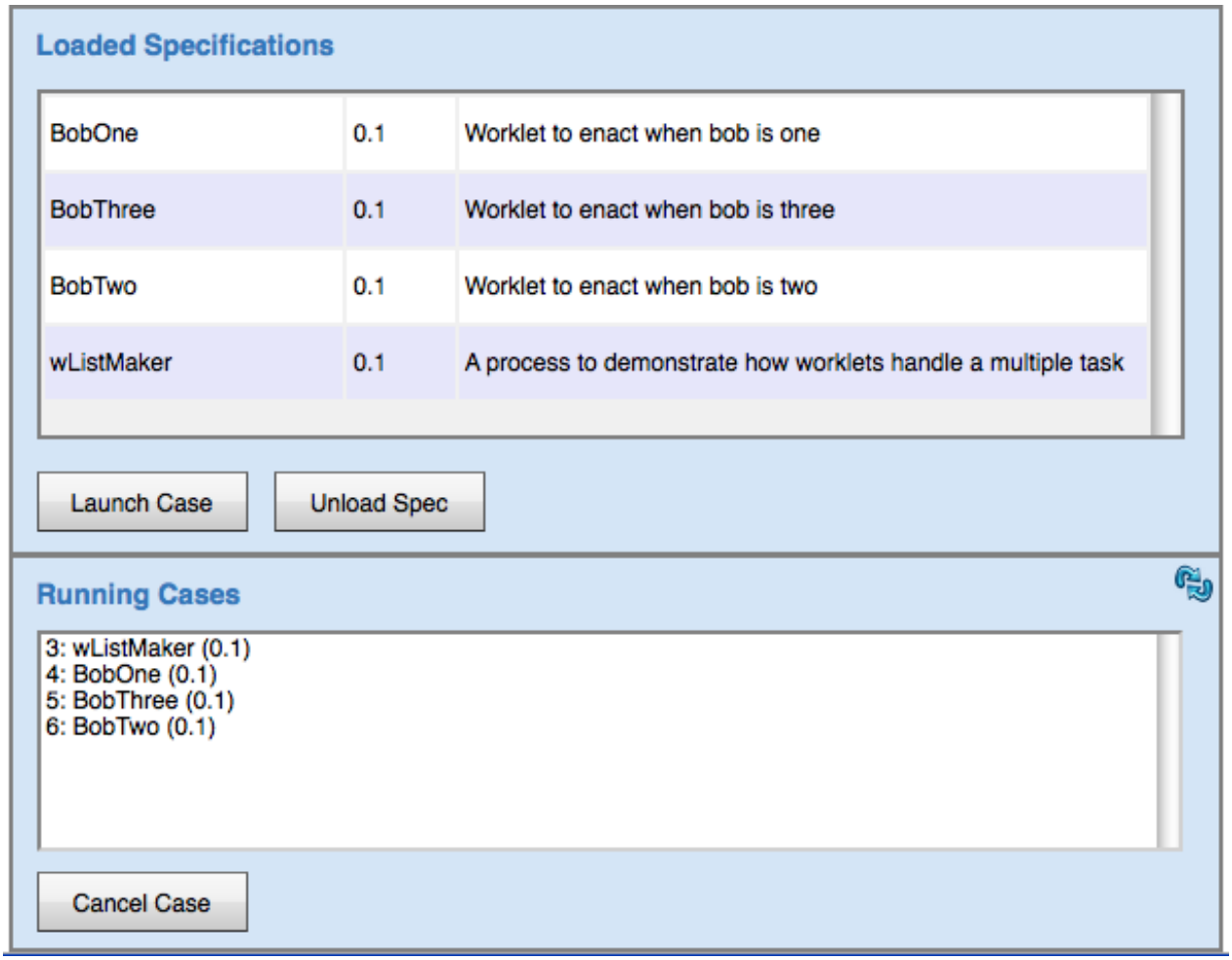


Figure 8.32: 'Bob' Specifications Loaded and Launched by the Worklet Service

C. Exception: Constraints Example

This walkthrough uses the *OrganiseConcert* specification to demonstrate a few features of the Worklet Exception Service. The *OrganiseConcert* specification is a very simple process modelling the planning and execution of a rock concert. Figure 8.34 shows the specification as it appears in the YAWL Editor.

First, ensure the Exception Service is enabled (see Section 8.1.2 for details). Navigate to the YAWL *Case Mgt* page and upload the *OrganiseConcert* specification from the *samples/parents* directory. Then, launch an *OrganiseConcert* case.

As soon as the Engine launches the case, it notifies the Exception Service via a *PreCaseConstraint* event. If the rule set for *OrganiseConcert* contains a rule tree for pre-case constraints, that tree will be queried using the initial case data to determine whether there are any pre-constraints not met by the case. In this example, there are no pre-case constraint rules defined, so the notification is simply ignored.

Tip: To follow what is happening, watch the log output in the Control Panel, the Tomcat command window, or the contents of the log file *catalina.out* in Tomcat's *logs* folder – both the exception and selection services log all interactions between themselves and the Engine to the Tomcat window and to a log file.

Pre-case constraints can be used, amongst other things, to ensure case data is valid or within certain ranges before the case proceeds, can be used to run compensatory worklets to correct any invalid data, or may even be used to cancel the case as soon as it starts (in certain circumstances). As a trivial example of the last point,

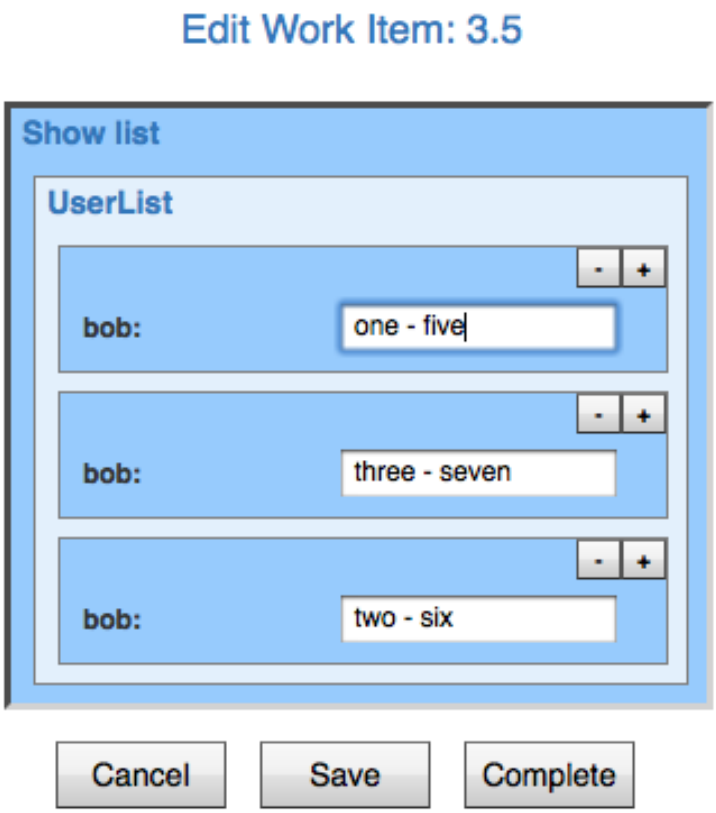


Figure 8.33: The Show List Workitem Showing the Changes to the Data Values

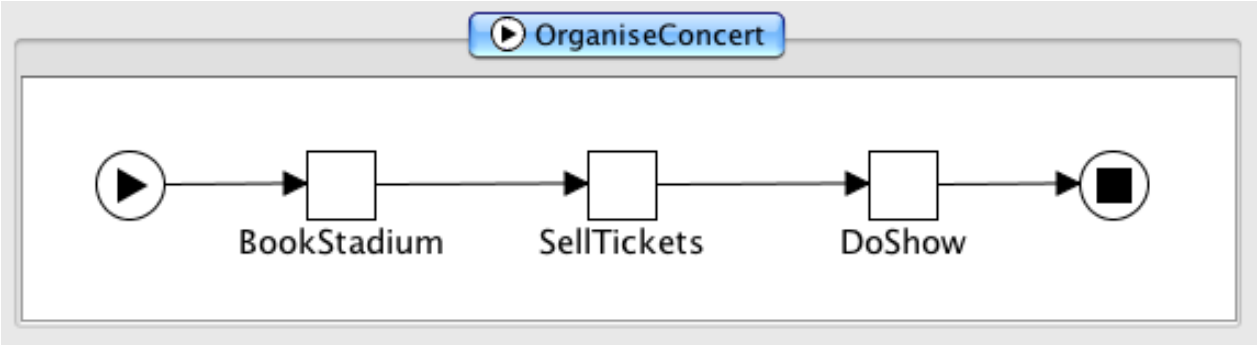


Figure 8.34: The *OrganiseConcert* Specification

launch an instance of the *Casualty Treatment* specification discussed in Walkthrough A, and enter "smith" for the patient name when the case starts. The *Casualty Treatment* rule set contains a pre-case constraint rule to cancel the case if the patient's name is "smith" (presumably smith is a hypochondriac!). This also serves as an example of exception rules and selection rules being defined within the same rule set.

Directly following the pre-case event, the Engine notifies the Service of a *PreItemConstraint* for the first workitem in the case (in this case, *Book Stadium*). The pre-item constraint event occurs immediately the workitem becomes enabled (i.e. ready to be checked out or executed). Like pre-case constraint rules, pre-item rules can be used to ensure workitems have valid data before they are executed. The entire set of case data is made available to the Exception Service - thus the values of any case variables may be queried in the ripple-down rules for any exception type rule. While there are pre-item constraint rule trees defined in the

rule set, there are none for the *Book Stadium* task, so this event is also ignored by the service.

The *Book Stadium* workitem may be started in the normal fashion. This workitem captures the seating capacity, cost and location of the proposed rock concert. These may be changed to any valid values, but for the purposes of this example, just accept the default values as given (Figure 8.35).

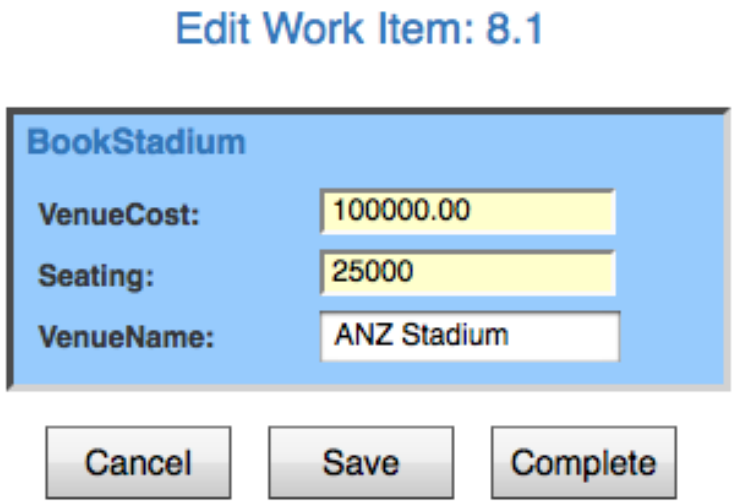


Figure 8.35: The Book Stadium Workitem (detail)

When the workitem is submitted, a *PostItemConstraint* event is generated for it by the Engine. There are no post-item constraint rules for this workitem, so again the event is just ignored. Then, a pre-item constraint notification is received for the next workitem (*Sell Tickets*). This workitem records the number of tickets sold, and the price of each ticket. Enter a price of $100 per ticket, and 12600 as the number of tickets sold, and then complete the workitem (Figure 8.36).

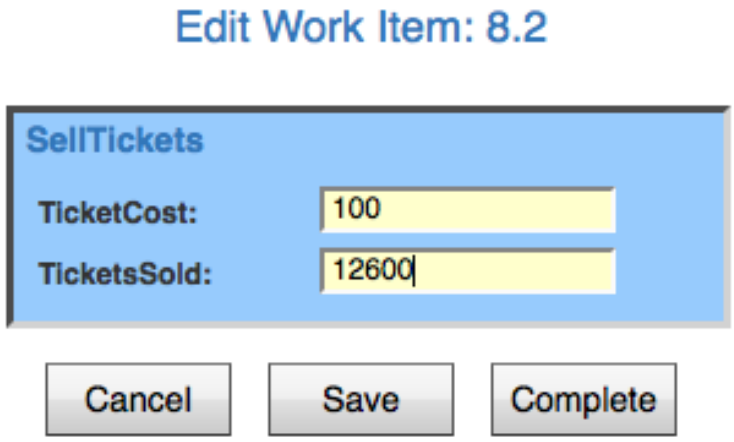


Figure 8.36: The Sell Tickets Workitem (detail)

Notice that the entered number of tickets sold (12600) is slightly more than 50% of the venue's seating capacity (25000). The next workitem, *Do Show*, does have a pre-item constraint rule tree, and so when it becomes enabled, the rule tree is queried. The effective composite rule for *Do Show*'s pre-item tree (as viewed in the Rules Editor), is shown in Figure 8.37.

In other words, when *Do Show* is enabled and the value of the case data attribute "TicketsSold" is less

Effective Composite Rule

```
IF TicketsSold < (Seating * 0.75) THEN suspend workitem -> compensate ChangeToMidVenue -> continue workitem
  EXCEPT IF TicketsSold < (Seating * 0.5) THEN suspend workitem -> compensate ChangeToSmallVenue -> continue workitem
    EXCEPT IF TicketsSold < (Seating * 0.2) THEN suspend case -> compensate CancelShow -> remove case
```

Figure 8.37: Effective Composite Rule for Do Shows Pre-Item Constraint Tree

than 75% of the seating capacity of the venue, we would like to suspend the workitem, run the compensatory worklet *ChangeToMidVenue*, and then, once the worklet has completed, continue (or unsuspend) the workitem. Following the logic of the ripple-down rule, if the tickets sold are also less than 50% of the capacity, then we want instead to suspend the workitem, run the *ChangeToSmallVenue* worklet, and then unsuspend the workitem. Finally, if there has been less than 20% of the tickets sold, we want instead to suspend the entire case, run a worklet to cancel the show, and then remove (i.e. cancel) the case.

In this example, the first rule's condition evaluates to true, for a "Tickets Sold" value of 12600 and a seating capacity of 25000, so the child rule node on the true branch of the parent is tested. Since this child node's condition evaluates to false for the case data, the rule evaluation is complete and the last true node returns its conclusion.

The result of all this can be seen in the *Work Queues* screen of the worklist. The *Do Show* workitem is marked as "Suspended" and thus is unable to be selected for starting; while the *ChangeToMidVenue* worklet has been launched and its first workitem, *Cancel Stadium*, is enabled and may be started.

By viewing the log file, you will see that the *ChangeToMidVenue* worklet is being treated by the Exception Service as just another case, and so receives notifications from the Engine for pre-case and pre-item constraint events also.

Start *Cancel Stadium*, accept the default values, and complete. Notice that the worklet has mapped the data attributes and values from the parent case. Next, start the *Book Ent Centre* workitem - by default, it contains the data values mapped from the parent case. Since we are moving the concert to a smaller venue, change the values to match those in Figure 8.38, then complete the workitem.

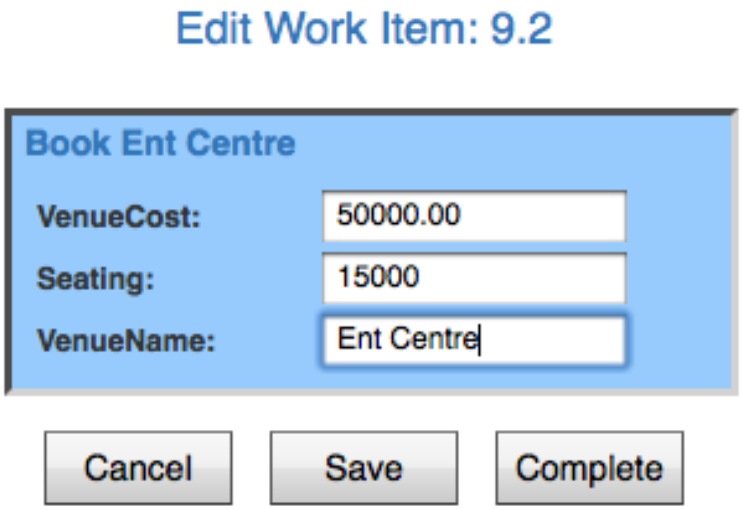


Figure 8.38: The Book Ent Centre Workitem (detail)

The third workitem in the worklet, *Tell Punters*, is designed for the marketing department to advise fans and existing ticket holders of the change in venue.

Start the workitem. Notice that the values here are read-only (since this item is meant to be a notification only, the person assigned does not need to change any values). This is the last workitem in the worklet, so when that is completed, the engine completes the case and notifies the Exception Service of the completion, at which time the service completes the third and final part of the exception handling process, i.e. continuing or unsuspending the *Do Show* workitem so that the parent case can continue.

Back at the *Work Queues* page, the *Do Show* workitem is now shown as enabled and thus is able to be started. Check it out now and notice that the data values entered in the worklet's *Book Ent Centre* workitem have been mapped back to the parent case.

D. Exception: External Trigger Example

It has been stated previously that almost every case instance involves some deviation from its specified process model. Sometimes, events occur completely removed from the actual process model itself, but affect the way the process instance proceeds. Typically, these kinds of events are handled "off-system" so there is no record of them, or the way they were handled, kept for future executions of the process specification.

The Worklet Exception Service allows for such events to be handled on-system by providing a means for exceptions to be raised by users externally to the process itself. The *Organise Concert* specification will again be used to illustrate how external triggers work.

Go to the *Case Mgt* page and launch another instance of the *Organise Concert* specification. Execute and submit the first workitem.

If the Worklet Exception Service has been correctly enabled in the Resource Service (cf. Section 8.2.2), two extra buttons will appear on the *Case Mgt* page: *Raise Exception* and *Reject Worklet*[2]. To raise a case-level external exception, go to the *Case Mgt* screen, and select the *Organise Concert* case from the list of running cases, then click the *Raise Exception* button (Figure 8.39).

The *Raise Case Level Exception* screen is now displayed. This screen is a member of a set of Worklet Service add-in screens for the worklist. Before this screen is displayed, the Exception Service retrieves from the rule set for the selected case the list of existing external exception triggers (if any) for the case's specification. See Figure 8.40 for the list of case-level external triggers defined for the *Organise Concert* specification.

This list contains all of the external triggers either conceived when the specification was first designed or added later as new kinds of exceptional events occurred and were added to the rule set. Notice that at the bottom of the list, the option to add a New External Exception is provided – that option is explained in detail in Walkthrough F.

For this example, let's assume the band has requested some refreshments for backstage. Select that exception trigger and submit the form. When that exception is selected, the conclusion for that trigger's rule is invoked by the service as an exception handling process for the current case. Go to the *Work Queues* form where it can be seen that the parent case has been suspended and the first workitem of the compensatory worklet, *Organise Refreshments*, has been enabled (Figure 8.41).

Organise Refreshments informs the staff member responsible to buy a certain number of bags of M & Ms (first workitem), then to remove all the candies except those of a specified colour, before delivering them to the venue (mapped in from the parent case). Once the worklet has completed, the parent case is continued.

Item-level external exceptions can be raised from the *Work Queue* page by selecting the relevant workitem from the list, then clicking the *Raise Exception* button at the top-right toolbar (the green 'forked arrow' within the tabbed area – see Figure 8.41). You will be taken to the *Raise Item Level Exception* screen where the procedure is identical to that described for case-level exceptions, except that the item-level external exception triggers, if any, will be displayed.

External exceptions can be raised at any time during the execution of a case – the way they are handled may depend on how far the process has progressed (via the defining of appropriate rule tree or trees).

[2]If the two buttons don't appear, the exception service has not been correctly enabled for the Resource Service (cf. Section 8.2.2).

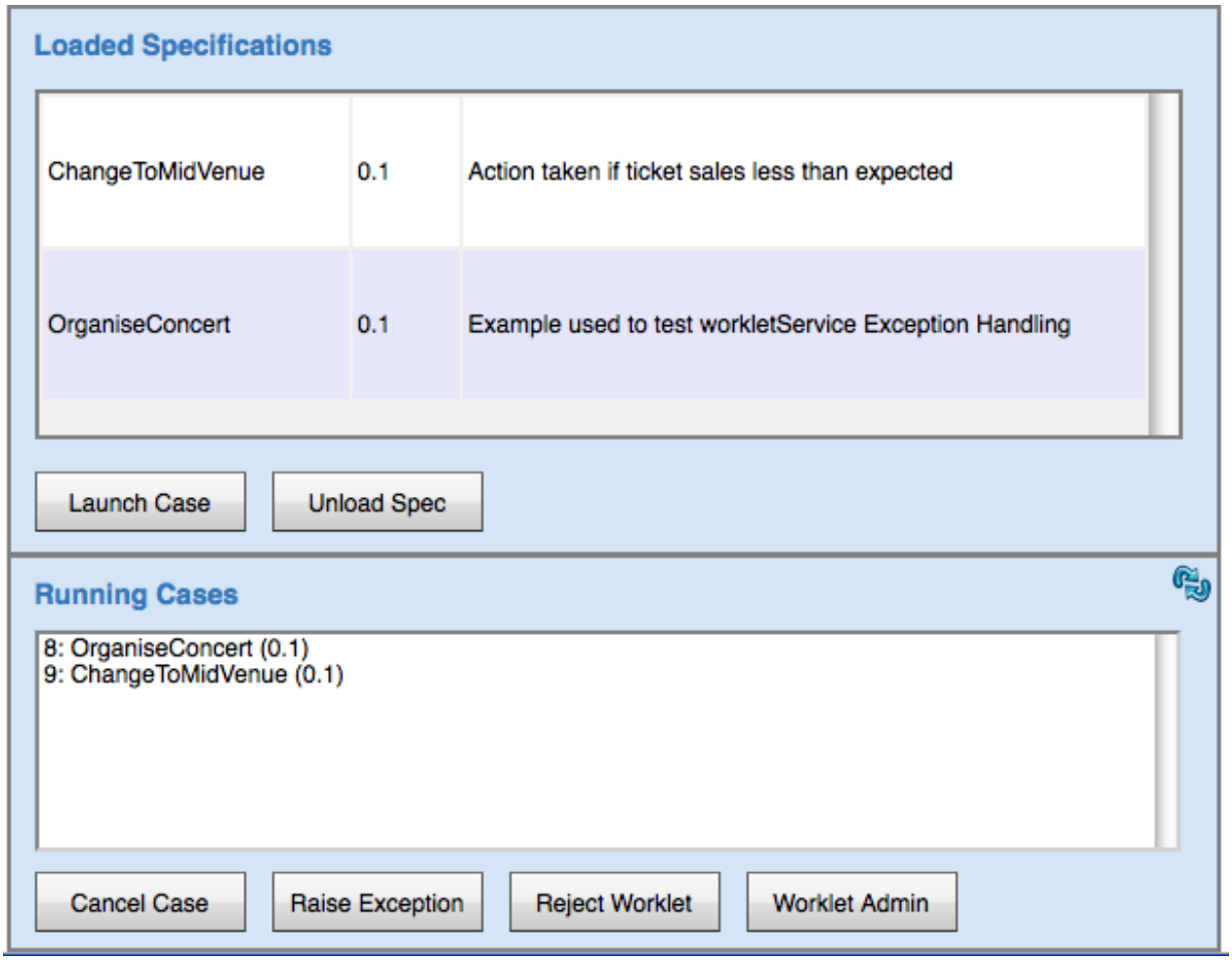


Figure 8.39: Case Mgt Screen, *OrganiseConcert* case running

E. Exception: Timeout Example

When a workitem has an associated timer that times out (expires), the Engine notifies the Exception Service and informs it of all the workitems running in parallel with the timed out item. Thus, rule trees can be defined to handle timeout events for all affected workitems (including the timed out item itself).

The specification *Sales* gives an simple example of how a timeout exception may be handled (Figure 8.42). Upload the specification via the *Case Mgt* screen, and then launch the case.

The first workitem, *Fill Order*, simulates a basic purchase order for a bike. Check out the *Fill Order* workitem, accept the default values, and submit it. Once the order has been filled, the process waits for payment to be received for the order, before it is archived. The *Receive Payment* task has an associated timer, and so waits for some specified time to receive payment. For the purposes of this example, the wait time is set to 5 seconds (Figure 8.43).

While the deadline is reached, the Engine notifies the Exception Service of the timeout event. The timeout tree set is queried for the *Receive Payment* workitem. There is a tree defined for the *Receive Payment* task with a single rule (see Figure 8.44).

Notice the rules condition: "isTimerExpired(this)":

- *isTimerExpired* is an example of a defined function that may be used as (or as part of) conditional expressions in rule nodes.

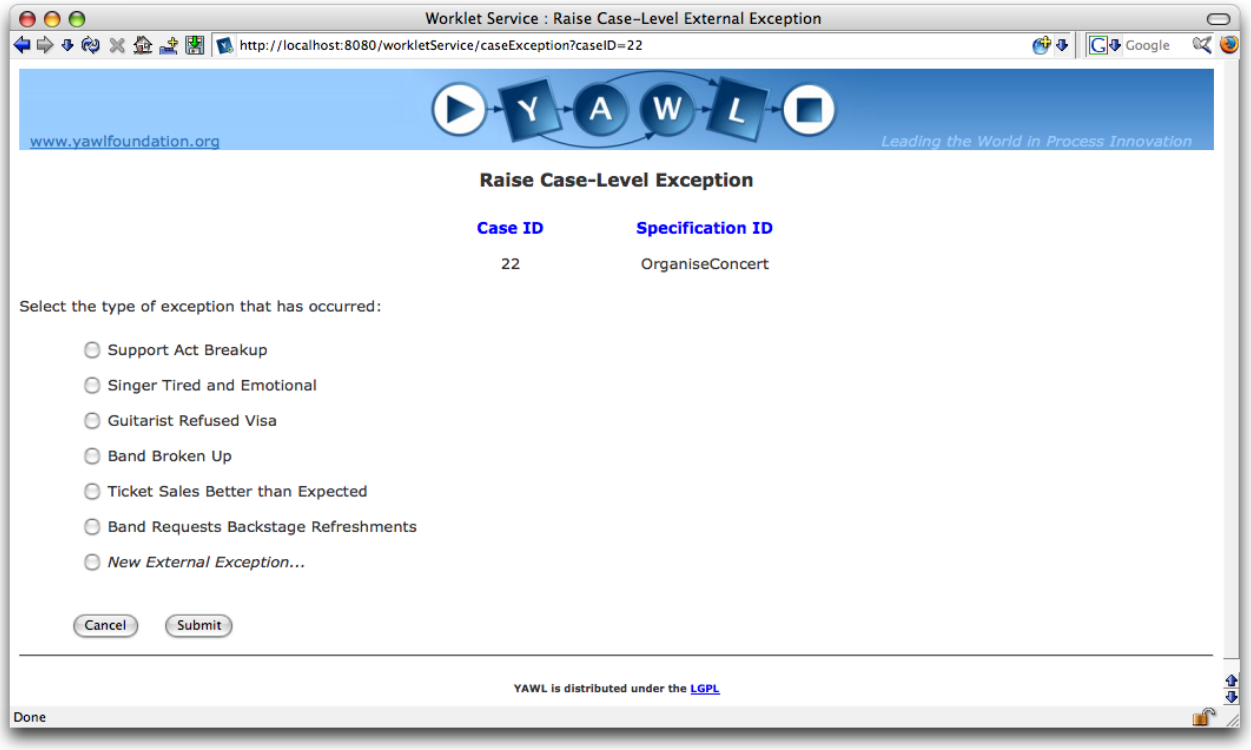


Figure 8.40: Raise Case-Level Exception Screen (Organise Concert example)

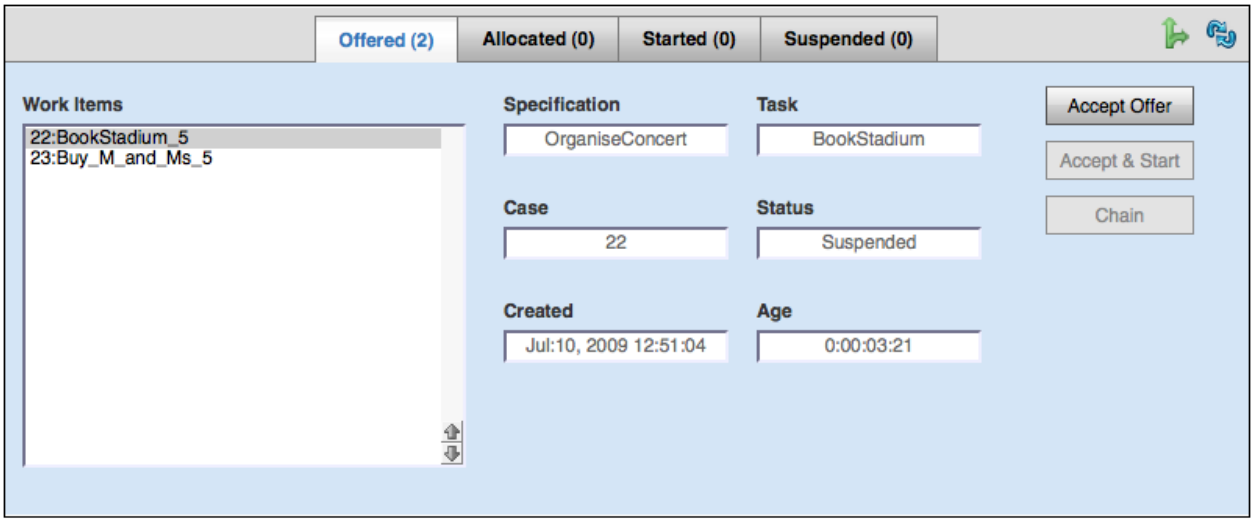


Figure 8.41: Available Work Items after External Exception Raised

- *this* is a special variable created by the Worklet Service that refers to the workitem that the rule is defined for and contains, amongst other things, all of the workitem's data attributes and values.

Tip: The Worklet Service provides developers with an easily extendible class where functions can be defined and then used in conditions. See Section 8.8 for more information about defining functions.

In this case, the condition tests if the timer has expired for the *Receive_Payment* workitem. If it has (thus payment for the order has not yet been received) then the conclusion will be executed as an exception handling process, launching of the worklet *SendReminder*.

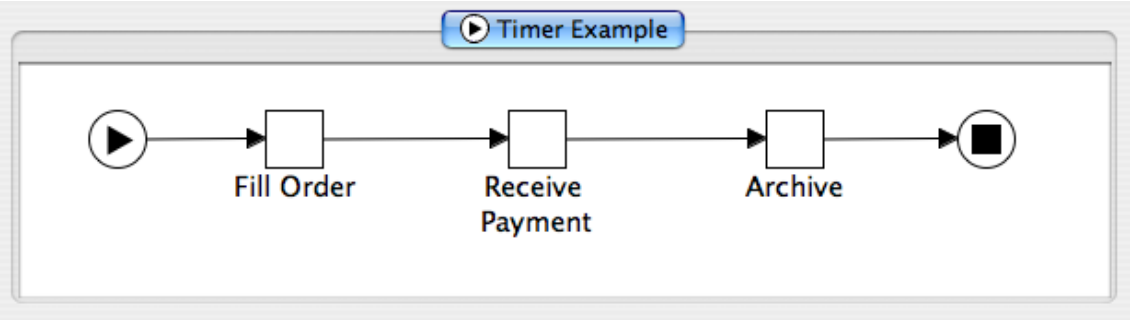


Figure 8.42: The Sales Specification

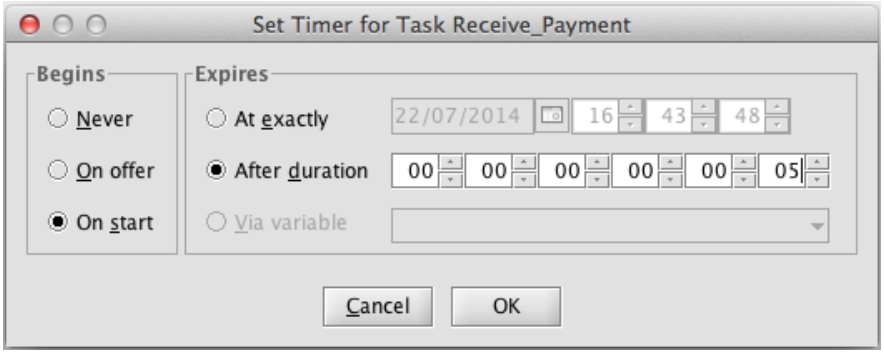


Figure 8.43: The Set Timer Detail dialog for the Receive Payment task

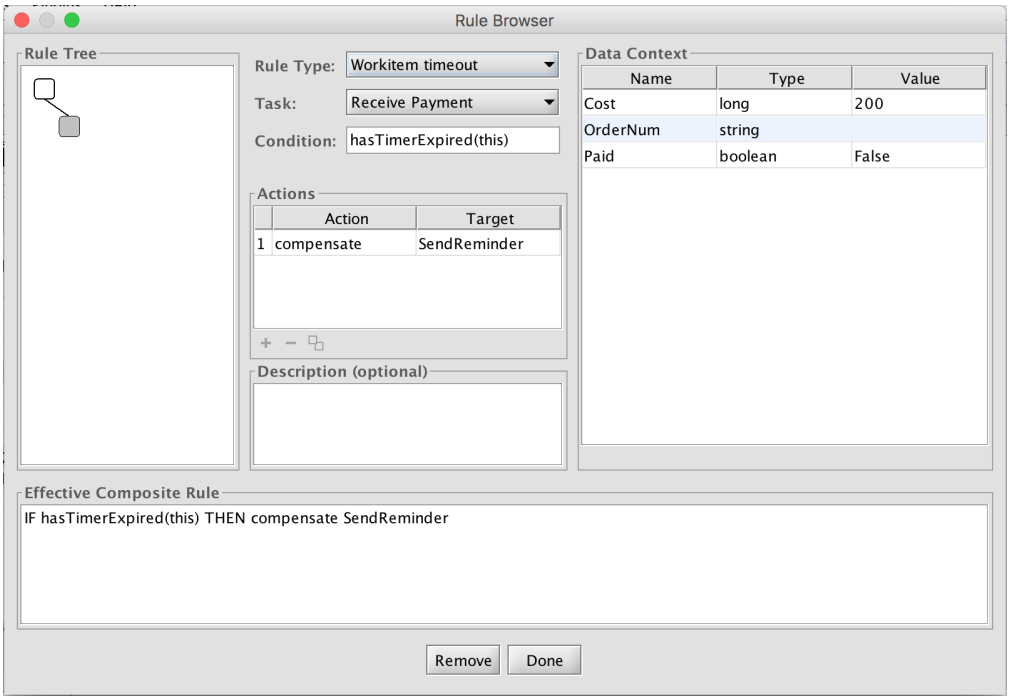


Figure 8.44: Rule Browser Showing Single Timeout Rule for Receive_Payment Task

The *SendReminder* worklet consists of three tasks: *Send Request*, the timer-enabled *Receive Reply*, and *Archive* – again, for the purposes of the example, the timed workitem waits for 5 seconds before timing out. When

the task times out, the Exception Service is notified. There is also a single timeout rule defined for the *Receive Reply* task – its condition is again "isTimerExpired(this)" but this time, the rule's conclusion differs, as can be seen in Figure 8.45.

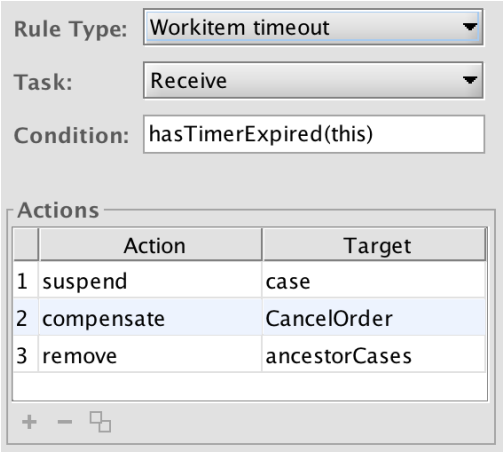


Figure 8.45: Rule detail for Receive Reply

File Cancellation is the first task of the *Cancel Order* worklet. What we now have is a hierarchy of worklets: case (*Sales*) is suspended pending completion of worklet case (*Send Reminder*) which itself is suspended pending completion of worklet case (*Cancel Order*). Worklets can invoke child worklets to any depth. Notice the third part of the handling process: "remove ancestorCases". *Ancestor Cases* are all cases from the current worklet case back up the hierarchy to the original parent case that began the exception chain (as opposed to "allCases" which refers to all currently executing cases of the same specification as the case which generates the exception). So, when the *Cancel Order* worklet completes, the *Send Reminder* case and the original parent *Sales* are both cancelled by the Exception Service.

F. Rejecting a Worklet and/or Raising a New External Exception

The processes involved in rejecting a worklet (launched either as a result of the Selection or the Exception Service) and raising a new external exception (that is, an external exception which has not yet been defined - formally an *unexpected exception*) are virtually identical and so are discussed together in this section.

When the Worklet Service launches a worklet, it selects the most appropriate one based on the current case context and the current rule set for the parent case. As discussed previously in this chapter, there may be occasions where the selected worklet does not best handle the current case's context (perhaps because of a new business rule or a more efficient method of achieving the goal of a task being found). In any event, a worker may choose to reject the worklet that was selected.

IMPORTANT: The rejection of a selected worklet is a legitimate and expected occurrence. Each rejection allows for the addition of a new exception rule (or a rule on the true branch of its parent) thus creating a 'learning' system where all events are handled online. When the new rule is added as a result of the rejection, it will return the correct worklet for every subsequent case that has a similar context. Thus, rejecting a worklet actually refines the rule set for a specification.

To reject a selected worklet, go to the *Case Mgt* screen and select in the list of running cases the worklet you wish to reject. Then, click the *Reject Worklet* button (see Figure 8.39). You will be redirected to the *Reject Worklet Selection* screen, another Worklet Service add-in screen (Figure 8.46). This screen displays the Specification and Case ID for the selected worklet. Enter a proposed title (or name) for the new worklet and an explanation of reason for the rejection (in plain text), and then submit the form.

To raise an unexpected exception at the case-level, follow a similar process at the *Case Mgt* screen, but instead click the *Raise Exception* button. On the *Raise Case-Level Exception* screen (discussed in Walkthrough D), select *New External Exception* from the list and submit the form. You will be redirected to the *Define*

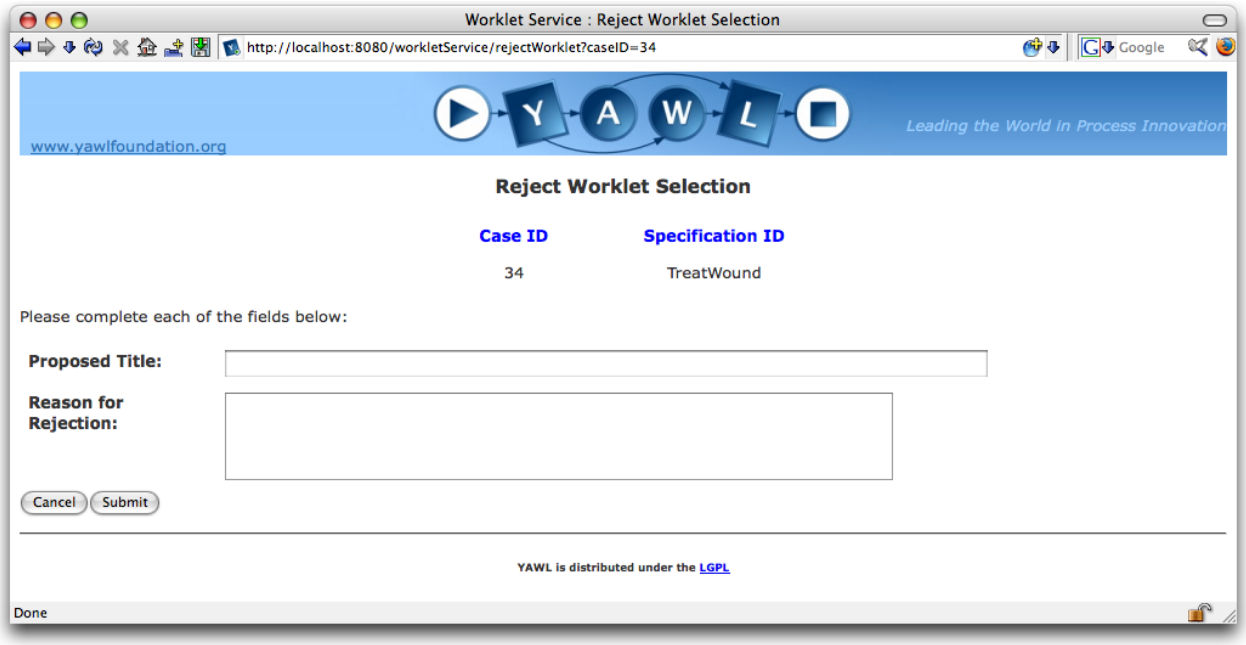


Figure 8.46: Reject Worklet Selection Screen

New Case Level Exception screen. Enter a proposed title, a description of the scenario (what has happened to cause the exception) and a (optionally) a proposal or description of how the new worklet will handle the exception (in plain text), and then submit the form. See Figure 8.47 for an example using the *Organise Concert* specification. Raising an item-level exception is identical, except that the *Raise Exception* button is clicked on the *Work Queue* screen, rather than the *Case Mgt* screen.

The information entered on the form is sent to a Worklet Service Administrator, who will action the rejection or new exception by adding a new rule to the rule set and (optionally) having the Rules Editor notify the service to reselect the new worklet using the updated rule set (see Section 8.6 on the Rules Editor for more detail). The process requires a user with administrator privileges to action the rejection request, rather than allowing all users access to update rule sets.

Note: Rejecting a worklet selection or raising a new unexpected exception will automatically suspend the parent case until such time as the rejection or unexpected exception is actioned by an administrator.

Back at the *Case Mgt* form, if the exception service is enabled, you will notice an extra button in the *Running Cases* panel called *Worklet Admin* (for example Figure 8.39). This button allows administrators to view the current list of outstanding worklet rejections and requests for new exception handlers. It also allows administrators to view the details of each outstanding rejection and exception request and to mark it as completed (removing them from the list) after it has been actioned (Figure 8.48).

8.8 Defining New Functions for Rule Node Conditions

In Section 8.5, we saw how rule conditions could be defined using a combination of arithmetic operators and operands consisting of data attributes and values found in workitems and at the case level of process instances. In *Walkthrough E*, an example of a defined function was given (*isTimerExpired*), using the special variable *this*.

The Worklet Service provides a discrete class that enables developers to extend the availability of such defined functions. That is, a developer may define new functions that can then be used as (or as part of composite) conditional expressions in rule nodes. That class is called *RdrConditionFunctions* - the source code for the class can be found in the *org.yawlfoundation.yawl.worklet.support* package. Currently, this class

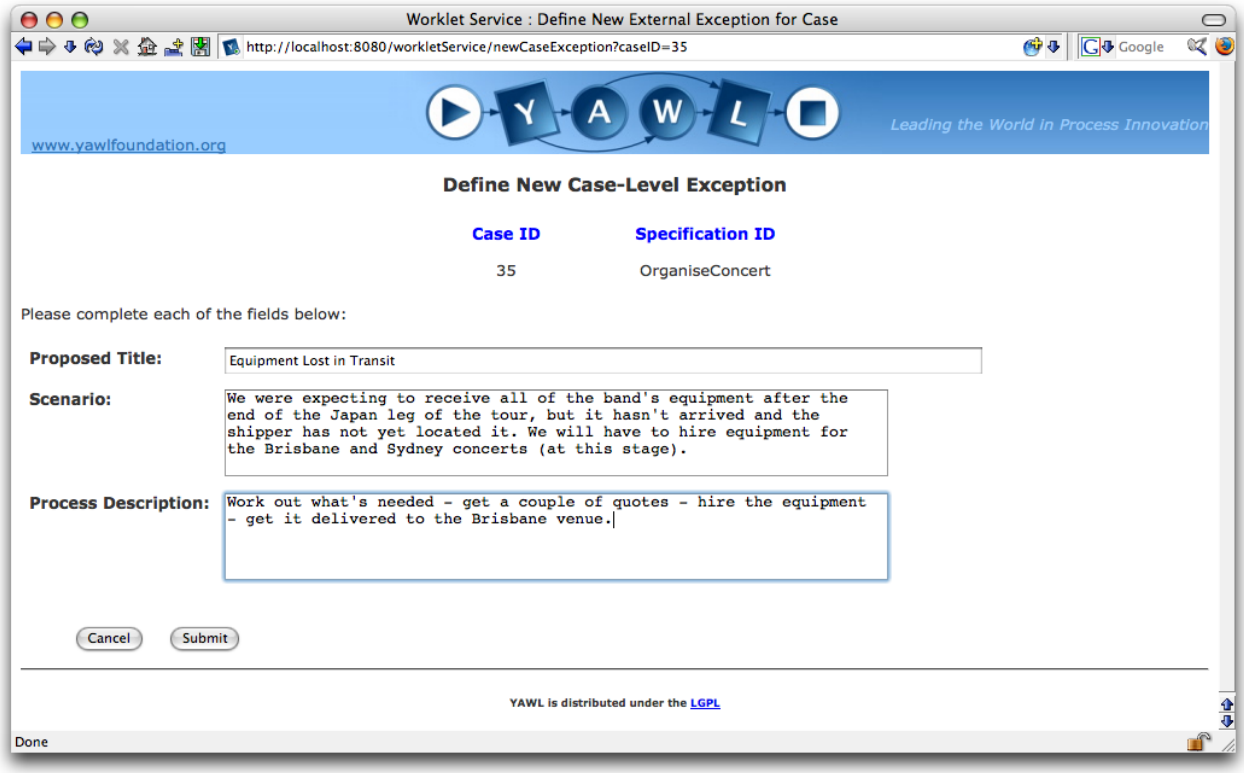


Figure 8.47: Example of a New Case-Level Exception Definition

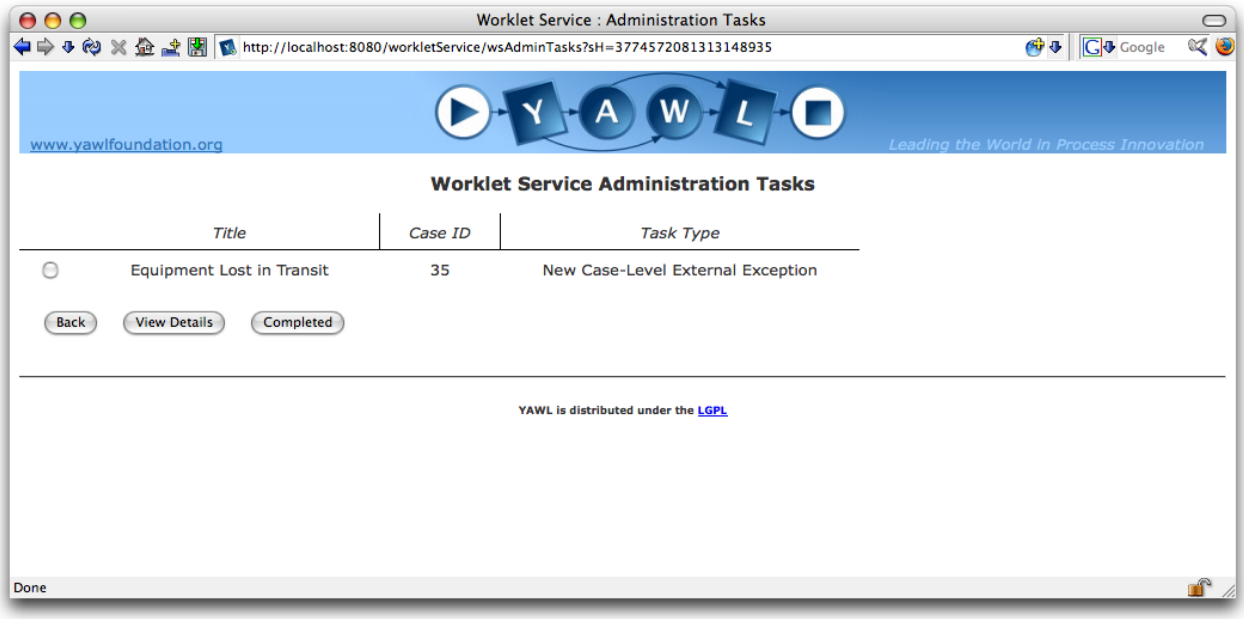


Figure 8.48: Administration Tasks Screen (detail)

contains a small number of examples to give developers an indication of the kinds of things you can do with the class and how to create your own functions.

The class code is split into four sections:

- Header;
- Execute Method;
- Function Definitions; and
- Implementation.

To successfully add a function, these rules must be followed:

1. Create the function (i.e. a normal Java method definition) and add it to the 'function definitions' section of the code. Ensure the function:
   - is declared with the access modifier keywords *private static*; and
   - returns a value of *String* type.
2. Add the function's name added to the array of '_functionNames' in the header section of the code.
3. Add a mapping for the function in the 'execute' method, using the examples as a guide.

Once the function is added, it can be used in any rule's conditional expression.

Let's use the *max* function as a simple example walkthrough (to be read in conjunction with the source code for the class). The first thing to do is define the actual function in the function definition section. The entire function is shown in Figure 8.49.

```
private static String max(int x, int y) {
    if (x >= y) return String.valueOf(x) ;
    else return String.valueOf(y) ;
}
```

Figure 8.49: Max function

Notice that the function has been declared as *private static* and returns a *String* value. Next, the name of the function, *max*, has to be added as a *String* value to the *_functionNames* array in the header section of the code, see Figure 8.50.

```
// add the name of each defined function here
public static final String[] _functionNames = { "cost",
                                                "max",
                                                "min",
                                                "isNotCompleted",
                                                "hasTimerExpired",
                                                "today"} ;
```

Figure 8.50: Adding the name

Finally, we need to map the function name to the *execute* method, which acts as the interface between the class's functions and the Worklet Service. The *execute* method receives as arguments the name of the function to execute, and a *HashMap* containing the functions parameters (all are passed as *String* values). The *execute* method is essentially an *if ... else if* block, the sub-blocks of which call the actual functions defined. The section of the *execute* method for the *max* function is shown in Figure 8.51.

The first line checks to see if the name of the function passed to the *execute* method is "max". If it is, the parameters passed with the function (as *String* values in the *HashMap* "args") are converted to integer values

```
else if (name.equalsIgnoreCase("max")) {
    int x = getArgAsInt(args, "x");
    int y = getArgAsInt(args, "y");
    return max(x, y);
}
```

Figure 8.51: Execute method for the max function

and finally the *max* function is called - its return value is passed back from the *execute* method to the calling Worklet Service.

The *getArgsAsInt* method called in the snippet above is defined in the *Implementation* section of the class's code. It is here that you can create private methods that carry out the external work of the any functions defined, as required.

The definition of *isNotCompleted* is slightly different, since the parameter passed is the special variable *this*. The *this* variable is essentially a *WorkItemRecord* that contains descriptors of the workitem the rule is testing, enabling developers to write methods that test the values in the variable and act on those values accordingly. If it is for a case-level rule, this contains the case data for the instance invoking the rule. Both versions of this are passed as a string-ified JDOM Element format. See the YAWL source code for more details of the *WorkItemRecord class*, if required.

What the execute methods sub-block for the *isNotCompleted* function looks like is shown in Figure 8.52.

```
private static String isNotCompleted(String itemInfo) {
    Element eItem = JDOMUtil.stringToElement(itemInfo);
    String status = eItem.getChildText("status");
    return String.valueOf(! isFinishedStatus(status) );
}
```

Figure 8.52: Execute method's sub-block for isNotCompleted function

The block gets *this* variable as a *String* from the "args" *HashMap* and then calls the actual *isNotCompleted* method (see Figure 8.53).

```
public static String execute(String name, Map<String, String> args) {
    String taskInfo = args.get("this");
    if (name.equalsIgnoreCase("isNotCompleted")) {
        return isNotCompleted(taskInfo);
    }
    else if (name.equalsIgnoreCase("hasTimerExpired")) {
        return hasTimerExpired(taskInfo);
    }
```

Figure 8.53: Calling the actual isNotCompleted method

Notice again that the function has been declared as *private static* and returns a *String* value. The first line of the function converts the *String* passed into the function to a JDOM Element, and then extracts from that Element a value for "status" (being one of the data attributes contained in the *this* variable). It then calls another method, defined in the *Implementation* section, called *isFinishedStatus* (see Figure 8.54).

All methods defined in the Implementation section must also be declared as private static methods - however, they can have any return type, so long as the value returned from the *execute* method back to the Worklet Service has been converted to a *String* value.

```
/** returns true if the status passed is one of the completed statuses */
private static boolean isFinishedStatus(String status) {
    return status.equals(WorkItemRecord.statusComplete) ||
           status.equals(WorkItemRecord.statusForcedComplete) ||
           status.equals(WorkItemRecord.statusFailed) ;
}
```

Figure 8.54: Definition of method isFinishedStatus

Of course, you are not restricted to querying the *this* variable as a *WorkItemRecord* - it is passed simply as a JDOM Element that has been converted to a *String* and so can be queried via a number of different methods.

The objective of the *RdrConditionFunctions* class is to allow developers to easily extend the capabilities of the Worklet Service by providing the means to test for other things in the conditional expressions of rule nodes other than the process instance's data attributes and values. It is envisaged that the class's functions can be extended into areas such as process mining, querying resource logs and external data sets.

Chapter 9

The Proclet Service

In this chapter we introduce the Inter-Workflow Service of YAWL. First, we provide in Section 9.1 an elaborate introduction to the Proclets framework and how inter-workflow support is achieved. Afterwards, in Section 9.1.4, its discussed how inter-workflow support is supported within YAWL by means of the Inter-Workflow Service.

9.1 Inter-Workflow Support

9.1.1 Introduction

Classical workflow notations primarily support monolithic processes. They are able to describe the life-cycle of individual cases and allow for hierarchical decomposition. Unfortunately, real-life processes are fragmented and are often composed of separate but intertwined life-cycles running at different speeds and coping with different levels of granularity. In order to provide the desired support, Proclets are an interesting means of modeling and executing complex and intertwined real-life processes. Proclets are *lightweight interacting processes* that can be used to divide complex entangled processes into simple fragments and, in doing so, place increased emphasis on interaction-related aspects of workflows. Proclets aim to address the following problems that existing workflow approaches are currently facing:

- Models need to be *artificially flattened* and are unable to account for the mix of *different perspectives and granularities* that coexist in real-life processes.
- Cases need to be *straightjacketed into a monolithic workflow* while it is more natural to see processes as intertwined loosely-coupled processes
- *One-to-many* and *many-to-many* relationships that exist between entities in a workflow cannot be captured.
- It is difficult to model interactions between processes.

Proclets were one of the first modeling languages to acknowledge above mentioned problems and are part of the Proclet framework which has been described in detail by van der Aalst et al. [6, 7]. In this chapter, the original Proclet framework and an extension to it will be presented. In the original framework, the concept of a Proclet, a performative, a port which has a cardinality and multiplicity, and a channel have already been introduced. Here, additional concepts such as an interaction point, an internal interaction, an external interaction, and an interaction graph will be introduced. Via interaction points, at design time, possible interactions between Proclet classes can be modeled without the need to define complex pre- and post-conditions. Subsequently, at run-time they allow users to nominate interactions between Proclet instances. Furthermore, via an interaction graph, interactions between Proclet instances and their state are recorded.

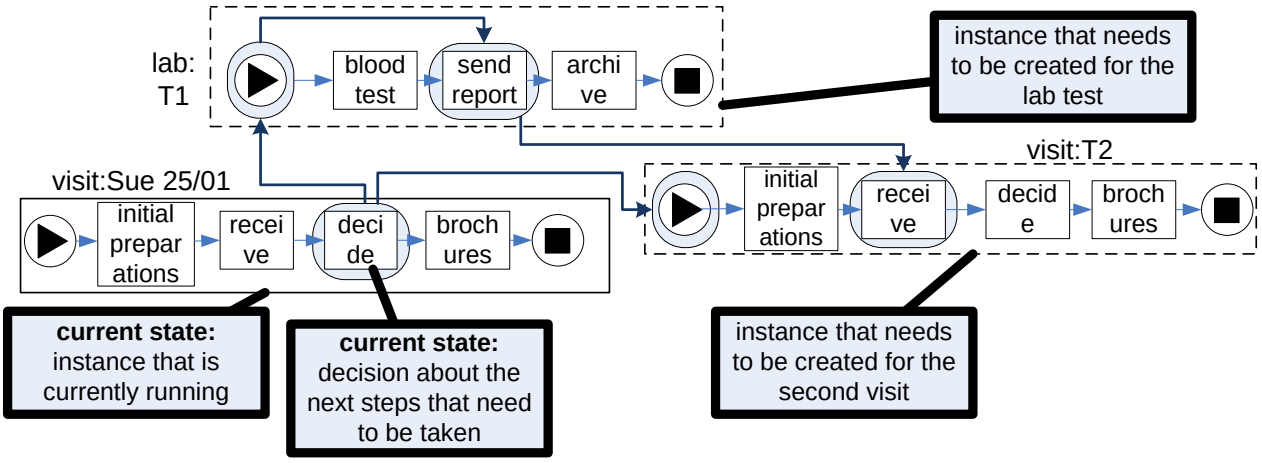


Figure 9.1: The first scenario. In this scenario, for patient "Sue" it is decided during the first visit that a lab test and a second visit are required.

9.1.2 The Proclet Framework

In this section, we discuss how Proclets provide a framework for modeling and executing workflows. First, the most important concepts of the framework will be introduced in Section 9.1.2. Then, in subsequent sections, particular aspects of the framework will be addressed.

Concepts

In this section, we discuss the main concepts of the Proclet framework. This will be done using two scenarios. One scenario is rather simple while the other one is more complex. These scenarios introduce the mechanisms that relate to the various concepts that are employed.

First Scenario Before introducing the framework, we first present the scenario which is shown in Figure 9.1. In this scenario, we schematically depict the process that needs to be followed by patient "Sue". Currently, Sue is in the process of having a first visit (fragment "visit:Sue 25/01"). As indicated by the outgoing arcs from the "decide" task, the doctor decides during the "decide" task that a subsequent visit is necessary (fragment "visit:T2") and that a lab test needs to be taken (fragment "lab:T1"). As the last two fragments need to be created in the future, the fragments for them are visualized with a dotted rectangle around them. Also, their instance identifier starts with a "T". As a subsequent action when creating an instance for the lab test, the result of the lab test needs to be used as input for the second visit. This is indicated by the arc leading from the input condition of the "lab" fragment to the "send report" task and the arc leading from the "send report" task to the "receive" task of the second visit.

Based on the scenario discussed above, we start our introduction of the Proclet framework. More specifically a framework that is centered around the notion of ***Proclets***. There is a distinction between a Proclet class and a Proclet instance. A *Proclet class* can best be seen as a process definition which describes which tasks need to be executed and in which order. For a Proclet class, instances can be created and destroyed. One instance is called a *Proclet instance*. For the definition of a Proclet class, a choice can be made between multiple graphical languages. Here, we use a graphical language based on the YAWL language [11], however, other languages, such as Petri Nets [1] or EPCs [3], can also be used. With regard to the choice of a graphical language, some limitations apply. First of all, Proclet instances need to have a state and they need to support the notion of a task. Second, a Proclet class needs to be sound, i.e., it must satisfy basic correctness requirements such as absence of deadlocks, proper termination, etc. [4].

In order to have interactions and collaboration among Proclets, *interaction points*, *channels*, *ports*, and *performatives* are important. The meaning of these will be discussed below. In addition, we describe how a Proclet

class and instances of it are defined.

- A Proclet class has a ***unique name***. In the same way, an instance of a Proclet class has an unique identifier.
- Proclet instances interact with each other via ***channels***. A channel can be used to send a ***performative*** to an individual Proclet instance or to a group of Proclet instances.
- A performative is a specific kind of message with several attributes which is exchanged between one or more Proclets. Two important attributes are the "sender" and "set of receivers" attributes. The *sender* attribute contains the identifier of the Proclet instance creating the performative. The *set of receivers* attribute contains the identifiers of the Proclet instances receiving the performative, i.e. a list of recipients. Additional attributes will be discussed in more detail later (Section 9.1.3).
- A Proclet class has ***ports***. Performatives are sent and received via these ports in order for a Proclet instance to be able to interact with other Proclet instances. A port is either an incoming or an outgoing port. Each outgoing port is connected with exactly one incoming port. We call such a connection, an ***external interaction***. Furthermore, every port is connected to one ***interaction point***. An interaction point represents a specific point in the Proclet class at which interactions with other Proclet classes may take place, i.e. via the associated ports performatives may be sent and received. An interaction point may be linked to an input condition and a task.
- Moreover, a port has two attributes.

  First, the *cardinality* specifies the number of recipients of performatives exchanged via the port. An $*$ denotes an arbitrary number of recipients, $+$ at least one recipient, 1 precisely one recipient, and $?$ denotes no or just one recipient. Note that by definition an input port has cardinality 1.

  Second, the *multiplicity* specifies the number of performatives exchanged via the port during the lifetime of an instance of the class. In a similar fashion to the cardinality, an $*$ denotes that an arbitrary number of performatives are exchanged, $+$ at least one, 1 precisely one, and $?$ denotes that either one or no performatives are exchanged.
- For an interaction point having only incoming ports, it may be desirable that the receipt of an individual performative is followed by the subsequent sending of a performative later in the process (e.g. the creation of a lab test needs to be followed by the execution of a task in the same process which sends the result of the test to the desired Proclet instance). Therefore, an interaction point with only incoming ports may be connected with an interaction point which has only outgoing ports. Such a connection is called an ***internal interaction***.

The above mentioned concepts are illustrated in Figure 9.2. Based on the first scenario, in Figure 9.2a, two Proclet classes are shown together with their interaction points, ports, and external interactions. In Figure 9.2b, a class diagram is shown containing the two Proclet classes. First, as can be seen in Figure 9.2a, the "visit" Proclet class models all the tasks related to a visit of the patient, whereas the "lab" Proclet class does the same for a lab test. The "decide" step of the "visit" Proclet class has an interaction point with two outgoing ports. One outgoing port leads to the interaction point that belongs to the input condition of the "lab" Proclet class. Sending a performative to the incoming port of this interaction point results in the creation of an instance of the "lab" Proclet class. Similarly, sending a performative via the second outgoing port of the "decide" task results in the creation of an instance of the "visit" Proclet class. As indicated by cardinality * for the two outgoing ports of the "decide" task multiple instances of the "lab" Proclet class and multiple instances of the "visit" Proclet class may be initiated. The multiplicity of the two ports is ? which means that a performative may optionally be sent in order to create an instance of the "lab" and "visit" Proclet class. Finally, performatives can be sent from the "send report" task to the "receive" task modeling that the result of a lab test may be used as input for a patient visit. The cardinality 1 and multiplicity ? of the outgoing port of the "send report" task indicate that it is optional to send a performative to one "visit" Proclet instance. In a similar fashion, the cardinality 1 and multiplicity * of the incoming port of the "receive" task indicate that performatives may optionally be received from the "send report" task.

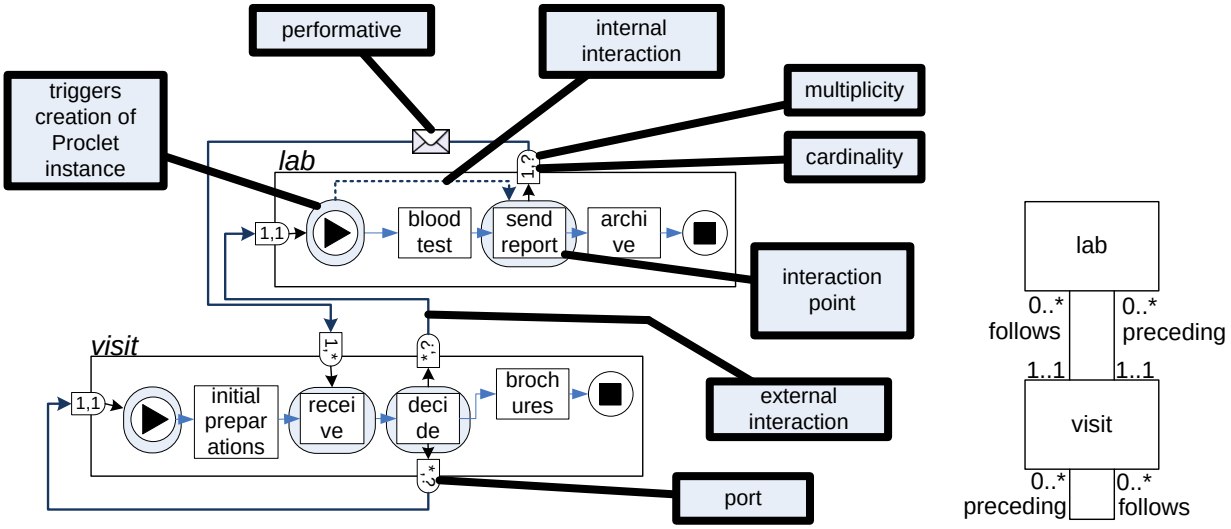


Figure 9.2: Based on the first scenario, the concepts of the Proclet framework introduced so far are illustrated. That is, two Proclet classes are modeled. For both of them, interaction points, channels, ports, and performatives are indicated.

However, although performatives can be sent to multiple receivers, there is still the issue that it needs to be *controlled which specific Proclet instance or instances a performative is sent to*. For example, in Figure 9.2, where an instance exists of the "lab" Proclet class and the "send report" task is executed, it is questionable to which "visit" Proclet instance a performative is sent. In particular, if we want to achieve the behavior for "Sue" which is defined during the execution of the "decide" task in the first scenario (which is visualized in Figure 9.1), the following is required for the two Proclet models shown in Figure 9.2

- One Proclet instance exists for the first visit which has "visit:Sue 25/01" as its instance identifier.
- A performative is sent from the "decide" task of the first visit to the initial condition of the "visit" Proclet such that one instance is created for the second visit of "Sue".
- A performative is sent from the "decide" task of the first visit to the initial condition of the "lab" Proclet such that one instance is created for the desired lab test for "Sue".
- The creation of an instance of the lab test should be followed by the execution of the "send report" task such that a performative is sent from that task to the "receive" task of the second visit.

In addition to this, for the above mentioned interactions it is important that it is known whether they have already taken place, i.e. the state of them needs to be known. For example, in order for the "receive" task of the second visit to take place it is important to know whether the performative from the "send report" task has already been received.

Entities and Interaction Graphs

In order to be able to precisely specify the interactions that need to take place for "Sue" and their current state, we need to introduce two additional concepts. The fist concept is called an *entity*. An entity is an object that exists in conjunction with existing and future Proclet instances. Examples of an entity are a patient, a claim, or a software product that needs to be developed. So, "Sue" can be an entity. For an entity, tasks in multiple Proclet instances need to be performed. In order for these tasks to be performed in the desired order, specific interactions are required between existing and future Proclet instances. Note that this also may also involve a sequence of interactions among multiple Proclet instances.

In order for an entity to store the interactions that need to take place between existing and future Proclet instances and their state, we introduce a so-called *interaction graph*. An interaction graph belongs to a specific entity and consists of *interaction nodes* and *interaction arcs*. An *interaction node* refers to an interaction point of a Proclet instance for which *one or more internal or external interactions will take place*, i.e. an instance of an interaction point. So, an interaction node is a triple in which the first value refers to the identifier of the Proclet class, the second value refers to the identifier of the Proclet instance, and the third value refers to the identifier of the interaction point for which one or more interactions take place. In turn, an *interaction arc* refers to an interaction, either internal or external, that needs to occur between two interaction points of a Proclet instance. In this way, the direction of the arc in the graph is the same as the direction of the arc for the associated internal or external interaction.

In Figure 9.3, for the first scenario, the corresponding interaction graph is given for entity "Sue". First, in Figure 9.3a, the instances for the first visit, second visit, and lab test are shown. However, they are now modeled using the terminology of the Proclet framework, i.e. using interaction points, ports, and so on. For example, the "decide" task of the first visit has two outgoing ports illustrating the performatives that will be sent in order to create an instance for the lab and the second visit. Also, for the first visit (Proclet instance with identifier "visit:Sue 25/01"), the "decide" task is currently executed. As a result, the "initial preparations" and "receive" tasks are already executed which is indicated by the ticks.

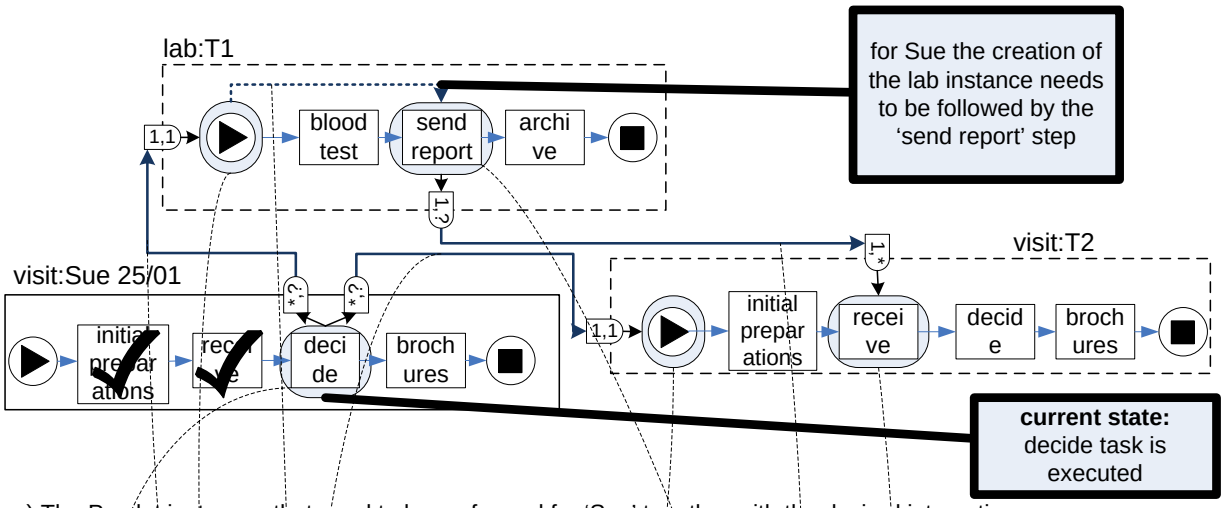
As a result of executing the "decide" task for the first visit, which necessitates interactions with existing and future Proclet instances, an interaction graph is created for entity "Sue". The graph is shown in Figure 9.3b. There are five interaction nodes and four interaction arcs. Note that by using dotted arcs, nodes of the interaction graph are linked with their corresponding interaction points in a Proclet instance. Additionally, via dotted arcs, arcs of the interaction graph are linked with their corresponding internal or external interactions. The meaning of each arc for the entity "Sue" is as follows:

- (visit,Sue 25/01,decide) $\rightarrow$ (lab,T1,create): from the "decide" task of the "visit" Proclet class with instance identifier "Sue 25/01", a performative is sent in order to create an instance of the lab Proclet class. As the lab instance still needs to be created a temporary instance identifier is used for it (i.e. T1). Note that the arc refers to an external interaction. For presentation reasons, input and output ports are not shown in an interaction graph. Instead, for an external interaction, the respective interaction nodes are immediately connected via an arc.
- (visit,Sue 25/01,decide) $\rightarrow$ (visit,T2,create): similar to the previous arc. This time an instance of the "visit" Proclet class needs to be created which represents the second visit. Note that also for the second visit a temporary instance identifier is used (i.e. "T2").
- (lab,T1,create) $\rightarrow$ (lab,T1,send report): the creation of an instance for the "lab" Proclet class needs to result in a subsequent interaction. This is represented by an internal interaction for which no performatives will be sent. Note that the subsequent interaction is the sending of a performative, starting from the "send report" task of the same instance to the "receive" task of the second visit.
- (lab,T1,send report) $\rightarrow$ (visit,T2,receive): from the "send report" task of the "lab" Proclet instance, a performative needs to be sent which is received by the "receive" task of the future "visit" Proclet instance for the second visit which has the temporary instance identifier "T2".

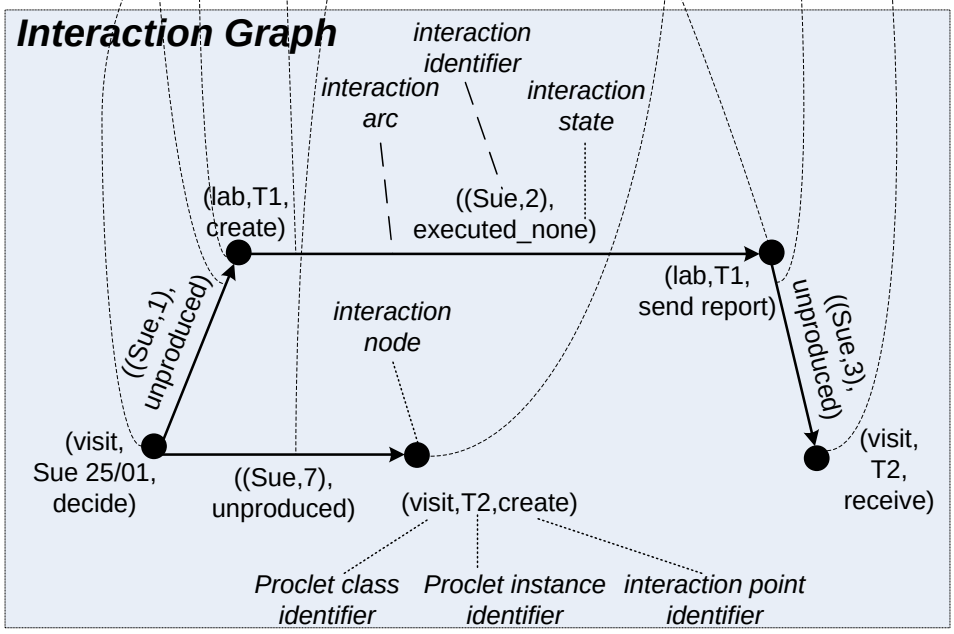
Obviously, the interaction graph of entity "Sue" captures all the interactions that need to take place between future and existing Proclet instances. In other words, the entity "Sue" is the linking pin between the three Proclet instances.

As indicated before, in an interaction graph we also save the state of the interactions for an entity. Therefore, every arc in the interaction graph has an *interaction identifier* and an *interaction state*.

The *interaction identifier* is an identifier in which the first value relates to the entity itself and of which the second value relates to a unique identifier for the interaction. These interaction identifiers allow for keeping track of the state of external interactions for entities, i.e. performatives that are exchanged. Additionally, in order to realize the latter, an additional attribute is added to a performative called the *set of interaction identifiers*. For the interaction arcs for which a performative is sent, the associated interaction identifier is added to this set. More details on this will be provided later.

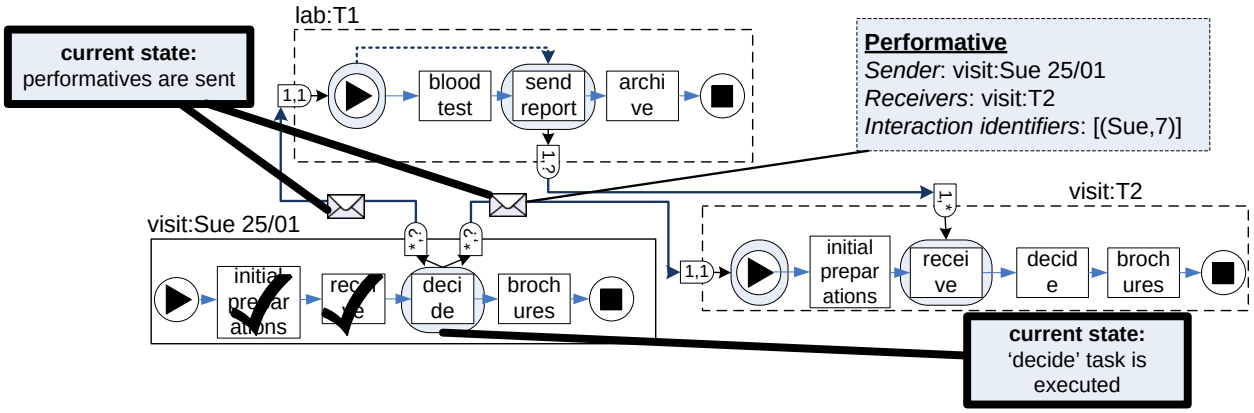


a) The Proclet instances that need to be performed for 'Sue' together with the desired interactions

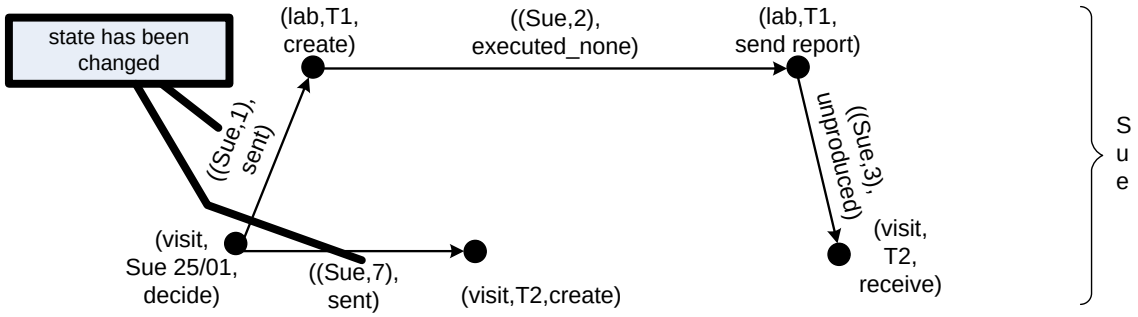


b) Interaction graph defined during execution of the 'decide' task. The graph saves the Proclet instances that need to be performed for 'Sue' together with the desired interactions.

Figure 9.3: For entity "Sue", using the Proclet terminology introduced so far, it is shown how the existing and future Proclet instances need to interact (Figure a). Additionally, for entity "Sue" the associated interaction graph is shown (Figure b).



a) The Proclet instances that need to be performed for 'Sue' together with the desired interactions



b) Current state of the interaction graph for 'Sue'. As a result of sending out performatives, due to the execution of the 'decide' task, the state of the outgoing interaction arcs of the '(visit,Sue,25/01)' node are changed into 'sent'

Figure 9.4: As a result of executing the "decide" task, two performatives are sent. As a consequence, the interaction graph is updated.

Next, the *interaction state* of an arc stores the specific state of an interaction for the respective entity. For example, for an external interaction it can be captured whether a performative has been sent. For an internal interaction, it indicates whether the task that is linked to the interaction point has already executed or not. In total, for an arc referring to an external interaction we distinguish four different states and for an arc referring to an internal interaction. In the example below, for the first scenario, tasks will be executed for different Proclet instances. In this way, it can be seen which performatives are exchanged. Moreover, it explains how the arcs of an interaction graph are updated and which states we distinguish. In this way, the "mechanisms" of an interaction graph can be illustrated along with the "mechanisms" of an interaction point, internal interaction, and external interaction.

Executing the First Scenario For the first scenario, visualized in Figure 9.1, tasks for the first visit, lab test, and second visit will be performed. In Figure 9.3 we show the current state of the Proclet instances and the current state of the interaction graph for entity "Sue". Tasks that are completed are indicated by a check mark.

Step 1:

In Figure 9.3, the "decide" task of the first visit is currently executing and an interaction graph is created. In Figure 9.4, the next step is visualized. That is, for the "decide" task that is executing, in the interaction graph it can be seen that two interactions need to take place. So, two performatives need to be sent in order to create an instance of the "lab" Proclet class and an instance of the "visit" Proclet class. As an example, we see the performative that is sent to create an instance of the "visit" Proclet class. The sender of the performative is the Proclet instance that is currently executed ("visit:25/01"). The receiver of the performative is the instance of the "visit" Proclet class which has temporary identifier "T2" as it still needs to be initiated. As an interaction identifier we see that "(Sue,7)" is added.

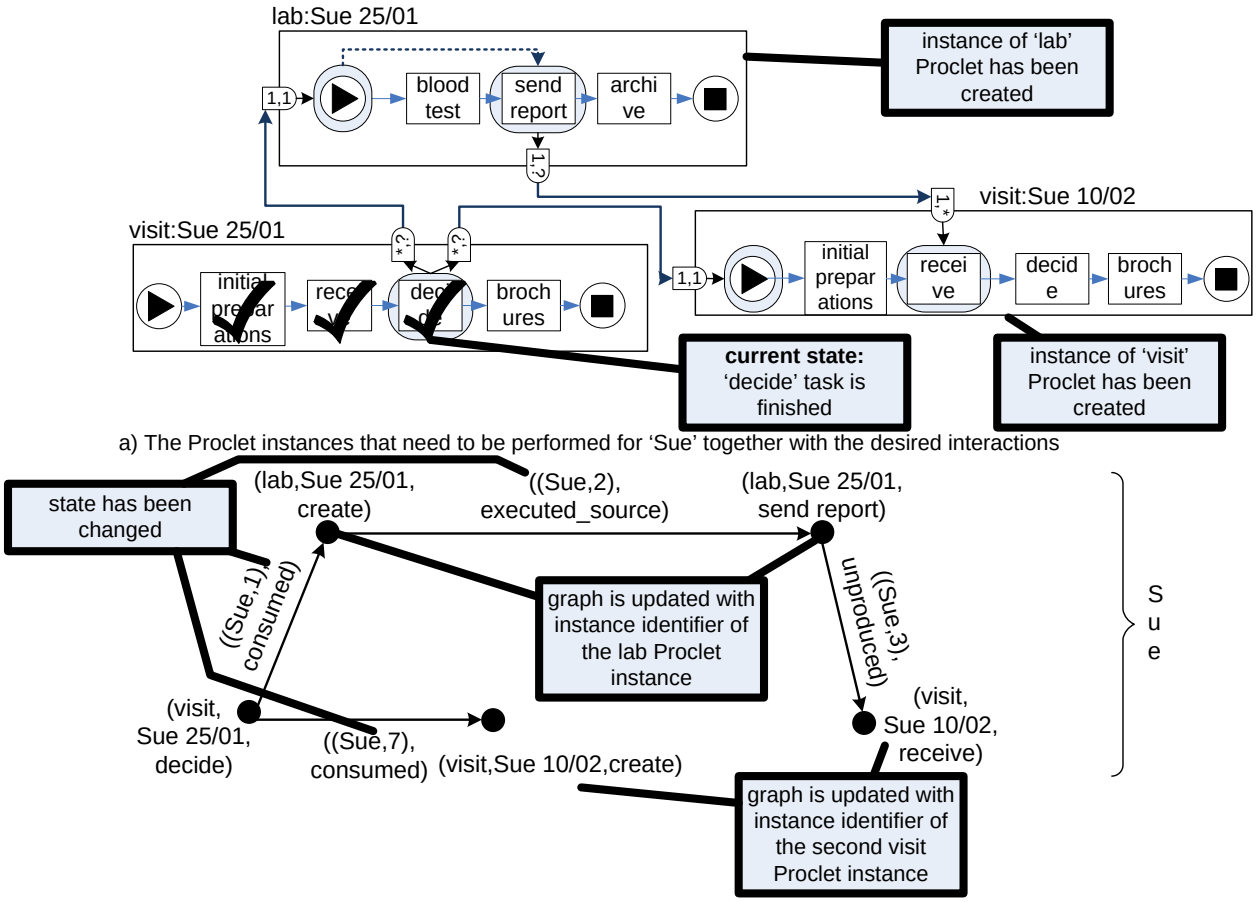


Figure 9.5: As a result of receiving the two performatives, instances for the "lab" and "visit" Proclet classes are created. As a consequence, the interaction graph is updated.

As a result, in the interaction graph for entity "Sue", we see that the interaction state of the arc leading from the "(visit,Sue 25/01,decide)" node to the "(lab,T1,create)" node has been changed to "sent". The same can be observed for the arc leading from the "(visit,Sue 25/01,decide)" node to the "(visit,T2,create)" node. This is due to the fact that a performative has been sent for that interaction. That is, the sender and receiver of the performative match with the sender and receiver of the associated interaction arc in the graph. Also, the performative contains the interaction identifier of the arc for which it is sent.

Step 2:

The next step is shown in Figure 9.5. The receival of the two performatives has resulted in the creation of an instance for the "lab" Proclet class and the "visit" Proclet class as well. Instead of instance identifier "T1" the instance of the "lab" Proclet class now has the identifier "lab:25/01". As a consequence, in the interaction graph for "Sue", the interaction nodes referring to the "lab" instance have been updated with the new instance identifier. Moreover, the interaction state of the arc leading from the "(visit,Sue 25/01,decide)" node to the "(lab,25/01,create)" node has been changed to "consumed". That is, a performative has been received for that interaction arc which resulted in the creation of an instance of a Proclet class, i.e. the performative can be considered to be "consumed" as its receival led to a certain action. Additionally, it can be seen that the interaction state of the arc from the "(lab,25/01,create)" node to the "(lab,25/01,send report)" node has been changed to "executed source". As an instance of the "lab" Proclet class has been

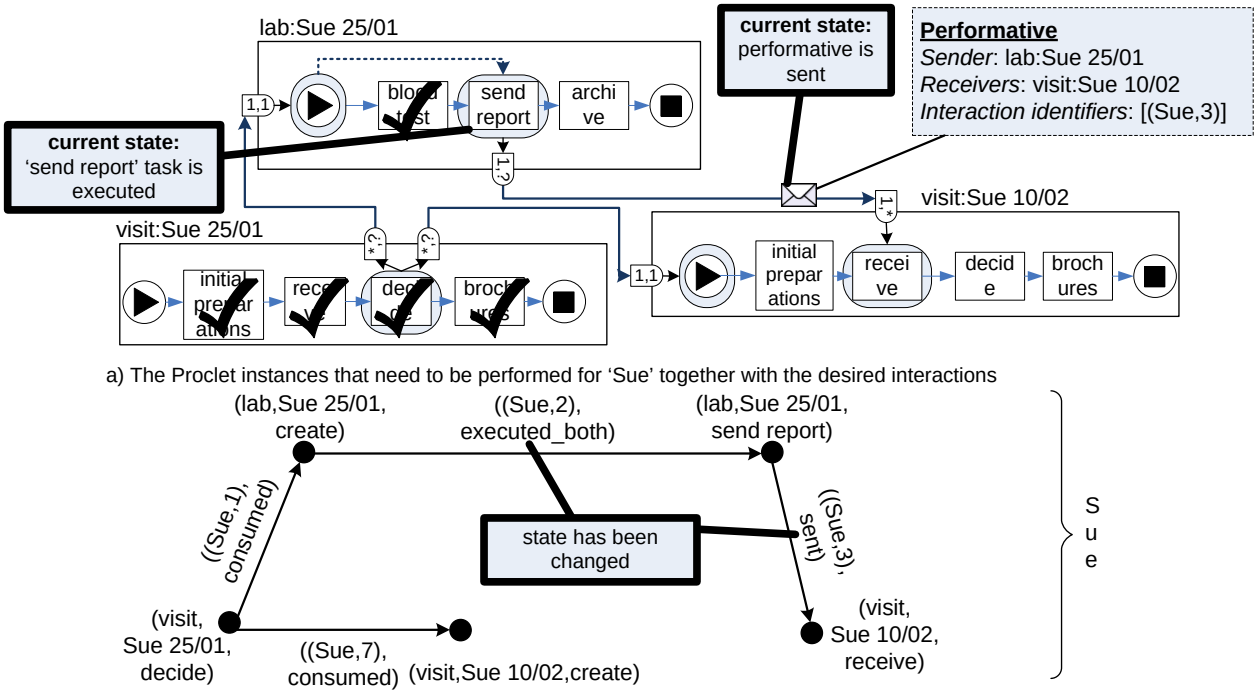


Figure 9.6: The "send report" task of the "lab" Proclet instance is executed. This results into a performative that is sent to the "receive" task of the "visit" Proclet instance for the second visit. As a consequence, the interaction graph is updated.

created, for this internal interaction the source interaction point has been executed.

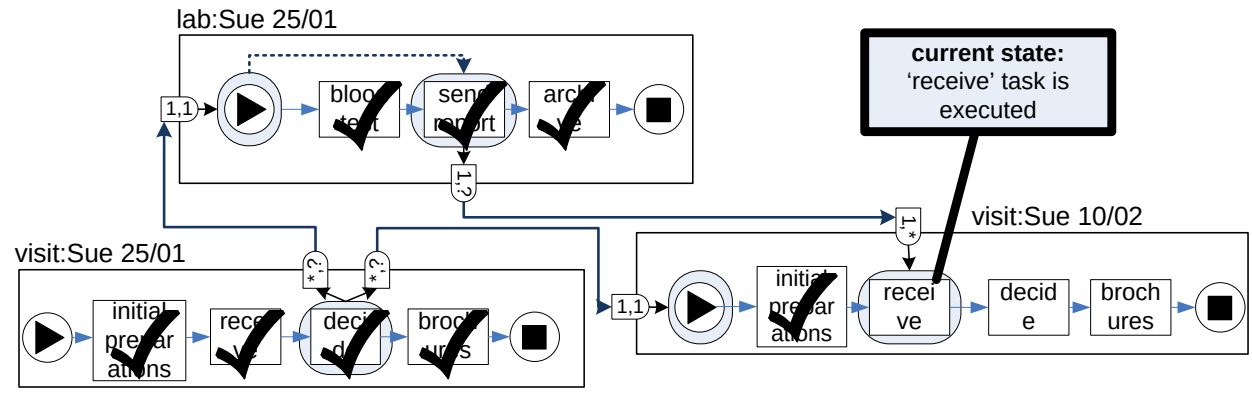
For the instance of the "visit" Proclet class that has been created, similar remarks can be made. As can be seen in the graph, the interaction state of the associated arc has been changed to "consumed" too. Moreover, instead of the instance identifier "T2", the instance of the "visit" Proclet class has now instance identifier "Sue 10/02".

Step 3:

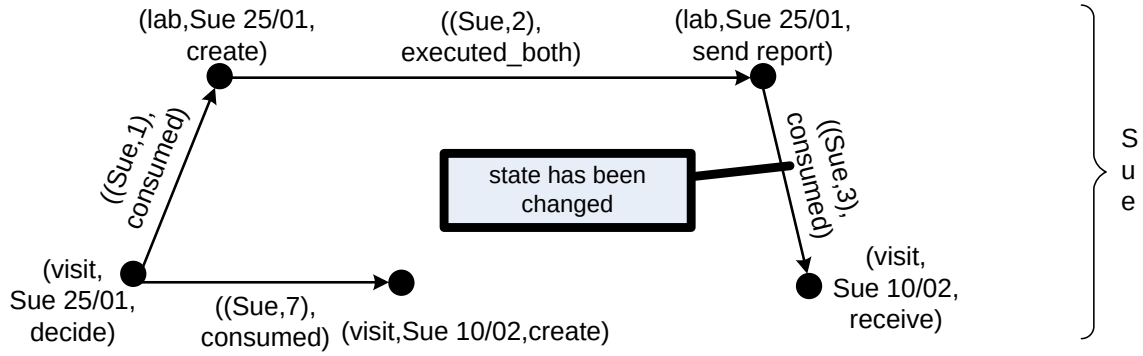
As the next step in the scenario, the "blood test" task of the "lab" Proclet instance is completed. As a result, the "send report" task may be performed. The result of performing the task can be seen in Figure 9.6. For, the arc in the interaction graph leading from the "(lab,25/01,send report)" node to the "(visit,Sue 10/02,receive)" node, a performative is sent to the "receive" task of the "visit" Proclet instance with the instance identifier "Sue 10/02". As a result, the interaction state of the arc has been updated to "sent". Moreover, the performative that is sent is visualized in the figure. That is, the sender and receiver of the performative match with the sender ("lab:Sue 25/01") and receiver ("visit:Sue 10/02") of the associated interaction arc in the graph. Also, the performative contains the interaction identifier of the arc for which it is sent ("(Sue,3)").

Furthermore, it can be seen that the interaction state of the arc from the "(lab,Sue 25/01,create)" node to the "(lab,Sue 25/01,send report)" node has been changed to "executed both". Due to the execution of the "send report" task both the source and destination interaction node of this internal interaction have now been executed. Consequently, the interaction state is updated to "executed both".

In the case where the arc did not have the state "executed source", i.e. the interaction point which is connected to the input condition was not executed, the "send report" task could not be executed. This is due to the fact that the meaning of an internal interaction is that first the source interaction point is executed, which is linked with either a task or an input condition, and then the task which belongs to the destination



a) The Proclet instances that need to be performed for 'Sue' together with the desired interactions



b) Updated interaction graph for 'Sue'. As a result of the receival of a performative for the 'receive' task of the second visit, the state of the incoming interaction arc of the '(visit,Sue 10/02,receive)' node is changed into 'consumed'

Figure 9.7: The "receive" task of the "visit" Proclet instance is executed. As a performative has been sent which contains "(Sue,3)" as interaction identifier, the task may be completed. Subsequently, the interaction graph is updated.

interaction point is executed.

Step 4:

The last step of the first scenario is to start executing steps for the second visit. As can be seen in Figure 9.7, the "receive" task is currently executing. For this task, we find in the interaction graph, the arc leading from the "(lab,Sue 25/01,send report)" node to the "(visit,Sue 10/02,receive)" node. This indicates that for completing this task it is necessary that a performative is received which contains "(Sue,3)" as an interaction identifier. As this performative has been sent as result of executing the "send report" task, the "receive" task may be completed. Consequently, the interaction state of the arc is updated to "consumed" indicating that the required performative was available and that it has been consumed in order to complete the task. Note that if the performative had not been available, it would not be possible to complete the task. So, although it is possible to complete the task according to the process definition, it is still necessary to wait till all required performatives are received. However, an exception to this rule is possible. This will be discussed later in Section 9.1.2.

Second Scenario Previously, we have considered a simple scenario for which we have shown some of the mechanisms of an interaction graph. In this, we only considered steps that are done for an individual patient. Now, as a follow-up we consider a more complex scenario in which we demonstrate that the framework can also deal with Proclet classes that operate at different levels of granularity. Therefore, for the second scenario we deal with one Proclet class in which an *individual patient* is taken as the basis for case. Additionally, we deal with another Proclet class in which a *group of patients* is taken as the basis for a case.

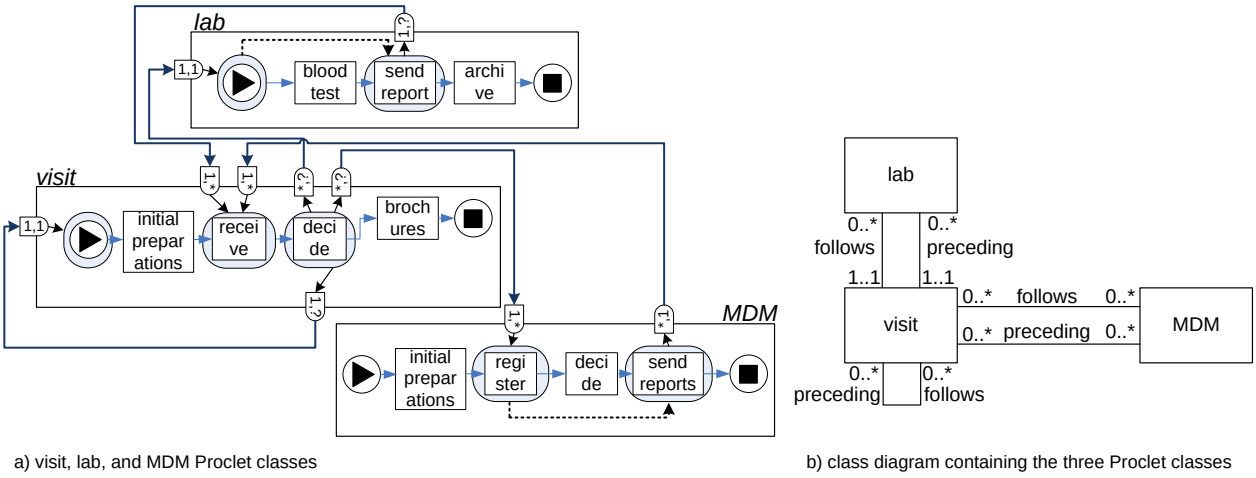


Figure 9.8: The Proclet classes that are used for the second scenario.

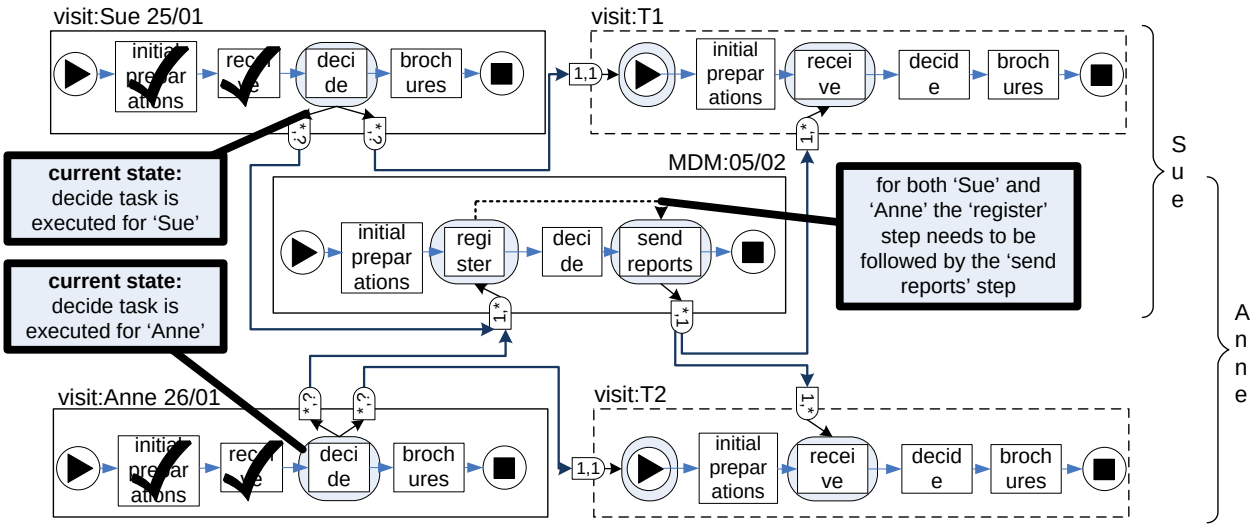
In Figure 9.8, we can see three Proclet classes. The "visit" and "lab" Proclet classes have already been discussed in the first scenario. The "MDM" Proclet class is concerned with a weekly meeting in which gynecological oncology doctors discuss the medical status of multiple patients. For this meeting, multiple patients may be registered (task "register"). This can be seen by the multiplicity * of the associated incoming port which indicates that multiple performatives may be received. During the "decide" task, the patients that are registered are discussed. Finally, for each patient that is discussed, a report may be sent out (task "send reports"). This is also represented by cardinality * of the associated outgoing port which indicates that a performative may be multicast to multiple "visit" Proclet instances.

Note that there is an internal interaction defined from the "register" task to the "send reports" task. This internal interaction has the meaning that for every patient that is registered, it can be decided that the subsequent report needs to be sent to a specific Proclet instance (e.g. the second visit of the patient).

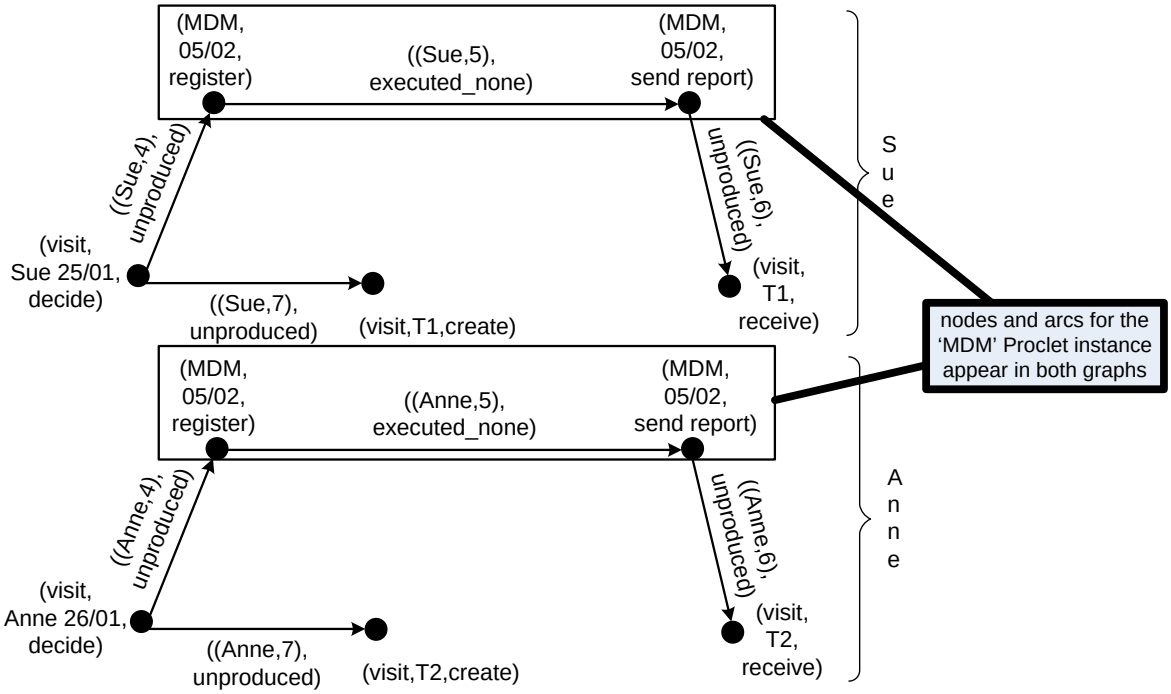
Obviously, the "MDM" Proclet class operates at a different level of granularity. That is, for the "MDM" Proclet class a group of patients is taken as the basis of an individual case whereas for the other Proclet classes this does not hold. For example, for the "visit" Proclet class an individual patient is taken as the basis for a case. This can also be seen in the Figure 9.8b which shows a class diagram containing the Proclet classes.

The scenario that will be executed is visualized in Figure 9.9 and deals with two different patients. Here, for both "Sue" and "Anne" tasks from multiple Proclet instances will be performed. In particular, for "Sue" during execution of the "decide" task at the first visit (Proclet instance "visit:Sue 25/01") it is decided that a subsequent visit is required (Proclet class "visit" with temporary instance identifier "visit:T1"). Moreover, "Sue" needs to be discussed during the multidisciplinary meeting for which an instance already exists with the identifier "MDM:05/02". Afterwards, the report needs to be used as input for the second visit.

For "Anne" exactly the same outcome is decided during execution of the "decide" task. So, she is also discussed during the multidisciplinary meeting for which an instance already exists with the identifier "MDM:05/02". However, for her the instance that exists for the first visit has the identifier "Anne 26/01" and the instance for the second visit has the temporary identifier "T2". As a result of executing the "decide" task for "Sue", which necessitates interactions with existing and future Proclet instances, an entity is created called "Sue" for which subsequently an interaction graph is created. For "Anne" exactly the same is done during the execution of the "decide" task but now an entity called "Anne" is created and a separate interaction graph. For both entities the corresponding interaction graphs are shown in Figure 9.9. These interaction graphs are very similar to the first scenario. However, instead of a lab test, now for both patients an interaction with the "register" task of the "MDM:05/02" Proclet instance is required. This is followed by the execution of the "send reports" task after which a performative is sent to the "receive" task of the second visit for both "Sue" and "Anne". Note that as no performatives have yet been sent, each arc in



a) The Proclet instances that need to be performed for 'Sue' and 'Anne'. For both the desired interactions are shown



b) Interaction graphs showing the Proclet instances that need to be performed for 'Sue' and 'Anne' together with the desired interactions

Figure 9.9: For both "Sue" and "Anne" it is shown how existing and future Proclet instances need to interact (Figure a). Additionally, for both, the interaction graph that is created during the execution of the "decide" task is shown (Figure b).

the interaction graph has either the interaction state "unproduced" or "executed none". It is important to mention that as both patients are discussed during the multidisciplinary meeting, similar interaction nodes and similar interaction arcs for the "MDM:05/02" Proclet instance appear in the graphs of both "Sue" and "Anne". Below, for both patients, tasks will be executed for different Proclet instances. In this way, the impact of having the same interaction node in multiple graphs can be illustrated.

Step 1:

In Figure 9.10, the result of executing the "decide" task for both "Sue" and "Anne" is shown. Performatives are sent in order to create for both of them an instance of the "visit" Proclet class. Additionally, performatives are sent in order to register both of them for the multidisciplinary meeting. Note that as for "Sue" and "Anne" their "decide" task is executed in different Proclet instances, the sending of performatives and updating the interaction graph for them occurs completely independently from each other.

Subsequently, both interaction graphs are updated as expected. So, all the outgoing arcs of an interaction node that refer to a "decide" task that has been executed, have received the "sent" state.

Step 2:

The next step is shown in Figure 9.11. As a consequence of receiving the required performatives, an instance of the "visit" Proclet class with instance identifier "Sue 10/02" has been initiated for "Sue" and an instance has been created for "Anne" with instance identifier "Anne 12/02". Note that in the interaction graphs, the related interaction nodes and the related interaction arcs for them are updated accordingly, i.e. they have the state "consumed". However, for the "register" task, the performatives have not been consumed yet as the preceding "initial preparations" task is still not complete. Therefore, the interaction arcs corresponding to these performatives still have the state "sent".

Step 3:

Subsequently, for the "MDM" Proclet instance we perform the "initial preparations" task. Afterwards, the "register" task may be executed. The result can be see in Figure 9.12. For the "register" task of the "MDM" Proclet with the instance identifier "05/02" we find in the interaction graph of both "Sue" and "Anne" an arc leading to the "(MDM,05/02,register)" node. In order to complete the task, for entity "Sue" a performative should be received with "(Sue,4)" as the interaction identifier and for entity "Anne" a performative should be received with "(Anne,4)" as the interaction identifier. As can be checked in Figure 9.11, these performatives have been sent. So, the task may be executed. For the receival of the performative which contains interaction identifier "(Sue,4)", the associated interaction arc with the same identifier is updated to "consumed" in the interaction graph of "Sue". For the performative which contains interaction identifier "(Anne,4)", the same is done for the interaction graph of "Anne".

For the "register" task, both for "Sue" and "Anne", there is an internal interaction defined for which the "(MDM,05/02,register)" interaction node is the source node. As this task will be completed, both for the interaction graphs of entities "Sue" and "Rose", the state of the corresponding outgoing interaction arc of the node is set to "executed source". Obviously, in order to perform a task it may be necessary to inspect and update multiple interaction graphs.

Note that if we abstract from the interaction identifier of an arc, then the internal interaction arc from the "(MDM,05/02,register)" node to the
"(MDM,05/02,send report)" node is the same in both the interaction graphs of "Sue" and "Anne". For these arcs it is important to see that the state is always the same and always changes simultaneously. This is due to the fact that both the head and the tail of these arcs refer to the same interaction node.

Step 4:

As the next step, the "send reports" task of the "MDM" Proclet instance is executed. First of all, as can be seen in Figure 9.13, for the "send reports" task, both for "Sue" and "Anne" an internal interaction is defined for which the "(MDM,05/02,send reports)" interaction node is the destination node. As we have seen earlier, the "register" task, which is the source of the two internal interactions, has already been executed, i.e. the state of the interaction arcs is "executed source". So, it is permissible to execute the task. Consequently, the state of both arcs is simultaneously updated to "executed both".

Note that if the task which belongs to the source node of an internal interaction has not yet been executed, it is not possible to execute the task which belongs to the destination node of the internal interaction. This

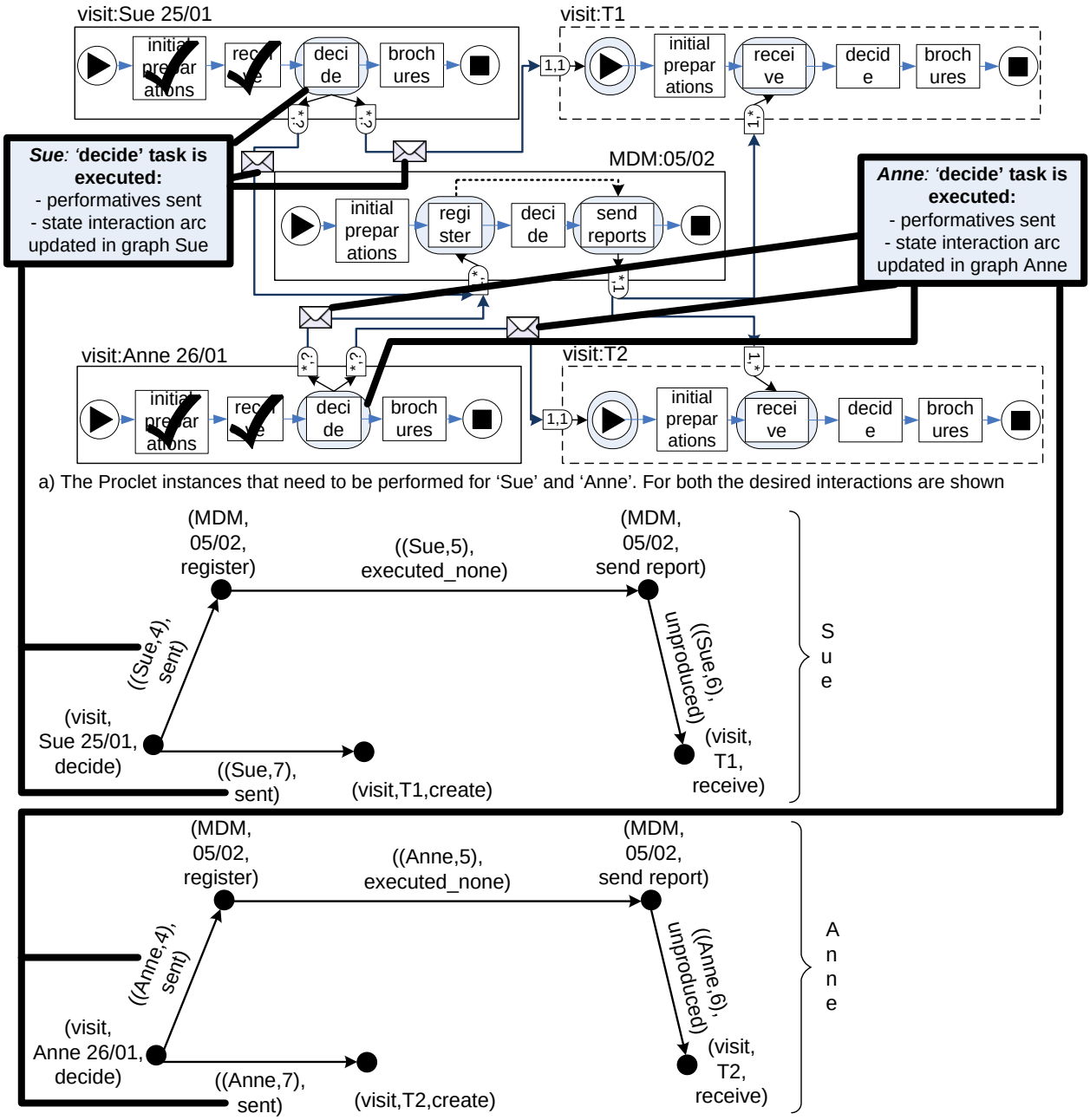


Figure 9.10: As a result of executing the "decide" task for both "Sue" and "Anne" in total four performatives are sent. As a consequence, both interaction graphs are updated.

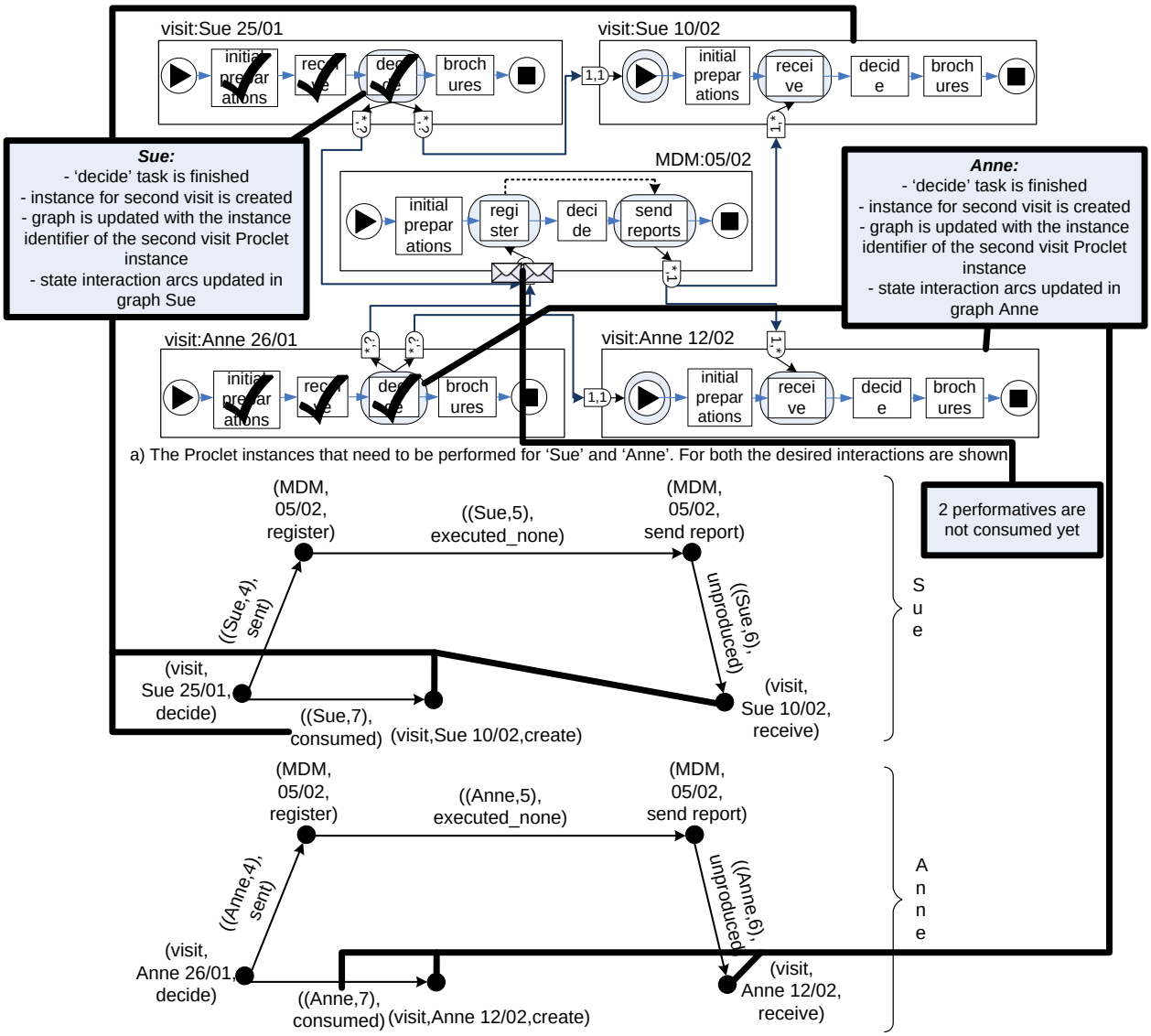
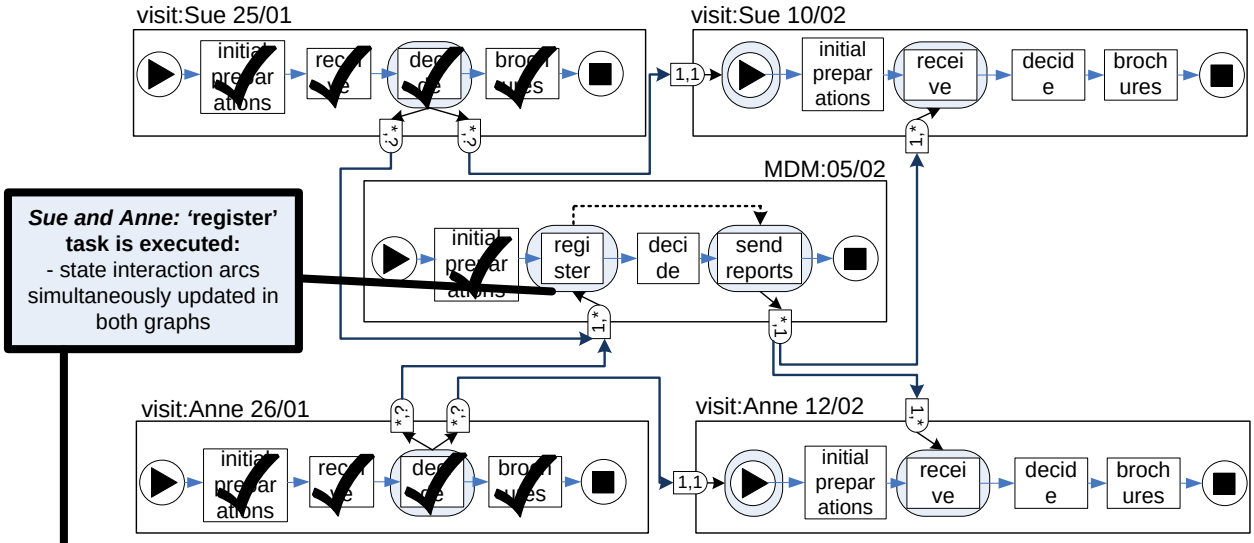
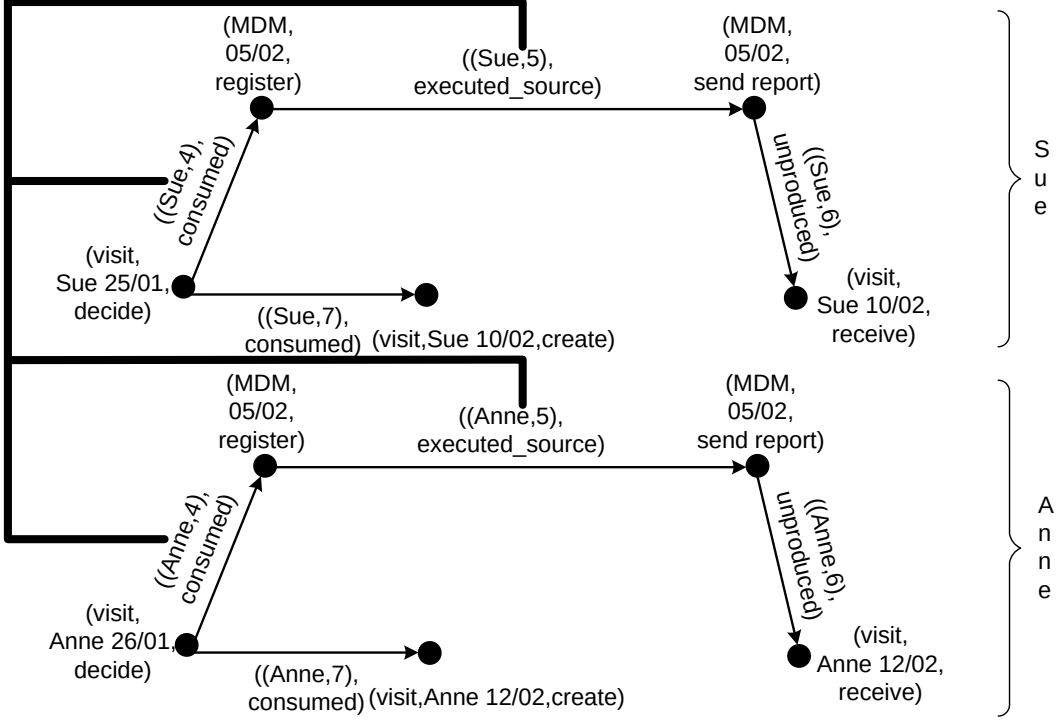


Figure 9.11: Both for "Sue" and "Anne" an instance of the "visit" Proclet class is created. For the "receive" task of the "MDM" proclet instance, the relevant performatives are not consumed yet.



a) The Proclet instances that need to be performed for 'Sue' and 'Anne'. For both the desired interactions are shown



b) Updated interaction graphs for 'Sue' and 'Anne'. As a result of executing the 'register' task, the state of the incoming interaction arc of the '(MDM,05/02,register)' node in both graphs is simultaneously updated to 'consumed'. Moreover, in both graphs the state of the outgoing interaction arc of the '(MDM,05/02,register)' node is simultaneously updated to 'executed source'

Figure 9.12: The "register" task of the multidisciplinary meeting is performed. As performatives have been sent earlier which contain either "(Sue,4)" or "(Anne,4)" as interaction identifiers, the task may be completed. Subsequently, the interaction graphs for both are updated.

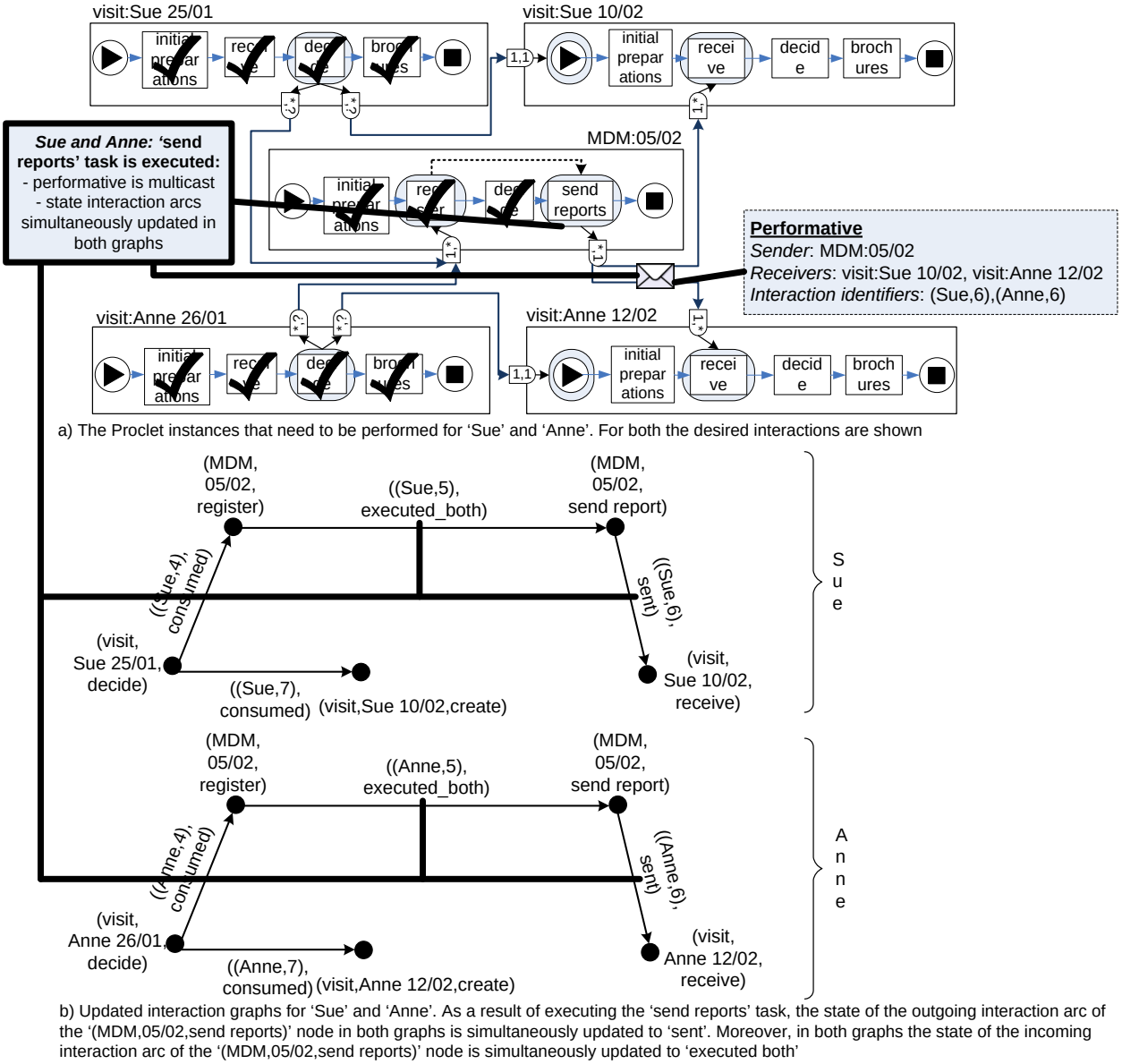


Figure 9.13: The "send reports" task of the multidisciplinary meeting is performed. As a result, one performative is multicast to the "visit:Sue 10/02" Proclet instance and the "visit:Anne 12/02" Proclet instance. Subsequently, the interaction graphs for both are updated.

is due to the fact that the meaning of an internal interaction is that first the task which corresponds to the source interaction node is executed, and then the task which belongs to the destination interaction node is executed.

In the interaction graphs of both "Anne" and "Sue", the "(MDM,05/02,send report)" node has exactly one outgoing arc. Both arcs point to the "receive" task of the "visit" Proclet class. So, as can be seen in Figure 9.8, this means that both interaction arcs refer to the same port. However, the instance identifiers are different. So, for "Sue" there needs to be a performative which is sent to the "(visit,Sue 10/02,receive)" Proclet instance with "(Sue,6)" as interaction identifier and for "Anne" there needs to be a performative which is sent to the "(visit,Anne 12/02,receive)" Proclet instance with "(Anne,6)" as its interaction identifier. However, as the two potential performatives have the same sender and the "receive" task of the "visit" Proclet class as their destination, only one performative will be

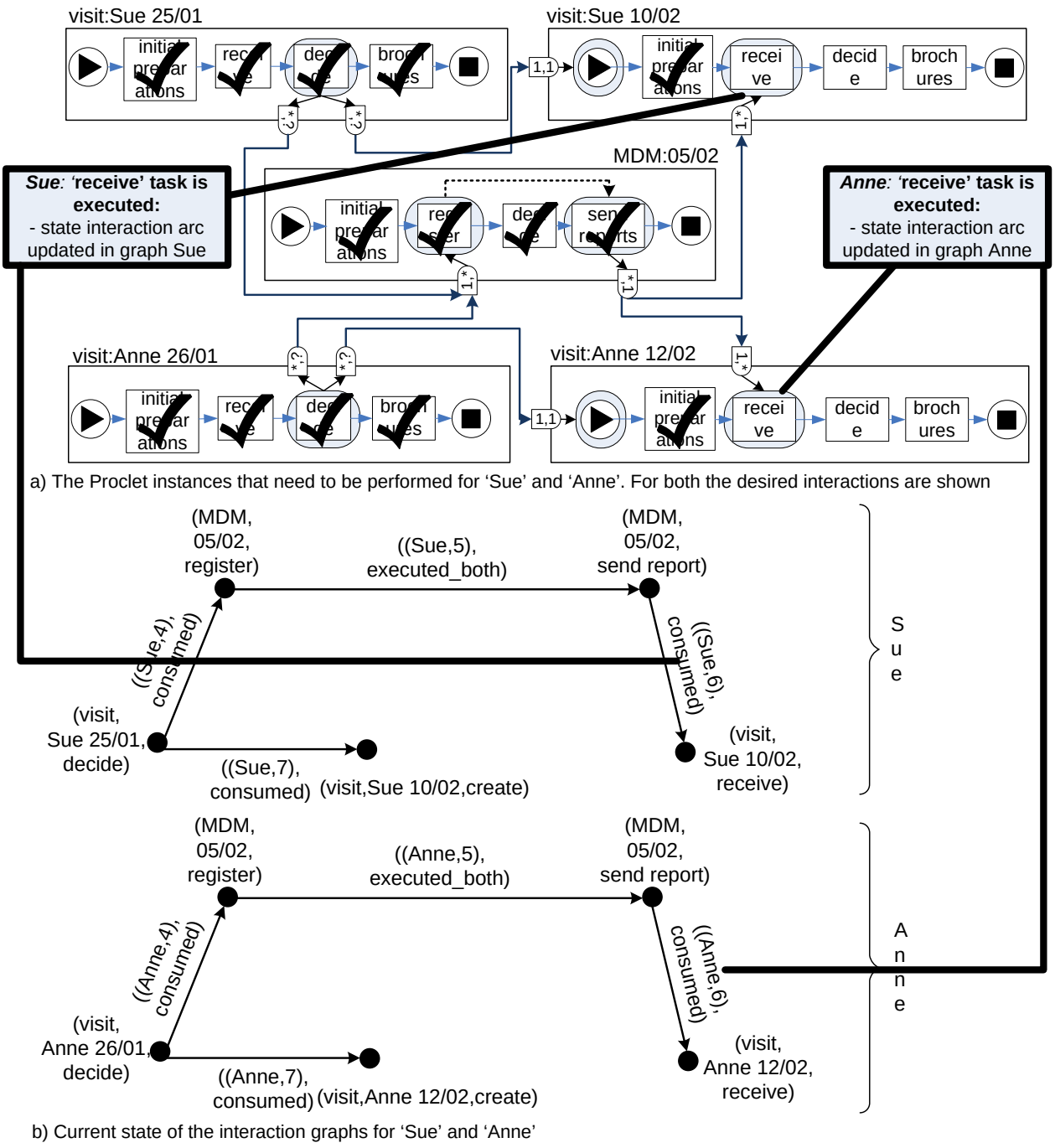


Figure 9.14: For both "Sue" and "Anne", the "receive" task is performed. As a performative has been sent which contains both "(Sue,6)" and "(Anne,6)" as its interaction identifier, the task may be completed. Subsequently, the interaction graphs for both are updated.

created which is *multicast* to the different receivers. So, as can be seen in Figure 9.13, there is a performative which has "MDM:05/02" as sender, "visit:Sue 10/02" and "visit:Anne 12/02" as receivers, and contains "(Sue,6)" and "(Anne,6)" as interaction identifiers. Note that the corresponding interaction arcs are updated accordingly, i.e. the state is set to "sent".

Step 4:

Finally, as the last step for both "Sue" and "Anne" the "receive" task of their second visit is executed (Figure 9.14). As a performative exists which contains the interaction identifiers for both of them, the "receive" tasks for the two may be executed. Subsequently, the interaction state of the corresponding arc in the two graphs are updated to "consumed". However, note that for both of them, the execution of the "receive" task and the subsequent update of the interaction graph occurs completely independently from each other.

Proclets Framework So Far In Figure 9.15, the concepts that have been introduced so far for the Proclets framework are visualized. For an interaction point we have indicated that it represents a specific point in a Proclet class at which interactions with other Proclets may take place. However, based on internal interactions that can be defined and interaction graphs of entities that can be extended, the notion of an interaction point can be further refined. Therefore, a distinction is made between a configuration, an inbox, and an outbox interaction point. The meaning of these is as follows.

Inbox Interaction Point: For one or more entities, performatives may be received. In this way, an inbox interaction point is only connected to input ports. For each input port, an arbitrary number of performatives may be received. An inbox interaction point is either connected to a task or an input condition.

In Figure 9.15a, inbox interaction points are marked with the abbreviation "IB". For example, the "receive" task in the "visit" Proclet class is an inbox interaction point as for an entity only performatives are received.

Outbox Interaction Point: For one or more entities, performatives may be sent to multiple receivers. So, an outbox interaction point is only connected to output ports. For each output port, an arbitrary number of performatives may be sent. Note that by definition an output port is only connected to a task.

In Figure 9.15a, outbox interaction points are marked with the abbreviation "OB". For example, the "send reports" task of the "MDM" Proclet class is an outbox interaction point as for multiple entities a performative may be sent.

Configuration Interaction Point: A configuration interaction point is similar to an outbox interaction point. It has the additional requirement that when an instance of a task is executed for such an interaction point, it is allowed to extend the interaction graph for multiple entities. In case for an entity such a graph does not yet exist, it will be created otherwise it will be extended. For each entity for which the interaction graph is extended, a human actor can nominate interactions that need to take place between existing and future Proclet instances. In Section 9.1.3 more details are provided about the extension of an interaction graph and the entities for which this may occur.

In Figure 9.15a, configuration interaction points are marked with the abbreviation "CP". For example, the "decide" task of the "visit" Proclet class is a configuration interaction point as for an entity multiple performatives may be sent and the associated interaction graph may be extended.

For an internal interaction this means that its source interaction point is always an inbox interaction point and its destination interaction point is always an outbox interaction point. For example, for the "MDM" Proclet class an internal interaction is defined which leads from the "register" inbox interaction point to the "send reports" outbox interaction point.

As indicated in Figure 9.15, interaction graphs that exist for entities, influence the behavior and interactions between existing and future Proclet instances. To this end, the interaction state of interaction arcs in these graphs are important. For arcs referring to external interactions, the state transitions from "unproduced", to "sent", to "consumed". The general meaning of each state is as follows.

unproduced: No performative has been produced yet for the interaction represented by the arc. Note that an interaction arc corresponds to a performative when the interaction identifier of the arc is contained in the "interaction identifiers" attribute of the performative.

sent: A performative has been produced for the interaction represented by the arc. In particular, the interaction identifier of the arc is contained in the "interaction identifiers" attribute of the performative that has been produced.

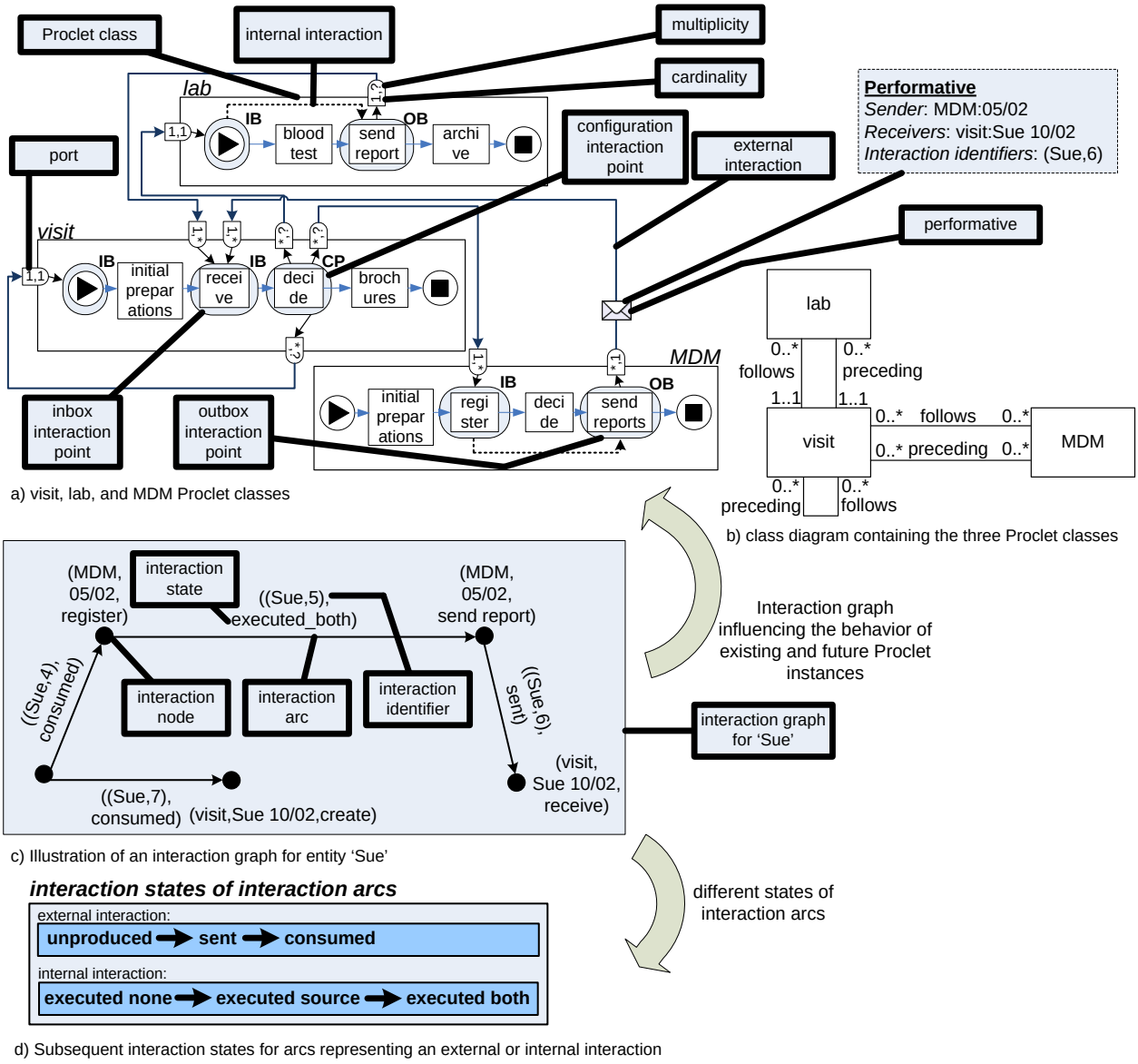


Figure 9.15: Illustration of the concepts that have been introduced for the Proclets framework so far. Additionally, for interaction points, a distinction has been made between configuration (CP), inbox (IB), and outbox(OB) interaction points.

consumed: Where the head of the arc refers to an input condition of a Proclet class, the "consumed" state occurs when the corresponding performative has been 'consumed' in order to create an instance of a particular Proclet class. Where the head of the arc refers to a task instance, state "consumed" is obtained when the corresponding performative has been "consumed" in order to complete the task instance. Note that a task instance may be completed if for all interaction nodes that belong to the task instance, all the incoming arcs have the state "sent". However, an exception to the latter is possible when an exception is handled. This will be discussed in more detail in Section 9.1.2.

For arcs referring to internal interactions, the state transitions from "executed none", to "executed source", to "executed both". The general meaning of each state for an arc is as follows.

executed none: if an arc has the state "executed none" both the source and destination interaction nodes have not been executed. That is, for a source interaction node which is linked to an input condition of a Proclet class, this means that no instance of that Proclet class has been created yet. For a source interaction node which is linked to a task instance, the task instance has not been executed yet. For the destination interaction point, which is always linked with a task instance, this also means that the task instance has not been completed yet.

executed source: For a source interaction node which is linked to an input condition of a Proclet class, this means that an instance of the Proclet class has been created. For a source interaction node which is linked to a task instance, this means that the task instance has been executed.

executed both: As an extension to the previous state, the task instance which is linked to the destination interaction point, has now been executed.

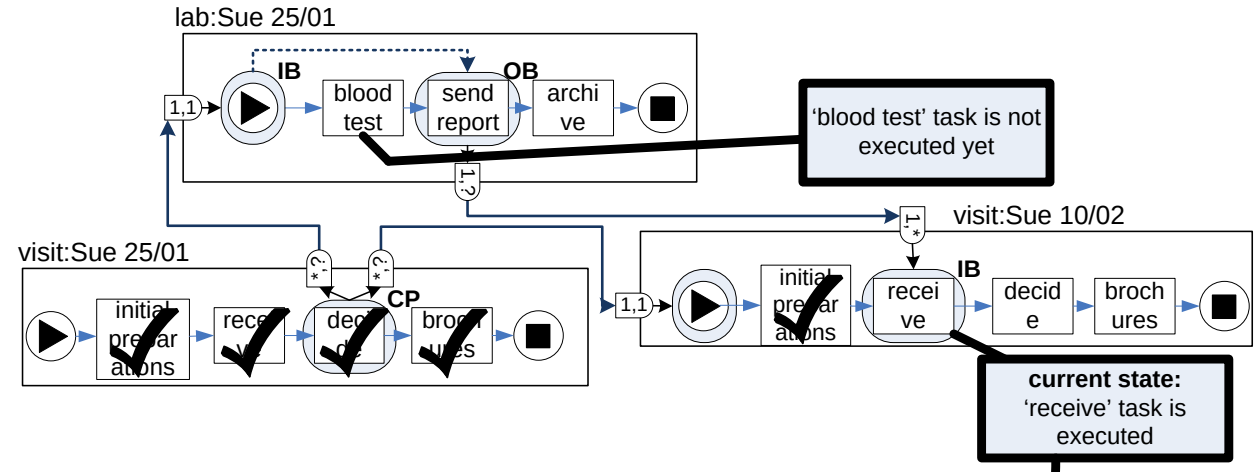
With regard to the state of an arc, it should be noted that the same interaction arc may be found in multiple interaction graphs (when abstracting from the interaction identifier). So, for these arcs, their tails refer to the same interaction node and their heads refer to the same interaction node. Consequently, the state of these arcs will always change simultaneously. For example, for the second scenario, we have seen in Figure 9.13, that the state of the arc from the "register" task to the "send report" task changed simultaneously from "executed source" to "executed both" in the interaction graphs of both "Sue" and "Anne".

Exception Handling

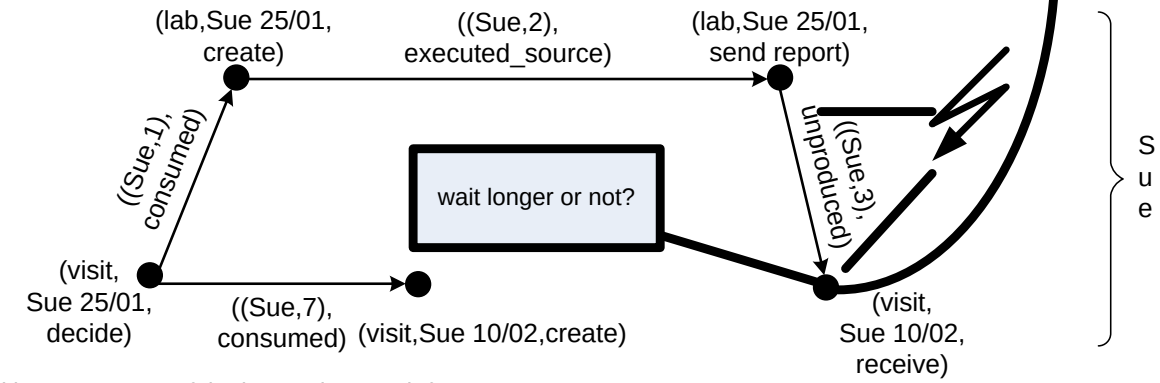
The interactions that are defined in the interaction graphs are nominated by a human actor. Therefore, these interactions need to occur. However, several kinds of exceptions may occur in which they cannot take place. In this section, we discuss the different situations in which an exception may occur and how they can be handled. First, in Section 9.1.2, exceptions that occur in the context of executing a task are discussed. Next, in Section 9.1.2, exceptions in the context of Proclet instances that are canceled or completed are elaborated upon. Note that the exceptions that may occur are discussed by referring to the two scenarios that have been presented earlier.

Execution of a Task In order to illustrate an exception that may occur in the context of executing a task we refer back to the first scenario. However, now we assume a situation in which the "send report" task of the "lab" Proclet instance has not been executed. Also, we assume that we are currently executing the "receive" task of the Proclet instance for the second visit. Note that the latter task is linked to an inbox interaction point. This situation is depicted in Figure 9.16.

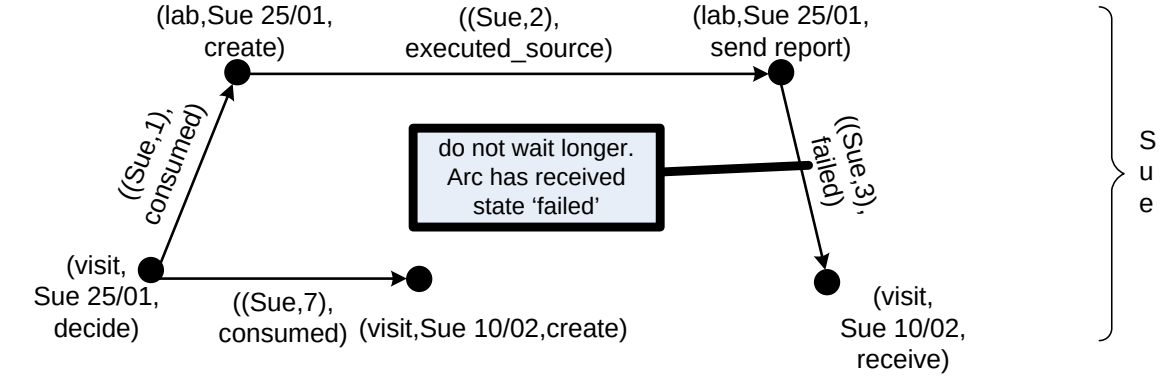
Time-Out Value As no performative has been sent from the "send report" task of the "lab" instance to the "receive" task of the second visit, it is not allowed to complete the "receive" task, i.e., the state of the interaction arc for the respective interaction is still "unproduced". As a result, an exception occurs for the "visit" Proclet instance. We distinguish two different approaches in order to deal with such a situation. The first approach is to reserve more time in which to receive the missing performative. This can be supported by defining a *time-out* value for an inbox interaction point. The value defines how much time needs to be reserved in which to wait for missing performatives. Once the waiting time has lapsed, a human user is



a) The Proclet instances that need to be performed for 'Sue' together with the desired interactions



b) Current state of the interaction graph for 'Sue'



c) State of the interaction graph when it is decided not to wait for the missing performative

Figure 9.16: Illustration of an exception that may occur in the context of a task that is executed. Here, no performative has been sent yet from the "lab" instance to the Proclet instance for the second visit. As a result, a problem occurs when executing the "receive" task for the second visit. One solution is to wait longer for the missing performative. Another solution is to force complete the task and to mark the corresponding interaction arc as having the state "failed".

requested how to deal with the situation. Note that the time-out value can be mapped to any unit of time. For example, a value of "5" may correspond to 5 minutes.

Another approach to dealing with the situation is to force complete the task and thus not to wait for missing performatives. This situation is illustrated in Figure 9.16c. As the interaction with the "send report" task will not take place, the arc from the "(lab,Sue 25/01,send report)" node to the "(visit,Sue 10/02,receive)" node has received the state "failed" in order to indicate that the interaction will not take place anymore.

Exception Interaction Point As a follow-up on the approach to force complete the task instance it may be desirable to use the result of the lab test as input for a third patient visit. This requires, for the entity that is affected by the exception, that it is possible to extend an interaction graph as part of the latter exception handling strategy. For the entity "Sue" this is illustrated in Figure 9.17.

In order to be able to extend an interaction graph in case an exception occurs for a certain Proclet instance, a so-called *exception interaction point* may be defined for a Proclet class. An exception interaction point is similar to a configuration interaction point. However, only for the entities that are affected by the exception, may the interaction graph be extended.

In Figure 9.17a, an exception interaction point has been defined for the "visit" Proclet class. By following the outgoing arcs, it can be seen that an instance of the "lab" Proclet class and an instance of the "visit" Proclet class may be created in case an exception occurs for an instance of the "visit" Proclet class. Subsequently, in Figure 9.17c, it is illustrated for the entity "Sue" how the interaction graph is extended using the exception interaction point of the "visit" Proclet class. That is, starting from the "visit:Sue 10/02" Proclet instance for which the exception occurred, it is decided to start an instance of the "visit" Proclet class. Next, the result of the "send report" task of the "lab" Proclet instance is used as input for the "receive" task of the new "visit" Proclet instance.

The resulting interaction graph for "Sue" can be seen in Figure 9.17c. As can be seen, the "(visit,Sue 10/02,exception)" node has been added representing the exception that has occurred. Starting from that node, the next instance of the visit Proclet class is created (node "(visit,T3,create)"). Finally, the result of the lab test is used as input for the third patient visit (node "(visit,T3,receive)").

General

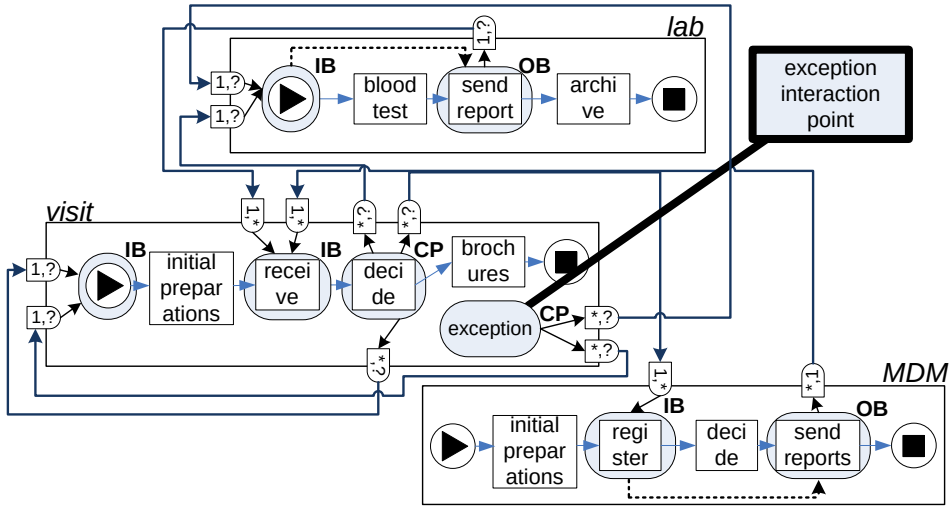
The approach for executing a task instance for which not all performatives have been received can be generalized as follows. This is schematically visualized in Figure 9.18 where two interaction graphs are shown. Remember that a task instance for which only performatives can be received is always linked to an inbox interaction point.

For a task instance "B", a corresponding interaction point "Bi" may be found in multiple interaction graphs. In case not all performatives have been received for task instance "B", i.e. not all incoming arcs for the interaction nodes named "Bi" have the state "sent", then an exception occurs for the Proclet instance in which the task instance occurs (in the figure, for entity "A" there is one incoming arc having the state "unproduced" and for entity "B" there are two incoming arcs having the state "unproduced"). Now, two options are possible.

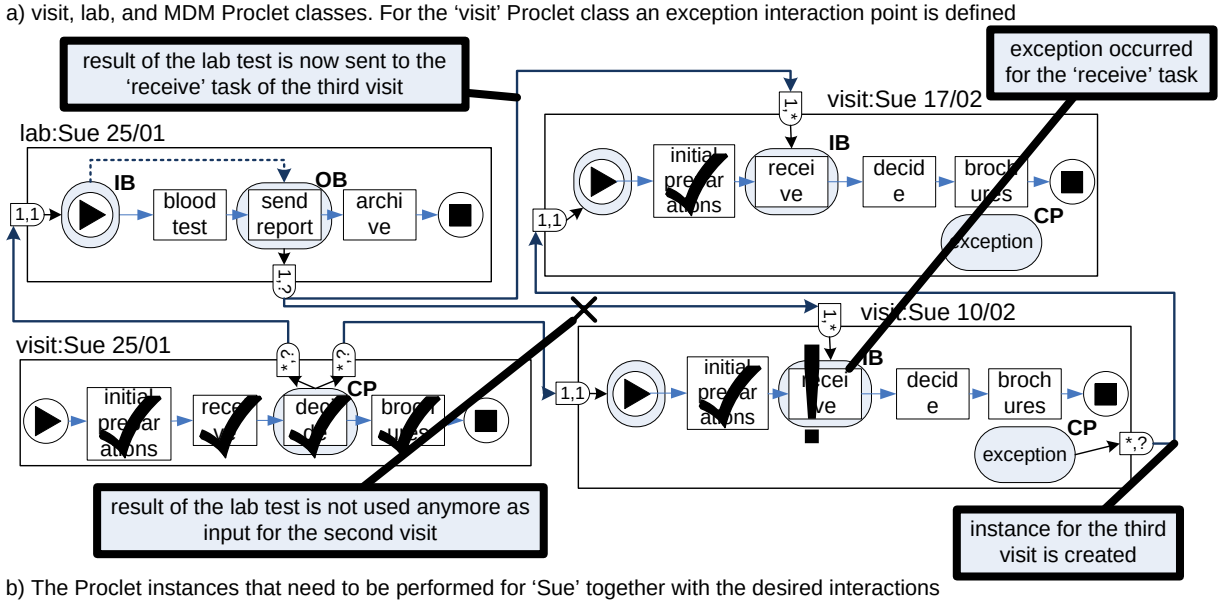
According to the time-out value defined for the interaction point that belongs to task instance "B", a human actor may decide to reserve more time in which to receive missing performatives.

Another option is to force complete the task instance thereby ignoring the performatives that still need to be received. In that case, for all incoming interaction arcs for the interaction nodes named "Bi" which have state "unproduced", the state is changed to "failed" (in Figure 9.18, for entity "A" there is now one incoming arc having state "failed" and for entity "B" there are two incoming arcs having state "failed"). Additionally, for all affected entities, the exception graph may be extended. An entity is affected by the exception if for one or more interaction arcs in the corresponding interaction graph the state had to be changed to "failed". Finally, the extension of the interaction graphs may be done via the exception interaction point of the Proclet class for which the exception occurred.

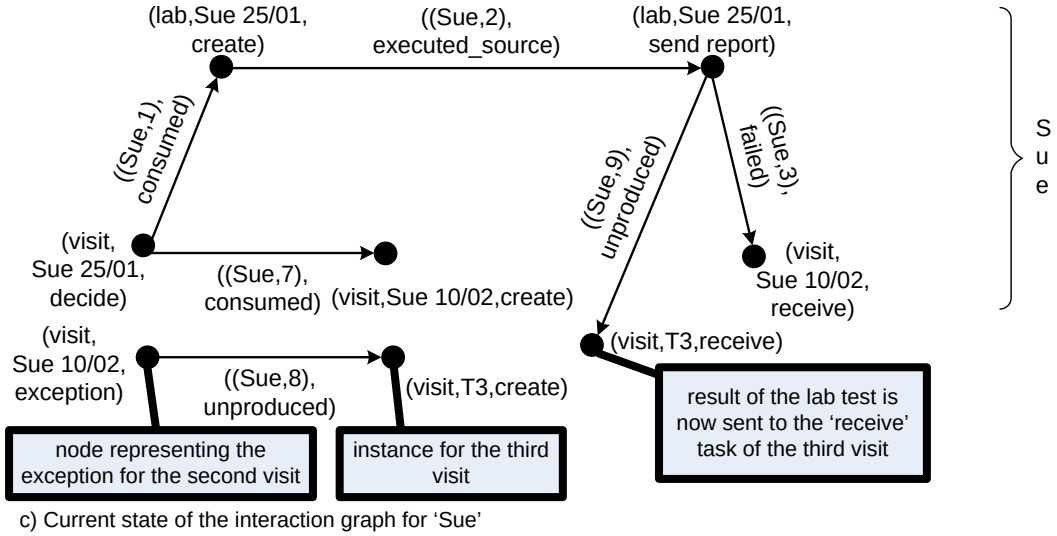
Instance Cancelation or Completion A Proclet instance may be canceled or completed before all desired interactions have occurred for it. First, we illustrate this kind of exception and the handling of it in the



a) visit, lab, and MDM Proclet classes. For the 'visit' Proclet class an exception interaction point is defined



b) The Proclet instances that need to be performed for 'Sue' together with the desired interactions



c) Current state of the interaction graph for 'Sue'

Figure 9.17: An exception interaction point may be defined for a Proclet class (Figure a). As for "Sue" an exception occurred for the "visit:Sue 10/02" Proclet instance, the exception interaction point has been used for creating the next instance of the "visit" Proclet class which represents the third visit (Figures b and c). Also, the result of the "send report" task is used as input for the third visit.

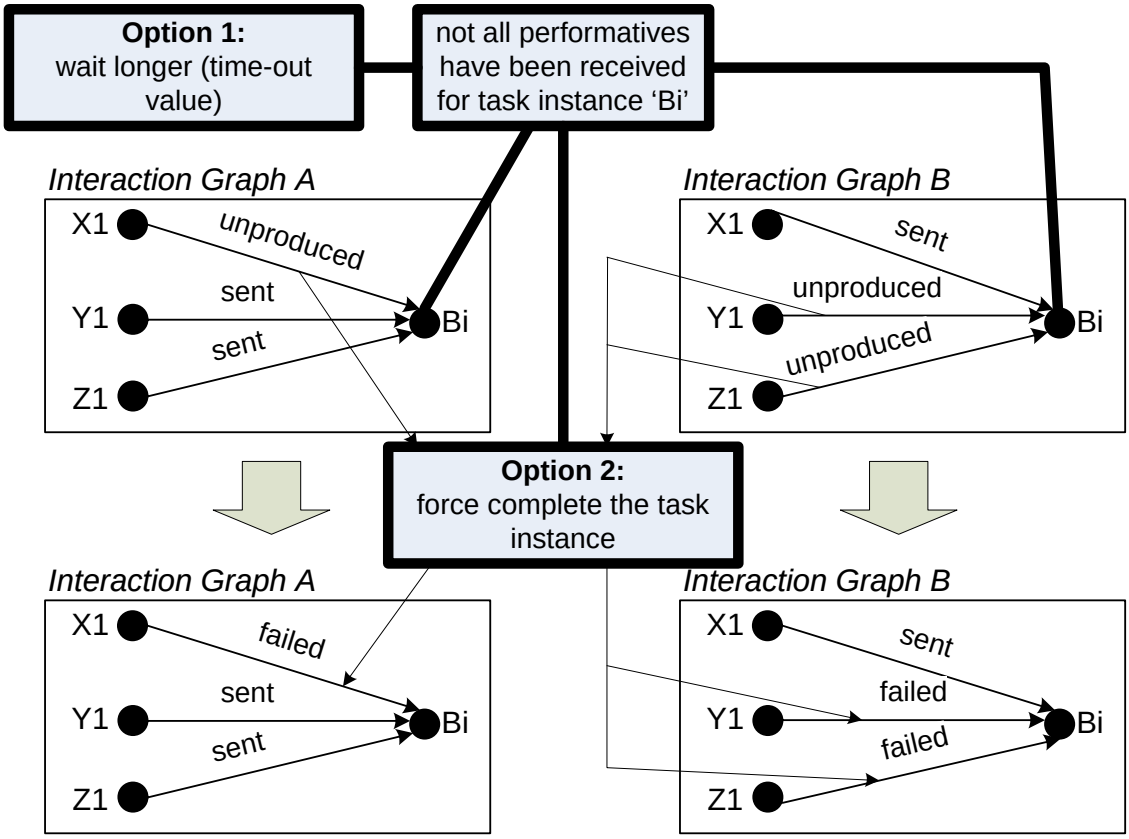


Figure 9.18: For the general case it is illustrated how an exception is handled in the interaction graphs if for a task instance not all required performatives have yet been received.

context of the first scenario. Then, the exception caused by case cancelation / completion and its handling is explained for the general case.

Outbox Interaction Point One kind of exception that may occur in the context of canceling a Proclet instance is related to an outbox interaction point. This is illustrated in Figure 9.19. Here, for the first scenario, we assume that the "lab" Proclet instance is canceled and that no tasks for it have yet been executed (Figure 9.19a). Also, for the second visit, no tasks have yet been performed.

If in the interaction graph of entity "Sue" we look at the arcs that relate to the canceled "lab" Proclet instance (Figure 9.19b) then we see that the arc from the "(lab,Sue 25/01,send report)" node to the "(visit,Sue 10/02,receive)" node has the state "unproduced". This means that no performative has been sent yet from the "send report" task to the "receive" task of the second visit. Note that the "send report" task is linked with an outbox interaction point. As a consequence, we have an exceptional situation as the respective performative will never be sent at any time in the future, i.e. the defined interaction will never occur.

Subsequently, the arc from the "(lab,Sue 25/01,send report)" node to the "(visit,Sue 10/02,receive)" node transitions to the state "failed" (Figure 9.19c). Moreover, similar to the previous exception, a human actor may extend the interaction graph for "Sue". As the exception occurred for the "lab" Proclet instance due to its cancelation, the graph may be extended by using the exception interaction point of the "lab" Proclet class (not shown in Figure 9.19a). Note that if the "lab" Proclet instance had been completed and no performative had been sent from the "send report" task (e.g. due to a choice in the process), then the same procedure as described above would be followed.

Inbox Interaction Point Another exception that may occur in the context of canceling a Proclet instance is related to an inbox interaction point. This is illustrated in Figure 9.20. Here, for the first scenario, we assume that the "visit" Proclet instance for the second visit is canceled and that no tasks for it have been executed yet (Figure 9.20a). Also, for the "lab" Proclet instance we assume that no tasks have been executed.

Looking at the interaction graph of "Sue" (Figure 9.20b), for the arcs that relate to the canceled "visit" instance we can see that the arc from the "(lab,Sue 25/01,send report)" node to the "(visit,Sue 10/02,receive)" node has the state "unproduced". So, no performative has yet been sent from the "send report" task to the "receive" task of the canceled "visit" Proclet instance. Note that the "receive" task is linked with an inbox interaction point. So, we have an exceptional situation as the performative which still needs to be sent from the "send report" task can never be consumed by the "received" task, i.e. the defined interaction will never occur.

Subsequently, the arc from the "(lab,Sue 25/01,send report)" node to the "(visit,Sue 10/02,receive)" node transitions to the state "failed" (Figure 9.20d). Also here, the interaction graph of "Sue" is offered for extension. However, as the exception occurred for the "visit" Proclet instance as a consequence of its cancelation, the graph may be extended by using the exception interaction point of the "visit" Proclet class.

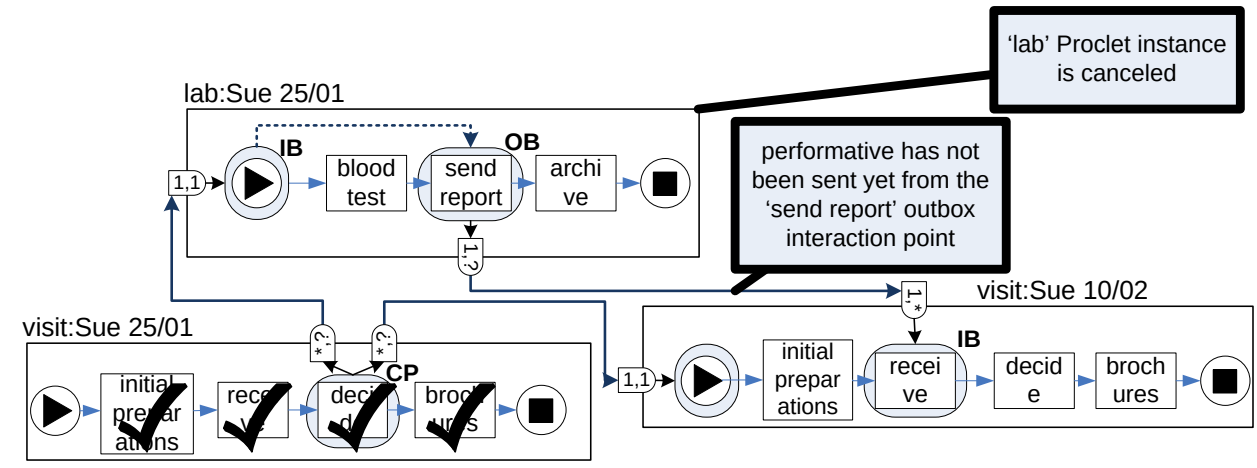
In Figure 9.20c, a comparable situation is shown. Here, the corresponding interaction graph is presented when a performative is sent from the "send report" task that has not yet been consumed by the "receive" task of the canceled "visit" Proclet instance. Here also, due to the cancelation of the Proclet instance for the second visit, the defined interaction will never occur. So, the respective interaction arc will change to the state "failed" (Figure 9.20d) and for the entity "Sue" the interaction graph may be extended.

Note that a similar procedure is followed if a Proclet instance is completed and for a task instance, corresponding to an inbox interaction point, all performatives have not yet been received.

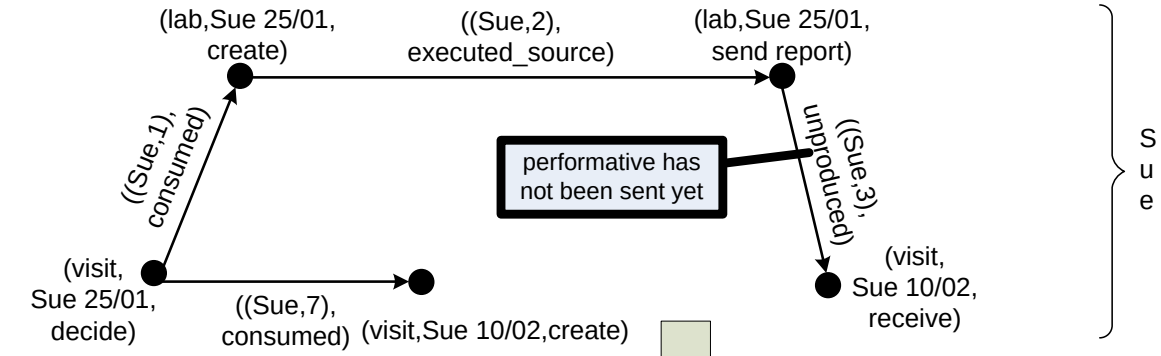
General

The approach for canceling / completing a Proclet instance where all desired interactions have not yet taken place can be generalized as follows. This is schematically visualized in Figure 9.21a and b where both interaction graphs are shown.

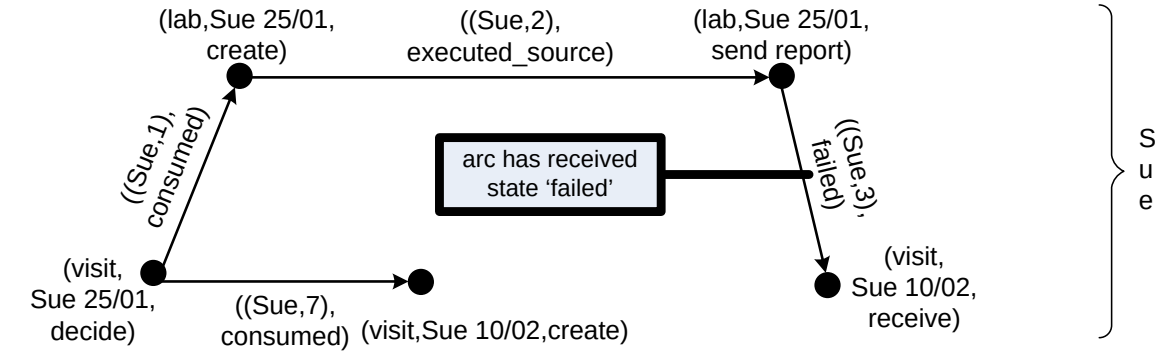
The first situation is depicted in Figure 9.21a where Proclet instance "Y" is canceled / completed. For Proclet instance "Y", multiple outbox interaction nodes may exist in multiple interaction graphs. For these interaction nodes (illustrated by nodes "Yi" and "Yj" in Figure 9.21a) if there is at least one outgoing arc with the state "unproduced", then an exception occurs for Proclet instance "Y', i.e. for such an arc a defined interaction can never occur.



a) The Proclet instances that need to be performed for 'Sue' together with the desired interactions

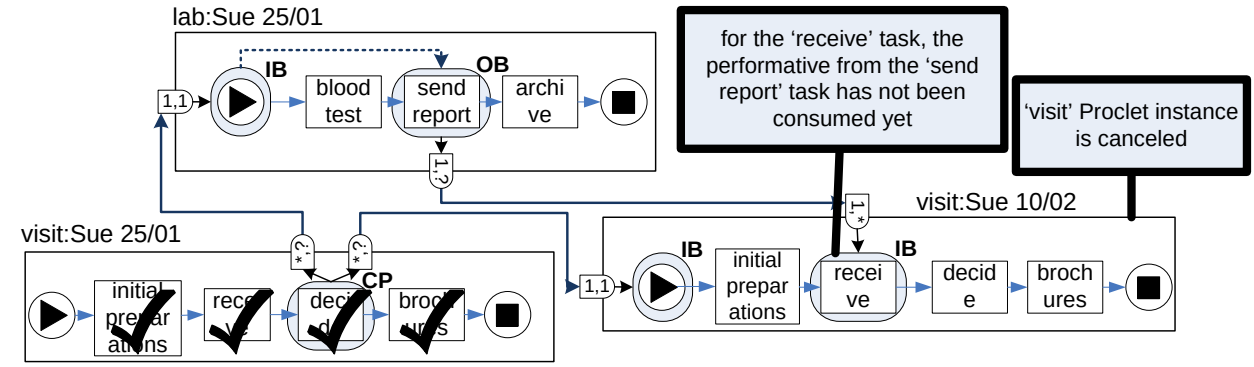


b) Current state of the interaction graph for 'Sue'

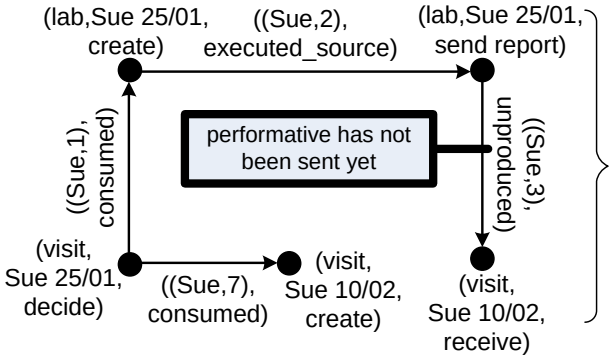


c) State of the interaction graph when the 'lab' Proclet instance is canceled

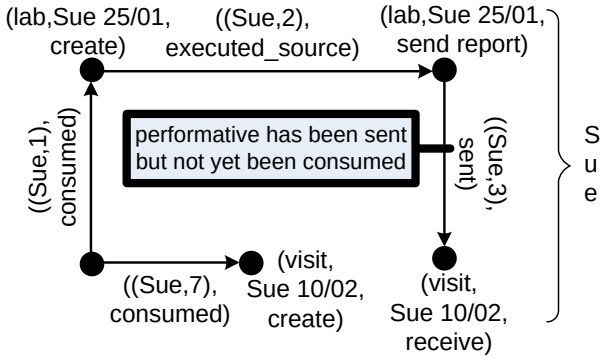
Figure 9.19: Illustration of an exception that may occur if a certain Proclet instance is canceled. For a task instance which relates to an outbox interaction point, the required performative has not been sent.



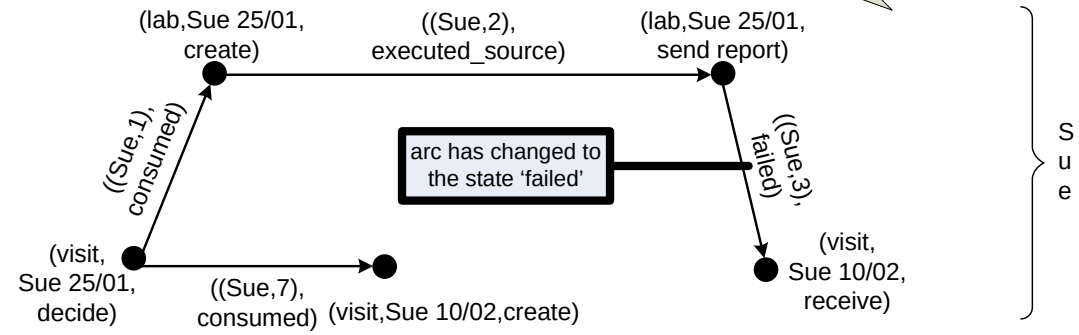
a) The Proclet instances that need to be performed for 'Sue' together with the desired interactions



b) Interaction graph for 'Sue' in which the performative from the 'lab' instance has not been produced yet

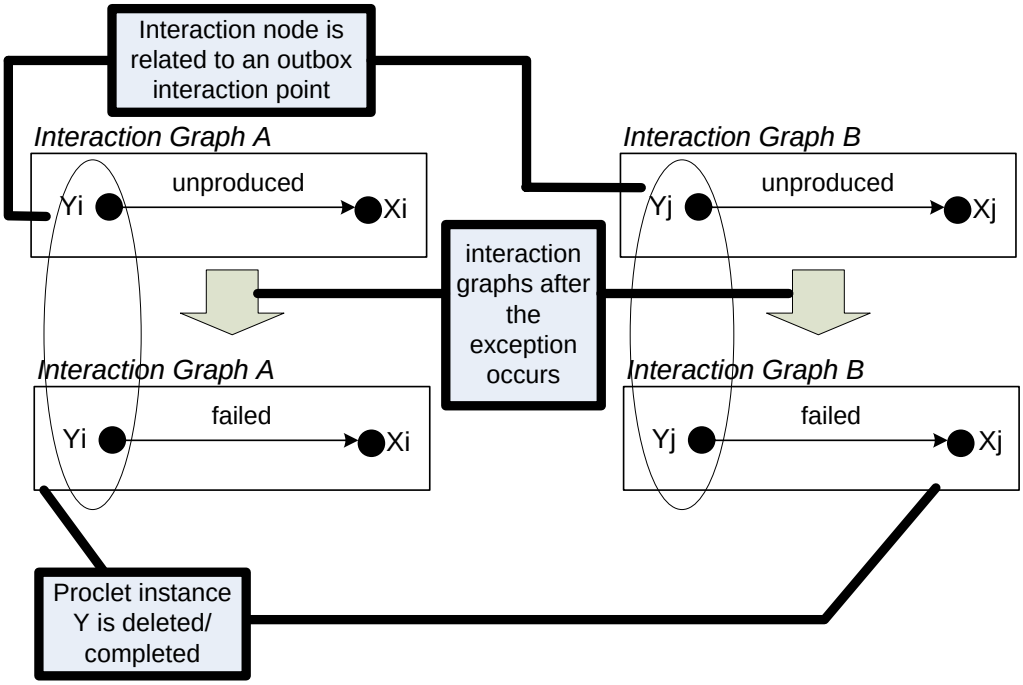


c) Interaction graph for 'Sue' in which the performative from the 'lab' instance has been produced but has not yet been consumed

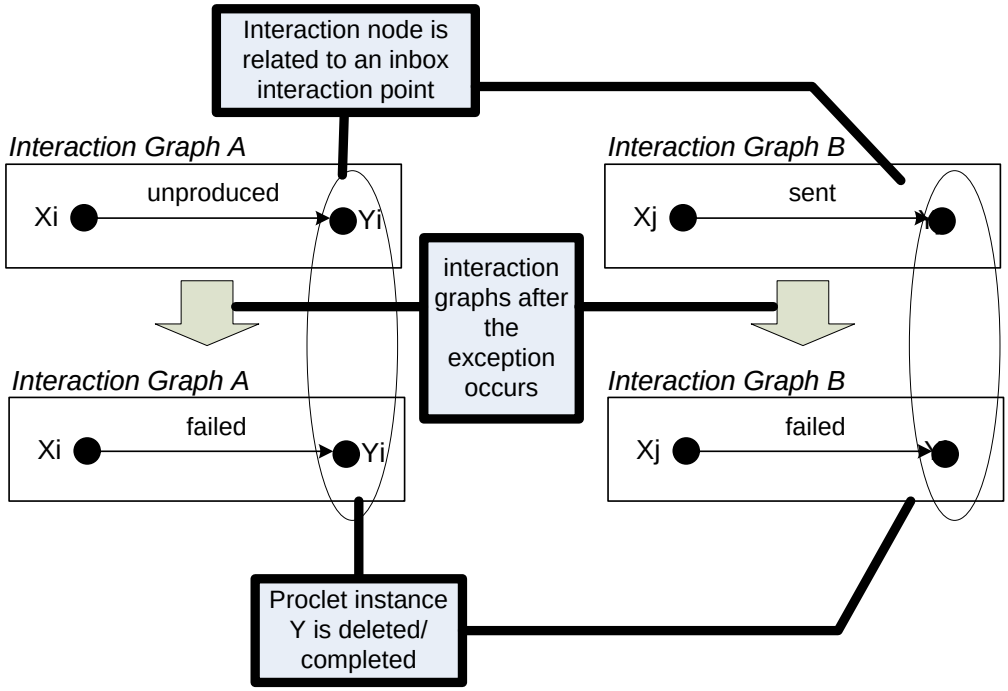


d) State of the interaction graph when the 'visit' Proclet instance is canceled

Figure 9.20: Illustration of an exception that may occur if a Proclet instance is canceled. For the task instance which relates to an inbox interaction point, the required performative has not been received.



a) Proclet instance Y is deleted. For the deleted instance, all outgoing interaction arcs related to an outbox interaction point which have the state 'unproduced' change to the state 'failed'.



b) Proclet instance Y is deleted. For that instance, all incoming interaction arcs related to an inbox interaction point which have the state 'unproduced' or 'sent' change to the state 'failed'.

Figure 9.21: For the general case it is illustrated how an exception is handled in the interaction graphs if a Proclet instance is canceled or completed.

Consequently, the latter mentioned arcs change to the state "failed" (illustrated at the bottom of Figure 9.21a). Furthermore, for the affected entities, the opportunity is offered to extend the interaction graph.

The second situation is depicted in Figure 9.21b where Proclet instance "Y" is canceled / completed. For Proclet instance "Y", multiple inbox interaction nodes may exist in multiple interaction graphs. For these interaction nodes (illustrated by nodes "Yi" and "Yj" in Figure 9.21b) if there is at least one incoming arc having the state "unproduced" or "sent", then an exception occurs for Proclet instance "Y", i.e. for such an arc the defined interaction can never occur. Consequently, the latter mentioned arcs change to the state "failed" (illustrated at the bottom of Figure 9.21b).

9.1.3 Extending an Interaction Graph

In previous sections, we have discussed different aspects of the Proclets framework. When elaborating on the two presented scenarios interaction graphs were given that were already defined. In this section, we elaborate upon how an interaction graph is extended for an entity.

The extension of an interaction graph is based on the current interaction graph of an entity. Also, it is based on the interaction points that exist for Proclet classes and how they are connected, i.e. internal and external interactions that exist between these interaction points. First, we illustrate the extension of an interaction graph in the context of the second scenario. Then, it is explained for the general case.

Scenario

As shown in Figure 9.22a, for the second scenario we assume that for the first visit the "decide" task is currently executing. Moreover, we assume that an instance of the "MDM" Proclet class is running. The "decide" task is linked with a configuration interaction point which means that the interaction graphs of multiple entities may be extended. For patient "Sue" we want to achieve the outcome that a second visit is created for her and that she is discussed during the multidisciplinary meeting. Also, the result of the discussion for her during the multidisciplinary meeting needs to be used as input for the second visit. In order to achieve this, an interaction graph needs to be created.

Step 1:

At the moment the "decide" task is executed, no interaction graph exists for "Sue". Therefore, as part of the "decide" task, we indicate that for the entity "Sue" an interaction graph needs to be created. The result of this action can be seen in Figure 9.22c. Here, we see that there is an interaction node with name "(visit,Sue 25/01,decide)" which refers to the "decide" task that is currently executing. Moreover, it is indicated that the node is *active*. This means that for the node a Proclet instance currently exists which has the same instance identifier. In this way, for the node interactions may be nominated which will potentially occur in the future. The interactions that may be nominated can be seen by looking at the Proclet classes and their interconnections in Figure 9.22b. In particular, if we look at the "decide" task in the "visit" Proclet class we see that it has three outgoing ports. For these the following can be observed which are indicated by the dotted arcs between the Proclet classes and the interaction graph of "Sue".

- An outgoing port is connected with an inbox interaction point which is then linked with the input condition of the "lab" Proclet class. As a result, for the "lab" Proclet class multiple instances may be created. Note however that this may be constrained by the cardinalities and multiplicities of the associated ports.
- An outgoing port is connected with an interaction point which is in turn linked with the input condition of the "visit" Proclet class. As a result, multiple instances of the "visit" Proclet class may be created.
- An outgoing port is connected with an inbox interaction point which is then linked with the "register" task of the "MDM" Proclet class. As currently an instance of the "MDM" Proclet class exists which has the instance identifier "05/02", it is possible to have an interaction with the "register" task of that Proclet instance. Note that we abstract from the current state of the "MDM:05/02" Proclet instance.

In Figure 9.22c, the interactions that may be nominated, starting from the "(visit,Sue 25/01,decide)" node, are indicated by dotted arcs. For each of them, a human actor decides to create one instance of the "visit"

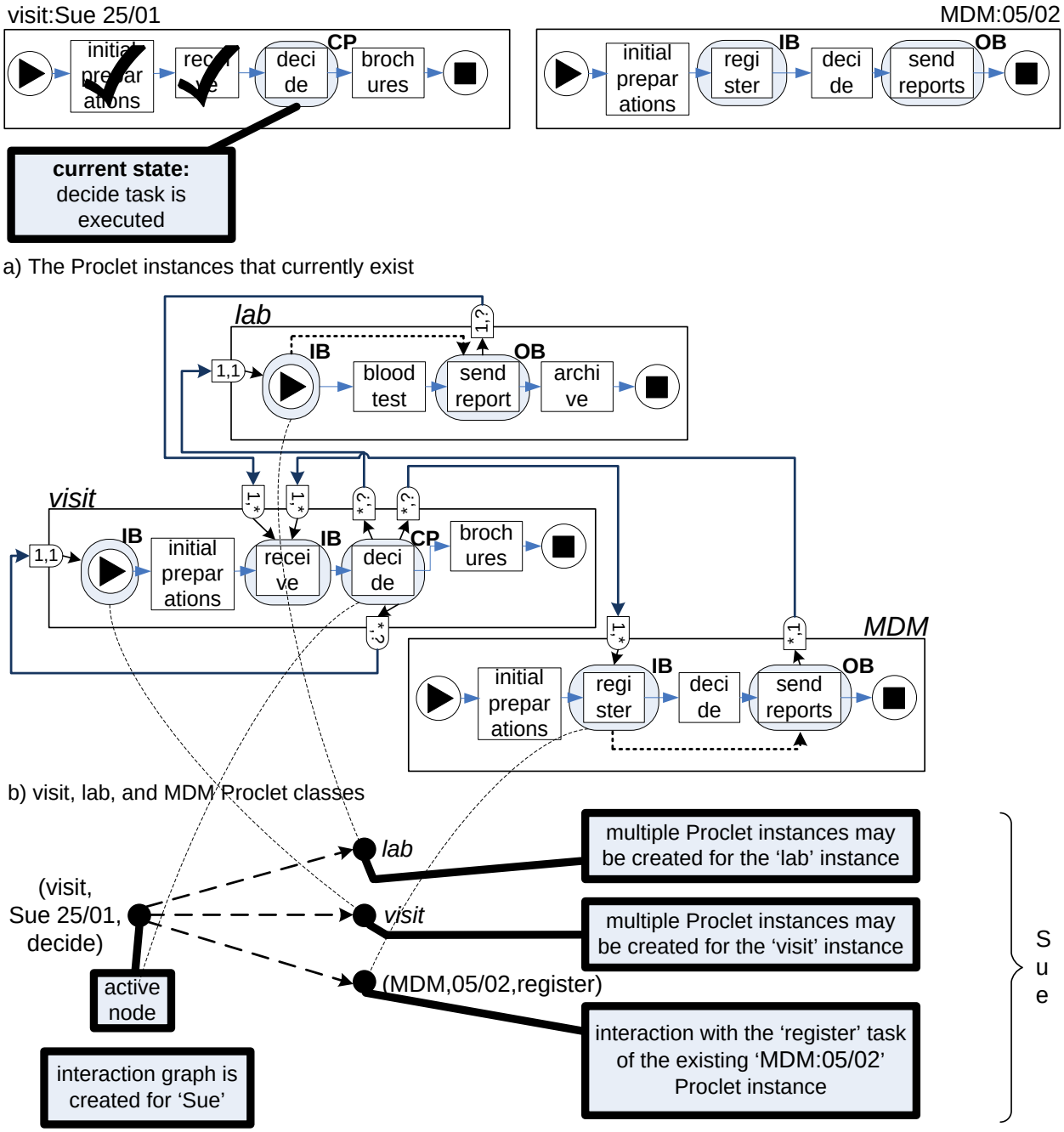


Figure 9.22: Creating an interaction graph for "Sue". The possible interactions starting from the "(visit,Sue 25/01,decide)" node are indicated by dotted arcs.

Proclet class which represents the second visit of "Sue". Additionally, it is decided to have an interaction with the "register" task of the existing "MDM" Proclet instance in order to register "Sue" for the multidisciplinary meeting.

Step 2:

The new interaction graph can be seen in Figure 9.23. As can be seen, there is an arc leading from the "(visit,Sue 25/01,decide)" node to the
"(MDM,05/02,register)" node which represents the interaction with the "register" task of the "MDM" instance. The arc leading to the "(visit,T1,create)" node represents the instance of the "visit" that will be created in order to have the second visit of the patient. Note that a temporary instance identifier is used ("T1") because the instance still needs to be created. For the new interaction arcs, it can be seen that their current state is "unproduced" as no performatives have yet been sent. Also, each arc has a unique instance identifier.

Additionally, in Figure 9.23a, we can see two nodes which are active. For the "(visit,Sue 25/01,decide)" node, the possible interactions are not shown in order to not clutter the graph. However, for example, it is still possible to create an additional instance of the "visit" Proclet class. The other active node is the "(MDM,05/02,register)" node for which the new interactions that can be nominated are indicated via dotted arcs. That is, the "(MDM,05/02,register)" node matches with the "register" task interaction point of the "MDM" Proclet class. For that interaction point, an internal interaction is defined that has the "send report" interaction point as its destination. So, an internal interaction with the "send report" task of the "MDM" Proclet instance is possible.

As can be seen in the figure, the "(visit,T1,create)" node is not active. The node matches with the interaction point that corresponds to the input condition of the "visit" Proclet class (Figure 9.23a). However, for that interaction point no outgoing external and internal interactions have been defined. So, no interactions are possible starting from the "(visit,T1,create)" node and consequently, the node is not active.

Step 3:

The new interaction graph can be seen in Figure 9.24b. The internal interaction for the "MDM" Proclet instance has been added in order to use the outcome of the multidisciplinary meeting as input for the second visit. Note that the associated arc has a unique identifier and that its state is "executed none" as nothing has happened yet.

There are three active nodes in the interaction graph of which only for the "(MDM,05/02,send reports)" node the new possible interactions are visualized. That is, the "(MDM,05/02,send report)" node matches with the "send reports" interaction point of the "MDM" Proclet class (see Figure 9.24a). For that outbox interaction point there is one outgoing port which is linked with the "receive" interaction point of the "visit" Proclet class. As this is an inbox interaction point, an interaction is possible with the existing "visit" Proclet class which has the instance identifier "25/01" (node "(visit,Sue 25/01,receive)"). However, in the graph of entity "Sue" we see an interaction node for a "visit" Proclet instance with the temporary instance identifier "T1" which represents the second visit. As it will exist at some time in the future, interactions may also be defined for it. As a consequence, an interaction with the "receive" task of this future instance is possible (node "(visit,T1,receive)").

The resultant interaction graph is shown in Figure 9.24c. As can be seen, an interaction has been added such that the result of the multidisciplinary meeting is used as input for the second visit. Note that a unique interaction identifier is used and that the state is "unproduced".

When during the second visit the "'decide" task is executed, the interaction graph can be extended again. However, here it is not necessary to provide "Sue" as the entity identifier in order to extend the graph. That is, in the interaction graph for "Sue", we can already find the "create" and "receive" node for the second visit. In this way, it is now denoted that the entity is relevant for the second visit and the graph for it may be extended when required.

General

In the scenario, we have seen how an interaction graph is extended by taking into account current and future Proclet instances. External and internal interactions that are defined for Proclet classes are also taken

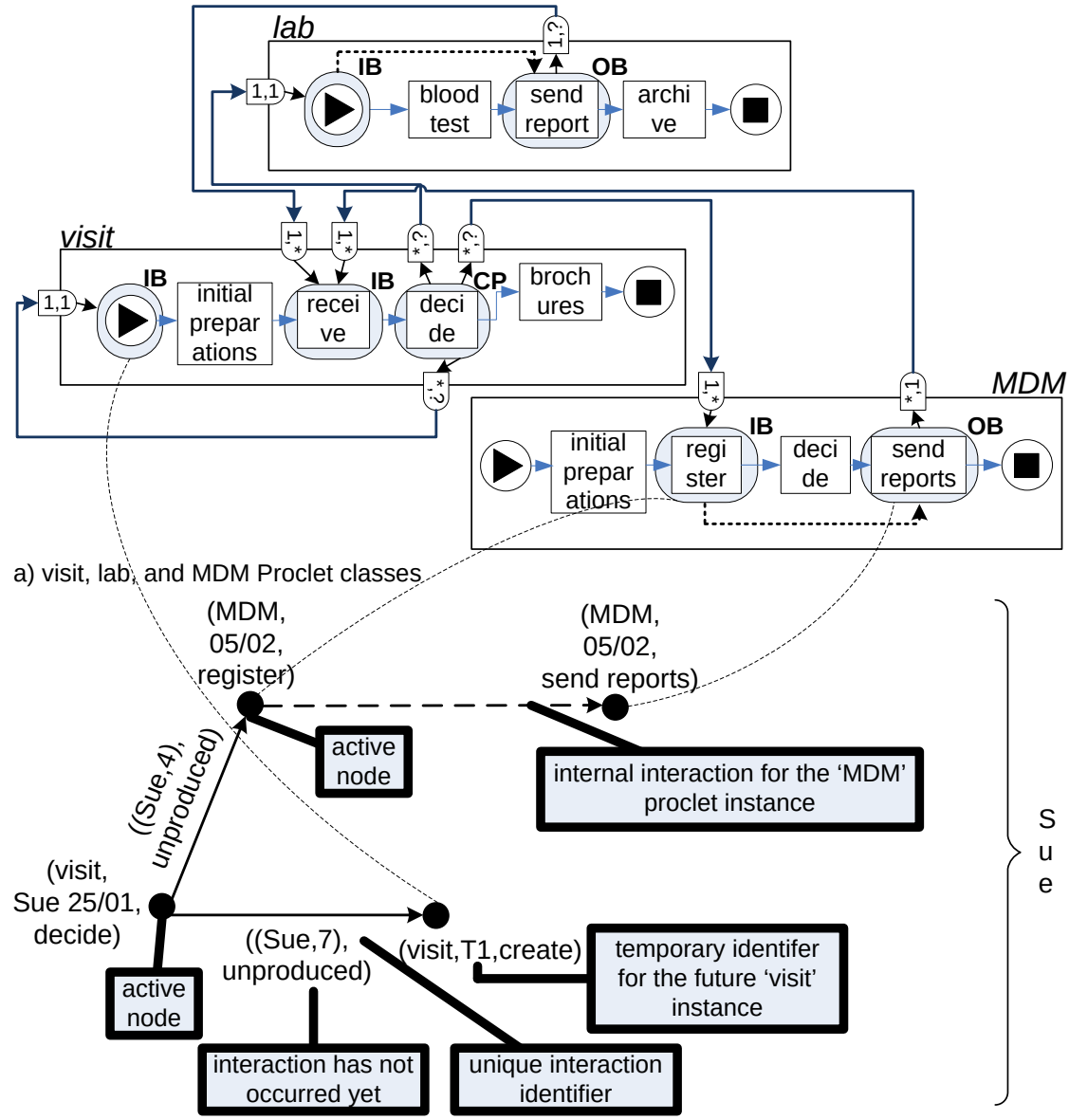
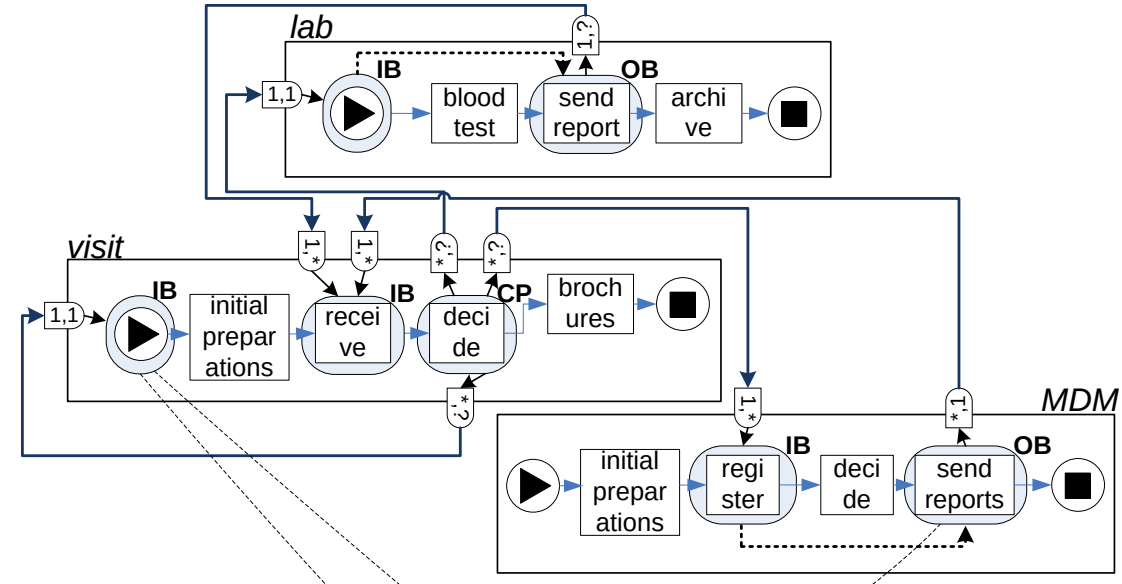
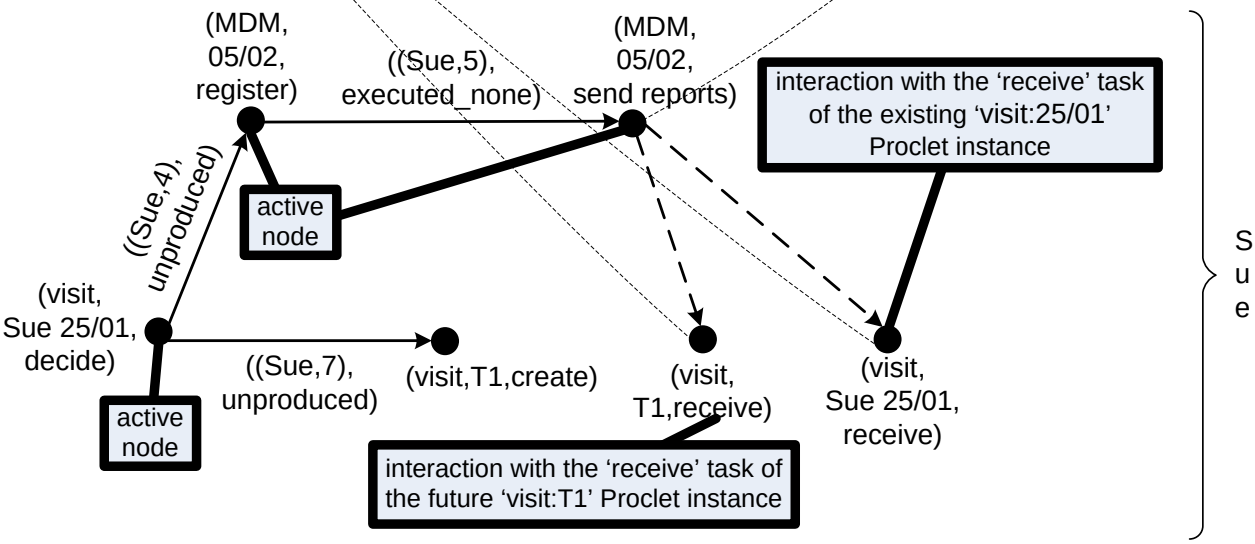


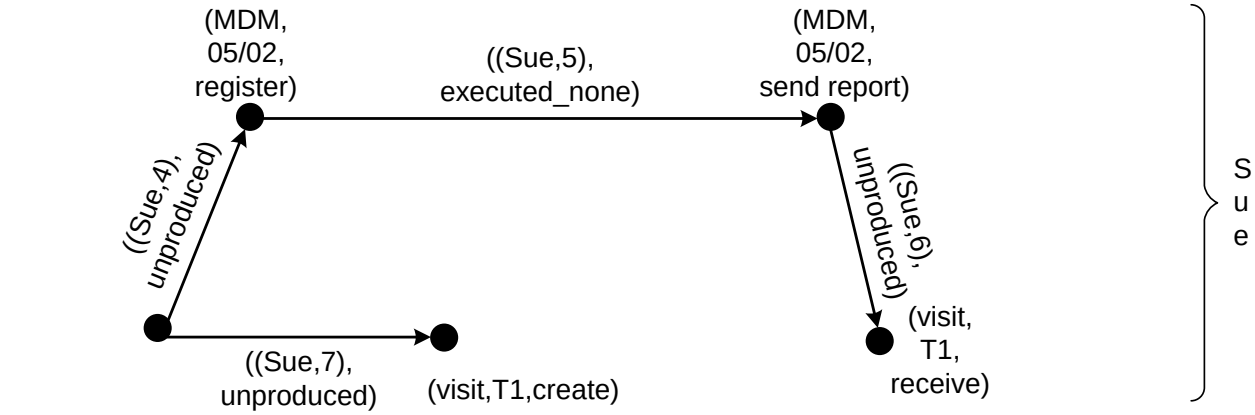
Figure 9.23: Extending the interaction graph for "Sue". The possible interactions starting from the "(MDM,05/02,register)" node and the "(visit,T1,create)" node are indicated by dotted arcs.



a) visit, lab, and MDM Proclet classes



b) Interaction graph defined for the entity 'Sue' so far. Additionally, for the '(MDM,05/02,register)' node the subsequent interactions that can be nominated are visualized by dotted arcs



c) resultant interaction graph for entity 'Sue'

Figure 9.24: Extending the interaction graph for "Sue". The possible interactions starting from the "(MDM,05/02,send report)" node are indicated by dotted arcs. The resultant graph is shown at the bottom.

into account. Now we explain for the general case how an interaction graph is extended.

Relevant entities:

When a task instance corresponding to a configuration interaction point is executed, the interaction graphs of entities may be extended. First, it is important to know for which entities the interaction graph may be extended. Here we distinguish two different cases:

- a human actor has provided the names of entities for which a corresponding interaction graph needs to be created and for which interactions can be defined.
- for the entity an interaction has already been defined for the Proclet instance for which the task instance is executed. So, in the graph of the entity, a node exists which has the same instance identifier as the Proclet instance for which the task instance is executed. In this case we say that the entity is relevant for the Proclet instance.

Furthermore, where an exception occurs for a certain Proclet instance, for the entities that are affected by the exception, the corresponding interaction graph may be extended. Note that an entity is affected by an exception if for one or more interaction arcs in the corresponding interaction graph the resultant state is changed to "failed".

Extension:

In Figure 9.25, the procedure that is followed for extending an interaction graph is visualized. Before starting the procedure, it is first identified whether the instance of the configuration interaction point is itself present in the interaction graph. If not, an interaction node for it is added. Afterwards, the procedure is started by determining which nodes in the graph are *active*. A node is active if for its Proclet instance identifier a Proclet instance exists with the same instance identifier. Also, a node is considered to be active if it has a temporary instance identifier, i.e. the Proclet instance still needs to be created. Obviously, for active interaction nodes, interactions that are defined for it may potentially occur at some time in the future.

Having determined all the active interaction nodes in the interaction graph, for each of them it is determined in which subsequent interactions the node can be involved, i.e. the candidate interactions. For identifying these candidate interactions for active nodes, three different situations can be distinguished which are indicated by the three numbers in Figure 9.25. For each situation, we elaborate on the kind of interaction that is possible and how a nominated interaction leads to an extension of the interaction graph.

- The first situation relates to an external interaction. Via this external interaction it is possible to create instances of a given Proclet class. That is, for active node "n1" that is under consideration, the following observations can be made. Looking at the corresponding interaction point of node "n1" in its Proclet class, it can be seen that via an external interaction, it is connected with the input condition of Proclet class "X". As a result, multiple instances of Proclet class "X" may be created.

  For each instance that needs to created, an interaction arc is added leading from node "n1" to the node that is added for the new instance of Proclet class "X". Note that the newly added node has a temporary instance identifier which has "T" as prefix. Also, the inserted arc has the interaction state "unproduced" and it has a unique interaction identifier.

- The second situation relates to an external interaction. Via this external interaction it is possible to have an interaction with a task instance of an existing or future Proclet instance. That is, for active node "n2" that is under consideration, the following can be observed. Note that "n2" may relate to an existing or future Proclet instance. Looking at the corresponding interaction point of node "n2" in its Proclet class, it can be seen that via an external interaction, it is connected with task "T" of Proclet class "X". Now, for Proclet class "X" an instance already exists which has the instance identifier "i". So, an interaction with task "T" of Proclet class "X" with instance identifier "i" is possible. Consequently, for node "n2" an interaction with node "(X,i,T)" may be nominated for extension in the graph.

  If selected, an interaction arc is added leading from node "n2" to the new node "(X,i,T)". Also, the inserted arc has the interaction state "unproduced" and it has a unique interaction identifier.

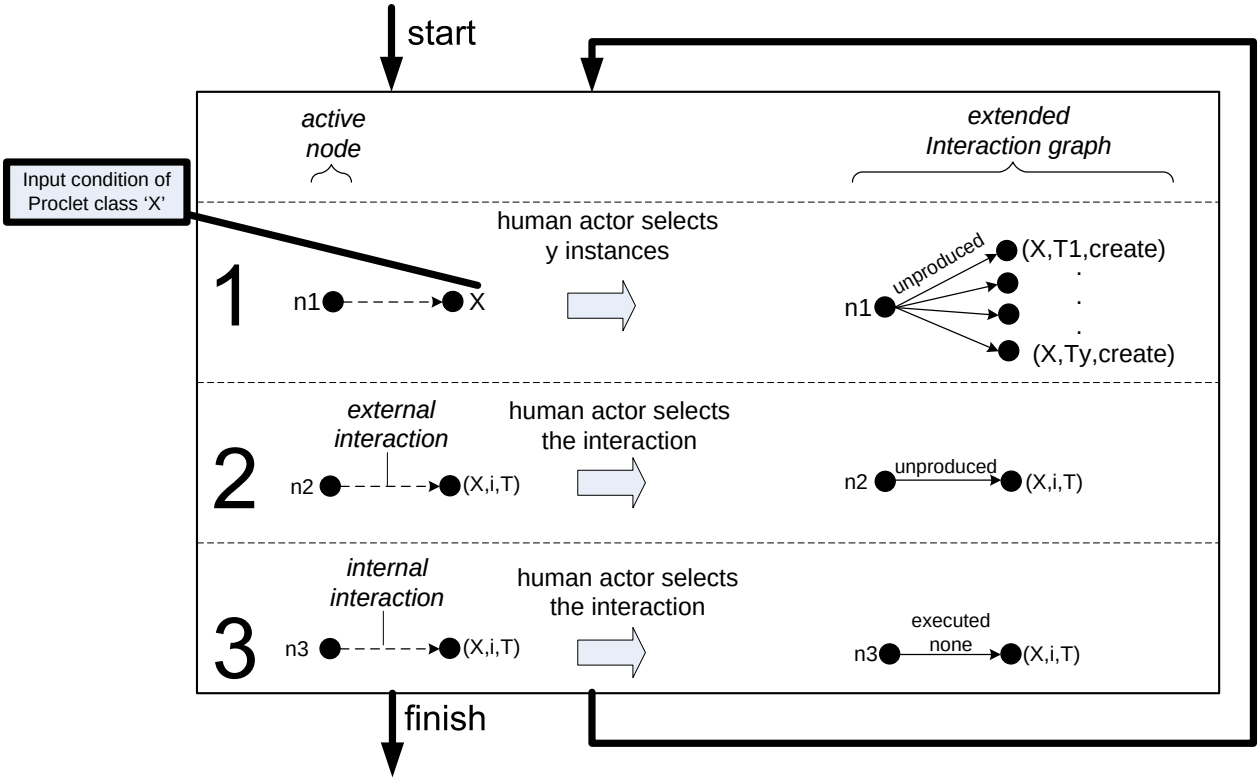


Figure 9.25: Schematic representation of the procedure for extending an interaction graph for an entity. Each dot represents an interaction point.

Note that for Proclet class "X" a future Proclet instance may also be found in the interaction graph. That is, there is an interaction node referring to Proclet class "X" which has the temporary instance identifier "Ti". In that case, the nomination of the interaction and the subsequent extension of the graph is done in a similar way to that for an existing Proclet instance.

- The third situation relates to an internal interaction. That is, for active node "n3" that is under consideration, the following can be observed. Note that "n3" may relate to an existing or future Proclet instance. Looking at the corresponding interaction point of node "n3" in its Proclet class "X", it can be seen that via an internal interaction, it is connected with task "T" of the same Proclet class. As we are dealing with an internal interaction, task "T" occurs in the same Proclet instance as that to which node "n3" refers (say (temporary) instance identifier "i"). Consequently, for node "n3" an internal interaction with node "(X,i,T)" may be nominated for extension in the graph.

  If selected, an interaction arc is added leading from node "n3" to the new node "(X,i,T)". Also, the inserted arc has interaction state "executed none" and it has a unique interaction identifier.

Subsequently the graph is extended in the manner described above, the procedure is repeated. That is, a new set of active nodes is identified and for each of them it is determined in which interactions the node can be involved in. In this way, a human actor can select new interactions or can indicate that he is finished.

Note that the interactions that are defined between interaction nodes in the graph might be limited by the cardinality and multiplicity values of the involved input and output ports. After a human actor is done with defining the interactions that need to take place, it is checked whether the new interactions are in line with the cardinality and multiplicity values of the ports. For that, the interaction graph of the entity itself needs to be considered together with the interaction graphs of other entities. If this is not consistent, then the extended graph for the entity is rejected and the human actor has the option to define the interactions again.

Performatives

Performatives are sent between Proclet instances. For such a performative we have already indicated that it contains three different attributes. However, more attributes are relevant for an entity. Therefore, we present the attributes, and their meaning below, that are most relevant to a performative. Note that these attributes are based on the ones that are presented in [6, 7].

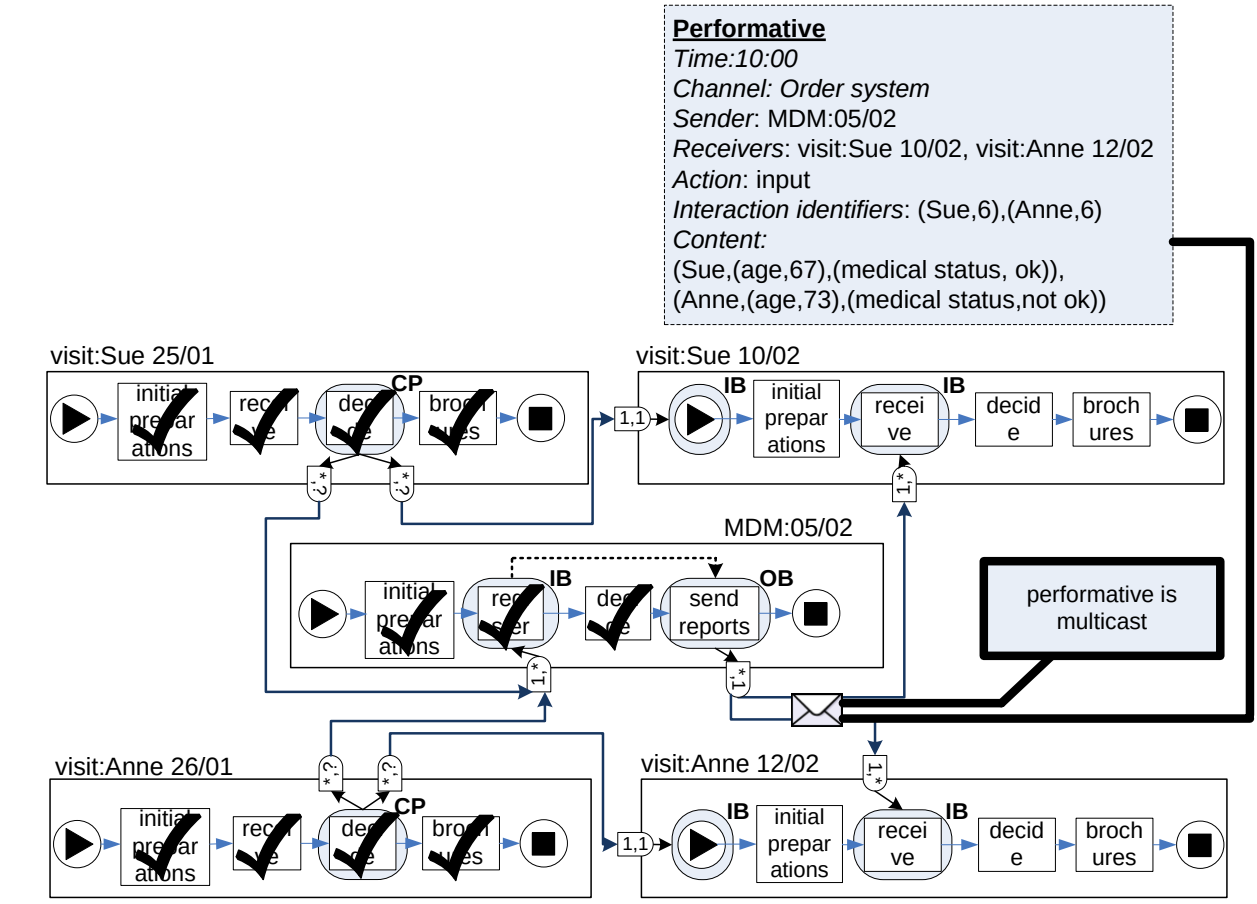
- *Time*: the moment the performative was created.
- *Channel*: the medium used to exchange the performative.
- *Sender*: the identifier of the Proclet instance creating the performative.
- *Set of receivers:* the identifiers of the Proclet instances receiving the performative, i.e. a list of recipients.
- *Action*: the type of the performative.
- *Content*: the actual information that is being exchanged.
- *Set of interaction identifiers*: a list of interaction identifiers. In particular, for the interaction arcs for which the performative is sent, the associated interaction identifier is added to this set.

The role of the action attribute deserves some special attention. This attribute can be used to specify the illocutionary point of the performative. The five illocutionary points identified by Searle [29] (assertive, directive, commissive, declarative, expressive) can be used to specify the intent of the performative. Examples of typed performatives identified by Winograd and Flores are request, offer, acknowledge, promise, decline, counter-offer, and commit-to-commit [32] which each represents a change in the state of a conversation. In the model no restriction is made as to any single classification of performatives (i.e. a fixed set of types). It is important to use the experience and results reported by researchers working on the language/action perspective [32] as these give an insight into the broader requirements in this area. Of course, it is possible to add more attributes to a performative.

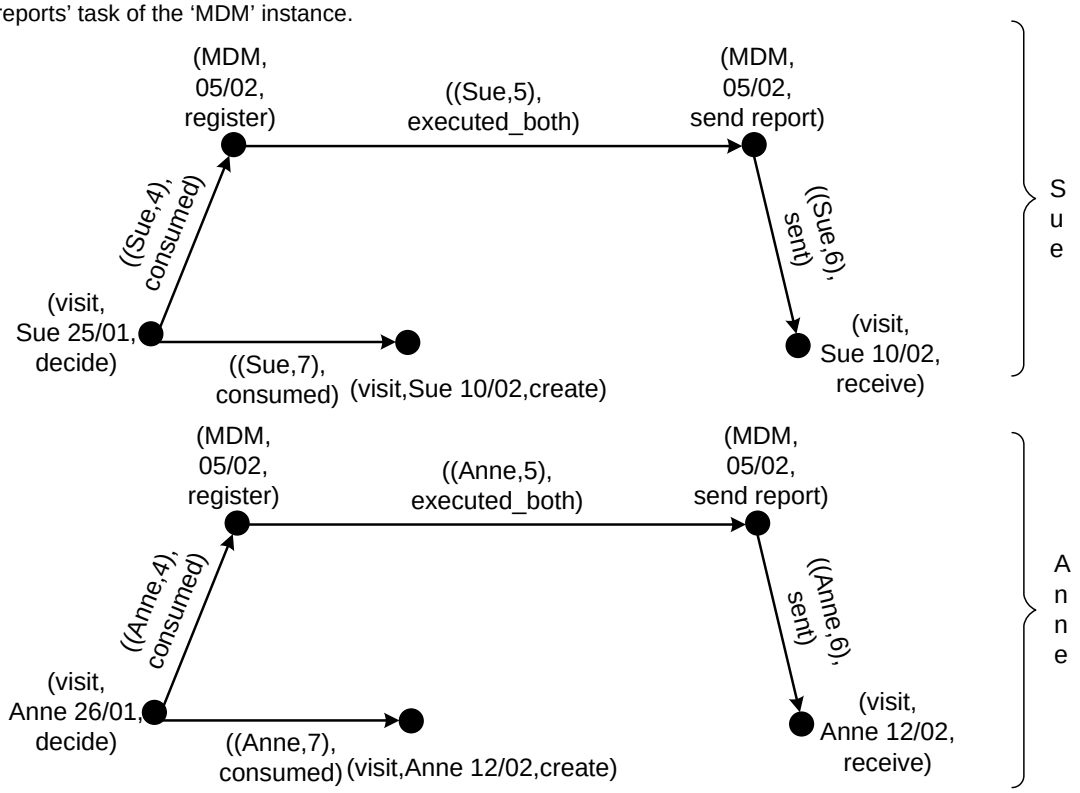
For entities, the "content" field of a performative can be used for exchanging data between Proclet instances in a structured way. However, first we need to remember that a performative may be multicast to multiple receivers. That is, for different Proclet instances of the same Proclet class, the performative has the same task as its destination. This is illustrated in Figure 9.26 in the context of the second scenario. Here, we see for both "Anne" and "Sue" that a performative is sent from the "send reports" task of the "MDM" Proclet instance. For both of them, the performative has the "receive" task of the "visit" Proclet class as its destination. However, for "Sue" the performative needs to be received by the Proclet instance which has "Sue 10/02" as instance identifier and for "Anne" the performative needs to be received by the Proclet instance which has "Anne 12/02" as its instance identifier.

Now, we explain how the "content" attribute is used for exchanging data between Proclet instances of an entity in a structured way. In order to do so, we require that for this attribute a fixed data structure is used. For this data structure, we may have a list of entity identifiers. In turn, for each entity identifier we may have a list of name-value pairs. This is illustrated in Figure 9.26 for the performative that is multicast to the "visit" Proclet instances of "Sue" and "Anne". For the "content" attribute we can see that there is an entity identifier element which has as its identifier "Sue" and that has two name-value pairs. These name-value pairs indicate that Sue is 67 years old and that her medical status is ok. Similarly, for "Anne" we can see that she is 73 years old and that her medical status is not ok.

Note that there is a close link between the information that is contained in the "content" attribute and the information contained in the "set of interaction identifiers" attribute. That is, for every interaction identifier in the "Interaction identifiers" field, a corresponding data element may be found in the "content" attribute which has the same entity identifier.



a) The Proclet instances that need to be performed for 'Sue' and 'Anne'. Currently, a performative is sent from the 'send reports' task of the 'MDM' instance.



b) Current state of the interaction graphs for 'Sue' and 'Anne'

Figure 9.26: Illustration of the attributes of a performative and their content.

9.1.4 Inter-Workflow Service

In this section, the inter-workflow support features of the YAWL4Healthcare WfMS are presented. These features are presented in the context of the scenario that is discussed in Section 9.1. Note that the scenario is shown again in Figure 9.27 for convenience reasons. Remember that for "Sue" during the first visit it was decided that a second visit is needed, and that she needs to be discussed during a multidisciplinary meeting. Moreover, the result of the multidisciplinary meeting is required as input for the second visit. For "Anne" the process is the same.

Note that in comparison to the scenario presented in Chapter 6 of [19], an exception interaction point has now been defined for the "MDM" Proclet class. When an exception occurs for the "MDM" Proclet class, this exception interaction point allows that for an entity affected by the exception, an interaction with the "register" task of an existing or future "MDM" Proclet instance can be defined, i.e. a patient can be registered for another multidisciplinary meeting.

First, we discuss in Section 9.1.4 how a model can be defined in order to allow for inter-workflow support. Then, in Section 9.1.4, it is explained how these models are enacted such that cases can be created and that interactions between Proclet instances can be realized. Next, in Section 9.1.4, it is elaborated upon how exceptions can be handled.

Modeling Support

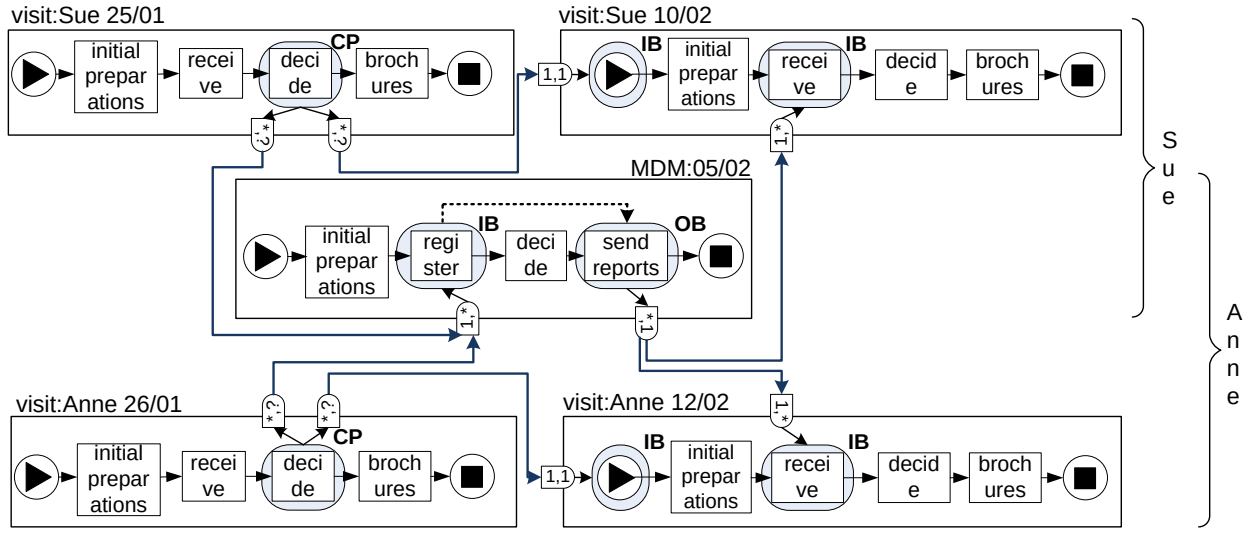
In order to make use of the inter-workflow support features of the YAWL4Healthcare WfMS, first Proclet classes and the external interactions between them need to be defined. Note that process definitions are defined at the engine side (YAWL) and that the extensions on top of these definitions are defined at the Inter-Workflow Service side (via the Interaction Definition Editor).

YAWL Editor Figure 9.28 shows how the process definition of the "visit" Proclet class is defined via the YAWL editor. Every task in the process definition is a flow task as they do not need to be scheduled. However, tasks for which interactions may be necessary are indicated by a plug-in icon. The execution of the corresponding workitem for them is delegated to the Inter-Workflow Service. This needs to be defined via the "Task Composition Details" of the task. Flow tasks which need to be performed via a worktray are indicated by a single person icon.

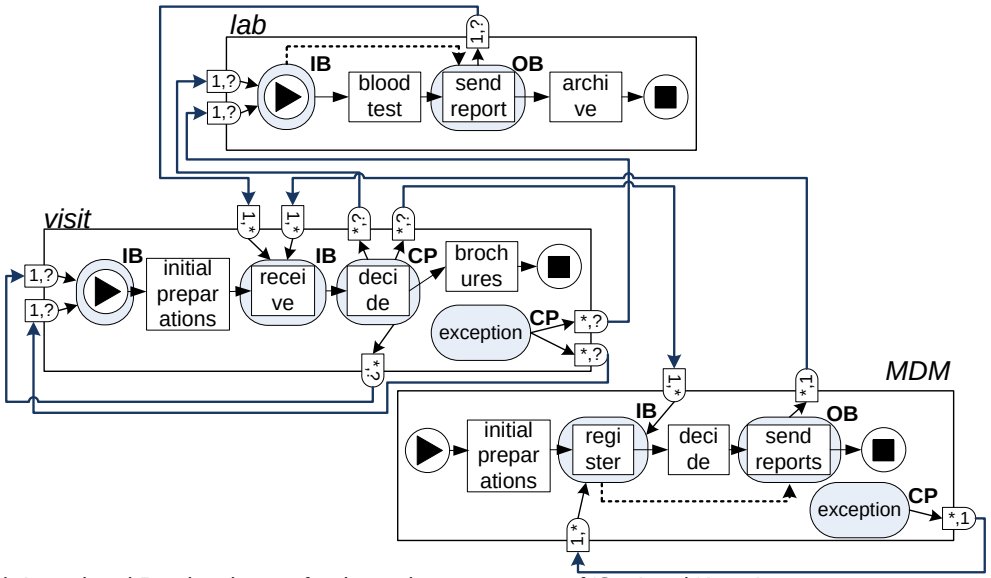
For both the "initial preparations" and "decide" task the corresponding task decomposition details are shown. It is important to note that the Inter-Workflow Service has been implemented as a YAWL Custom Service. This means that the execution of a workitem for a task can be delegated to a service if needed. The "YAWL Registered Service Detail" field of the decomposition details for the "decide" task shows that the execution of the workitem is delegated to the Inter-Workflow Service. For the "initial preparations" task it is defined that it can be performed via the Workflow Client Application. Moreover, it can be seen that for both tasks, an "entities" data variable has been defined which is a complex data type. The reason for defining the variable is as follows.

In order to create an interaction graph, a user needs to provide at run-time the names of the entities for which such a graph needs to be created. Then, the graphs can be created if a workitem for a configuration interaction point is performed, i.e. in this case if a workitem for the "decide" task is performed. As the Inter-Workflow Service has been implemented as a YAWL Custom Service this has the consequence that, at run-time, the names of the entities can only be passed on to the service via a workitem whose execution is delegated to the service. More precisely, this can only be achieved via a data variable that has been defined for the task of the workitem.

Consequently, the names of the entities need to be provided if the workitem for the "initial preparations" task is performed. As the workitem is performed via the Workflow Client Application, the names of the entities can easily be filled in via the form that is automatically generated. Afterwards, these names are passed on the service via the "entities" variable if the workitem for the "decide" task is performed.



a) Patient processes of patients ‘Sue’ and ‘Anne’.



b) Associated Proclet classes for the patient processes of ‘Sue’ and ‘Anne’.

Figure 9.27: The inter-workflow support features of the YAWL4Healthcare WfMS will be presented using the running example shown.

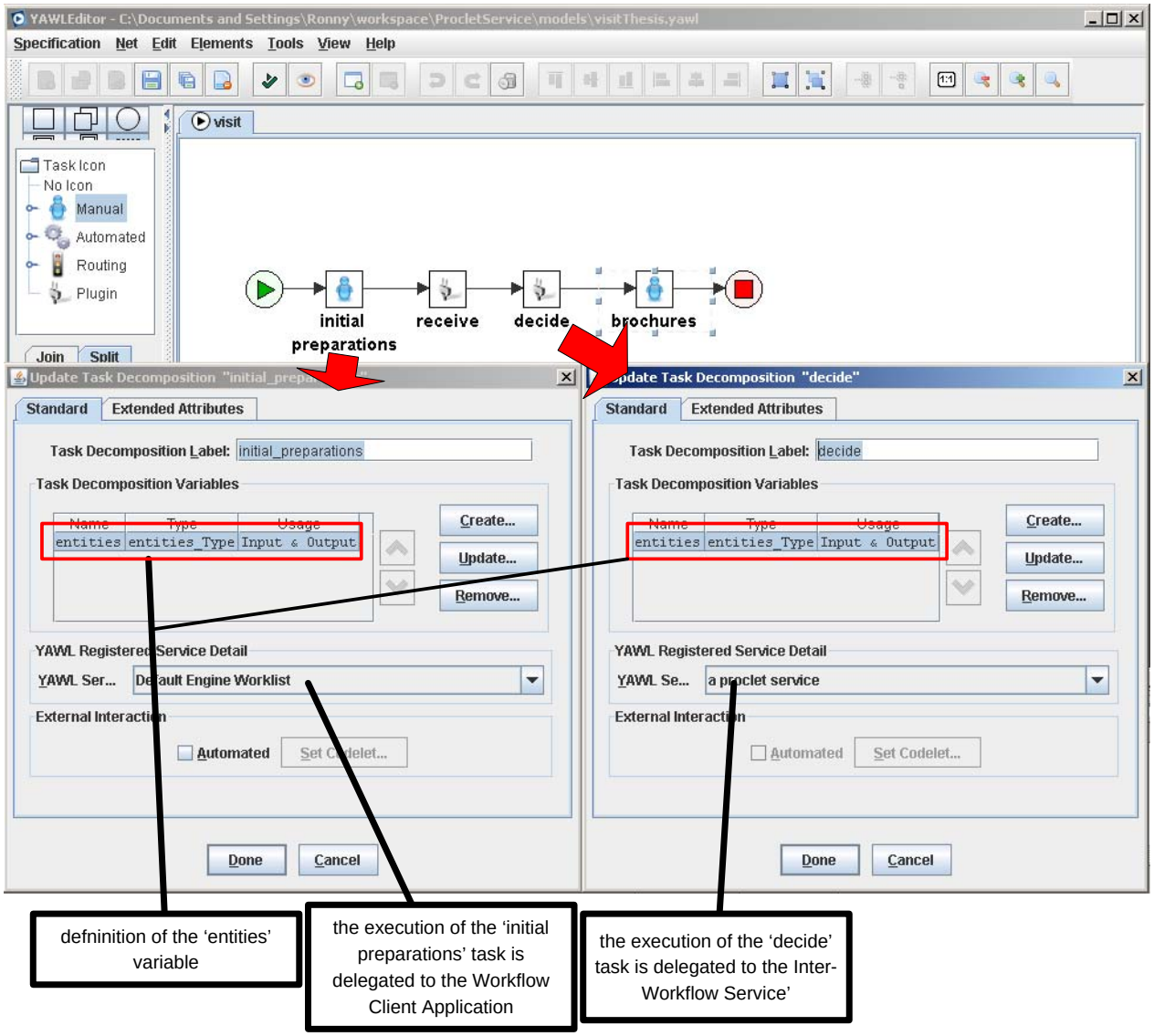
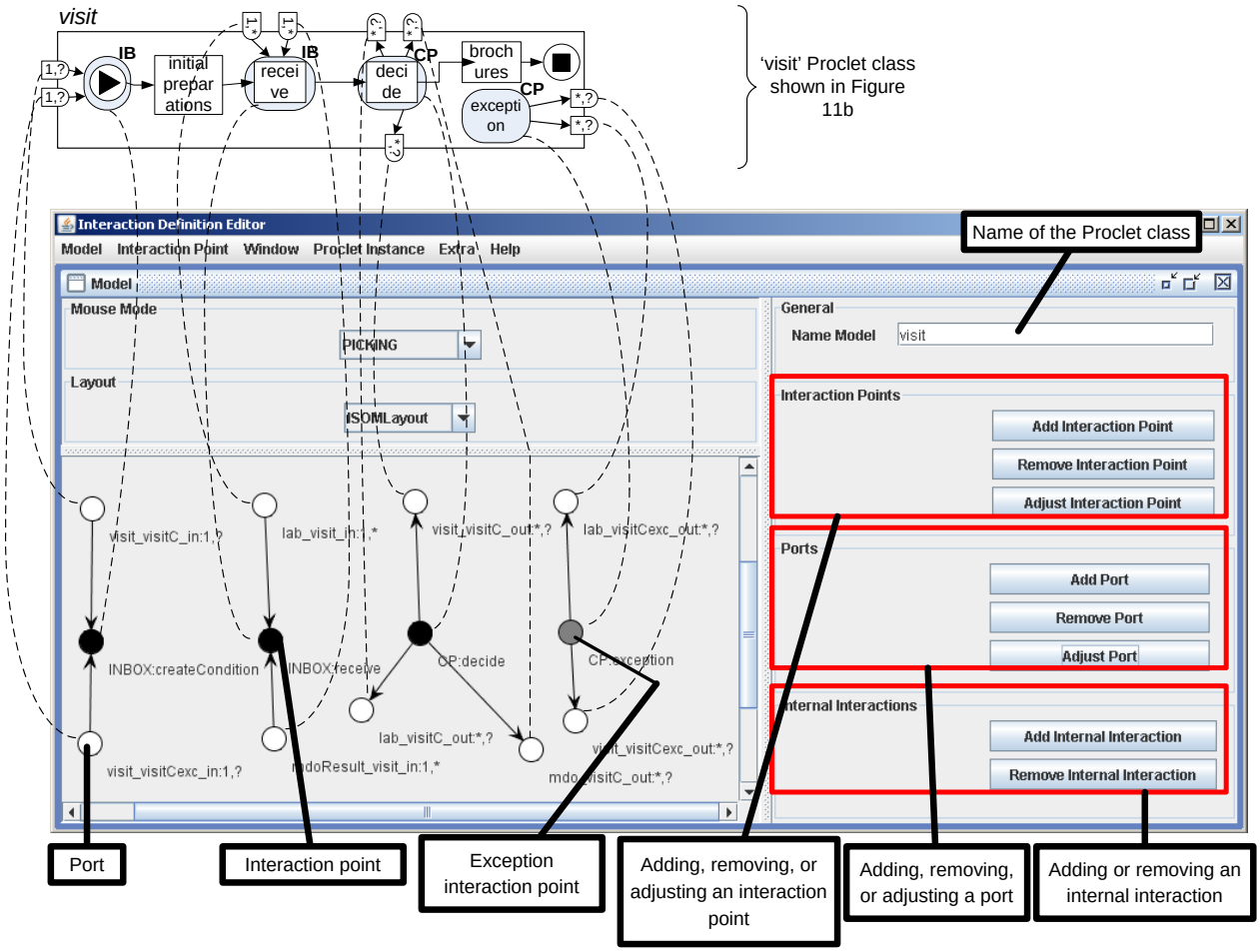
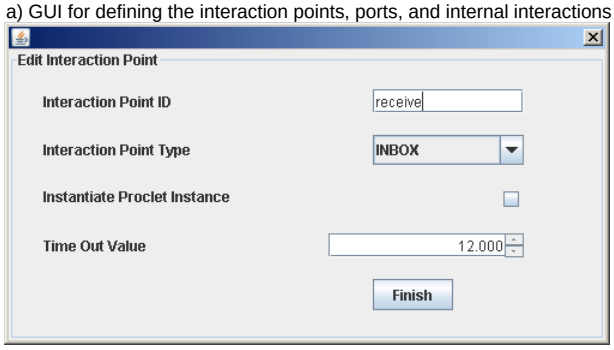


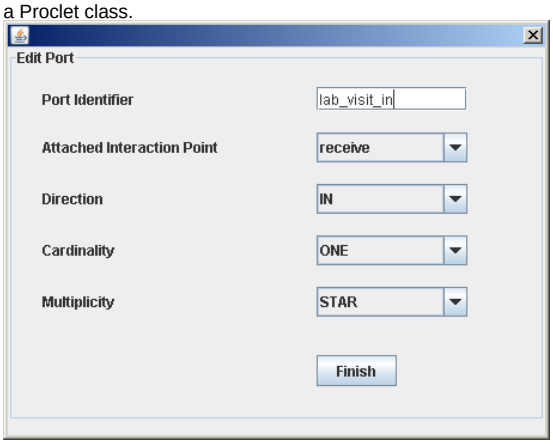
Figure 9.28: Defining the process definition of a Proclet class in the YAWL editor.



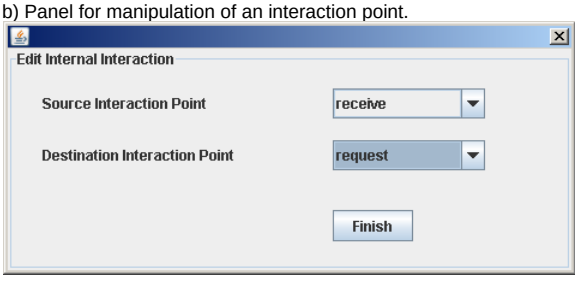
a) GUI for defining the interaction points, ports, and internal interactions of a Proclet class.



b) Panel for manipulation of an interaction point.



c) Panel for manipulation of a port.



d) Panel for defining an internal interaction.

Figure 9.29: Definition of the interaction points, ports, and internal interactions of a Proclet class.

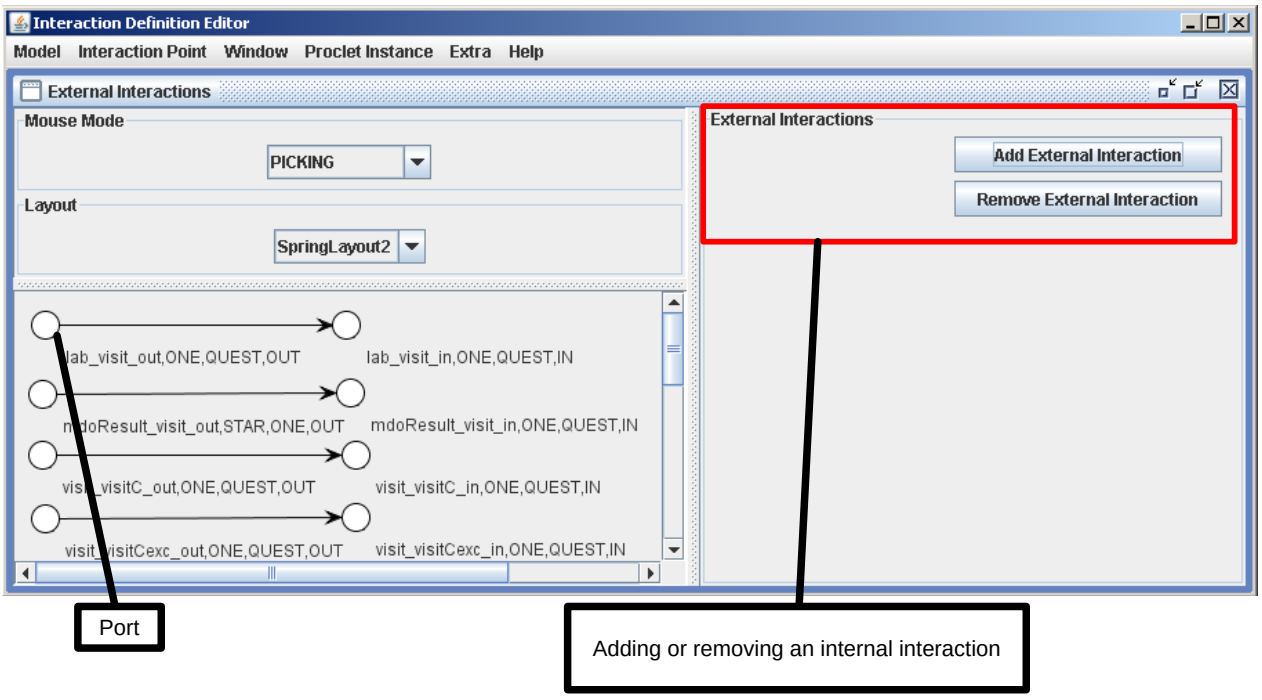


Figure 9.30: Definition of the external interactions of Proclet classes.

Interaction Definition Editor The Interaction Definition Editor is distributed as part of the YAWL Proclet Service, and can be found in the `tomcat/webapps/procletService` directory along with two launchers, 'runEditor.bat' for Windows based systems and 'runEditor.sh' for *nix based systems, and an `editor.properties` file. The editor depends on a number of external libraries, which are normally located either in the `tomcat/lib` directory of YAWL4Study installations or the `tomcat/webapps/procletService/WEB-INF/lib` directory of Enterprise installations, and using the appropriate launcher to start the editor will locate the required libraries, assuming the editor is being run on the same machine as the proclet service is installed. If that is not the case, download a copy of the proclet service from the YAWL download pages, unzip it and copy all of the jar files from its `tomcat/webapps/procletService/WEB-INF/lib` directory to a directory on the local classpath.

Figure 9.29a shows the GUI of the Interaction Definition Editor that allows for the definition of interaction points, ports, and internal interactions for an existing process definition. More specifically, at the right side of the GUI, these details are defined whereas at the left side they are visualized.

In order to illustrate the definition of these details, Figure 9.29a shows the interaction points, ports, and internal interactions that are defined for the "visit" Proclet class (this is defined in the "Name Model" field at the top right). Note that at the top of Figure 9.29a the "visit" Proclet class is shown. Via dotted arcs, interaction points and ports are linked with the associated interaction points and ports that are shown in the GUI. At the left side, interaction points are visualized by a black dot together with the type of the interaction point and its identifier. As a special case of this, an exception interaction point is visualized as a grey dot. Ports are visualized by a white dot together with its identifier and the associated cardinality and multiplicity. Moreover, via an arc it is indicated to which interaction point a port belongs to. If an arc is leading from an interaction point to a port then we are dealing with an output port whereas for an input port this is exactly the opposite. Note that an internal interaction has not been defined for the "visit" Proclet class. Internal interactions are visualized via a dotted arc that leads from the source to the destination interaction point.

Interaction points, ports, and internal interactions can be manipulated via the "Interaction Points", "Ports", and "Internal Interactions" panels respectively. The associated panels for editing the details of an interaction point, a port, and an internal interaction are shown in Figures 9.29b to 9.29d respectively. In particular,

in Figure 9.29b it is defined that the "receive" interaction point is an inbox interaction point, that no instance of the Proclet class needs to be created once it is triggered, and that the timeout value is 12.000 milliseconds. In Figure 9.29c it is defined that the "lab_visit_in" port is an input port and attached to the "receive" interaction point. Moreover, it has cardinality "1" and multiplicity "*". Finally, in Figure 9.29d, the source and destination interaction point of an internal interaction can be selected.

Once Proclet classes have been defined, external interactions can be defined. The corresponding GUI is shown in Figure 9.30. At the right side of the GUI external interactions can be specified or removed whereas at the left side they are visualized.

In order to illustrate the definition of these details, Figure 9.30 shows a part of the external interactions that are defined for the example in Figure 9.27b. At the left side, ports are visualized by a green dot together with the associated identifier. Moreover, the direction of the arc between two ports indicates the source (tail) and the destination (head) of the external interaction. For example, the "lab_visit_out" output port is connected with the "lab_visit_in" input port. Via this connection, the result of a lab test is sent to the "receive" task of a "visit" Proclet.

Enactment Support

In this section, we demonstrate how Proclet classes can be enacted in the YAWL4Healthcare WfMS. This is done in the context of the running example that is discussed at the beginning of Section 9.1.4. For the demonstration we assume that all the Proclet classes of Figure 9.27b and their relationships have been configured in the system.

The demonstration is started by "Sue" for whom a visit to the outpatient clinic is required. So, for "Sue" an instance of the "visit" Proclet class exists which has "64" as instance identifier. However, in order to be able to have interactions with other Proclets, first an entity identifier should be created. As discussed before, this can be done during the "initial preparations" task. The corresponding form in the Outlook client is shown in Figure 9.31a. Via the "entities" data variable, data for entities can be filled in. In the "entity_id" field, the entity identifier can be filled in. Moreover, some additional data can be filled in via the "name value pair" elements. As the value of the "entities" data variable has a complex data type, in Figure 9.31a this is represented as XML code. However, in the Workflow Client Application of the YAWL WfMS, a form is automatically created for any (complex) data type which is more user friendly. The corresponding form is shown in Figure 9.31b.

For both forms, we see for "Sue" that "Sue" has been filled in as the entity identifier, she has age 65, and she is 1.75 m long. Remember that the additional data that has been filled in is included in the performatives that are sent from this Proclet instance. In this way, the data becomes available to the Proclet instances to which the performative is sent.

Afterwards, during the "decide" task, it can be defined what needs to be done next. As the "decide" task is associated with a "configuration" interaction point, the interaction graph for "Sue" can be extended. Below we demonstrate via several screenshots how this is supported in our system. Note that we assume that the reader is familiar with how an interaction graph can be extended. More details can be found in Chapter 6 of [19].

First, in the Interaction Definition Editor, "Sue" is selected as the entity (see Figure 9.32a). Afterwards, the panel shown in Figure 9.32b is presented. At the left side, the interaction graph that has been defined so far is shown. In particular, for nodes that have been colored white, new interactions can be selected. For nodes that have been colored black this is not possible. Following on, at the right side, one of these white colored nodes can be selected in order to define new interactions. For example, for "Sue" we see at the left side a node in the graph for the "decide" workitem. At the left, this node can be chosen for selecting new interactions. Note that for the arc states in an interaction graph, abbreviations will be used. So, for the arc states "unproduced", "consumed", "sent", "executed none", "executed single", "executed both", and "failed", the abbreviations "U", "C", "S", "EN", "ES", "EB", and "F" will be used respectively.

For a node that is selected, a new screen is presented which is shown in Figure 9.32c. At the left side, the interaction graph that has been defined so far is shown. In particular, the node that has been selected for defining new interactions is colored white. The other ones are colored black. At the right side, for the se-

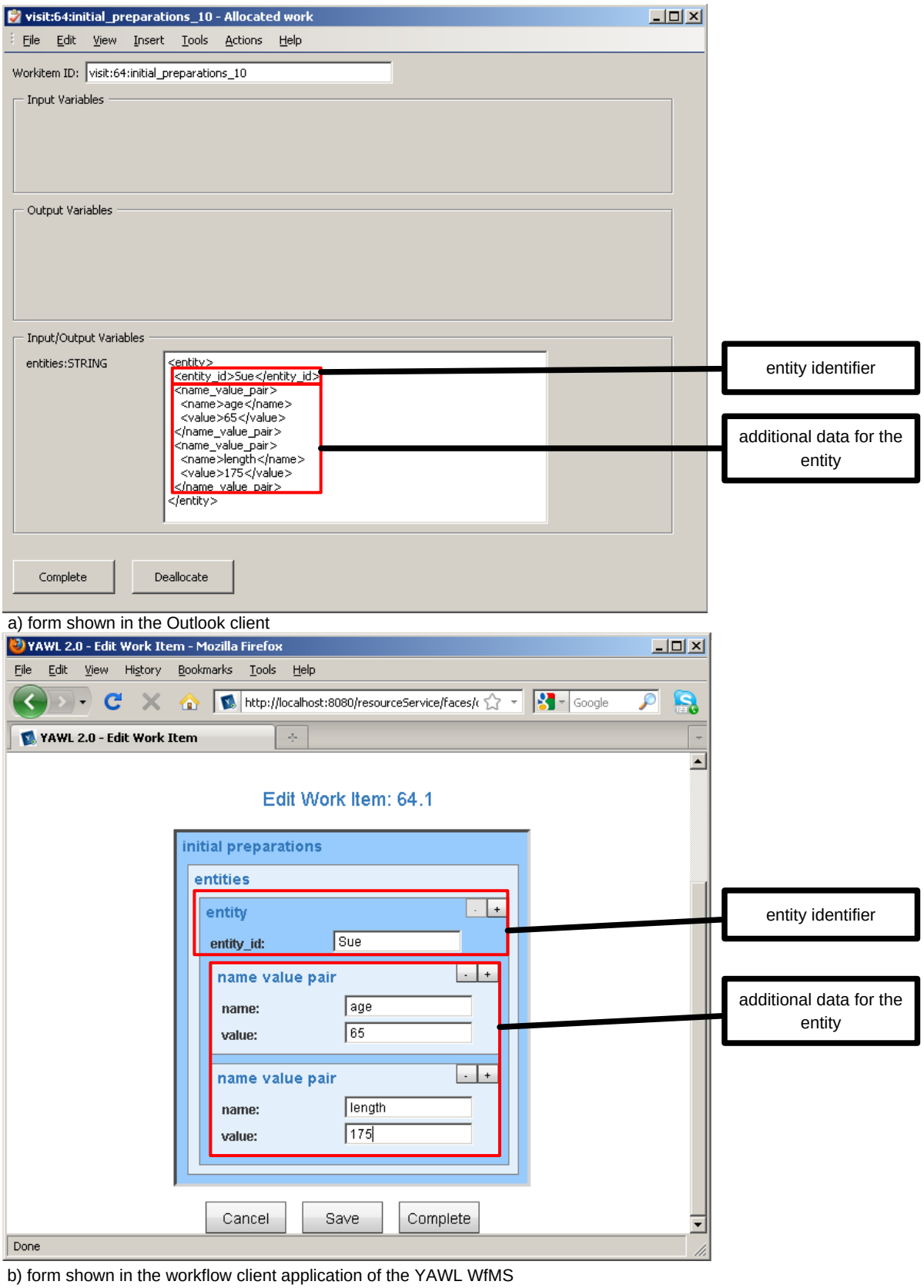


Figure 9.31: Form in which the identifier for an entity can be filled in. Moreover, some additional data can be filled in. If later the workitem for the "decide" task is executed, the information can be used for creating an interaction graph.

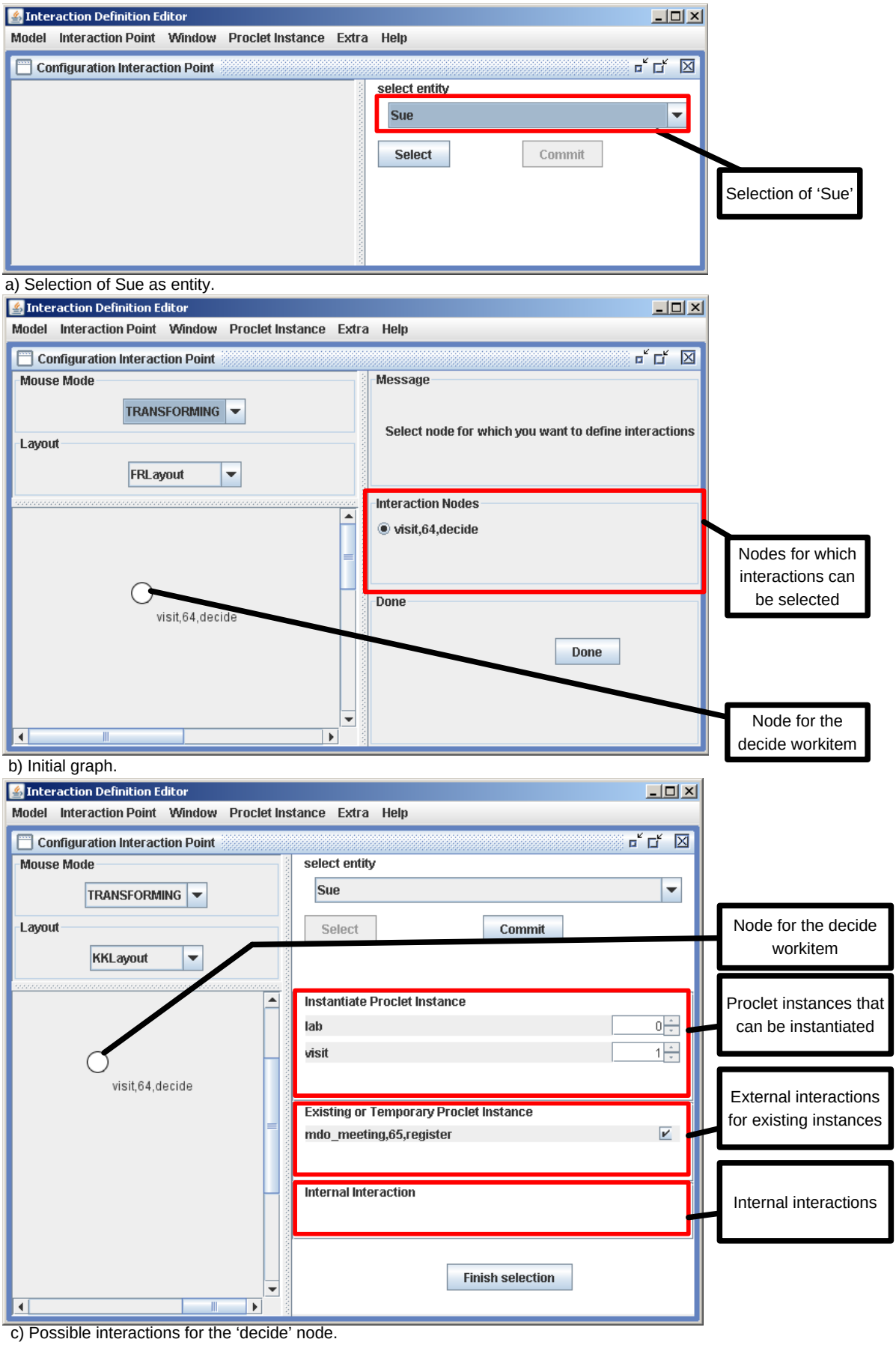
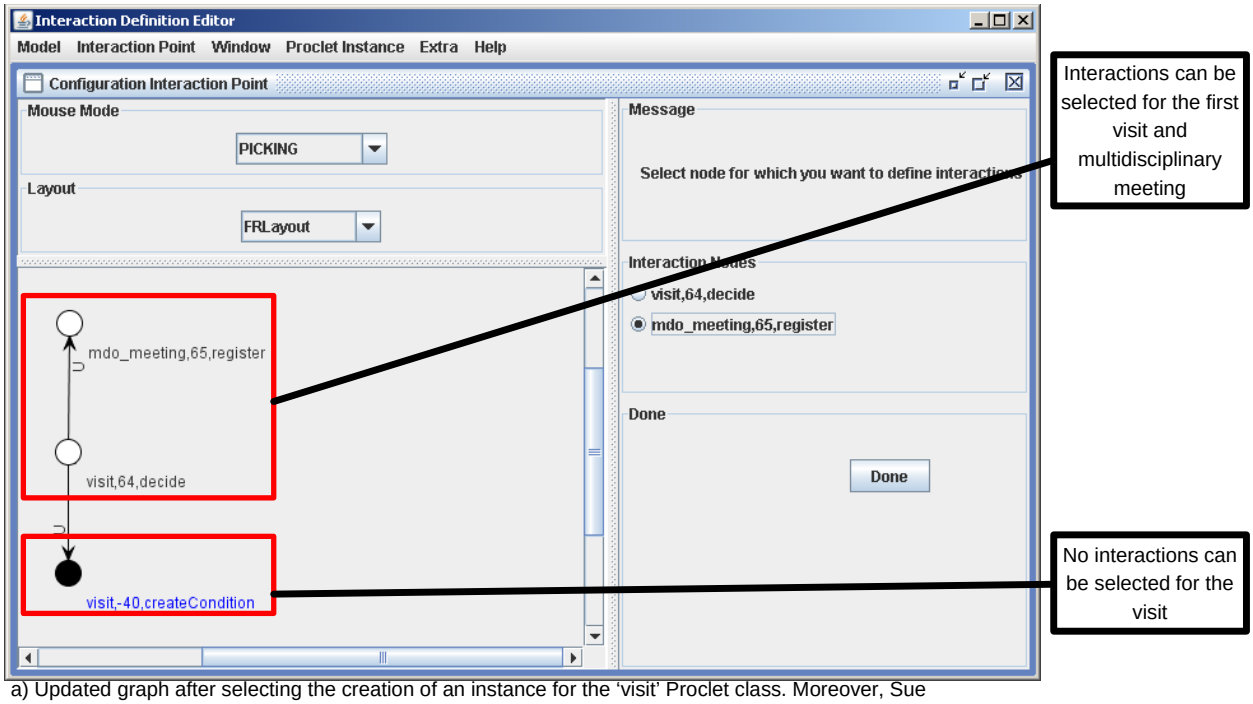
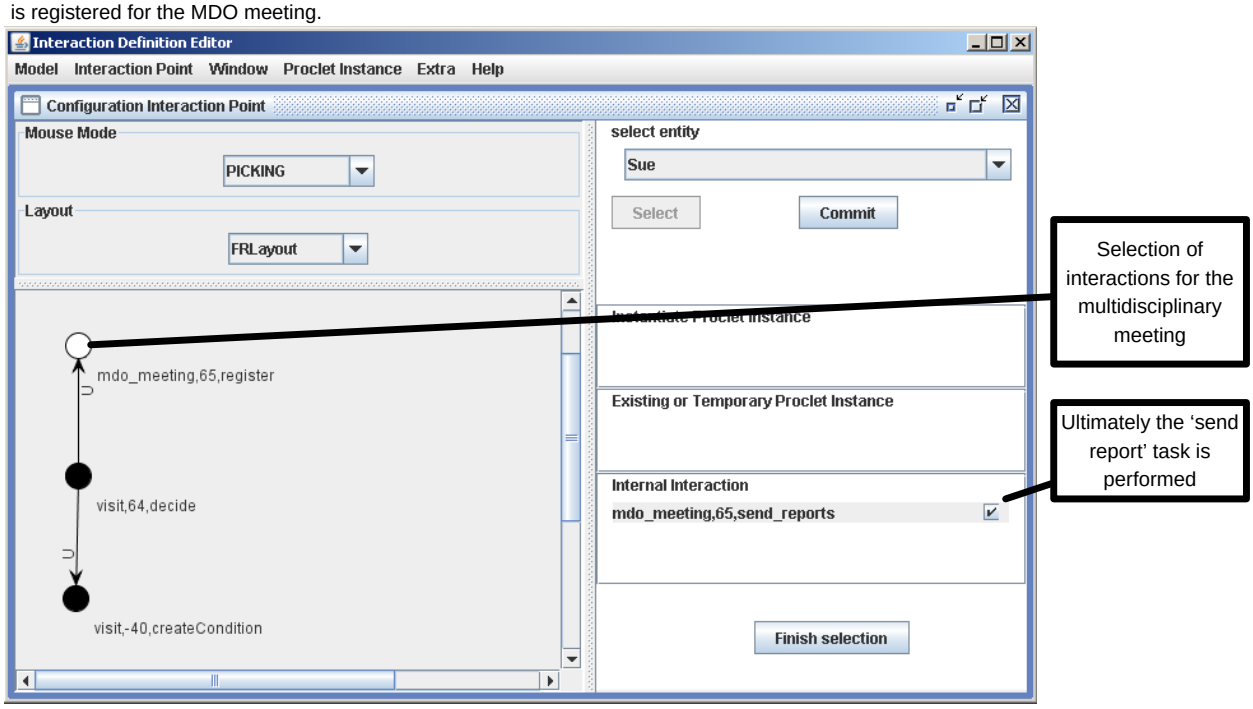


Figure 9.32: Extending the interaction graph for Sue.

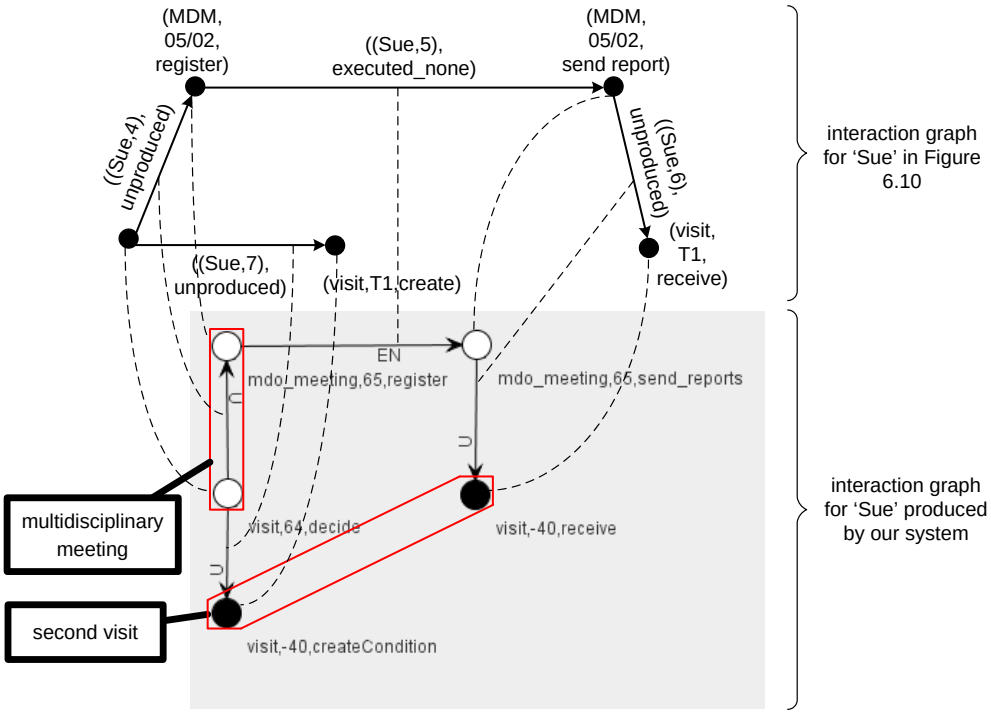


a) Updated graph after selecting the creation of an instance for the 'visit' Proclet class. Moreover, Sue is registered for the MDO meeting.

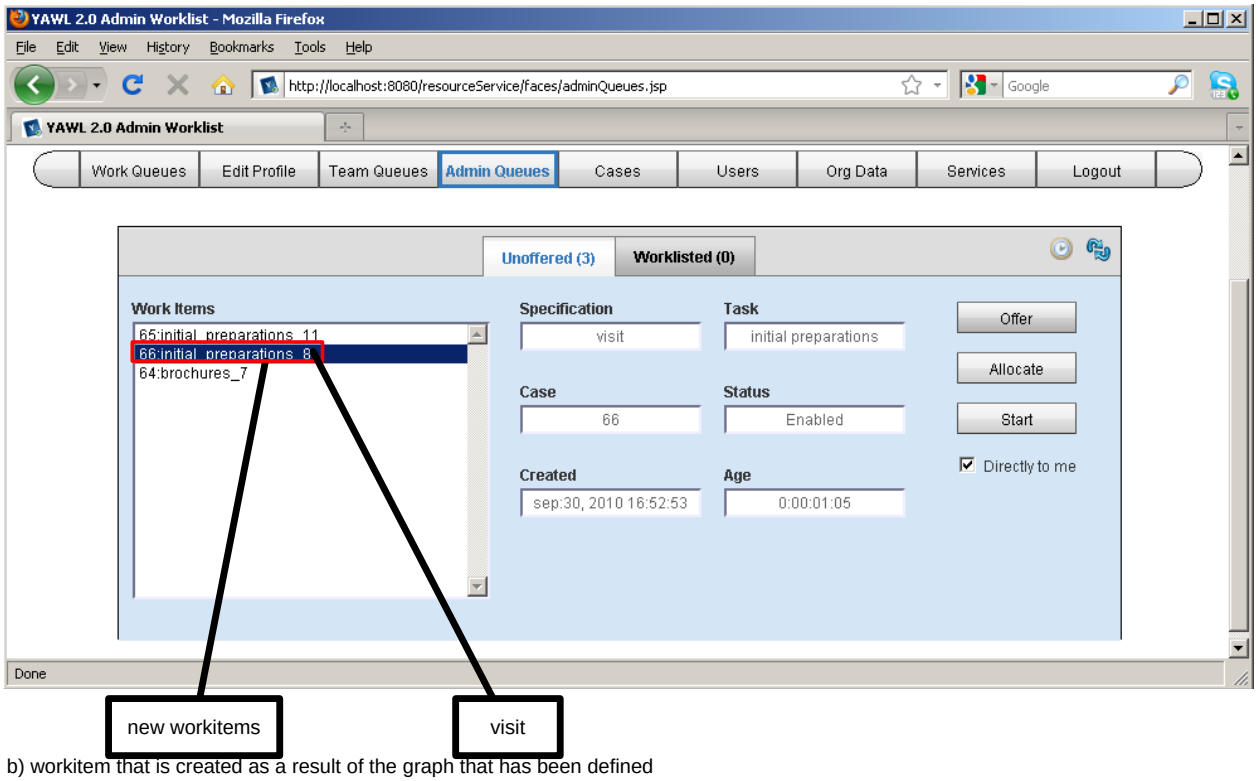


b) Possible interactions for the multidisciplinary meeting.

Figure 9.33: Extending the interaction graph of Sue with an internal interaction.



a) resultant graph for 'Sue'



b) workitem that is created as a result of the graph that has been defined

Figure 9.34: Final graph for Sue. Moreover, as a result of the graph, the workitems that need to be performed are shown.

lected node, the possible interactions are presented via three different panels. In the "Instantiate Proclet Instance" panel the interactions are shown which lead to the instantiation of a Proclet class. For each of them, it can be indicated how many instances need to be instantiated. In the "Existing or Temporary Proclet Instance" panel, interactions with existing or future Proclet instances can be selected by checking the checkbox of the respective interaction. Note that future Proclet instances have a negative case identifier. In the "Internal Interaction" panel, internal interactions can be selected by checking the checkbox of the respective internal interaction. For example, for "Sue" one instance of the "visit" Proclet class will be created. Moreover, for the multidisciplinary meeting an instance exists which has "65" as the instance identifier. As a result, she is also registered for this meeting.

After selection of the interactions, the panel is shown again which allows for the selection of a node for which interactions can be selected. So, the process can be repeated until no more interactions are required. An example of this can be seen in Figure 9.33 which is a follow up to Figure 9.32. In particular, in Figure 9.33a a node is shown for the future instance of the "visit" Proclet class. Note that this node has a temporary instance identifier which is a negative number. Moreover, via the "mdo_meeting,65,register" node, the patient is registered for the multidisciplinary meeting. For the "mdo_meeting,65,register" node that has been selected, the next screen is shown in Figure 9.33b. As can be seen on the right side of this figure, there is only an internal interaction that can be selected. By selecting this interaction, it will be assumed that after registering "Sue" for the multidisciplinary meeting, the "send report" task is performed for her.

The resultant graph for "Sue" is shown in Figure 9.34a. Via rectangles it is indicated which nodes are related. For example, for the multidisciplinary meeting for which "Sue" is registered, the result will serve as input for the next visit ("visit,-40,receive" node). Note that the graph is in line with the example shown in Figure 9.27a. Also, at the top, the resultant interaction graph for "Sue" is shown. Via dotted arcs, corresponding interaction nodes and interaction arcs are connected. For example, the "(MDM,05/02,register)" interaction node is connected with the "mdo_meeting,65,register" interaction node of the scenario that is executed in the YAWL4Healthcare system.

Moreover, as a result of the interactions that are specified, one new workitem needs to be performed. This is shown in the worklist presented in Figure 9.34b. Here, the "66:initial_preparations_8" workitem relates to the second visit that is required.

In a similar fashion as for "Sue", an interaction graph is defined for "Anne" which is shown in Figure 9.35. Also here, via rectangles it is indicated which nodes are related. Moreover, at the top, the resultant interaction graph for "Anne" is shown. Also here, via dotted arcs, corresponding interaction nodes and interaction arcs are connected. Note the graph is in line with the example shown in Figure 9.27.

Exception Handling

In the previous section, we have illustrated the operation of the system under normal circumstances. In this case, we will illustrate the operation of the system where exceptions occur. Therefore, we first demonstrate how an exception at case level is handled followed by the manner in which an exception at workitem level is handled.

Exception at Case Level For the demonstration assume that for both "Sue" and "Anne" we have just finished defining the interactions as a result of the "decide" task during the first visit. So, the interaction graphs shown in Figures 9.34a and 9.35 apply.

Moreover, we assume that the multidisciplinary meeting with instance identifier "65" can not take place anymore because some of the doctors involved need to attend a conference. As a result, the patients that are discussed during the meeting now need to be discussed during the next multidisciplinary meeting which has instance identifier "69". In the system this is supported in the following way.

As a result of the cancelation of the multidisciplinary meeting with instance identifier "65" an exception occurred. This is because both for "Anne" and "Sue" it has been defined that the result of their multidisciplinary examination need to be used as input for the second visit for each of them. This is now not possible anymore.

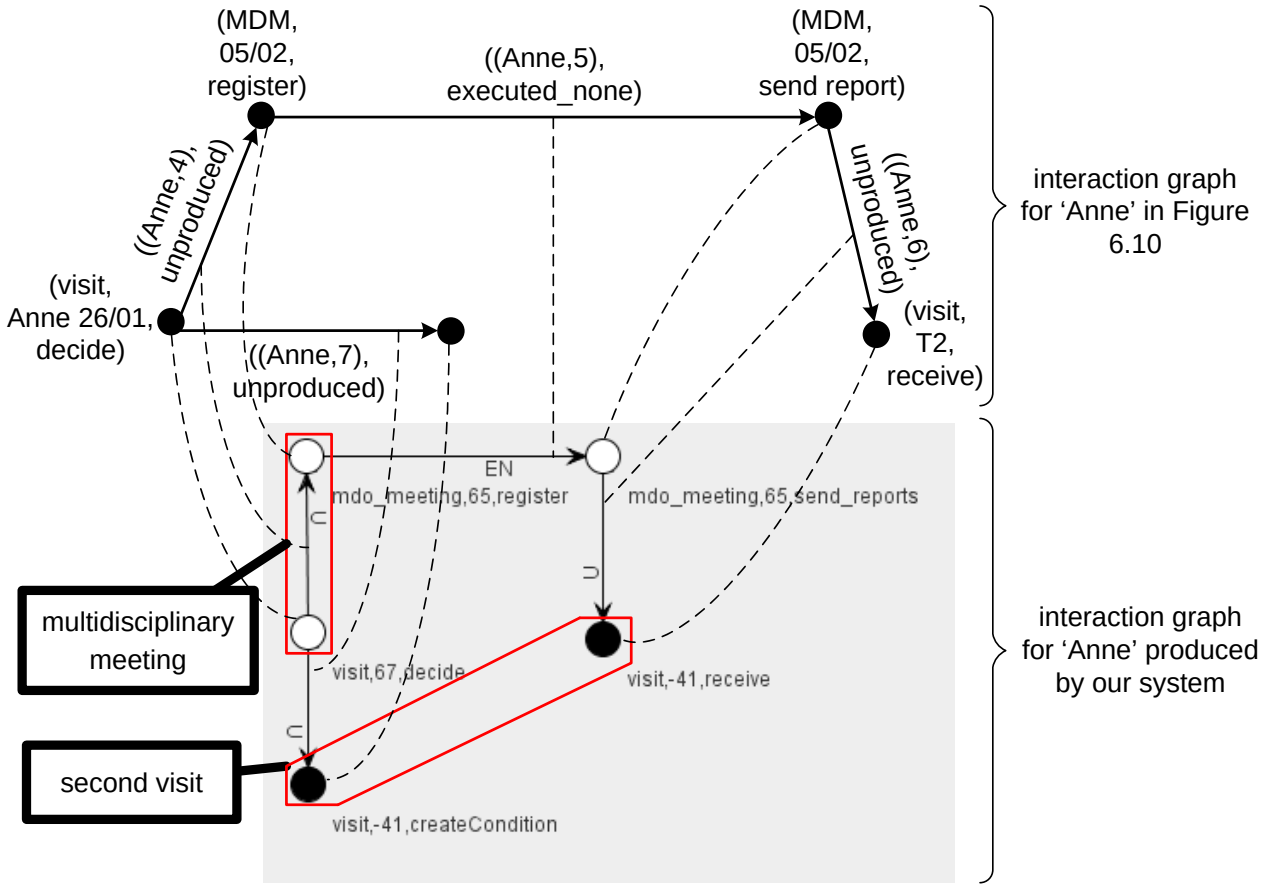


Figure 9.35: Final graph for Anne. Via rectangles it is shown which nodes are related.

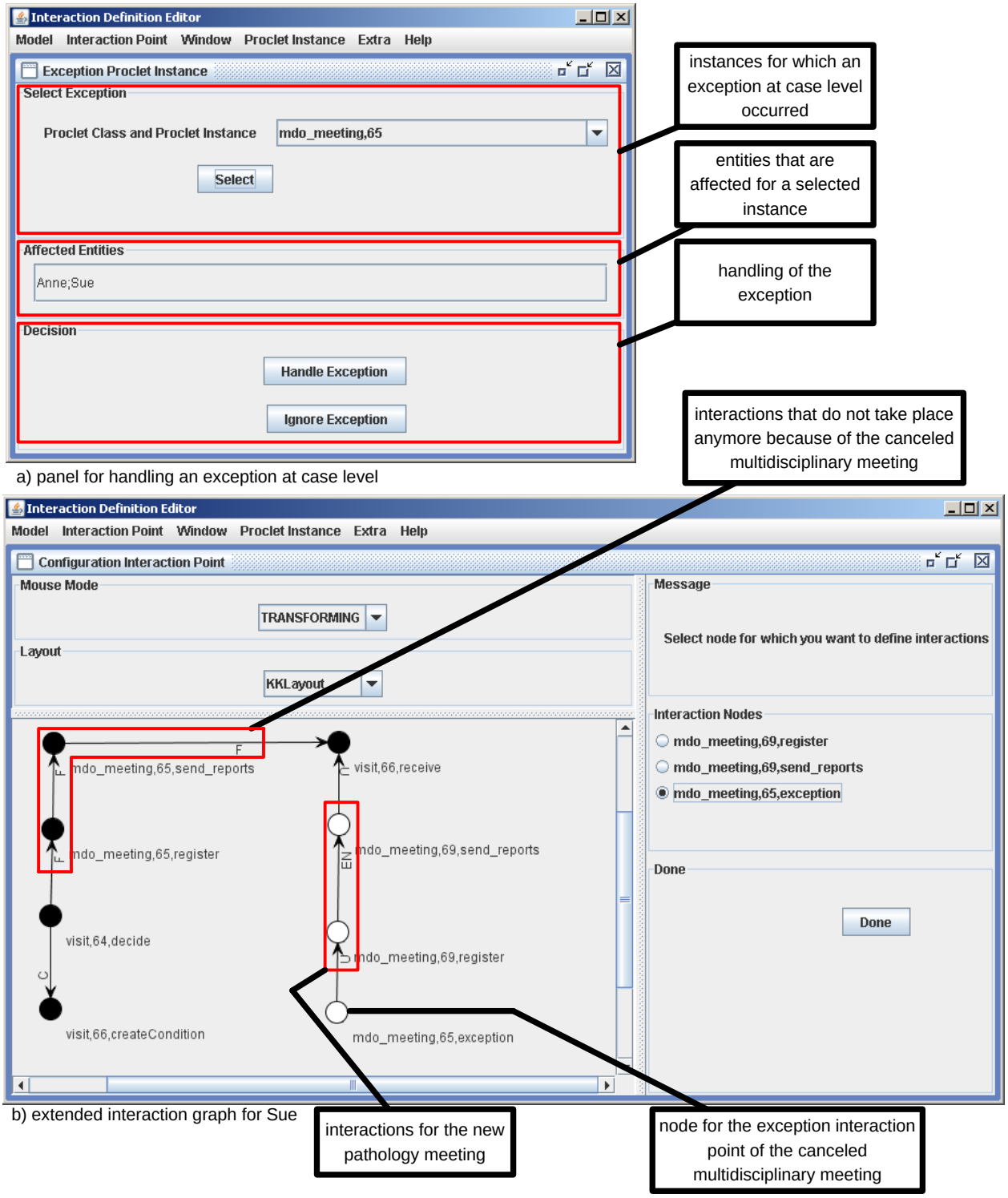


Figure 9.36: Handling of the exception which involves the cancelation of the pathology meeting with instance identifier "65".

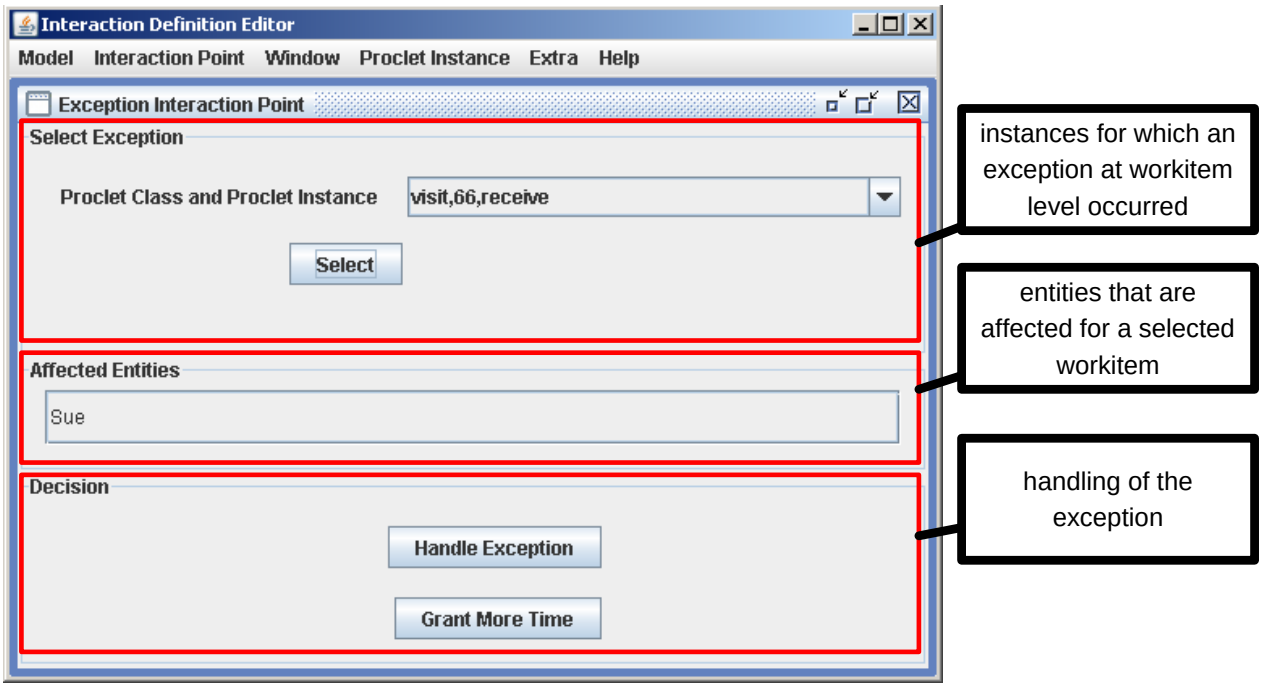


Figure 9.37: Handling of the exception which involves a workitem for which not all required performatives are received.

In Figure 9.36a, the panel is presented which shows exceptions that occurred at case level and how they can be handled. In the top panel, via the drop down box, instances are shown for which the cancelation or completion resulted in an exception. As a follow up to that, for a selected case, the affected entities are shown in the middle panel. Finally, in the bottom panel it can be decided how the exception needs to be handled. Either the exception is ignored ("Ignore Exception" button) or for the affected entities the interaction graphs may be extended ("Handle Exception" button).

For the canceled multidisciplinary meeting we see that both "Sue" and "Anne" are affected. For them we decide to click on the "Handle Exception" button. As an example, in Figure 9.36b it is shown how the interaction graph for "Sue" is extended. That is, using the exception interaction point of the canceled multidisciplinary meeting (node "mdo_meeting,65,exception"), "Sue" is registered for the multidisciplinary meeting which has "69" as its instance identifier (node "mdo_meeting,69,register"). Moreover, the result of the meeting is used as input for the second visit. Note that for the interactions that relate to the canceled multidisciplinary meeting it is indicated that they do not take place anymore, i.e the arcs have state "failed".

The interaction graph of "Anne" is extended in a similar way. Therefore, it is not shown here.

Exception at Workitem Level In this section, we demonstrate how an exception at workitem level is handled by the system. For the demonstration assume that for both "Sue" and "Anne" we have just finished defining the interactions as a result of the "decide" task during the first visit. So, the interaction graphs shown in Figures 9.34b and 9.35 apply.

Moreover, assume that now for "Sue" the second visit takes place. As a result, the "initial preparations" task is performed. Next, the result of the multidisciplinary meeting needs to be received at the "receive" task. However, as the multidisciplinary meeting has not been performed yet, an exception is raised.

In Figure 9.37, the panel is presented which shows exceptions that occurred on workitem level and how they can be handled. In the top panel, via the drop down box, workitems are shown for which the required interactions have not taken place in time. As a follow up to that, for a selected workitem, the affected entities are shown in the middle panel. Finally, in the bottom panel it can be decided how the exception needs to be handled. Either more time is granted for receiving missing performatives ("Grant More Time" button) or

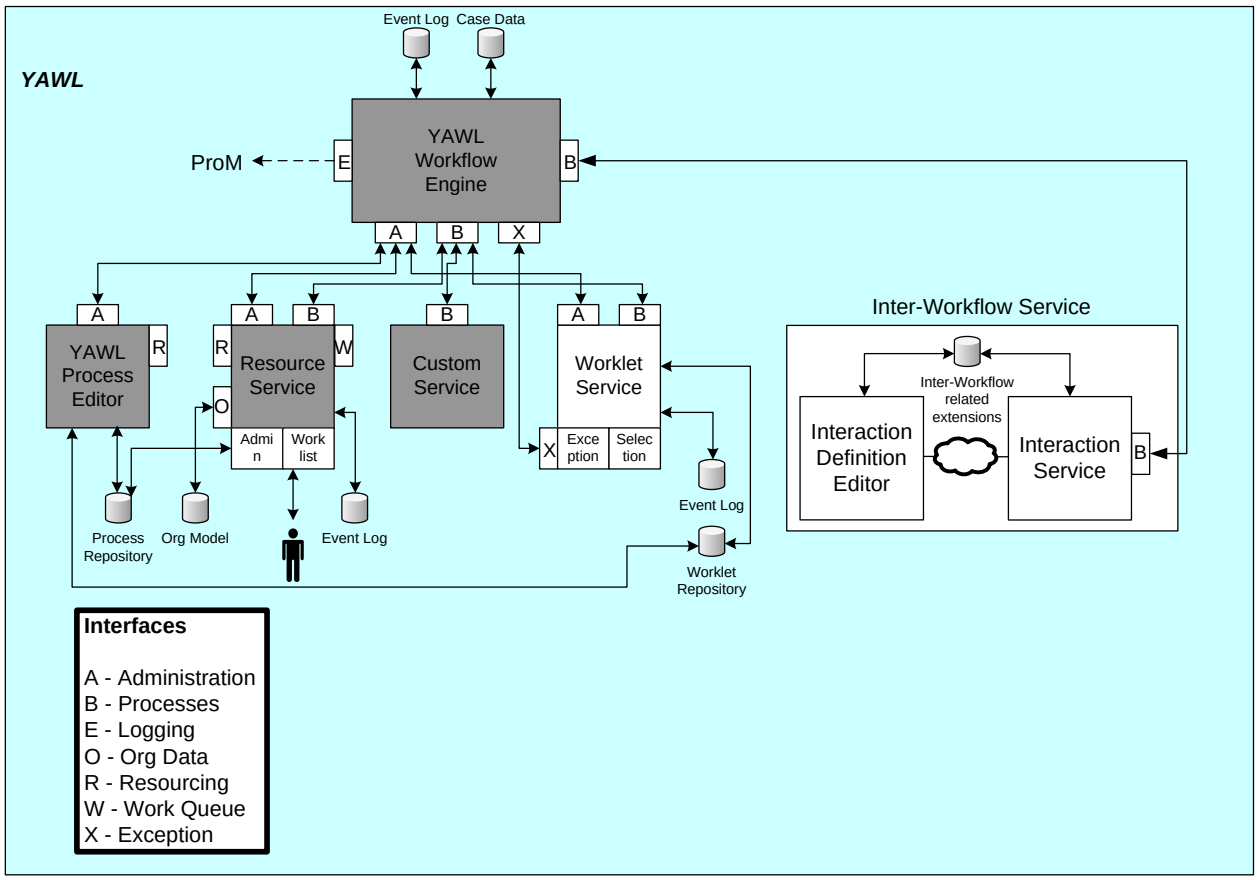


Figure 9.38: Architecture outlining which components of the YAWL WfMS have been used for the Workflow Engine of our system.

for the affected entities the interaction graphs may be extended ("Handle Exception" button).

For example, for the workitem exception of the "receive" task of the second visit for "Sue", we see that indeed only "Sue" is affected and not "Anne". In case it is decided to extend their interaction graphs, this would proceed in a similar way as for the exception at case level. Therefore, it is not shown here.

Architecture

In Figure 9.38c it is shown how the Inter-Workflow Service is connected to the YAWL WfMS. As can be seen in the figure, the service consists of a Interaction Service and an Interaction Definition Editor.

The *Interaction Service* component is responsible for storing and taking care of the interactions that take place between Proclet instances. More specifically, for tasks for which interactions are necessary, a corresponding interaction point is defined at the service side which means that the execution of these tasks is deployed to the Inter-Workflow Service. In this way, for such a task instance the service identifies which interactions are necessary, i.e. whether the sending and receiving of performatives is necessary. If it is, then these interactions are taken care of which also may involve the instantiation of Proclet classes. In addition to this, the Interaction Service takes care of identifying whether exceptions occur (e.g. the cancelation or completion of a Proclet instance). Based on the decision of a human actor, an exception is handled. Finally, based on previously defined interactions for an entity, subsequent interactions that are possible are determined in case the opportunity is offered to extend an interaction graph.

A human actor will only have contact with the Inter-Workflow Service via the *Interaction Definition Editor*. In this way, the component offers the ability to define interactions for an entity, i.e. extending the correspond-

ing interaction graph, both in normal and exceptional situations. Here it should be noted that possible interactions for an entity, which are determined by the Interaction Service, are offered to a human actor via the editor. From these possible interactions, a selection is made and send to the Interaction Service such that new possible interactions are calculated and offered again. In addition to this, identified exceptions are presented such that a human actor can decide how they need to be handled (e.g. take no action or extend the interaction graph for an entity).

Furthermore, for the service, Proclet classes have been defined as an extension of YAWL-nets (e.g. interaction points, ports, cardinality, and multiplicity). These extensions are stored at the Inter-Workflow Service side. In particular, for a YAWL-net these extensions can be defined via the Interaction Definition Editor. Moreover, they can be accessed by both the Interaction Definition Editor and the Interaction Service. Note that the interaction graphs are also stored at the Inter-Workflow Service side.

Chapter 10

Other Services

Because of the open design of the YAWL interfaces, it is possible to design a YAWL Custom Service that will perform the work of a task instance using a wide variety of techniques to meet particular needs. Previous chapters have described the Resource, Worklet and Proclet Services, which, while quite complex, are examples of the sorts of things that can be achieved using Custom Services. This chapter briefly describes a few other custom services in the deployed YAWL set, their varied purposes and their use. Please consult the YAWL Technical Manual for detailed information on the development of YAWL Custom Services.

10.1 Document Store

The Document Store service maintains uploaded files passed as data in process instances. That is, a task may contain a variable of *YDocumentType* that is used to store a reference to an uploaded data file (e.g. documents, spreadsheets, in fact any type of file) (cf. Section 4.14). That file can then be accessed (downloaded), and updated (uploaded) during the life of the process instance.

To minimise the amount of data actually passed to and from the YAWL Engine, uploaded files are stored directly in the Document Store and only a reference to the file is passed to/from the Engine.

To enable this file storage feature, the Document Store must first be installed (cf. Section 2.4.3) and registered with the Engine as a client application (cf. Section 6.4.4). The Document Store has a configuration setting in its `web.xml` file labelled *RetainStoredDocsOnCaseCompletion* that determines whether an uploaded filed is archived or removed from the Document Store when a case completes or is cancelled (cf. Section 6.1). The setting defaults to a 'false' value.

10.2 Web Service Invoker Service

The Web Service Invoker Service (WSInvoker) provides a mediation layer between the Engine and external SOAP web services. In this way, a task can be associated with an operation of a SOAP web service at design-time, and at runtime task instances are routed to the specified SOAP web service through the WSInvoker. Without this layer, a Custom Service would have to be developed between each and every SOAP web service and the Engine on an individual web service basis.

A task is associated with WSInvoker by selecting it from the dropdown list of the *Custom Service* property in the Decomposition section of the Property Pane in the Editor and choosing the WSInvoker from the list of available services. The service will supply to the task three variable definitions which will require supplied values when the service is invoked at runtime:

- **YawlWSInvokerWSDLLocation**: specifies the URI of the WSDL file describing the web service to invoked;

- **YawlWSInvokerPortName**: specifies the port binding that the web service listens on for interaction with external clients and protocols; and
- **YawlWSInvokerOperationName**: specifies the name of the operation to be executed within the web service.

In addition, any data values that are required for the web service's operation must also be specified within the task-level variables; these are passed to the specified web service when it is invoked. The WSInvoker Service then waits until the external service responds, then returns a mapping of the resultant data to the task instance's output data variables. Note that when a task is registered with the Invoker Service at design time, the Editor automatically populates the task's input variables with the required data attributes above.

At present, the WS-Invoker Service supports only SOAP over HTTP and request-response and one-way interactions (out-in and out-only message exchange patterns).

10.3 SMS Service

The SMS service can use any third-party SMS Gateway web service to send and receive SMS messages and pass the data into and out of the YAWL Engine. In this way, participants can view, update and complete task instances via mobile phones and other SMS capable devices. The service is pre-configured in its web.xml file with four values – a userid and password for an SMS account known to the specified SMS Gateway web service, and its send and receive URI's. These values will be particular to each SMS Gateway service provider.

Like the WSInvoker Service, when a task is associated with SMS Service via its *Custom Service* property in the Editor and choosing the SMS Service from the list of available services, the service will supply three variables required for the successful operation of the service:

- **SMSMessage**: The message text to send to the mobile device;
- **SMSPhoneNumber**: The phone number of the mobile device to call; and
- **SMSReplyMessage**: The message text that is returned from the mobile device.

When invoked at runtime, the SMS Service will logon to the SMS Gateway provider using the userid and password supplied via the web.xml file, then if successful will pass the SMSMessage and SMSPhoneNumber values to to the service's 'Send' URI. The SMS Gateway provider will send the text message to the mobile device identified by the phone number. It will then wait for a reply message from the device, which it will pass back through the SMSReplyMessage variable which can then be mapped back to a corresponding net-level variable in the process for display in a subsequent task.

10.4 Mail Service

The Mail Service was introduced in YAWL 2.2 as a replacement for or alternative to the Mail Sender Service. The new Mail Service allows a task to be designated as an *email* task, which can then be easily configured to send an email to a recipient, using SMTP.

The Mail Service has several parameters that are required to be configured to successfully send an email. Most of the parameters can have default values set for them in the service's *web.xml* file, so that at runtime a user only has to supply values for those remaining parameters that haven't had default values set. The full list of parameters are:

- **host**: The URL of the SMTP mail host. In the service's web.xml, a sample host value of `smtp.gmail.com` has been supplied, but should of course be changed to whatever host you use to send mail.

- **port**: The port number that the host listens on for SMTP traffic. Again, a default value is supplied (for the gmail host) but should be changed to the correct port for your host.
- **mailUserName**: The username of a registered account on the host that is capable of sending mail.
- **mailUserPassword**: The corresponding password for the supplied username.
- **senderName**: the actual name to associate with the email sender.
- **sendAddress**: the email address of the sender, that is the address of the email account corresponding to the *mailUserName* account on the given host.

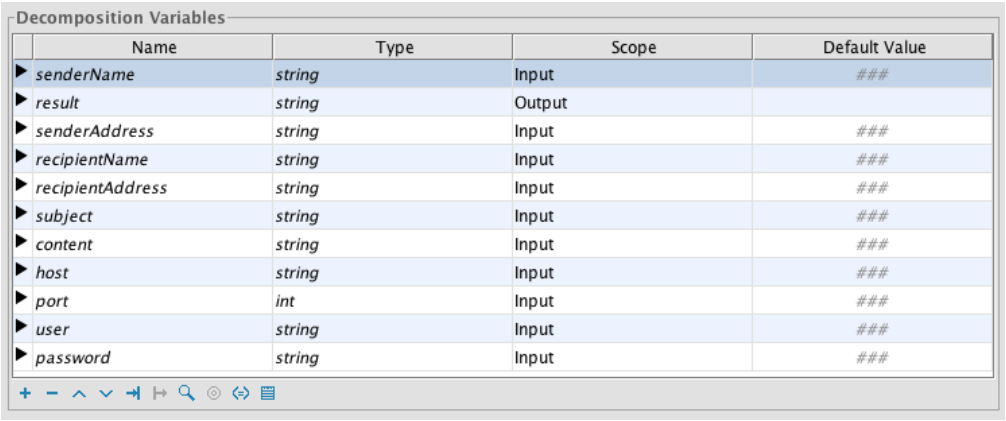


Figure 10.1: Task-level variables for a Mail Service associated task

Note that any, all or none of the parameters listed above may be given default values in the service's web.xml. Each one without a default value will be requested from the user at runtime. Note also that you may include parameters that already have default values in the task decomposition so that at runtime values are requested from the user – when the user supplies a value for which a default value already exists, the runtime user supplied value will take precedence.

Figure 10.1 shows the Editor's Data Variable dialog for a task that has been associated with the Mail Service. Notice that the list of task variables has been populated by the service. For each of the default parameters supplied via web.xml, you may simply remove the matching task variable from the list for all those parameters you don't want the user to override at runtime (e.g. Figure 10.2 has removed the variables for all the default values). The output-only variable *result* is used by the service to return the outcome of the send (i.e. either a success or error message).

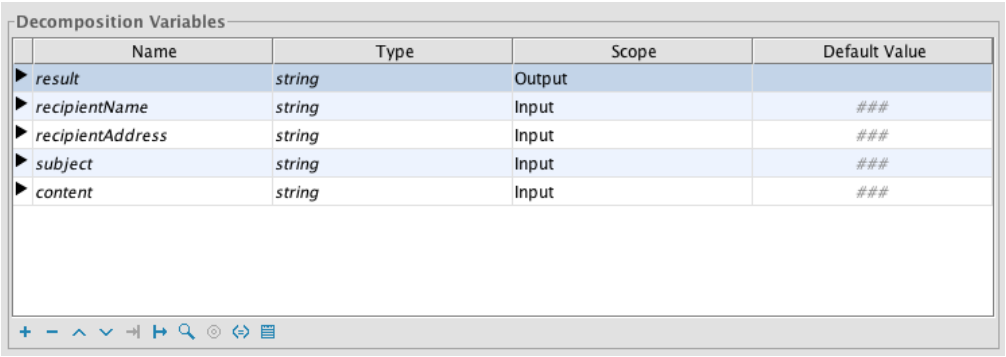


Figure 10.2: Task-level variables for a Mail Service task, default variables removed

Generally, a *Send Email* task associated with the Mail Service will be preceded by another task the will collect the required non-default values from a user at runtime. An example of a dynamic form for such a task is

shown in Figure 10.3 – of course, a custom form may also be used if desired. The *content* field in figure 10.3 uses a *textarea* extended attribute to provide ample space to input the required email content.

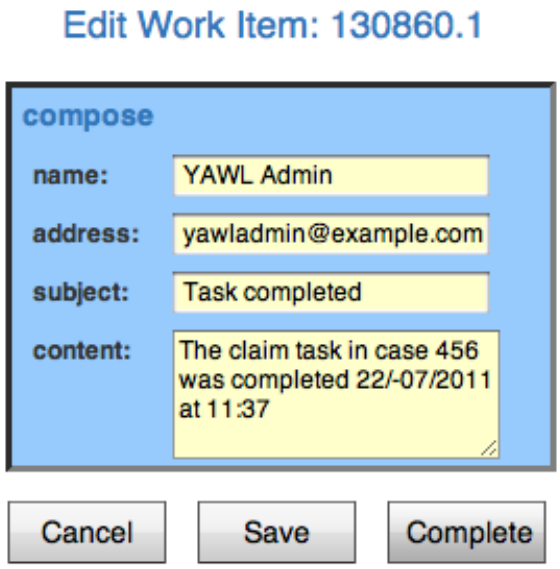


Figure 10.3: A dynamic form for a typical 'compose email ' task

To send HTML formatted content, the text entered into the *content* field should be wrapped in CDATA tags. For example, to send the content "Hello World!" in bold text:

```
<![CDATA[ <b>Hello World!</b> ]]>
```

10.5 Twitter Service

The Twitter Service is a simple service that provides for the posting of status updates (i.e. 'tweets') to Twitter.

When a task is associated with Twitter Service in the Editor's Task Decomposition dialog, the service will supply two required variables:

- **status**: (Input-Only) The message text to send to Twitter;
- **result**: (Output Only) A response message received from Twitter that indicates the success or failure of the status update.

When invoked at runtime, the Twitter Service will connect to Twitter and, through its API using the configured userid and password, post the status update (if connection was successful) and put Twitter's response text in the result variable.

Note: the Twitter API requires a set of four authorised OAUTH tokens to log on, which are stored in the service's *twitter4j.properties* file (twitter4j[1] is the name of the third party library the service uses to communicate with Twitter). Two OAUTH tokens, *oauth.consumerKey* and *oauth.consumerSecret* identify the YAWL TwitterService and should not be changed. The other two tokens, *oauth.accessToken* and *oauth.accessTokenSecret* define the username and password of the Twitter account being used to logon to Twitter – those in the properties file are for the default twitter user *YAWLProc*. To use the service with a different Twitter user account, replace the *oauth.accessToken* and *oauth.accessTokenSecret* values with those generated from the other account. A simple guide to generating tokens for another twitter account can be found here: http://goo.gl/cyHaR

[1] twitter4j.org

10.6 Digital Signature Service

The purpose of the digital signature is not to hide the data on the form (captured as an XML ComplexType) but to ensure the authenticity of the information. This custom service is composed of two functions, the first one is to sign the XML form and the second one is to check the validity of the signature created by the first one.

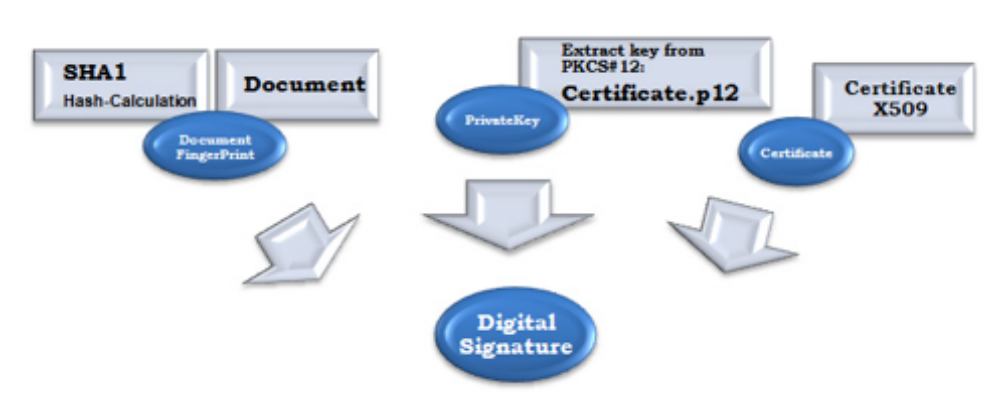


Figure 10.4: An overview of the production of a digital signature

10.6.1 Signing a Document

1. The Document is hashed with a hashing algorithm to encrypt it and to reduce its volume.
2. The Private Key is extracted from inside the key store certificate 'p12'. To do this, we need the publisher's password which is only known by him/her.
3. Combine the private key, the Document fingerprint and the X.509 certificate to create the digital signature using 'PKCS#7' encryption.
4. The fingerprint and the private key are used to calculate the signature itself and the X.509 certificate gives the publisher details.

A digest-SHA1 of the document is included in the signature. Since the Document has a unique fingerprint, the digital signature only applies to this document.

The p12 certificate is protected by a password and is only used to sign the document. The public certificate X.509 contains the information about the signer and the public key that can be used to check the validity of the signature. It is public because anybody should be able to access the content of the signature to verify it. But without the private key it cant be reproduced.

10.6.2 Verification of the Digital Signature

1. Calculate the document fingerprint with the same algorithm than the signer used.
2. To verify the Digital Signature you need first to extract the certificate X.509. The certificate X.509 contains all the information needed to identify a user. It is not encrypted and can be seen freely. We use the public key contained in this certificate to decrypt the signature using the same algorithm.
3. If the signature matches the digest message of the document then the signature is considered valid.

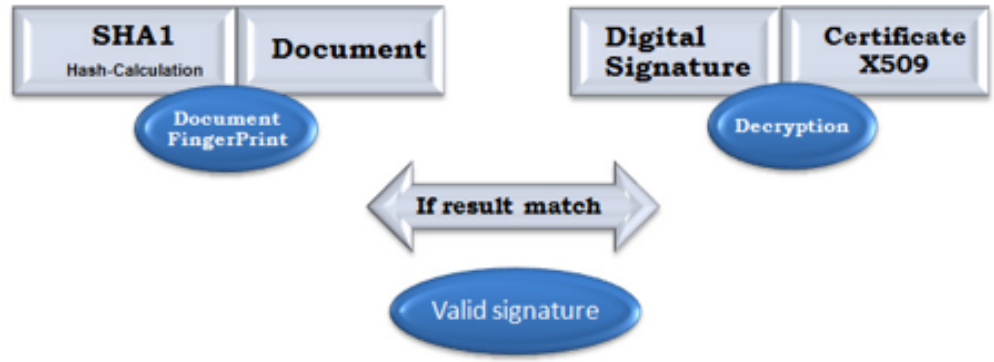


Figure 10.5: An overview of the verification of a digital signature

Anybody can verify the signature since the public key is contained in the X.509 certificate. It is hard to reproduce this signature as the private key is only contained in the certificate PKCS#12 and it needs the owners password to be extracted. Also if the signature is slightly modified the digest function won't match the signature anymore.

10.6.3 Interaction between a YAWL custom form and the service

Before you can use the digital signature function, the user will need to create a digital certificate. These certificates are provided by a CA (certificate authority), for example Thawte Consulting for X.509 certificates. It is advisable to use a certificate provided by a trusted third party like CA but you can also create your own certificates using some open source tools.

10.6.4 Creating a New Certificate

New certificates can be easily created using *Key tool IUI*, which can be downloaded from: `www.softpedia.com/get/Security/Security-Related/KeyTool-IUI.shtml`.

The first step is to create an empty Key Store PKCS12 locked with your password. Save the empty key store in a chosen location (Figure 10.6).

Then use this key Store to generate the key pair and the X.509 certificate by filling in the owner information (Figure 10.7).

When the key pair is created you can review the certificate information produced (Figure 10.8).

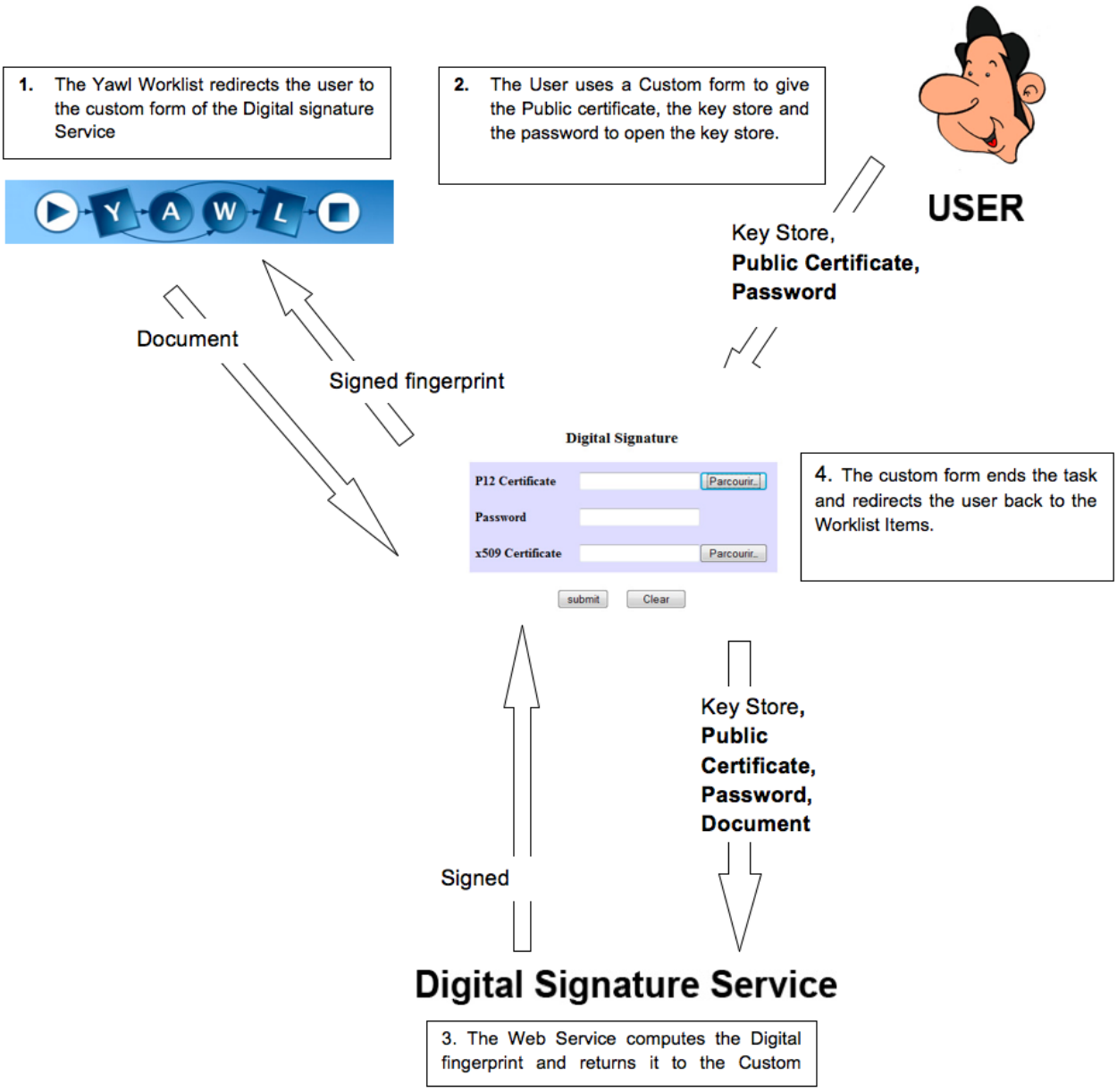
When the key pair is ready you can export the certificate via Export → Private Key's first signing certificate file → as simple certificate file (Figure 10.9).

10.6.5 Using the Digital Signature Service

First please make sure that you have copied the DigitalSignature.war in your tomcat/webapps folder. In the simplest case of using the Digital signature service you will need at least three tasks (Figure 10.10).

The first task 'Fulfil Document' is a user task that can contain any complex type of document (i.e. a variable) you want to be signed. You can define your complex type in the 'Data Type Definition' dialog of the Editor, as the example in Figure 10.11.

The second task 'Sign Document' redirects the user to a custom form which is located inside the 'DigitalSignature.war' deployment. We use a custom form to directly deal with the user instead of a standard custom service because private data like the key store password needs to be hidden from YAWL as the data is passed from one task to another as a net variable, which is accessible by any other task. You can define the Custom



Form URI for the task by selecting the task's Custom Form property in the Properties Pane of the Editor, and entering the URI of the form in the dialog shown.

You also have to define the task data that will be used by the custom form. In this example, their names should be 'Document' and 'Signature'. Note that scope of the 'Signature' variable is InputOutput.

You may also have to change the paths in the 'upload.jsp' file: where the default values are 'localhost', they may have to be changed to the actual address where the service runs (if it is not running locally):

```
String Path = "http://localhost:8080/DigitalSignature/files/";
String redirectURL = "http://localhost:8080/resourceService/" +
                     "faces/userWorkQueues.jsp?workitem=" +
                     wir.toXML();
```

The Signature will be checked with the certificate loaded in the jsp page and it will provide the document

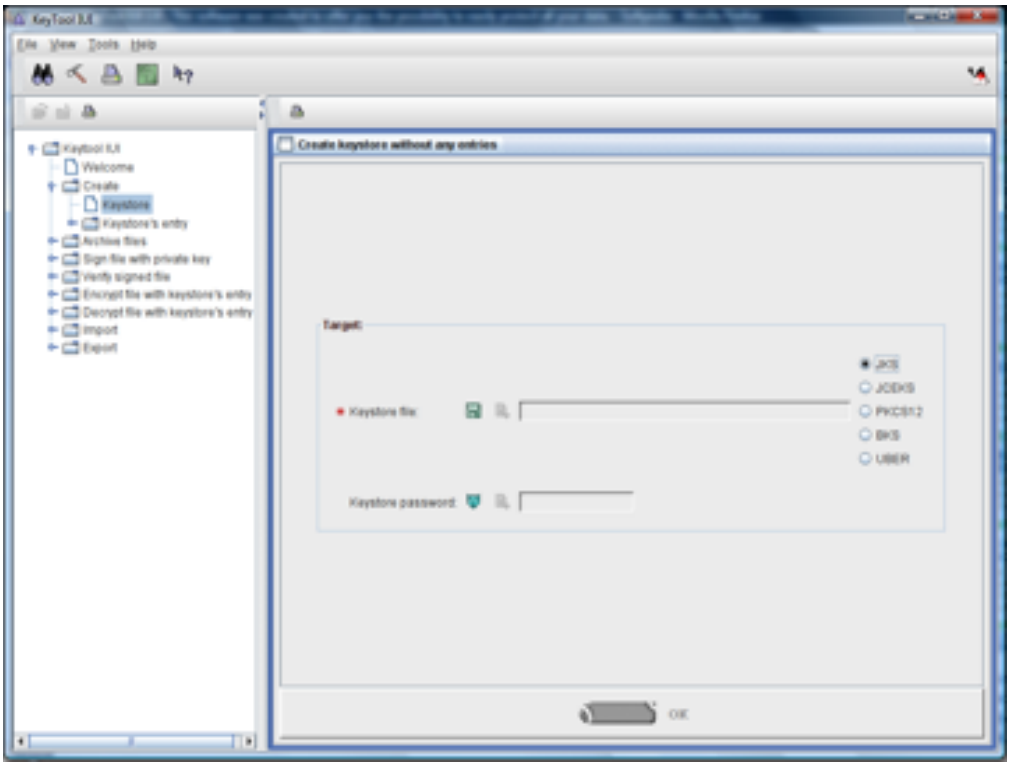


Figure 10.6: Creating an Empty Keystore

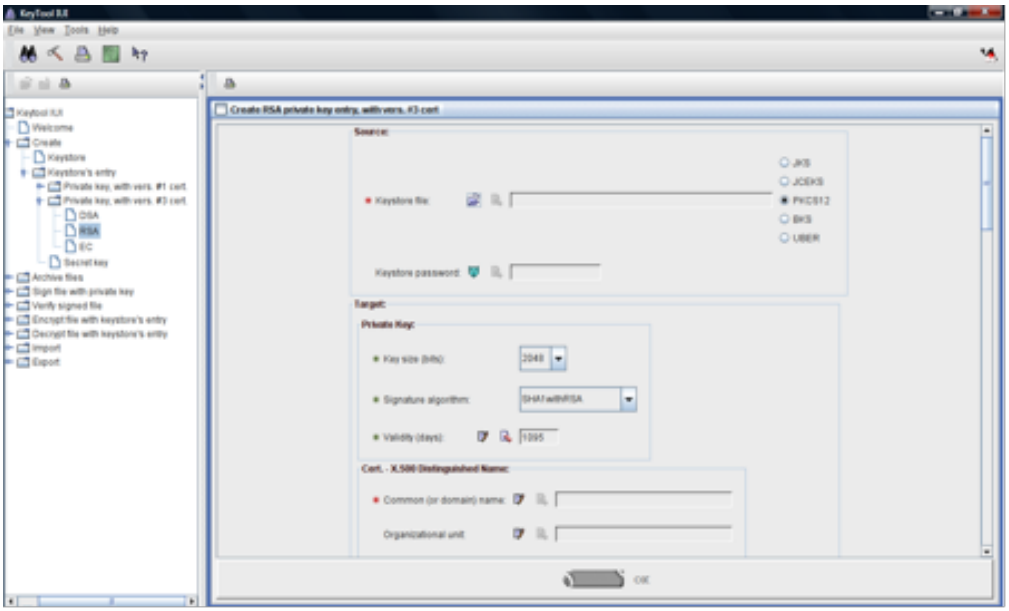


Figure 10.7: Generating the Key Pair

which has been signed. Note that the 'Document' variable is an "anyType" type, to be able to check back any complex type you may have signed.

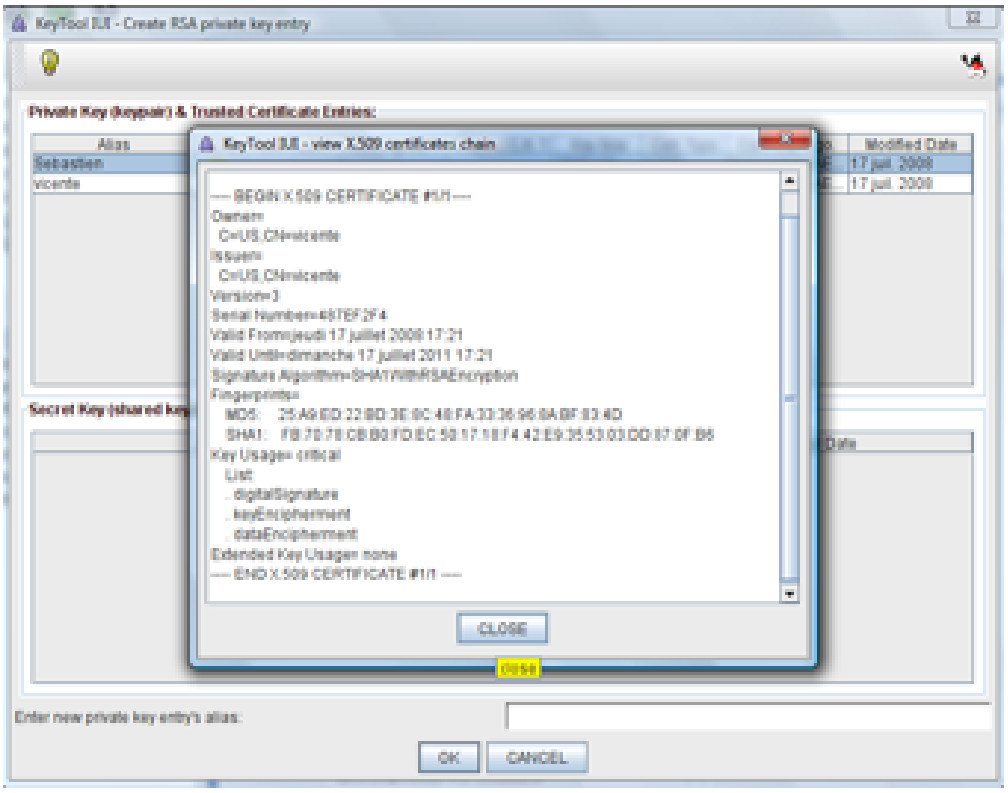


Figure 10.8: The Created Certificate

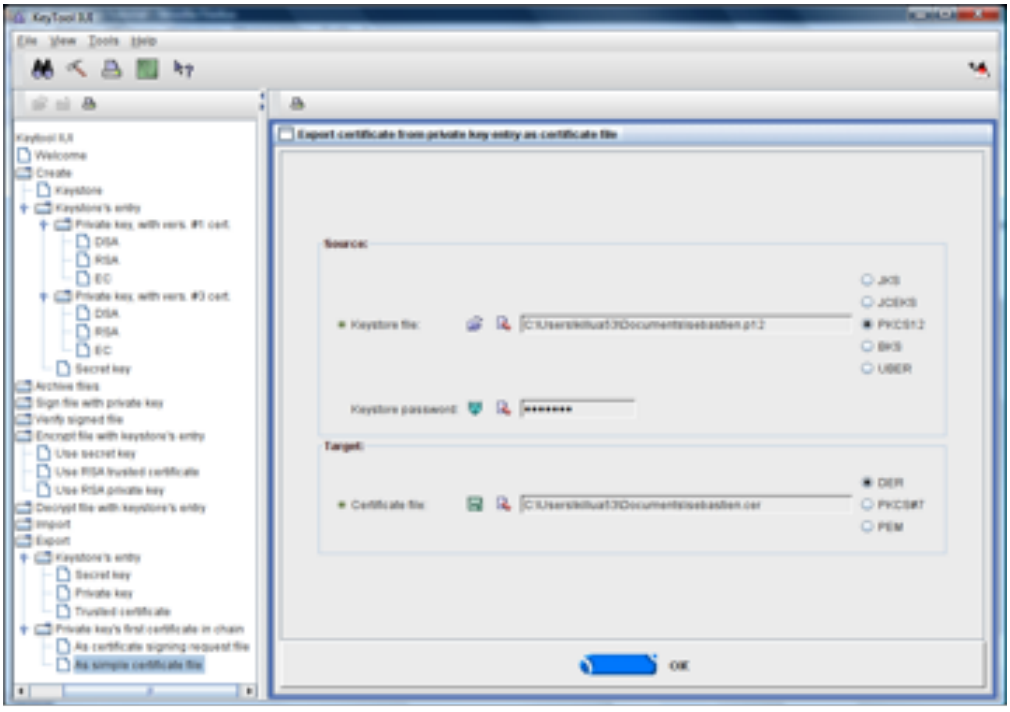


Figure 10.9: Exporting the Certificate

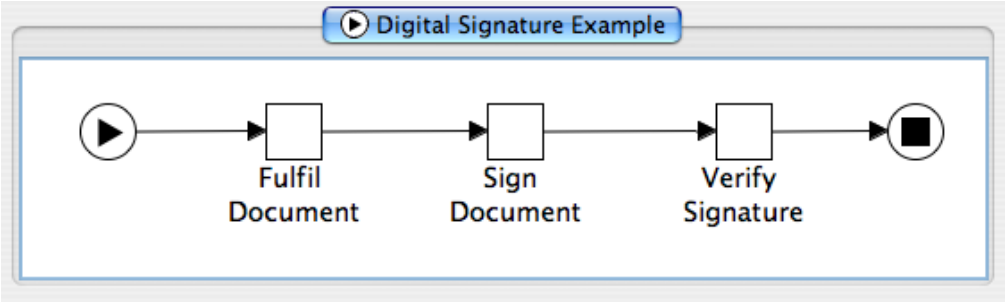


Figure 10.10: Example YAWL Process

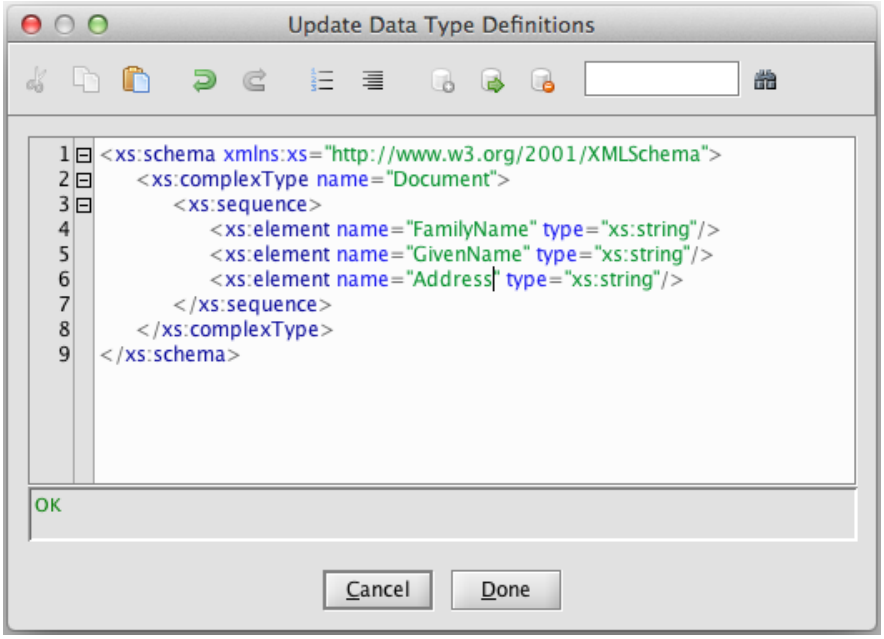


Figure 10.11: Data Definition for 'Document' type

10.7 Email Sender Service

NOTE: This service has been superseded by the Mail Service (cf. Section 10.4).

The Email Sender Service allows users to send simple emails from within the workflow process instance.

10.7.1 How to use the Service

First please make sure that you have the 'mailSender.war' file deployed in your tomcat/webapps folder. The Mail Sender Service uses a YAWL custom form for sending a simple mail notification. You just need one task and set the custom form of that task to call the Mail Sender jsp file. First create a net that will use the Mail Sender Service (Figure 10.12).

Note that you don't need to create a variable to use the Mail Sender but the task that uses it will need at least one variable defined so that the YAWL worklist's view/edit button will not be available and you will not be able to access the Custom form menu.

Second define the Custom form for the task, by right-clicking on the task and selection 'Set Custom Form' from the menu. The address of the custom form is: `http://localhost:8080/mailSender/WebMail.`

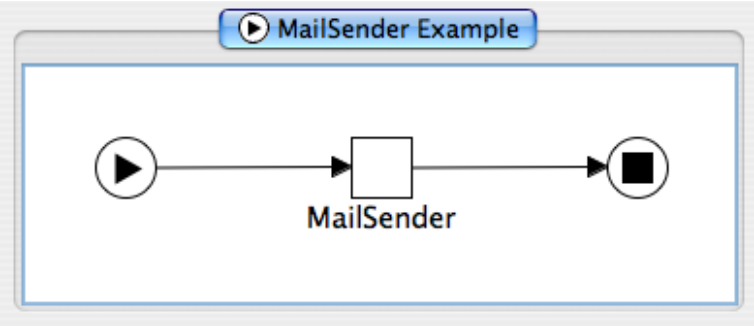


Figure 10.12: Example Mail Sender Process

jsp (remember to replace 'localhost' with the specific hostname when the service is not installed locally). When invoked at runtime, the custom form will look like the example in Figure 10.13.

Figure 10.13: Example Mail Sender Custom Form

You can see that the form in Figure 10.13 contains two parts. The first is to set the SMTP parameters to send email:

1. You need to chose in the dropdown list which SMTP you would like to use to send emails.
2. You also need to enter login and password to connect to the SMTP server.

To add another SMTP server, you have to edit WebMail.jsp and add an option=value pair to the dropdown list of servers:

```
<select name="SMTP" onChange="messageValue()">
   <option value="smtp.qut.edu.au">QUT - WebMail</option>
```

```
    <option value="smtp.gmail.com">Gmail</option>
    <option value="smtp.mail.yahoo.com">Yahoo</option>
    <option value="New.SMTP">Name in the dropdown list</option>
</select>
```

The second part of the form is for entering the details of the email. In the field 'Send To' enter the email address to send the mail to. The 'Alias' will be the name appearing on the receiver's mailbox. Then you can add the subject of the mail and the content. You can also attach a file, if desired.

The service will then attempt to send the email using the smtp server details provided in the form.

Chapter 11

Seeking Help

Manuals, like the software they describe, can never be considered to be complete. It is quite possible that you run into an issue for which the documentation is lacking, or find a problem with the YAWL environment that constitutes a bug. Alternately, you may come up with a new idea on how to enhance the system. Here we briefly address the questions of how and where to seek help.

For queries about the YAWL software, we recommend that people use the help and discussion forums at the YAWL Foundation (http://www.yawlfoundation.org/forum) or alternately, the YAWL User Group site (http://yaug.org). In posting a question, request or comment, please help us as much as possible in answering you by explicitly stating which versions of various software components you are using, providing the specification that is causing a problem if applicable, listing messages produced in the console window and/or log files, providing a backup of the database, and so on. Allow up to a few days for someone to respond to your request.

The YAWL Issues Lists can be used for reporting bugs or proposing enhancements – use https://github.com/yawlfoundation/yawl/issues for matters relating to the YAWL engine and services, and https://github.com/yawlfoundation/editor/issues for Editor related matters. As attachments can be easily provided, this is the preferred method for reporting bugs or requesting enhancements. Again, please provide as much relevant information as possible. Also, before reporting an issue, please thoroughly check the list to ensure that the issue has not already been reported.

Any feedback regarding this manual is most welcome and may be sent to yawlmanual@gmail.com.

We encourage forum posts, bug reports and enhancement requests. Providing these centrally means that others can learn from the answers provided and, hopefully, people are inspired to respond to other people's requests. In this way we can manage progress on YAWL more efficiently.

Bibliography

[1] W.M.P. van der Aalst. Verification of Workflow Nets. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426. Springer-Verlag, Berlin, 1997.

[2] W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[3] W.M.P. van der Aalst. Formalization and Verification of Event-driven Process Chains. *Information and Software Technology*, 41(10):639–650, 1999.

[4] W.M.P. van der Aalst. Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 1–65. Springer-Verlag, Berlin, 2004.

[5] W.M.P. van der Aalst, L. Aldred, M. Dumas, and A.H.M. ter Hofstede. Design and implementation of the YAWL system. In A. Persson and J. Stirna, editors, *Proceedings of the 16th International Conference on Advanced Information Systems Engineering (CAiSE 04)*, pages 142–159, Riga, Latvia, 2004. Springer-Verlag, Berlin.

[6] W.M.P. van der Aalst, P. Barthelmess, C.A. Ellis, and J. Wainer. Workflow Modeling using Proclets. In O. Etzion and P. Scheuermann, editors, *7th International Conference on Cooperative Information Systems (CoopIS 2000)*, volume 1901 of *Lecture Notes in Computer Science*, pages 198–209. Springer-Verlag, Berlin, 2000.

[7] W.M.P. van der Aalst, P. Barthelmess, C.A. Ellis, and J. Wainer. Proclets: A Framework for Lightweight Interacting Workflow Processes. *International Journal of Cooperative Information Systems*, 10(4):443–482, 2001.

[8] W.M.P. van der Aalst, B.F. van Dongen, C.W. Günther, R.S. Mans, A.K. Alves de Medeiros, A. Rozinat, V. Rubin, M. Song, H.M.W. Verbeek, and A.J.M.M. Weijters. ProM 4.0: Comprehensive Support for Real Process Analysis. In J. Kleijn and A. Yakovlev, editors, *Application and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2007)*, volume 4546 of *Lecture Notes in Computer Science*, pages 484–494. Springer-Verlag, Berlin, 2007.

[9] W.M.P van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods and Systems*. MIT Press, Cambridge, MA, USA, 2002.

[10] W.M.P. van der Aalst and A.H.M. ter Hofstede. Workflow Patterns: On the Expressive Power of (Petri-net-based) Workflow Languages. In Kurt Jensen, editor, *Proceedings of the Fourth International Workshop on Practical Use of Coloured Petri Nets and the CPN Tools*, volume 560 of DAIMI, pages 1–20, Aarhus, Denmark, August 2002. University of Aarhus.

[11] W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet another workflow language. *Information Systems*, 30(4):245–275, 2005.

[12] M. Adams. *Facilitating Dynamic Flexibility and Exception Handling for Workflows*. PhD Thesis, Queensland University of Technology, Brisbane, Australia, 2007. Available through http://yawlfoundation.org.

[13] M. Adams, A.H.M. ter Hofstede, W.M.P. van der Aalst, and D. Edmond. Dynamic, Extensible and Context-Aware Exception Handling for Workflows. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS, OTM Confederated International Conferences CoopIS, DOA, ODBASE, GADA, and IS 2007, Vilamoura, Portugal, November 25-30, 2007, Proceedings, Part I*, volume 4803 of *Lecture Notes in Computer Science*, pages 95–112. Springer-Verlag, Berlin, 2007.

[14] M. Adams, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Worklets: A service-oriented implementation of dynamic flexibility in workflows. In R. Meersman and Z. Tari et. al., editors, *Proceedings of the 14th International Conference on Cooperative Information Systems (CoopIS'06)*, volume 4275 of *Lecture Notes in Computer Science*, pages 291–308, Montpellier, France, November 2006. Springer-Verlag, Berlin.

[15] M. de Leoni, W.M.P. van der Aalst, and A.H.M. ter Hofstede. Visual Support for Work Assignment in Process-Aware Information Systems. In M. Dumas, M. Reichert, and M.-C. Shan, editors, *BPM 2008*, volume 5240 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2008.

[16] F. Gottschalk, W.M.P. van der Aalst, M.H. Jansen-Vullers, and M. La Rosa. Configurable Workflow Models. *International Journal of Cooperative Information Systems*, 17(2):177–221, 2008.

[17] A.H.M. ter Hofstede, W.M.P van der Aalst, M. Adams, and N. Russell, editors. *Modern Business Process Automation: YAWL and its Support Environment*. Springer, 2010.

[18] N. Lohmann and D. Weinberg. Wendy: A tool to synthesize partners for services. In J. Lilius and W. Penczek, editors, *31st Int. Conference on Applications and Theory of Petri Nets and Other Models of Concurrency*, volume 6128 of *Lecture Notes in Computer Science*, pages 297–307. Springer-Verlag, Berlin, 2010.

[19] R.S. Mans. *Workflow Support for the Healthcare Domain*. PhD thesis, Eindhoven University of Technology, June 2011. See http://www.processmining.org/blogs/pub2011/workflow_support_for_the_healthcare_domain.

[20] T. Murata. Petri nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[21] M. Pesic, H. Schonenberg, and W.M.P. van der Aalst. DECLARE: Full Support for Loosely-Structured Processes. In M. Spies and M.B. Blake, editors, *Proceedings of the Eleventh IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, pages 287–298. IEEE Computer Society, 2007.

[22] J.L. Peterson. *Petri net theory and the modeling of systems*. Prentice-Hall, Englewood Cliffs, USA, 1981.

[23] A. Rozinat, M. Wynn, W.M.P. van der Aalst, A.H.M. ter Hofstede, and C. Fidge. Workflow Simulation for Operational Decision Support Using Design, Historic and State Information. In M. Dumas, M. Reichert, and M.-C. Shan, editors, *BPM 2008*, volume 5240 of *Lecture Notes in Computer Science*, pages 196–211. Springer-Verlag, Berlin, 2008.

[24] N. Russell, W.M.P van der Aalst, and A.H.M. ter Hofstede. Workflow exception patterns. In E. Dubois and K. Pohl, editors, *Proceedings of the 18th International Conference on Advanced Information Systems Engineering (CAiSE'06)*, volume 4001 of *Lecture Notes in Computer Science*, pages 288–302, Luxembourg, Luxembourg, 2006. Springer-Verlag, Berlin.

[25] N. Russell, W.M.P. van der Aalst, A.H.M. ter Hofstede, and D. Edmond. Workflow resource patterns: Identification, representation and tool support. In O. Pastor and J. Falcão e Cunha, editors, *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05)*, volume 3520 of *Lecture Notes in Computer Science*, pages 216–232, Porto, Portugal, 2005. Springer-Verlag, Berlin.

[26] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P. van der Aalst. Workflow data patterns: Identification, representation and tool support. In L. Delcambre, C. Kop, H.C. Mayr, J. Mylopoulos, and O. Pastor, editors, *Proceedings of the 24th International Conference on Conceptual Modeling (ER 2005)*, volume 3716 of *Lecture Notes in Computer Science*, pages 353–368, Klagenfurt, Austria, 2005. Springer-Verlag, Berlin.

[27] N. Russell, A.H.M. ter Hofstede, D. Edmond, and W.M.P van der Aalst. newYAWL: achieving comprehensive patterns support in workflow for the control-flow, data and resource perspectives. Technical Report BPM-07-05, BPM Center, 2007. http://www.BPMcenter.org.

[28] N.C. Russell. *Foundations of Process-Aware Information Systems*. PhD Thesis, Queensland University of Technology, Brisbane, Australia, 2007. Available through http://yawlfoundation.org.

[29] J.R. Searle. *Speech Acts*. Cambridge University Press, Cambridge, 1969.

[30] H.M.W. Verbeek, Wil M.P. van der Aalst, and Arthur H.M. ter Hofstede. Verifying Workflows with Cancellation Regions and OR-joins: An Approach Based on Relaxed Soundness and Invariants. *Computer Journal*, 50(3):294–314, 2007.

[31] M. Weske. *Business Process Management: Concepts, Languages, Architectures* . Springer-Verlag, Berlin, 2007.

[32] T. Winograd and F. Flores. *Understanding Computers and Cognition: A New Foundation for Design*. Ablex, Norwood, 1986.

[33] M.T. Wynn. *Semantics, Verification, and Implementation of Workflows with Cancellation Regions and OR-joins*. PhD Thesis, Queensland University of Technology, Brisbane, Australia, 2006. Available through http://yawlfoundation.org.

[34] M.T. Wynn, D. Edmond, W.M.P. van der Aalst, and A.H.M. ter Hofstede. Achieving a general, formal and decidable approach to the OR-join in workflow using Reset nets. In G. Ciardo and P. Darondeau, editors, *Proceedings of the 26th International Conference on Application and Theory of Petri nets and Other Models of Concurrency (Petri Nets 2005)*, volume 3536 of *Lecture Notes in Computer Science*, pages 423–443, Miami, USA, 2005. Springer-Verlag, Berlin.