YAWL - User Manual

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 addtional 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.

Feedback?

Any feedback regarding this manual is very much appreciated. If you find there is a topic that is missing or has not been sufficiently well-explained, please send your feedback to `yawlmanual@gmail.com`. All suggested improvements will be incorporated into future versions of the manual.

Sources

The first version of this document 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 forms 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 copied into Chapter 4.

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 was copied into Chapter 7.

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 forms the basis for Chapter 2. Its introduction forms 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.

- Through the BPMN2YAWL component, BPMN models can be mapped to the YAWL environment for execution.

- The Declare component (released through `declare.sf.net`) provides unique support for specifying workflows in terms of constraints. This approach can be combined with the Worklet approach thus providing very powerful flexibility support.

- 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 Engine are released to the public, they will be available for download at the YAWL Sourceforge website (`sourceforge.net/projects/yawl`). 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 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. [5] 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 [6]. 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 [20]. For a formalisation of the OR-join, a complex synchronisation concept in YAWL, we refer to [24]. This definition supersedes the definition provided in [6].

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 [14, 12].

Wil van der Aalst has written much about the application of Petri nets to workflow, see e.g. [1]. 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 [4].

A recent textbook on Business Process Management (BPM), which covers the original control-flow patterns and also YAWL, was written by Mathias Weske [22]. 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 [20].

If you would like to know more about how verification of YAWL specifications really works, we refer you to [21] and to [23]. 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 [20, 16] and from an implementation aspect in [7, 8]. The YAWL's worklet approach to dealing with on-the-fly changes to workflows has been discussed in [7, 9].

The reader that is interested in declarative specification of workflow is referred to [13]. On the Declare web site, `declare.sf.net`, the Declare service for YAWL can be downloaded. Further documentation about this approach can also be found there.

YAWL has a close link to the Process Mining environment ProM [3], `www.processmining.org`. This link is for example exploited in [15] 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 [10]. 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* is to be published by Springer in October 2009.

Chapter 2

Installation

The YAWL System comprises a number of web services and a java-based Editor. It requires a *Servlet Container* to host the web services and a back-end database system for process data storage and archiving. Individual YAWL components may be installed manually (see Section 2.5), 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:

- **YAWL4Study** is available for Windows, Linux and Mac OSX platforms. This is the installer to choose if YAWL is to be used within a single user environment. So, your personal purposes could be to write a workflow research paper based on YAWL or to prepare your next workflow lecture. Furthermore YAWL is the right choice if you intend to learn about or to experiment with YAWL. The resulting pre-configured installation will have full functionality and contains the same YAWL components as *YAWL4Enterprise*.

- **YAWL4Enterprise** is an installer for Windows only. Third-party components like Apache Tomcat and PostgreSQL are installed as services so that YAWL can be used as a long-running server for production purposes. Furthermore, the installer allows the configuration of the YAWL environment. For production purposes on a Windows platform this is the installer to choose. See Section 2.5 for instructions on how to install for multi-user production environments on other platforms.

- **YAWLive** is a LiveCD which contains a Linux operating system and a fully pre-configured YAWL environment. No installation is required, all that needs to be done is to boot the .iso image from a virtual machine or a previously burned YAWLive CDROM. This is a great way to use YAWL in an educational setting.

The complete YAWL environment is installed whatever installer is chosen. Further information on the different installer types can be found on the YAWL project web-page on SourceForge[1].

	Installing	Maintaining	YAWL Engine	YAWL Editor	Purposes
YAWL4Study	Easy	Easy	✔	✔	Fits for all purposes; except long running workflows
YAWL4Enterprise	Tricky	Tricky	✔	✔	Long running workflows
YAWLive	Medium	Easy	✔	✔	Sandbox; minimal persistence possible

Figure 2.1: YAWL product comparison

Official and stable versions of the YAWL installation files are found on the YAWL project web-page on sourceforge[2] under *Download*, release package *YAWL Complete*.

---

[1]http://downloads.sourceforge.net/yawl/right_yawl_version_for_YOU.pdf
[2]http://sourceforge.net/projects/yawl/

2.1   Requirements

Pre-requisites depend on the type of installation used.

YAWLive, distributed as an .iso image, is a complete OS and pre-installed YAWL package that will run on any computer that can load it in a virtual machine (e.g. VirtualBox or VMware). While the image can be burned to a CD and booted from there, some hardware configurations may experience a freeze during boot phase.

All other installation types require the Java SE Runtime Environment (JRE), 1.5 or greater (`java.sun.com`).

YAWL4Study for Linux requires installed xdg-utils (`portland.freedesktop.org`), which should be supported by your desktop environment (GNOME and KDE do support xdg-utils), in order to visualise YAWL Editor menu entries.

YAWL 2.0 has been successfully tested on:

- Windows: XP, Vista, Windows 7 beta, Server;

- Linux: Ubuntu (8.10), sidux, Debian (Etch);

- Mac OSX: Tiger, Leopard.

2.2   YAWL4Study

The release package provides installation files for the operating systems Windows, Linux and Mac OSX. They have names similar to those shown in Table 2.1.

	Windows	YAWL4Study_**windows**_2.0.exe
	Linux	YAWL4Study_**linux**_2.0.bin
	Mac OSX	YAWL4Study_**osx**_2.0.app.zip

Table 2.1: YAWL4Study is available for three different operating systems

Preparation   YAWL4Study can be installed from an ordinary user account without administration rights as long as Java version 1.5 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 YAWL4Study_linux_2.0_RC1.bin
```

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

```
user@host:/tmp$ ./YAWL4Study_linux_2.0_RC1.bin
```

Installation   You will be greeted by a Welcome screen. Pressing *Next* will take you to the License Agreement page, shown in figure 2.2. The license agreement contains the Gnu Lesser General Public License (Version 2.1), the Apache License (Version 2.0), the BSD License and Sun's Binary Code License Agreement. You need to accept the agreements to continue the installation.

**Figure 2.2:** License Agreement

The next step is to select the installation directory (figure 2.3). 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.
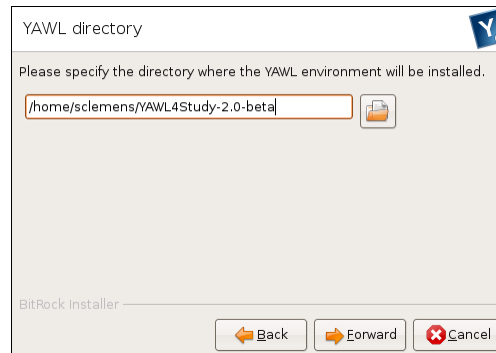
Figure 2.3: Select Installation Directory

After clicking *Next*, the installer will look for a valid Java installation. The dialogue box in figure 2.4 will only appear if Java has been detected on your system.
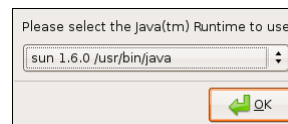
Figure 2.4: Select Java Version

⊞ If no Java version greater or equal 1.5 is found and the YAWL4Study for Windows installer is executed, the supplied Java version will be installed.

△ The installation will stop if no Java version greater or equal 1.5 is found. Install the latest Java version before executing 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 sourceforge forum[3].

2.3   YAWL4Enterprise

The release package on the YAWL project web-page on sourceforge[4] provides an installation file for Windows. In contrast to YAWL4Study, YAWL4Enterprise allows the configuration of third-party components like the application servlet container and the database server.

Preparation   YAWL4Enterprise should be installed with an administrator account. Installation is started by simply double-clicking on the installation file.

Installation   After being greeted by the Welcome screen and pressing *Next*, you'll see the License Agreement page. The license agreement contains the Gnu Lesser General Public License (Version 2.1), the Apache License (Version 2.0), the BSD License and Sun's Binary Code License Agreement. You need to accept the agreements to continue the installation. In the next step you are given the choice to update YAWL (but only if the same YAWL version is already installed), to do a standard installation, or to install YAWL and conduct a comprehensive configuration of its third party components. If it is chosen to update YAWL, existing configurations are used to update the current YAWL installation.
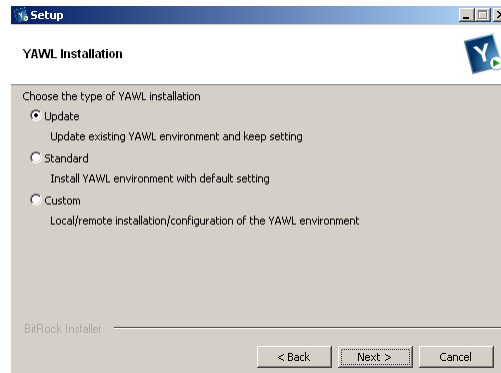

Figure 2.5: Type of YAWL4Enteprise Installation

The next step is to select the installation directory and to choose a Java installation. The dialogue box in figure 2.6 will only appear if Java greater or equal 1.5 is detected on your system.
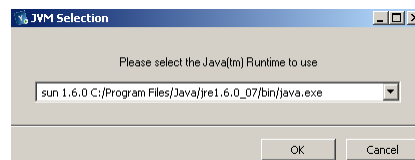
Figure 2.6: Select Java Version

Within the next steps the installation path and port numbers (see figure 2.7) of Apache Tomcat are set. The default port numbers are fine in most cases, and only need to be changed if those ports are already in use.

The YAWL4Enterprise installer allows to install a supplied version of PostgreSQL or to use an already installed PostgreSQL or MySQL version (see figure 2.8).

---

[3]http://sourceforge.net/forum/forum.php?forum_id=391803
[4]http://sourceforge.net/projects/yawl/
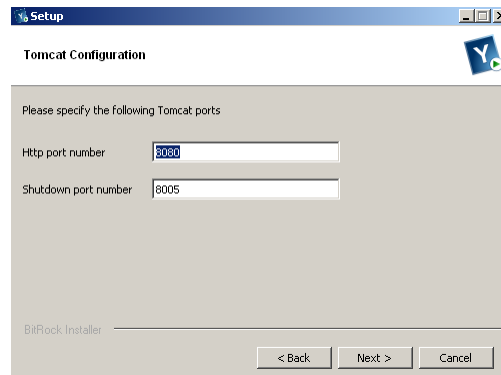
Figure 2.7: Set up of Tomcat Ports
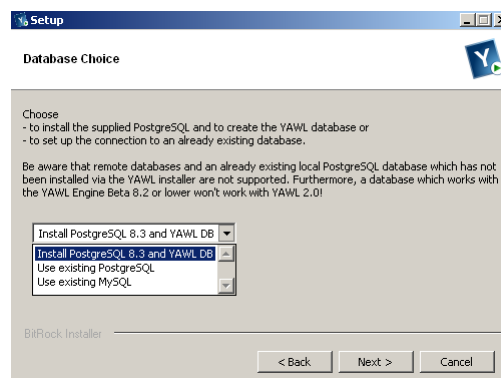
Figure 2.8: Specification of Database

If you choose to install PostgreSQL and the YAWL DB, the PostgreSQL installation directory and different PostgreSQL settings can be specified (see figure 2.9). Again the default settings are suitable in most cases.
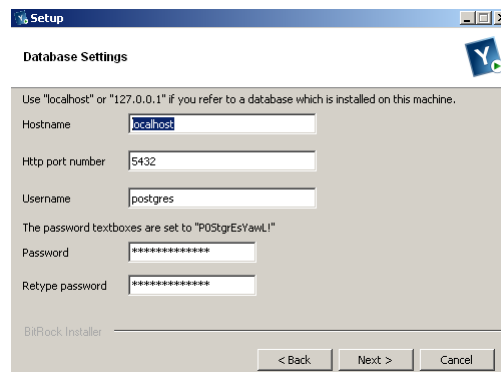
Figure 2.9: Specification of PostgreSQL settings

The last installation panel allows specifying whether Apache Tomcat and/or PostgreSQL should run as services.

2.4   YAWLive

The release package on the YAWL project web-page on sourceforge[5] provides an iso image with a complete operating system and preconfigured YAWL environment. The advantage of using YAWLive is that it can be used within a virtual machine or booted from a CDROM. No installation is required.

Preparation   Booting YAWLive from a CDROM does not work on all computers. Therefore it is recommended to start the YAWLive iso image out of a virtual machine (like VirtualBox). No installation of an additional operating system into the virtual machine image is necessary. The virtual machine image needs to boot the YAWLive iso image only. Please read the instructions of your preferred virtual machine to boot an iso image.
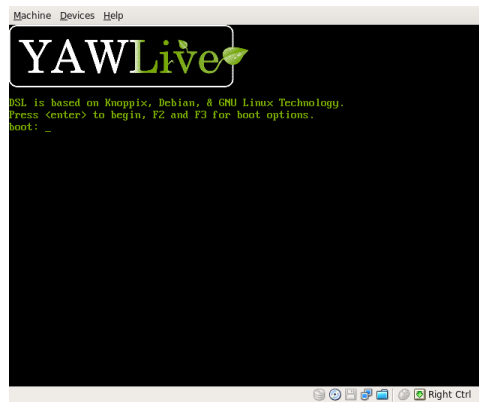

Figure 2.10: YAWLive Welcome Boot Prompt

Usage of YAWLive   After showing a YAWLive Welcome screen and entering Return at the prompt (see figure 2.10), the YAWLive iso image is booted as shown in figure 2.11.

As soon as the desktop environment appears, you can immediately start using YAWL (see figure 2.12)

2.5   Manual Installation

If you already have Apache Tomcat (version 5.5.26 or greater) and/or PostgreSQL (version 8.1 or greater) and/or a previous version of YAWL, you may prefer to install specific components only. Alternately, you may wish to use a different database back-end than PostgreSQL, or a different servlet container than Apache Tomcat. Manual installation is also required if you want a multi-user, production-level installation on an OS platform other than Windows. This section details how to install YAWL 2.0 on a component-by-component basis.

2.5.1   Installing Tomcat

YAWL mainly consists of a number of web services, and so needs a servlet container installed to host them. We recommend Apache Tomcat be used – it if free, stable and fully tested as a YAWL host over a number of years.

YAWL requires Tomcat version 5.5.26 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.
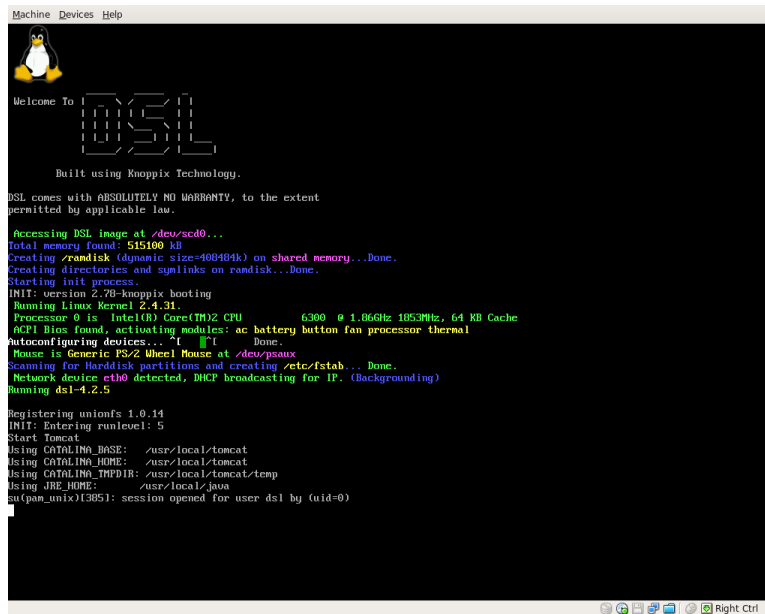
---
[5]http://sourceforge.net/projects/yawl/

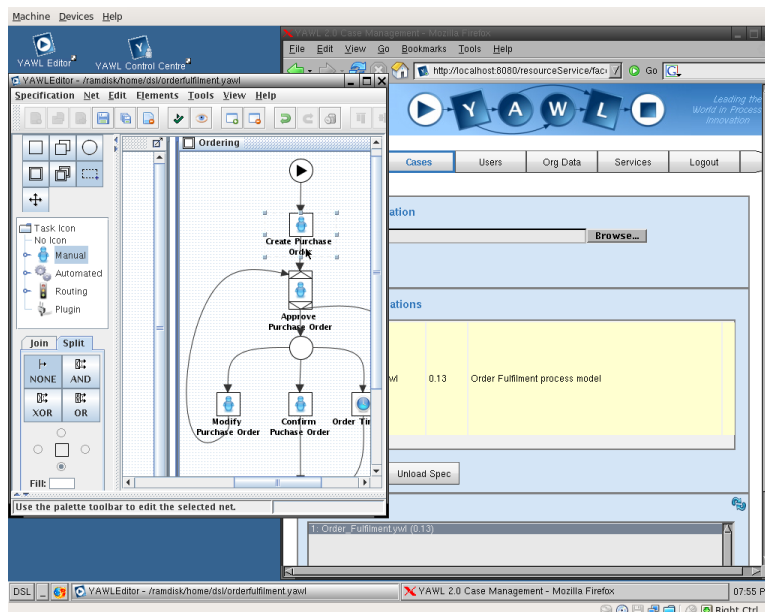**Figure 2.11:** YAWLive Bootphase

Figure 2.12: YAWLive Desktop with Editor and Worklist running

For installation on Mac OSX, follow the instructions at wiki.apache.org/tomcat/TomcatOnMacOS. Ignore the first half of the page; start from the heading 'Updated for 2006'. The steps for starting the firewall and enabling port-forwarding are not required from a YAWL perspective.

Linux installation is quite 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 fourth parameter, *URIEncoding="UTF-8"*, has been added):

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

2.5.2   Installing YAWL Services

Go to the YAWL Engine download page on SourceForge (sourceforge.net/project/showfiles.php?group_id=114611&package_id=124238), and download the file *YAWL_2.0_CoreWebServices.zip*, (and optionally the file *YAWL_2.0_OptionalWebServices.zip*). 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 2.13). There are four 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.

The Resource Service is described in detail in Chapter 6. The Worklet Service is described in detail in Chapter 7. The WS Invoker Service is described in Chapter 8.

There are three optional web services:

1. **yawlSMSInvoker.war**: allows tasks to be read and modified via SMS Services.

2. **digitalSignature.war**: authenticates the information provided on a form using a digital signature via X.509 certificates and private keys.

3. **mailSender.war**: provides a custom form for a task, from which an email can be sent.

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 may run out of resources while doing so, resulting in an *OutOfMemoryError*.
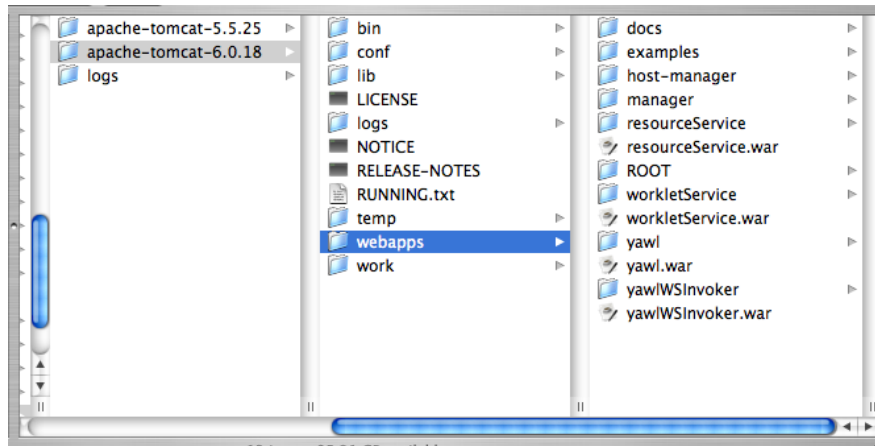
Figure 2.13: YAWL Core Services deployed in /webapps directory (OSX Example)

2.5.3 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 the next section for details).

Download the latest version of PostgreSQL from www.postgresql.org/download/ – there is a one-click installer 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 2.14). 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.
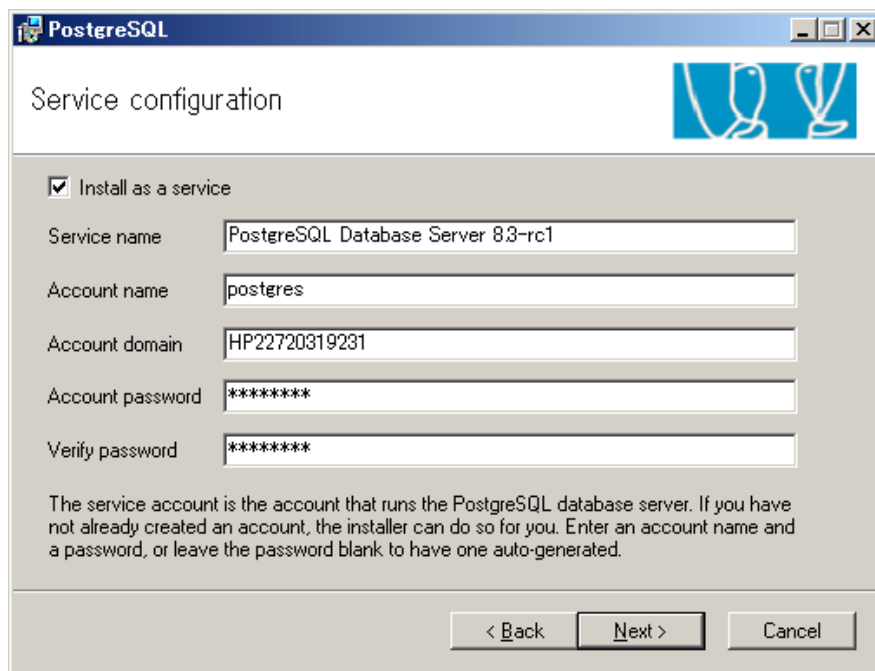
Figure 2.14: Choose *Install as a Service* and enter any password

On the next screen, *Initialise Database Cluster* (Figure 2.15), 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. Leave the settings for all other screens at their defaults and click through to completion.
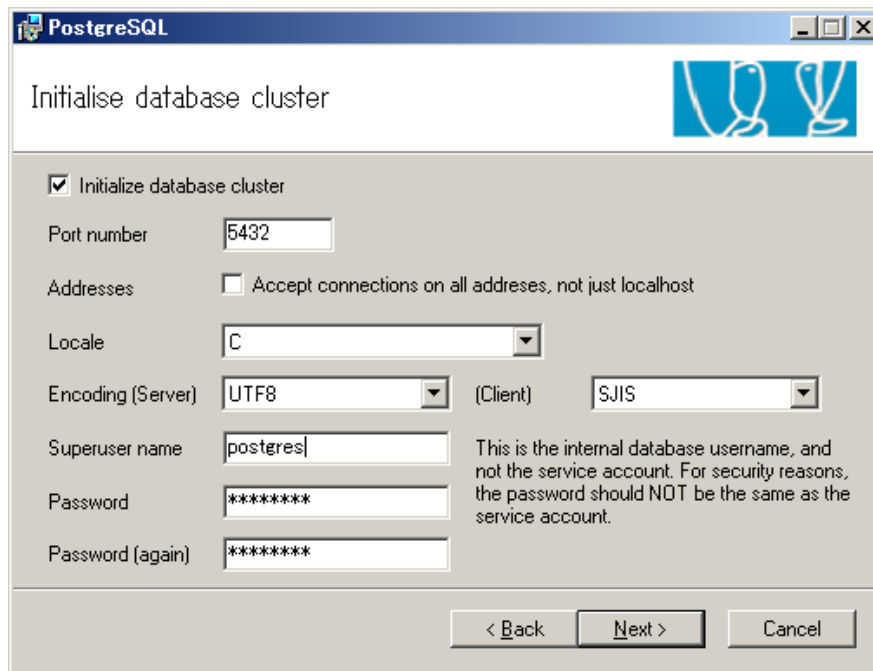
Figure 2.15: Keep *postgres* as the Superuser name, and enter *yawl* as the password
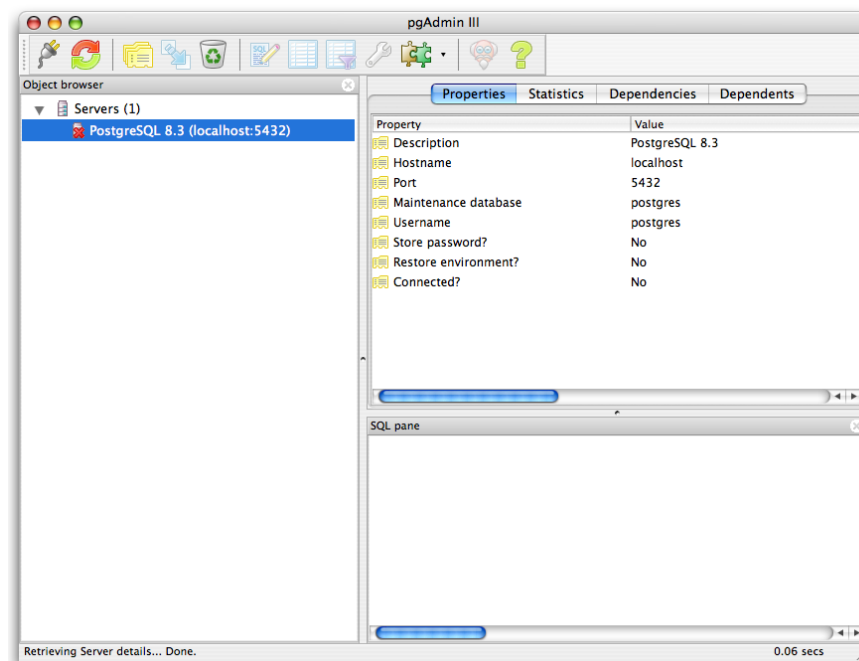
Figure 2.16: The pgAdmin tool, Object Browser panel on the left

Next, run the administration tool *pgAdmin*[6], which was installed along with the postgreSQL installation.

_____
[6]www.pgadmin.org

In the *Object Browser* panel on the left (cf. Figure 2.16), double-click on the *Postgres* server, and, when prompted, 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.

2.5.4   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 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 and Apache Derby (used by YAWL4Study), but other database platforms are known to work with Hibernate (see `www.hibernate.org/80.html` for the complete list) and so should have no trouble working with YAWL too[7].

Each YAWL service that communicates with the database (i.e. the Engine, Resource Service and Worklet Service) 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.

---

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

```
## 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)

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 "business process

---

[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.

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 in the YAWL Sourceforge 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]See https://sourceforge.net/docman/?group_id=114611

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
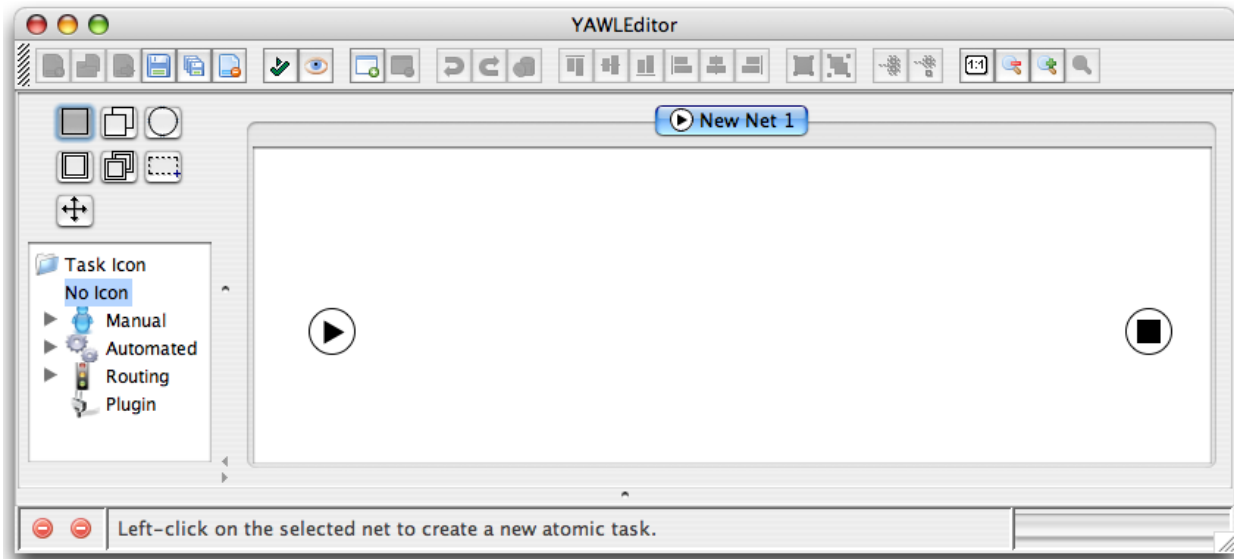

Figure 3.1: A New Specification and its Starting Net

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. 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 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*.
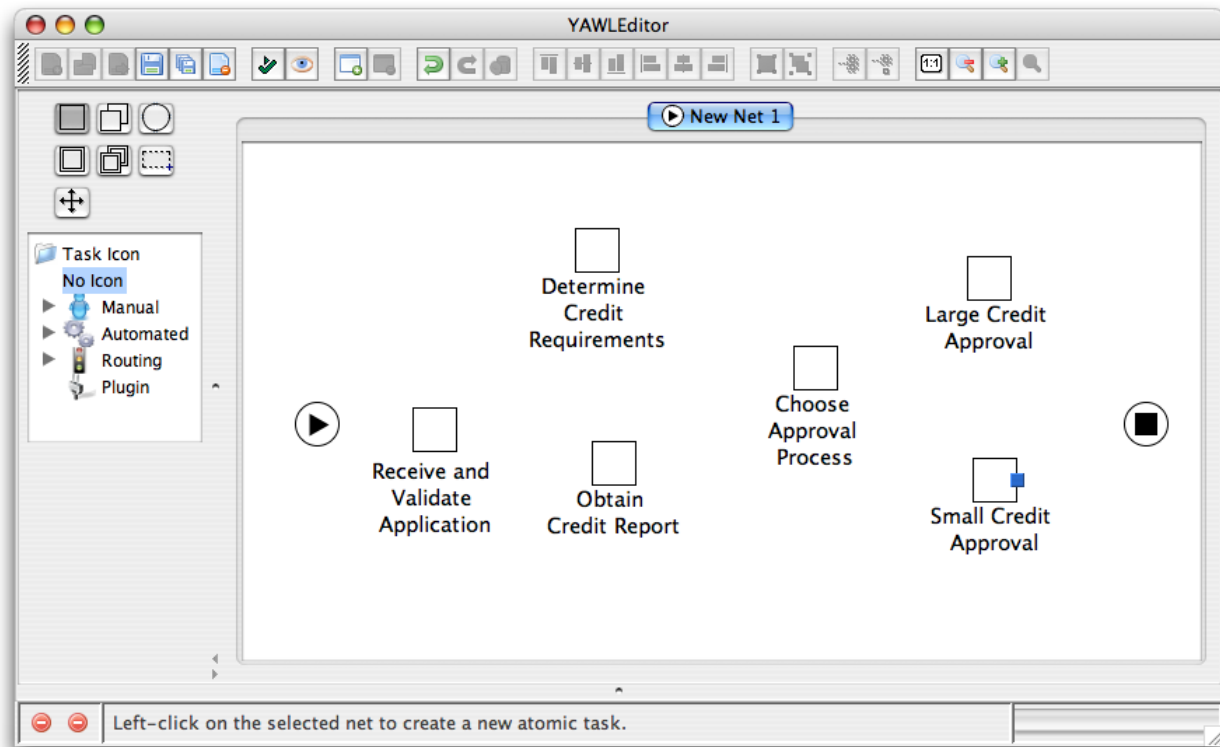
Figure 3.2: Atomic Tasks Added to the Starting Net

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 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.

We now 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 Work-list* (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 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 to a Web Service that can, for example, send notifications to
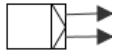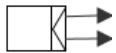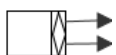
Name:	Symbol:	Description:
Split Types:		
XOR-Split		The XOR-Split is used to trigger *only one* outgoing flow. It is best used for automatically choosing between a number of possible exclusive alternatives once a task completes.
AND-Split		The AND-Split is used to start a number of task instances simultaneously. 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

people via SMS and receive replies from them also via SMS.

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. 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.
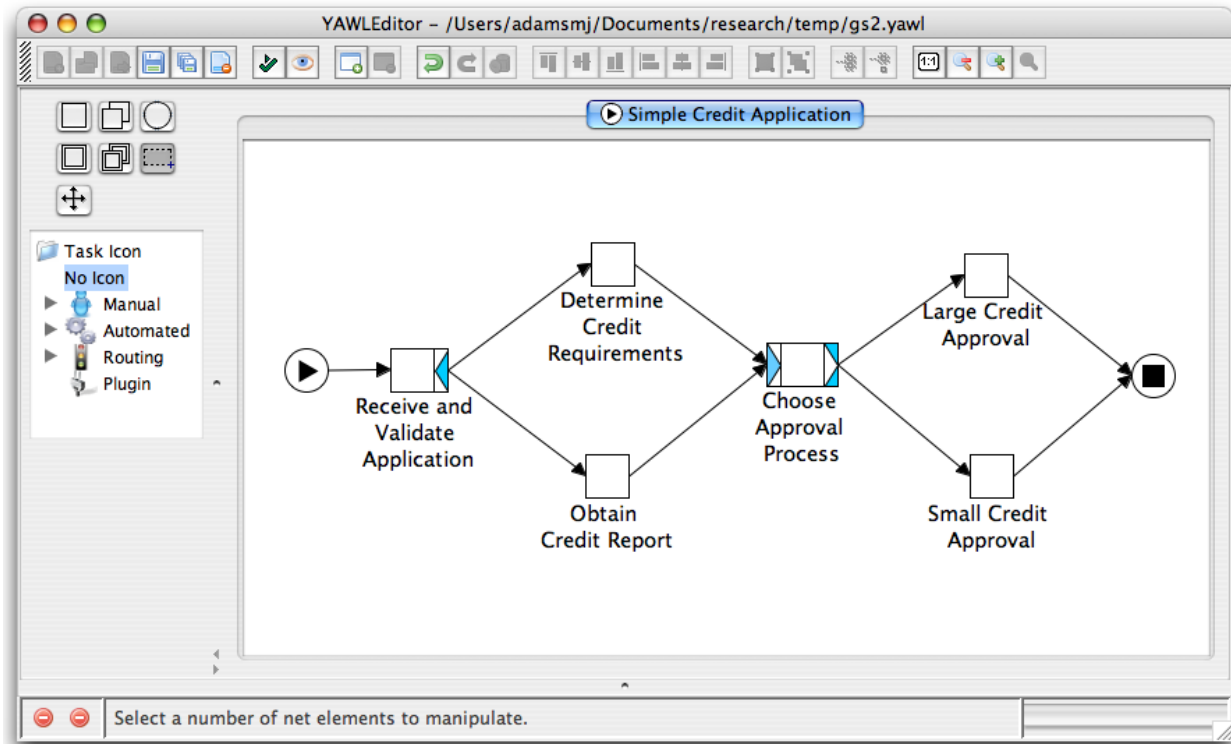
Figure 3.4: Multiple Flows Between Tasks

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 starting the Resource Service (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 tasks. Right click with your mouse on a task and choose "Manage Resourcing" from the popup menu (this menu
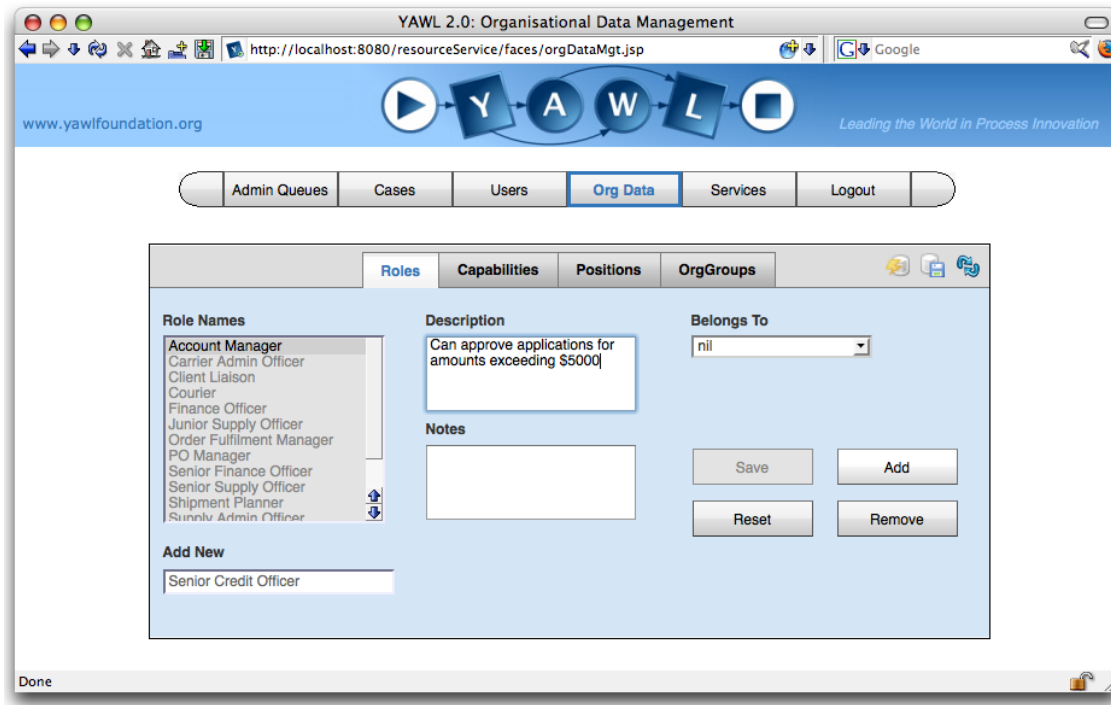
Figure 3.5: Defining a Role

item will be disabled for tasks without decompositions defined). The Resourcing Wizard will begin and you can choose the interaction strategy for that task. In Figure 3.7 an interaction strategy is chosen where, at runtime, the system offers a work item to all authorised participants, from where one of these participants can then choose to perform this work item 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.

Step 2 of the Resourcing Wizard then allows you to choose who is authorised to execute work items of the particular task. In Figure 3.8 the "Senior Credit Officer" role is assigned to the task, thus guaranteeing that at runtime work items of this task are offered only to participants that perform that role.

Resourcing requirements can be 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 nets of a YAWL specification along with all tasks that require interaction with the external environment need a decomposition. Decompositions can have a number of *parameters* (or variables) 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 parameter has a *name* it may be referenced by, a *type* dictating valid values it may store, a *designation* indicating how that data may be used, and a *scope* defining the visibility of the parameter.

Figure 3.6: Defining a Participant

Parameters can belong to one of two scopes, which we'll name *net scope*, and *task scope*. At runtime, every net and task instance with a decomposition will have data stored as a number of parameters belonging to it. To get data from a net instance to a task instance it contains, 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 parameter are *input*, *output* or both *Input & Output*. A task parameter with an input designation is one where we expect data to be delivered from a net-level parameter to that parameter at run-time. A task parameter with an output designation can have its data output to a containing net-level parameter once a task instance has completed.

Just like task parameters, net parameters may have *input* and/or *output* designations. But in addition to these two options, net parameters may have a *local* designation. You can think of a net parameter with a local designation as a local variable. They are used to store intermediate data during the execution of a process instance.

Data transfer from a net a task is achieved via *inbound mappings*. An inbound mapping is a statement that says how to transfer data from the net's parameters to a task's input parameters. 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 the task's output parameters to its containing net's parameters. XPath expressions[4] are

---

[4]For more advanced workflows, XPath expressions may prove too limiting. Accordingly, YAWL allows developers to also use
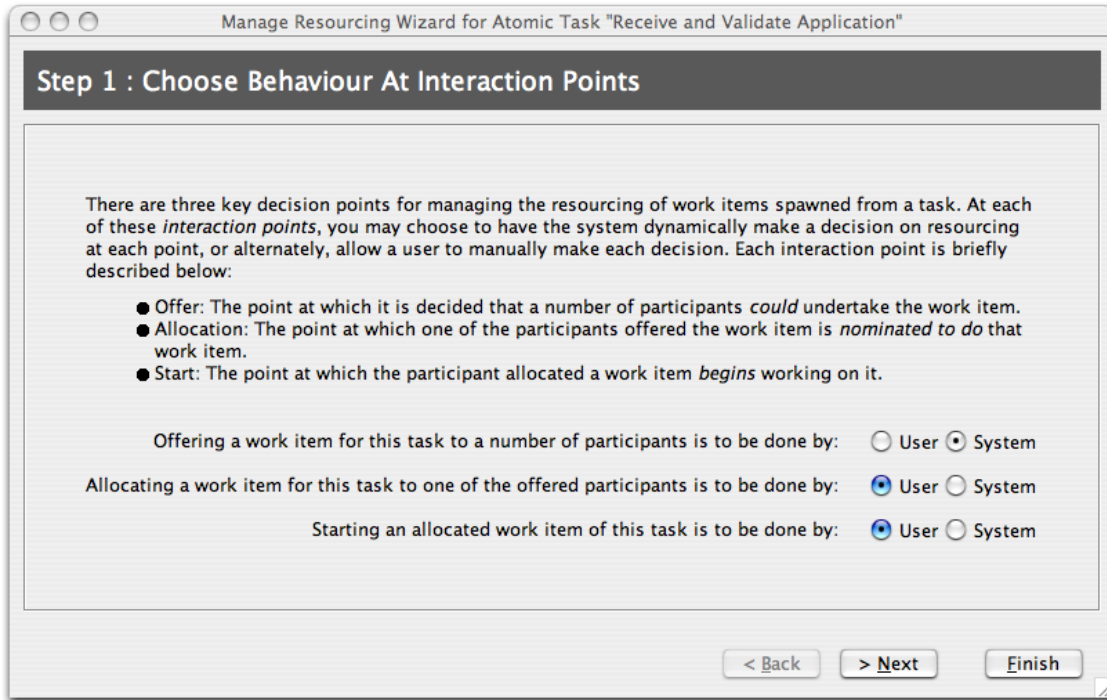
Figure 3.7: Specifying the Interaction Strategy
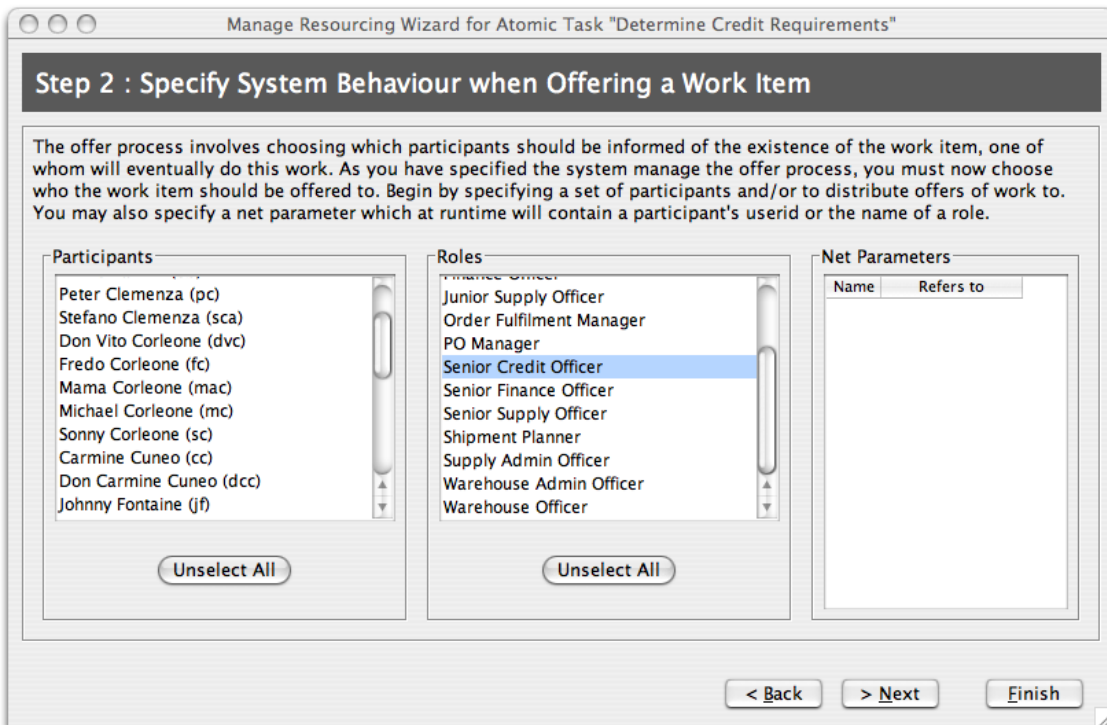

Figure 3.8: Assigning a Role

used to describe inbound and outbound mappings. Accordingly, the parameters of nets and tasks in YAWL

XQuery expressions for data transfer in such cases.
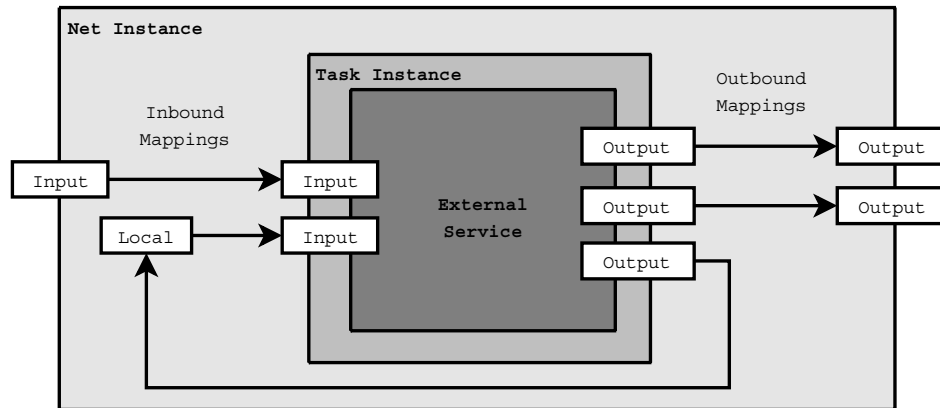
are all encoded as XML documents.

Figure 3.9: Example Data Transfer between a Net and Task

Figure 3.9 depicts example data transfers over the lifetime of a task instance. The task's decomposition defines two input parameters and three output parameters. When the task instance starts, values for its input parameters are populated by executing the input mappings for the task, which are then passed onto the task's designated YAWL Service. The default worklist handler is an example of an external service, but there are many others and advanced users are able to define and add virtually any service they want. The external service eventually finishes its execution, resulting in values being supplied to the output parameters of the task instance. The output mappings for this task instance are then executed, resulting in a number of parameters in the task's containing net instance being updated with values from the task instance's output parameters.

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 users to also define their own). For our example, will limit our parameters to be of either XML Schema *string* or *double* simple types. We'll go through our atomic task decompositions now and add parameters to each task decomposition first before we add 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 parameter requirement per decomposition as per figure 3.10.

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 3.10: Parameters Required for Task Decompositions

To give you some idea of how this might look, figure 3.11 is a screenshot of the Editor showing the decomposition for the task *Determine Credit Requirements* with an input parameter *ApplicationID*, and an output parameter *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 3.12 shows how the default worklist handler displays an instance of this task for a participant to work with.
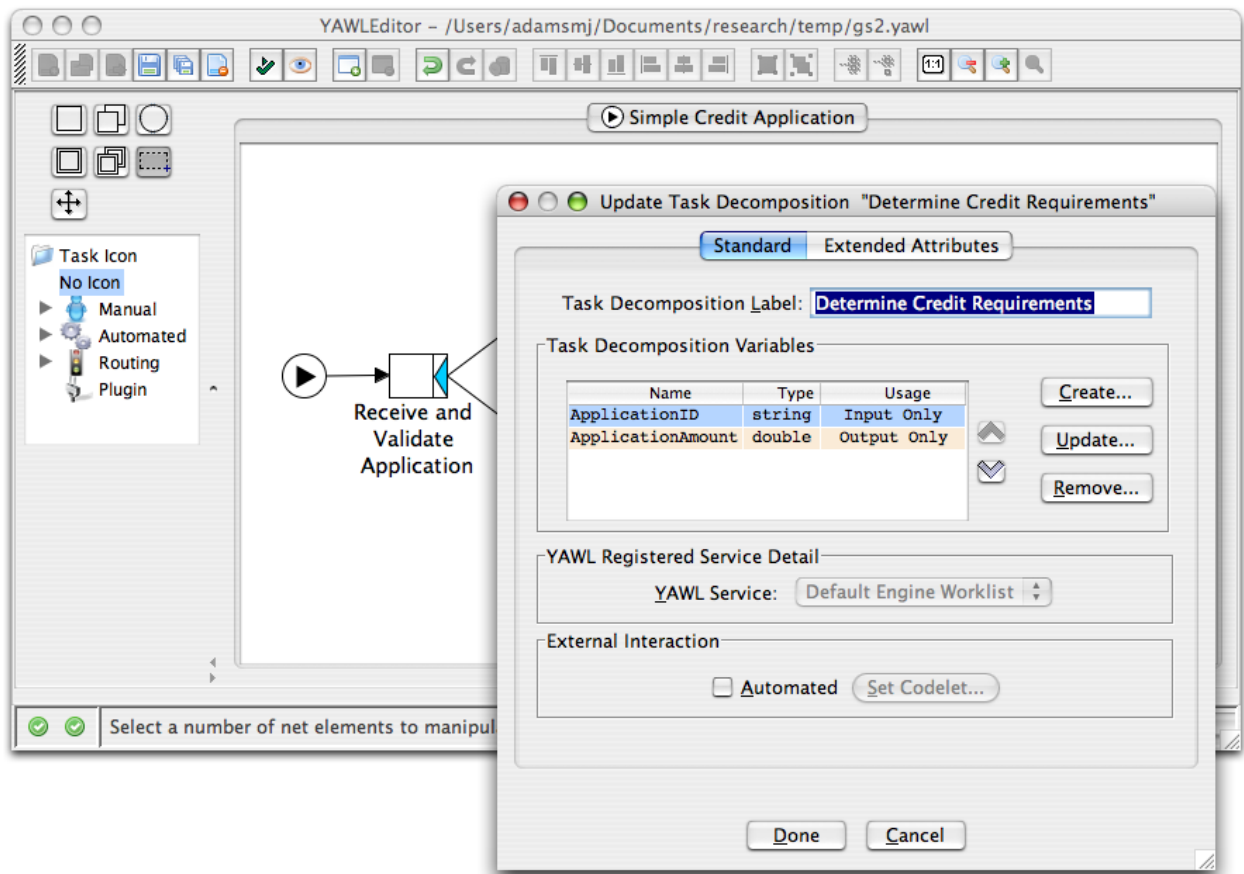

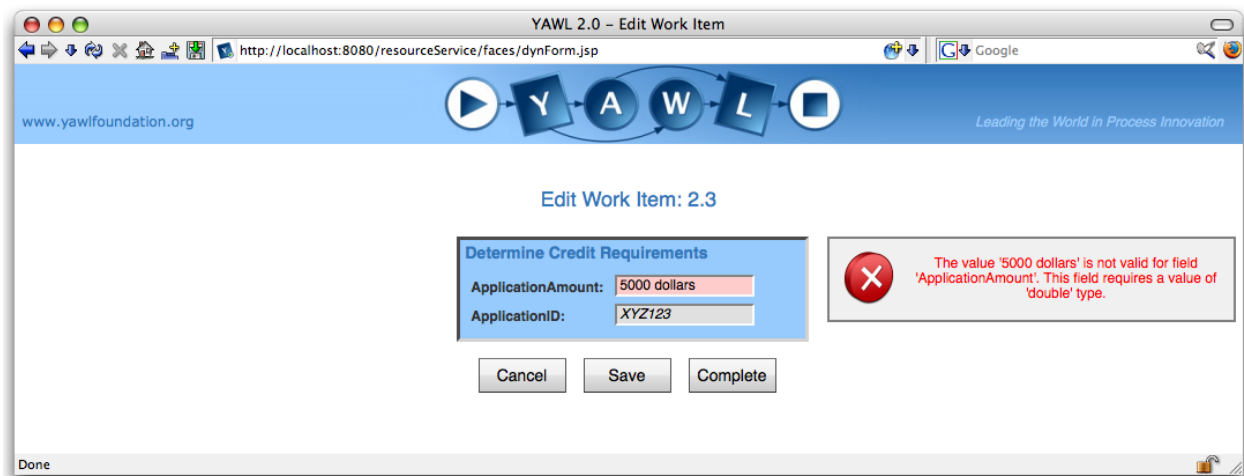**Figure 3.11:** Establishing Parameters on a Task Decomposition

Figure 3.12: A *Determine Credit Requirements* work-item shown on a dynamic form

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're also going to need a number of *local parameters* for the decomposition of 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.
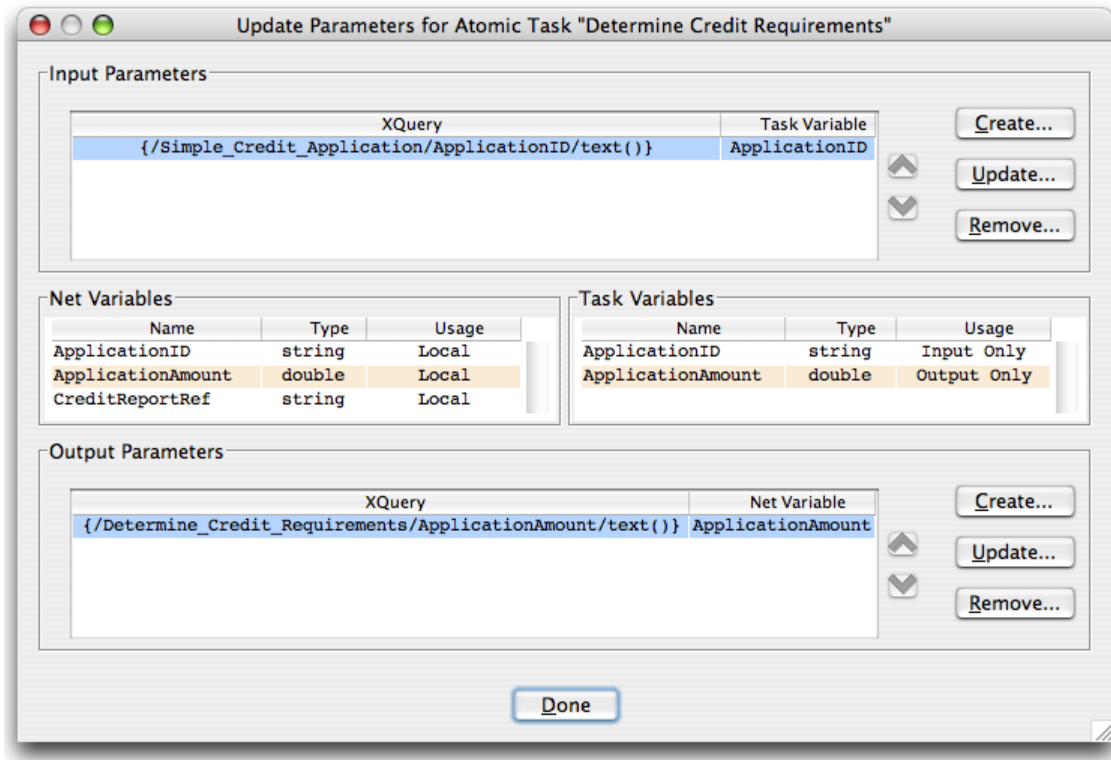

Figure 3.13: Data Mappings for *Determine Credit Requirements* Parameters

With these local net parameters in place, we now need to specify how data is passed between the net and tasks with XPath expressions. The XPath expressions needed are fairly straightforward. For a task input parameter, we need the expression to fetch and populate the value of this parameter with that of its corresponding net parameter. For a task output parameter, the matching net parameter needs an expression to retrieve the value of the task parameter. Figure 3.13 shows an example of the queries needed for the *Determine Credit Requirements* task (note that XPath expressions of this kind are generated with a couple of mouse clicks in the Editor). A similar exercise can then be conducted for the parameters of the remaining tasks.

The only thing remaining to do with data in our specification is to deal with the XOR-Split. XOR-splits need a boolean XPath expression to be associated with each outgoing flow of the split[5]. These expressions are evaluated once a task instance completes. Thus, the expressions can only interrogate the state of a net that a recently completed task has updated. 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,

---

[5]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 workflow can continue even when all other flow conditions evaluate to false.`

approval amounts of less than \$5000 are to be routed to the *Small Credit Approval* task. Anything more requires *Large Credit Approval* to be run. The XPath expressions needed to capture this choice are shown in Figure 3.14. 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.
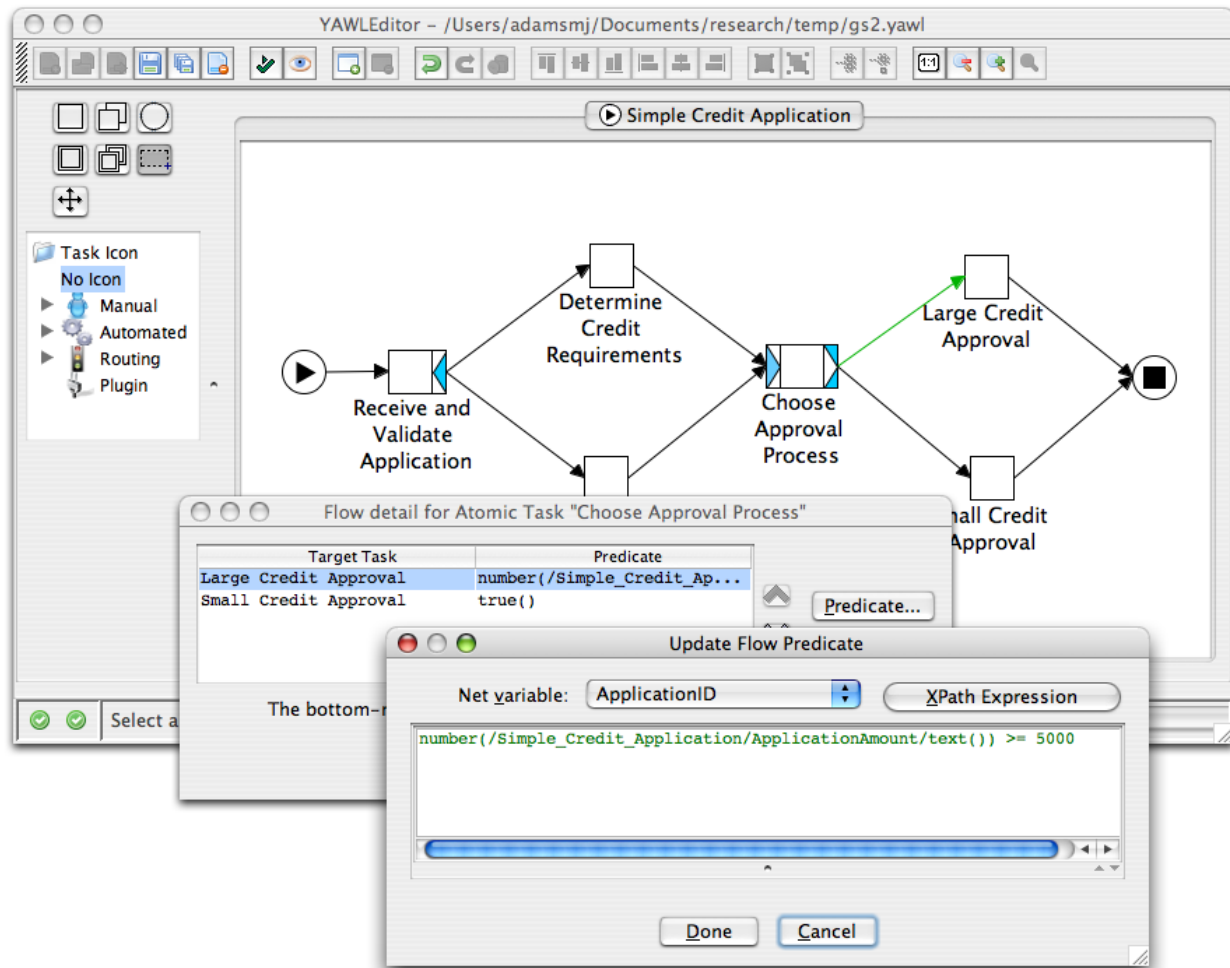

Figure 3.14: XPath predicates to choose between flows of an XOR-Split task

3.5   Where To From Here

You've now seen how we can construct a 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 2.0 of the Editor and of the Engine for the screenshots in this chapter. The components of YAWL can all be found at Sourceforge, via the URL http://sourceforge.net/projects/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. Mailing lists and forums around the components of YAWL can also be accessed via this Sourceforge URL. 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 http://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 2.0), a tool for creating, editing, 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.
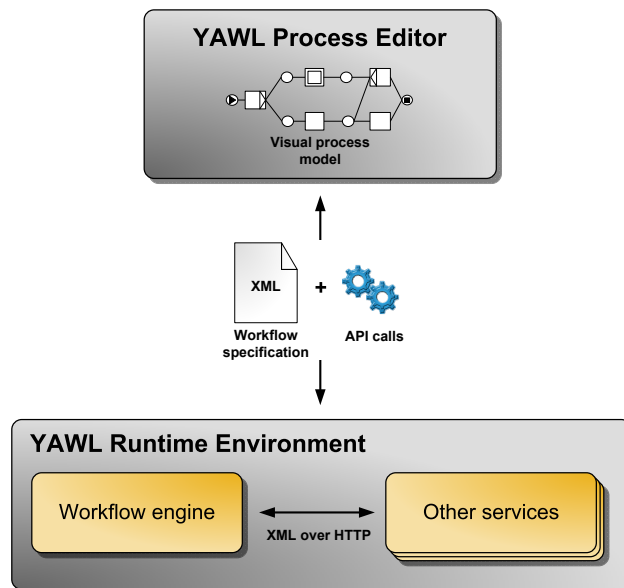

Figure 4.1: The YAWL Components

In this chapter, this icon 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 SourceForge website: http://sourceforge.net/projects/yawl/. Be sure that the version number of the Editor you are using matches the version of YAWL installed.

The YAWL Editor is distributed as a Java Archive (jar). Double click on the *YAWLEditor2.0.jar* file to start

the application (where supported). The YAWL Editor can also be started from a command line or Terminal prompt:

    java –jar YAWLEditor2.0.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 asking you to open or create specification to begin.

Before you create your first specification, let us 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.
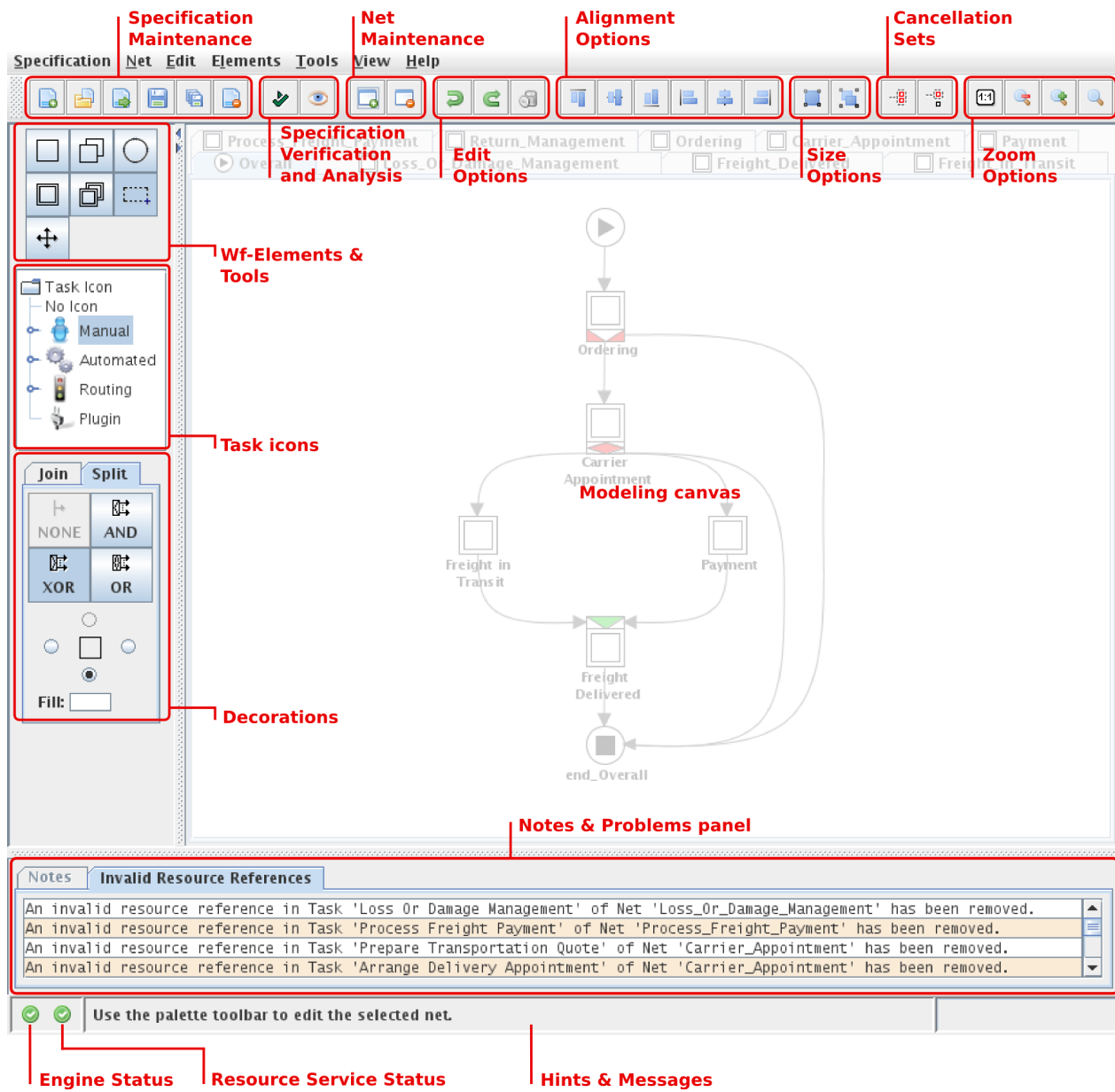

Figure 4.2: The YAWL Editor Workspace

4.2.1 The Toolbar

The Menu Toolbar contains eight groups of buttons to assist you in maintaining your YAWL specification. The menu can be repositioned by dragging the left-hand anchor bar. Each button may be enabled or disabled at certain times depending on what you are doing in the Editor.

Specification Maintenance

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

- Create a new specification;

- Open an existing specification file. Specification files will have a .yawl extension (or sometimes a .xml extension, if they are an 'engine' file created with a pre-2.0 Editor version);

- Import a specification file created with a pre-2.0 version of the Editor. These files have a .ywl extension.

- Save the currently loaded specification to file. For newly created specifications, this behaves the same as *Save As*;

- Save As a new file name;

- Close the loaded specification. If there are any unsaved changes, you will be prompted to save the file first before closing.

Specification Verification & Analysis

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

Net Maintenance

Each workflow specification consists of one or more nets. You can use these buttons to add a new Net to or remove an existing Net from your specification.

Edit Options

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

Alignment Options

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;

- right sides

The first selected object is used as the reference to align the other objects to.

Object Sizes

To increase or decrease the size of an object or objects within your specification, select the object(s) and then use these buttons.

Cancellation Sets

These buttons allow you to include in and/or exclude elements from the cancellation set of a task.

Zoom Options

These buttons allow you to apply zoom functionality to the currently selected net. From left-to-right, reset the zoom to the actual size, zoom the entire net out, zoom the entire net in, and zoom into the currently selected net elements.

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 menu toolbar.
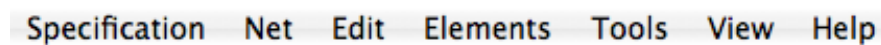
Specification

In addition to the Specification Maintenance, Verification and Analysis toolbar items, this menu also contains these sub-items:

- Open Recent: show 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);

- Update the Specification Properties: such as specification name, author, description and so on;

- Update Datatype Definitions: where you can define your own data types to be used in the specification.

Net

In addition to the Net Maintenance toolbar items, this menu also contains these sub-items:

- Set Starting Net: for specifications containing several nets, this item allows you to specify which of them is the starting net (i.e. the net that begins execution of the workflow instance);

- Update Net Detail: shows a dialog where you can set the name of the net, and create/update/remove net-level variables;

- Export to PNG Image: saves a graphical image of the net to a file;

- Net Background Colour: set the background colour of the selected net;

- Print Net: prints the currently selected net (graphically).

Edit

In addition to the Edit Options toolbar items, this menu also contains sub-items to Cut, Copy and Paste objects to/from the canvas.

Elements

This menu contains the Alignment Options, Object Sizes and Cancellation Set toolbar item sets. You can also set the fill colour for all selected tasks and conditions using this menu.

Tools

This menu contains the following items:

- Engine Connection Settings: The Editor must connect to a running Engine to obtain a list of the available services that a task can be assigned to. This menu item allows you to set the parameters for a connection to the Engine and to proceed with a connection.

- Resource Service Connection Settings: The Editor must connect to a running Resource Service to obtain a list of the available resources that task can be allocated to (amongst other things discussed in later sections). This menu item allows you to set the parameters for a connection to the Resource Service and to proceed with a connection.

- Configure Specification Analysis: This item will display a dialog where various verification and analysis techniques may be chosen. If the **wofyawl** analysis utility is also supplied in the same directory as the editor, the configuration dialog will allow specification designers to configure and use wofyawl for additional specification analysis[1].

---

[1]Only available when installed in a Windows environment.

View

You can use this menu to toggle:

- Tooltips, which provide useful hints when your mouse is positioned over various items;

- Anti-aliasing of graphical components; and

- Grid on the canvas background – useful for aligning objects visually.

This menu also provides options to set the font size used for element labels, the default background colour for nets, and the default background colour for elements (i.e. tasks and conditions). Finally, it shows a list of all the nets of the loaded specification, allowing the selection of one from those available for editing.

Help

The Help Menu provides an "About the Editor" dialog, describing components used in the editor's construction, a list of source code contributors, and the version and build date of the Editor in use.

4.2.3   Workflow Elements and Tools

The Workflow Elements and Tools panel contains seven selectable buttons – five YAWL language icons and two selection tools – that assist with creation, selection and positioning of objects within your specification. This panel is also accessible by right-clicking on any blank area of the canvas.

Once an element is selected, it is possible to place objects of that type on the canvas by left-clicking the mouse button.

Atomic Task

Select this button to create an Atomic Task, which represents a single task to be performed by a human participant or an external application.

Composite Task

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

Multiple Instance Atomic Task

Select this button to create a Multiple Instance Atomic Task, which allows you to run multiple instances of a task concurrently.

Multiple Instance Composite Tasks

Select this button to create a Multiple Instance Composite Task, which allows you to run multiple instances of a composite task concurrently.

Condition

Select this button to create a Condition, which is a way to represent state for the Net.

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 using the Marquee Selector.

Drag Net Window

Select this button to drag the visible window of a net around that net.

4.2.4 Other Components

The Canvas

The Canvas is where elements are placed to create and modify a workflow specification.

Task Icons Panel

This panel shows a set of icons that can be selected and placed on the tasks of your specification to add visual cues to aid in the understanding of your models by others. The Editor comes with a standard set of icons, and you may also provide your own icons and access them via this panel. Note that the icons are grouped for ease-of-use only; you are not limited in how you actually use the icons in your model.

Decorations Panel

The Decorations panel provides a set if decorator types that may be attached to a task. You can select the type of decorator, what edge it is to be positioned on the task and choose a colour to use for each decoration.

Notes and Problem Panel

This panel consists of two tabs:

- On the *Notes* pane, you can add freeform text to accompany the selected task or condition;

- The *Problems* pane will list problems or messages that may occur while you are building your model, when you validate it or when you analyse it.

Status Bar

The Status Bar consists of three parts:

- On the left are two icons that indicate whether there are currently valid connections to the Engine and the Resource Service (required for certain design activities discussed later in this chapter). A connection will show a green indicator, a disconnection as a red indicator;

- Next there is a status message area that provides useful contextual hints throughout the creation of your specification;

- On the right is a progress bar, which shows the progress of various events at different times.

4.3   Creating Your First Specification

Overview

This section will lead you through the process of creating a YAWL specification from beginning to end, through a series of brief lessons following a scenario.

You can either follow all the instructions including the scenario provided, from beginning to end, or skip straight to the section that you are interested in and follow the instructions.

Look for the student icon next to the instructions for specific details of the scenario.

The Scenario

The scenario that we will be following throughout this section is the workflow of a student who has just completed their secondary study and is now looking to start their career. The scenario will follow the path of a student who either enrols in a University to complete their tertiary education, or undertakes private study that will eventually lead them to getting a job and starting their new career.

4.3.1   Creating Your First Specification

1. Click on the Create a New Specification button, , at the top left of the Menu Toolbar, or click on **Specification** in the Menu and choose **Create Specification**. This will create a blank Net called "New Net 1" which will be, by default, the starting net of the workflow. For details on selecting a starting net, consult Section 4.3.6.

2. Click on **Specification** in the main Menu and choose **Update Specification Properties**. A screen as per Figure 4.3 will appear. Alter the specification's properties as you feel appropriate.

3. Rename this Net by clicking on the **Net** Menu and choosing **Update Net Detail**.

4. Enter the new name of the Net in the "Decomposition Label" field, then click the **Done** button.

   Decomposition Variables will be explained later in Section 4.5.3.

   Change the name of the Net in the Decomposition Label, to "My Career". This Net will be the primary net for our scenario.

5. You are now ready to start drawing your specification.

4.3.2   Atomic Tasks

1. Click on the "Add an Atomic Task" button, , in the Workflow Elements Panel, or right click in an empty area of the canvas, and choose **Atomic Task**.
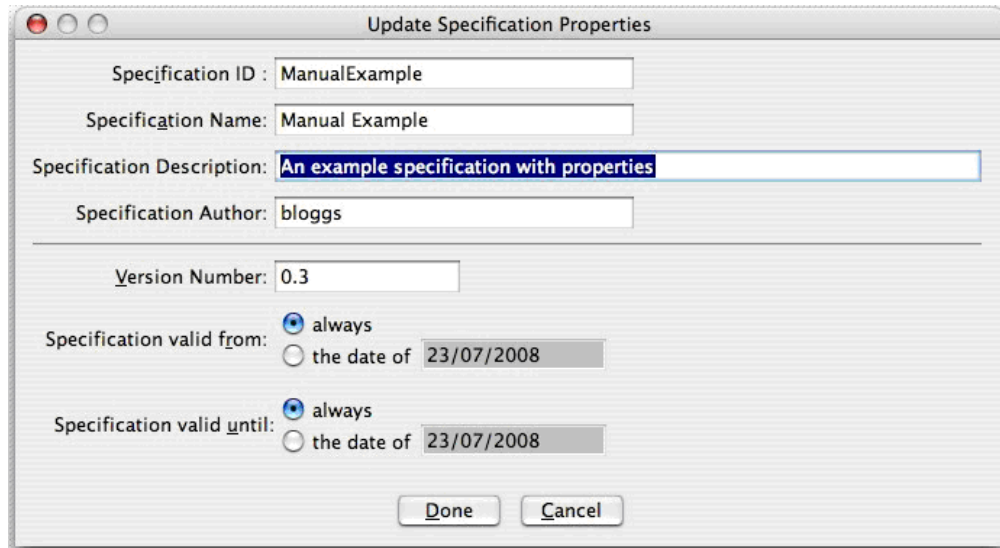
Figure 4.3: Specification Properties Dialog

2. Position your mouse just to the right of the Input Condition (the ▶ symbol), and click the left mouse button once to place an Atomic Task.

3. Set the decomposition of this task by right clicking on the Atomic Task and choosing **Select Task Decomposition**. You should see a dialog as per Figure 4.4.
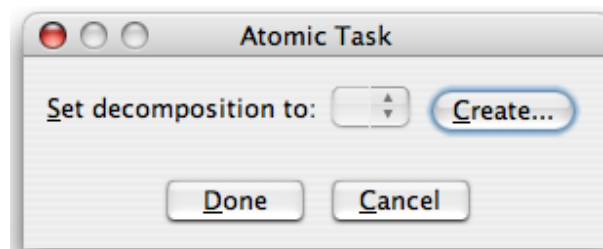

Figure 4.4: The "Select Task Decomposition" dialog

4. Press the **Create...** button, and in the following *Update Task Decomposition* dialog, enter the decomposition's label.

   See Section 4.5.5 for a full explanation of this dialog's features.

   Set the label to "Begin My Career", and click the **Done** button.

5. Note that by default, a 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 right-clicking on the task and choosing **Set Label...** from the pop-up menu. This will not change the name of the decomposition with which the task is associated.

6. Connect the Input Condition to your Atomic Task, as shown in Figure 4.5, by finding the flow connectors that appear as small blue boxes as you hover your mouse over the sides of the objects. Hold the left mouse button down over a flow connector and draw a line by dragging the mouse from the flow connector on the Input Condition to the one on the Atomic Task (which will appear when the mouse hovers over

the edges of the task). The editor will only show a connection point if it is valid to draw a flow connection between the objects. The directed arc (arrow) between two objects is referred to as a *flow*
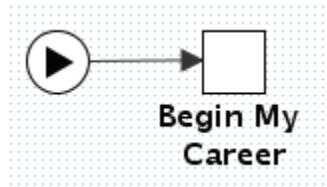

Figure 4.5: An established flow relation

relation, or most often simply a *flow* – it shows the 'flow' of execution from one object in the net to the next. That's it! Your Atomic Task is set.

Repeat the process for the following Atomic Tasks in order: Go to University, Get A Job, Career Started.

Link the 'Career Started' task to the Output Condition (the ⬤ symbol), as per Figure 4.6.
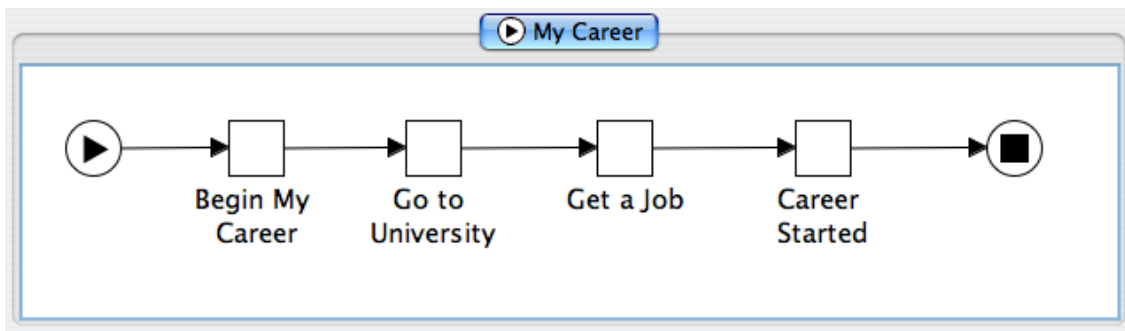

Figure 4.6: The "My Career" Net

7. 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 Specification**. If all things are going to plan, then you should receive a confirmation saying that there were no errors detected.

Task Indicators

Task indicators are mini-icons that appear across the top of a task to provide a visual cue regarding settings that have been applied to the task. An example of a task with all three available indicators is shown in Figure 4.7.
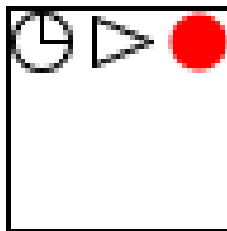

Figure 4.7: Task Indicators

The three task indicators (left to right in Figure 4.7) are:

- **Timer**: This task has had a timer set (see Section 4.11).

- **Automated**: This task has been set as automated (see Section 4.9). If this automated task also has a codelet specified, it will be filled green (see Section 4.9.1).

- **Cancellation Set**: This task has had a cancellation set defined (see Section 4.5.1).

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

Task Decoration

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 possible 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 a number of outgoing flows; when the task completes, it will activate *each and every* outgoing flow;

- XOR split: The task may have a number of outgoing flows, each with an associated 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 a number of outgoing flows, each with an associated 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 flows 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). Below are possible 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 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 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 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 technical papers on the YAWL website.

Creating Splits and Joins

To create a split or join:

1. Select a task. When a single task is selected the Decorator panel will appear with two tabs that allow you to decorate a task with a split and/or join. You can also choose a fill-colour to help visually differentiate splits from joins with the expanded colour palette.

2. Choose the required split or join and the orientation (which edge of the task to attach the decoration to) for the split or join to appear.

In our example, select the "Begin My Career" task and, in the Decorator panel, select an **XOR** split. Then set the orientation to eastern edge of the task, as per Figure 4.8.

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

Finally, select your "Get A Job" task and decorate it with an **XOR** join. Then set the orientation of the join 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 4.8.
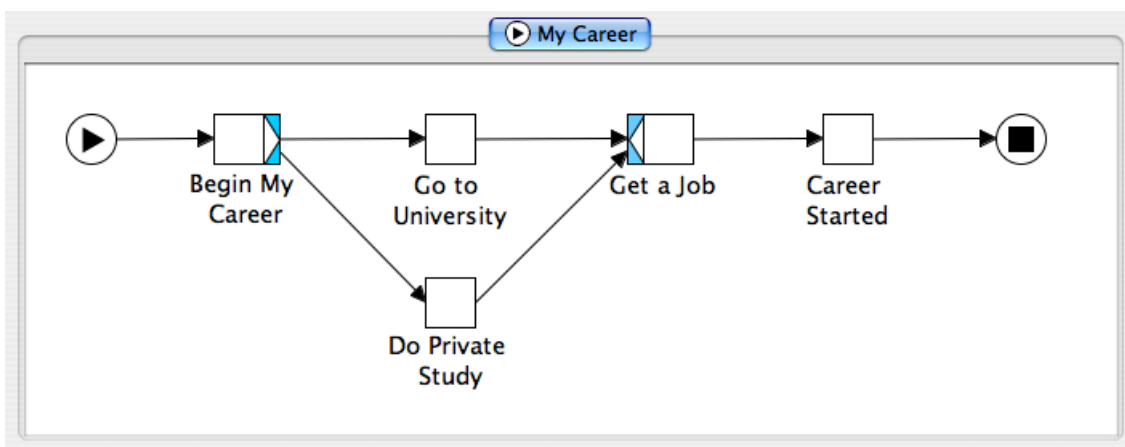

Figure 4.8: XOR Split and Join

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.

When "Begin My Career" has been completed, a choice must be made on which of the two tasks ("Go To University" or "Do Private Study") will be followed (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 at the point of the XOR split.

Composite Tasks

1. Composite tasks are placeholders for other sub-nets. That is, you can create another workflow in a separate Net, which is represented in the first net by the composite task. When a composite task is activated, control branches to the sub-net; when the sub-net completes, control passes back to the parent net.

2. To create a Composite Task: click on the Composite Task button, ⬜, in the Elements panel 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, so click on the "Go to University Atomic Task" and click the trash bin on the toolbar or press the Delete key on the keyboard. We will add in the new composite task next.

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

4. Create a new Net by clicking on the Create a New Net button, , on the Menu Toolbar, or click on the **Net** Menu and choose **Create Net**.

5. Choose a name for this Net by clicking on the **Net** Menu and choosing **Update Net Detail**.

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

6. Return to your original Net and right click on your Composite Task and choose **Unfold to net....** You will then be given a drop-down list with all the Nets available – choose the Net this task is to represent and then click **Done**.

   *Tip: you can also combine the last three steps by simply right clicking on the newly added composite task, selecting **Unfold to net...**, then clicking the "Create" button in the dialog that appears.*

   Choose "Attend University".
   You can now fill out the detail of your new "Attend University" Net.

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

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

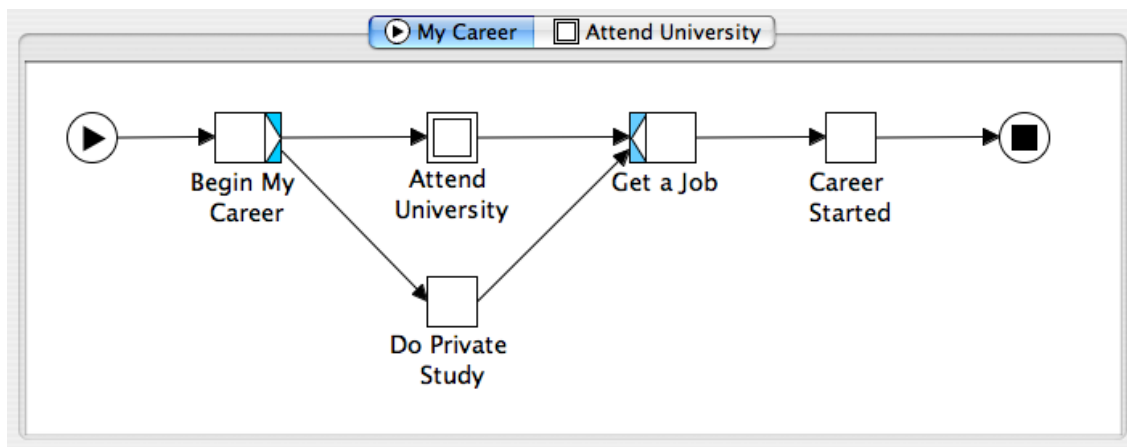   The resulting nets are shown in Figures 4.9 and 4.10.

Figure 4.9: Parent net with "Attend University" Composite Task

4.3.3 Multiple Instance Atomic Tasks

Multiple Instance Atomic Tasks (MI Tasks) allow you to run multiple instances of a task concurrently.
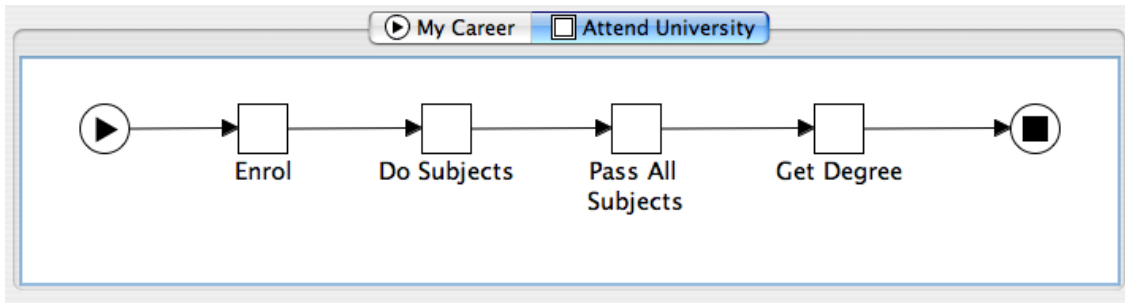
To create a Multiple Instance Atomic Task:

Figure 4.10: The "Attend University" sub-net

1. Click on the Add Multiple Instance Atomic Task button, ⊡, in the Elements panel 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 your Multiple Instance Atomic Task in your Net and set the name of this task by right clicking on the task and choosing **Select Task Decomposition**.

   Give this task the same decomposition as before by selecting "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 4.11.
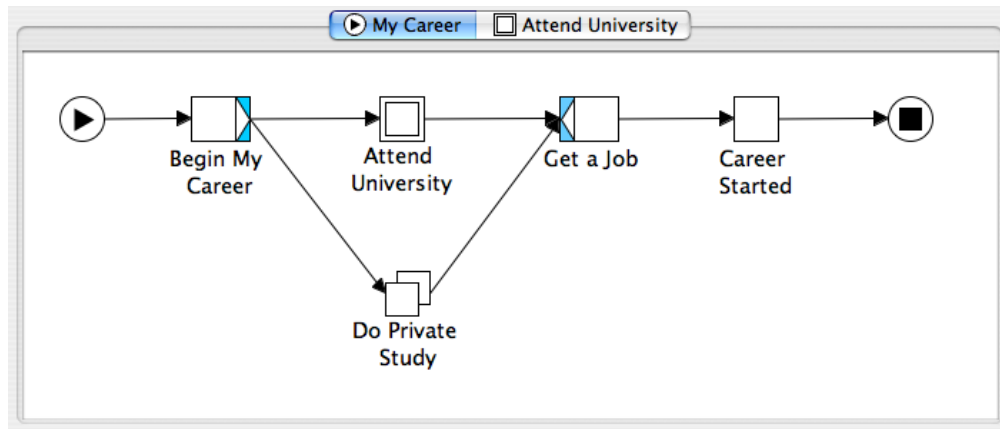

Figure 4.11: Adding a Multiple Atomic Task

3. You will now need to set the parameters of the MI Task, which, being multiple instance, needs a few more values set than a simple atomic task. Right click on the task and choose **Set Instance Detail**. Ensure that you are viewing the "Bounds" tab of the dialog, as per Figure 4.12.

4. Choose the Instance Creation mode. 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". *Static* mode means the number of task instances started cannot vary once the task is activated. *Dynamic* mode 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".
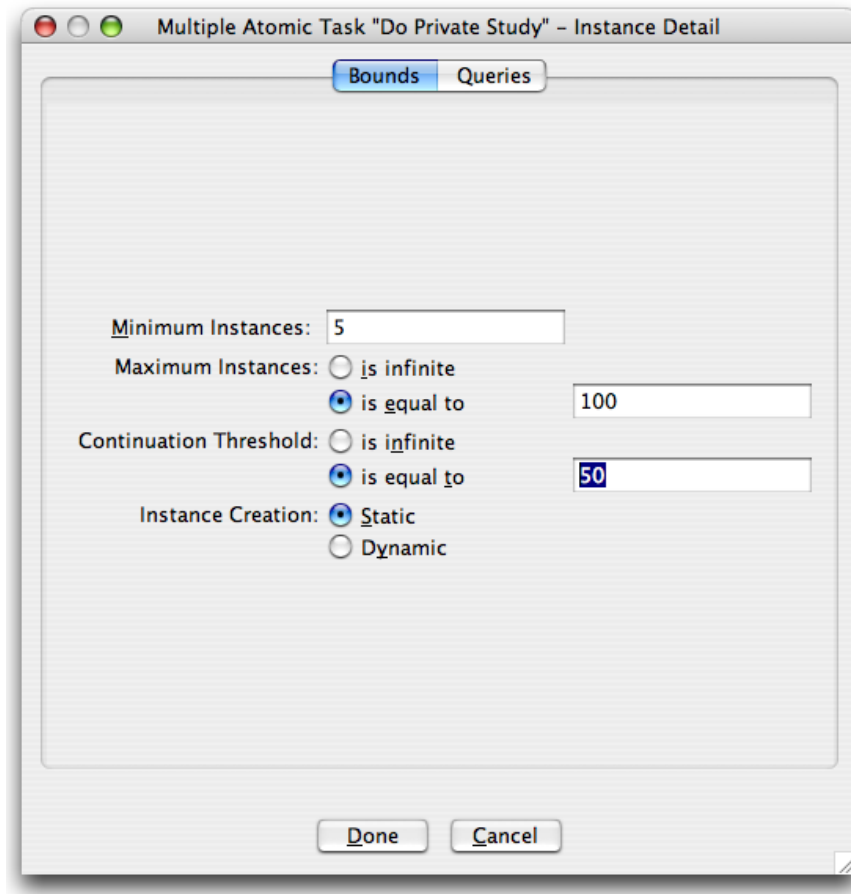
Figure 4.12: Instance Bounds on Multiple Instance Tasks

Set the Instance Creation type to "Static".

5. 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.

6. 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.

7. Set the Continuation Threshold value. The moment all task instances created have completed, or if the number of instance created exceeds the Continuation Threshold, the number specified for the Continuation Threshold have completed, the multiple instance task itself will be considered complete, and will trigger relevant outgoing flow relations from this task.

Set the Continuation Threshold to 50.

8. Click **Done**.

With the values set in the scenario, it has been 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.

We will revisit the setting of parameters for multiple instance tasks, in particular the details of the "Queries" tab of the Multiple Instance Detail dialog, in Section 4.5.9, after the basics of queries have been introduced.

4.3.4   Multiple Instance Composite Tasks

Multiple Instance Composite Tasks allow you to run multiple instances of the sub-net represented by a multiple composite task, concurrently.

To create a Multiple Composite Task:

1. Click on the Add Multiple Composite Task button, [image], in the Elements panel 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. You will now need to set the parameters of the Multiple Composite Task, in the same manner as those set previously for the Multiple Atomic Task. Right click on the task and choose **Set Instance Detail**.

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

4. Click **Done**.

5. Create a new Net by clicking on the Create a new Net button, [image], on the Menu Toolbar, or click on **Net** in the Menu and choose **Create Net**.

6. Give the new Net a name by clicking on the **Net** Menu and choosing **Update Net Detail**.

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

7. Return to your original Net and right click on your Multiple Composite Task and choose **Unfold to Net**. You will then be given a drop-down list with all the Nets Available – choose the Net for this task to initiate and then click **Done**.

   Choose "Study Privately".

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

   Create the following Atomic Tasks in order and then link them with Flow Relations as per Figure 4.13:

   - Read a Book
   - Feel Smarter

   Don't forget to validate your specification.

4.3.5 Conditions

Conditions represent states of the workflow and can be located in-between tasks. To create a Condition:

1. Click on the Add a Condition button, ⬡, in the Elements panel 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 loop Condition after the Read a Book atomic task, to determine whether we gained any knowledge from the book. We will add the new Condition next.

2. Place your Condition in your Net and set the name by right clicking on the Condition and choosing **Set <u>L</u>abel**.

    Call this Condition "Knowledge Gained?".

3. Now link to 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 An Easier Book".

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

    Finally create the Flow Relation from the "Knowledge Gained?" condition back to the XOR join of the "Read A Book" atomic task, as per Figure 4.13.

6. Validate your specification.

    Validation should fail and report errors as per Figure 4.13. The problem here is that the "Study Privately" multiple instance composite task needs to have more information specified for it to be valid.

    For setting data detail of multiple-instance tasks, please see Section 4.5.9.

7. Remove the "Study Privately" multiple-instance composite task and replace it with an atomic composite task using the same decomposition, and re-drawing flows from "Begin My Career" and to "Get A Job" tasks. Your updated net should look like Figure 4.14, and should validate successfully.

The *Knowledge Gained?* condition in Figure 4.13 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 (note that a condition may have any number of outgoing flows). This results in both the "Look for an Easier Book" and "Feel Smarter" tasks appearing in the user's worklist, 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 worklist.
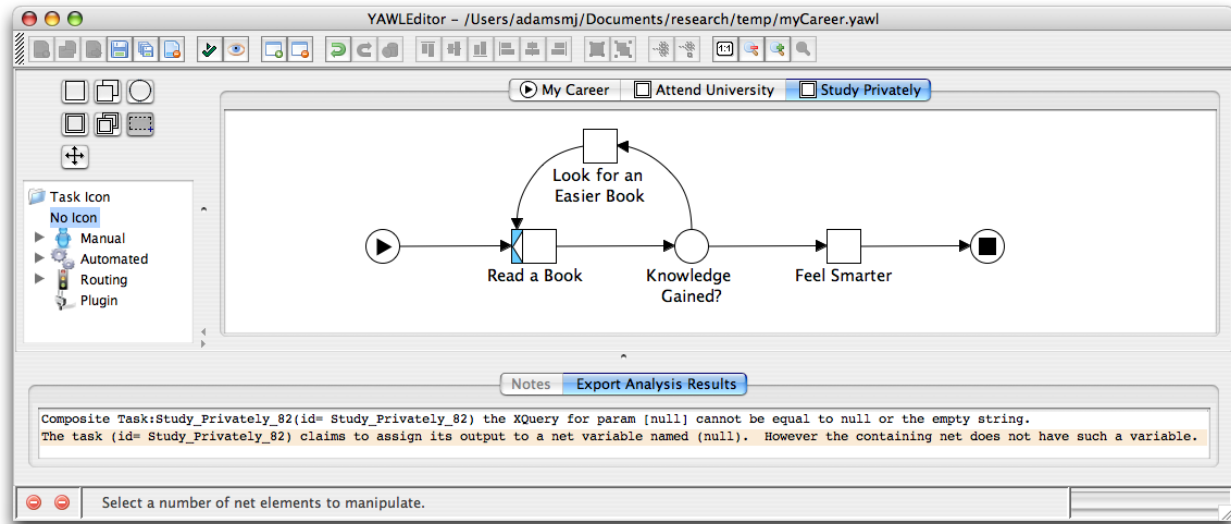
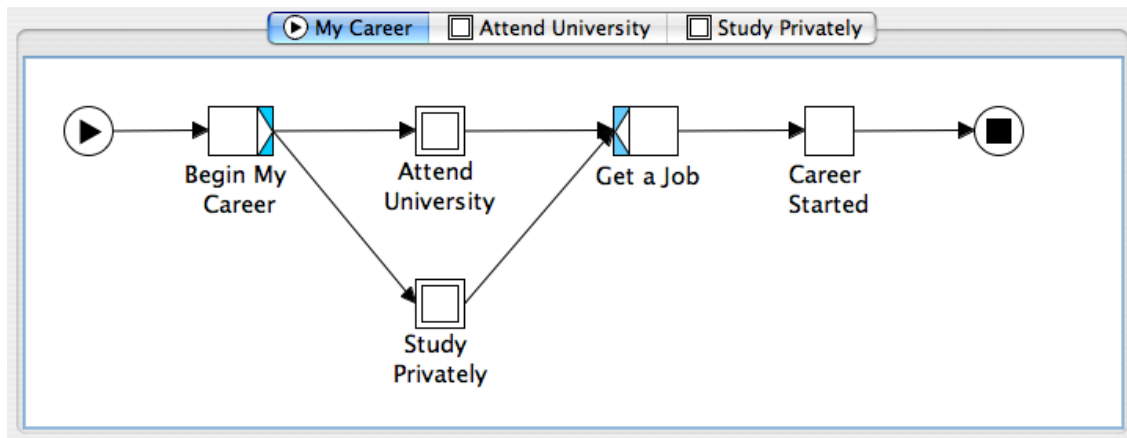**Figure 4.13:** Validation with unfinished Multiple-Instance Tasks

Figure 4.14: Making the "Study Privately" task an atomic composite

4.3.6   Changing the Starting Net

At any stage you can change the Starting Net of the specification. To change the Starting Net:

1. Select **Net** from the Menu.

2. Choose **Set Starting Net** (Figure 4.15).

3. From the Choose Starting Net window, click on the drop-down list and select a new starting Net.

4. Click **Done**.

Note that the starting net has an input condition symbol, ▶, in its title tab, and in the View menu list. All sub-nets have a composite task symbol, ▣, in their title tab, and in the View menu list.
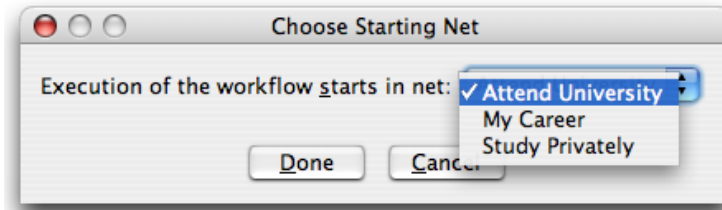
Figure 4.15: Changing the Starting Net

4.4 Changing the Appearance of Your Specification

4.4.1 Changing Flow Relations

Bends and Curves

You can control and improve the look of the flows between tasks by adding "bends" in them.

Go to the "Study Privately" Net.

Right click on the position in the flow where you want to add a bend, which will be denoted by a small blue square, ⎯⎯, in the Flow. A popup menu will appear, allowing you to add and remove bends, as well as change the line style of the flow.

Create a bend somewhere towards the middle of the flow going from the "Knowledge Gained?" condition to the "Look for an Easier Book" atomic task. Then left click on the bend marker created and drag it out to a more desirable location. You can add as many bends to a flow as you like.

Repeat the process for the flow between "Look For an Easier Book" and "Read A Book" tasks (see Figure 4.16).

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 to another. If a connection is possible at some other element, connection points will become visible as described earlier. Release the mouse button to attach the flow to its new home.

Take the current flow relation, and move it from the top of the task to its side, as depicted in Figure 4.16.

Adding Labels

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.
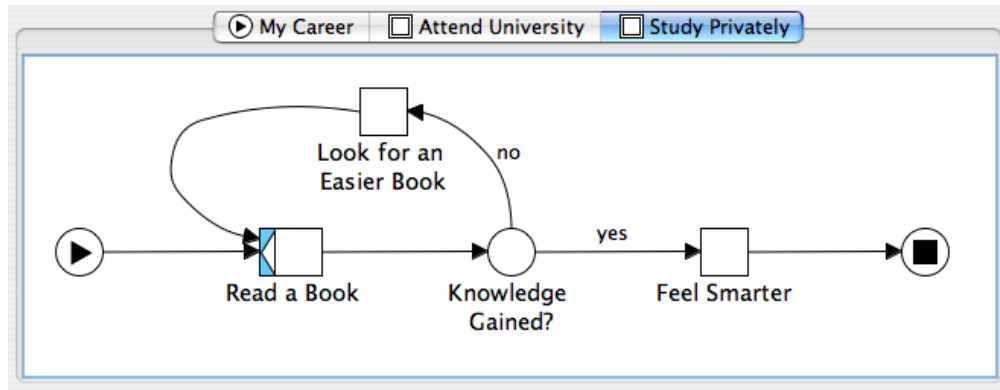
Figure 4.16: Adding bends to a Flow Relation

Take the two flow relations that have recently had bends added to them. 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 4.16.

Note that Figure 4.17 shows flows using two different line styles. The flow running from "Look for an Easier Book" has been given the "spline" line style in this figure, 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.

Setting Colours

For nets, the default background colour can be set (i.e. applied to all nets) by choosing *Default Net Background Colour...* from the *View* menu. To set the background colour of individual nets, choose *Net Background Colour...* from the *Net* menu.

For tasks and conditions, the default fill colour (i.e. for all newly added tasks and conditions) can be set by choosing *Default Element Fill Colour...* from the *View* menu. For individual tasks and conditions, right click on it then choose *Set Fill Colour...* from the popup menu. 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.

4.4.2   Editing Objects

You can edit more than one object at a time by using the Marquee Selection tool. See Figure 4.17.

1. Select the Marquee Selection tool, ▣, from the Elements panel.

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 edit.

3. Alternately, click and drag the Marquee tool to include multiple items in the drag rectangle.

4. Now choose the **Edit** option from the Menu or continue holding down the shift key and right click on the mouse button. Below are the edit options:
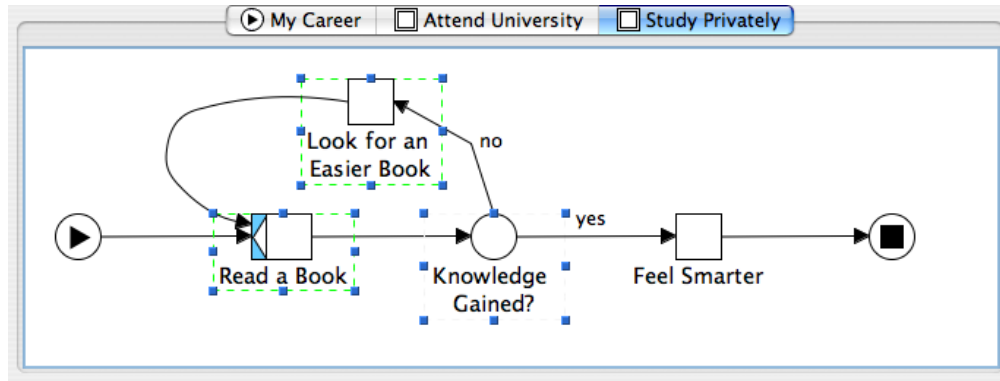
   - Cut, Copy, Delete;
   - Align;

Figure 4.17: Changing the Size of Multiple Objects

- Size Increase / Decrease to change the appearance of the objects. This can also be done using the CTRL key plus Up or Down arrow on your keyboard.

Note also, that whenever you have selected a number of net elements, pressing one of the arrow keys will move the selected elements in the direction of the arrow key, and pressing the CTRL key plus the A key will select all the elements in the currently selected net.

4.4.3 Changing Font Size

You can change the size of the font used to label tasks and conditions.

1. Change the font size by clicking on the **<u>V</u>iew** Menu and choosing the **Label Font <u>S</u>ize. . .** option.

2. Change the font size to that desired. The specified font size applies to all text drawn on the canvas.

4.4.4 Changing Task Icons

You can add or change the icon that is shown on atomic tasks.

1. Select any single atomic task in your workflow. The palette will expand to include a task icon tree, depicted in Figure 4.13, where you can an icon from the tree to the task. You are free to assign any icon. Icons have no runtime effect on the engine, and are provided simply to make specifications more easily understood by people looking at the specification in the editor.

4.4.5 Using Custom Icons

Workflow designers can plug in and use their own icons for specification design. Icons must be of the PNG file format, and be a maximum of 24 × 24 pixels to render properly within editor task boundaries.

When the editor starts, it checks the installation folder for the plugin directory:

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

for user-defined icons, and if found adds them into the plugin branch of the task icon tree widget of the editor's palette. Sub-directories are supported, and will form new sub-trees of the same name when the plugin sub-tree is being created. If an icon cannot be found that was previously used for a specification, 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   Additional Specification Features

4.5.1   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 that will initiate the Cancellation Set.

2. Right-click on the task, then choose **<u>V</u>iew Cancellation Set** from the context menu. The task will be fill with a grey colour to indicate that this is the task that 'owns' the cancellation set currently on view.

   Create the "Purchase Book" specification as shown in Figure 4.19. 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.
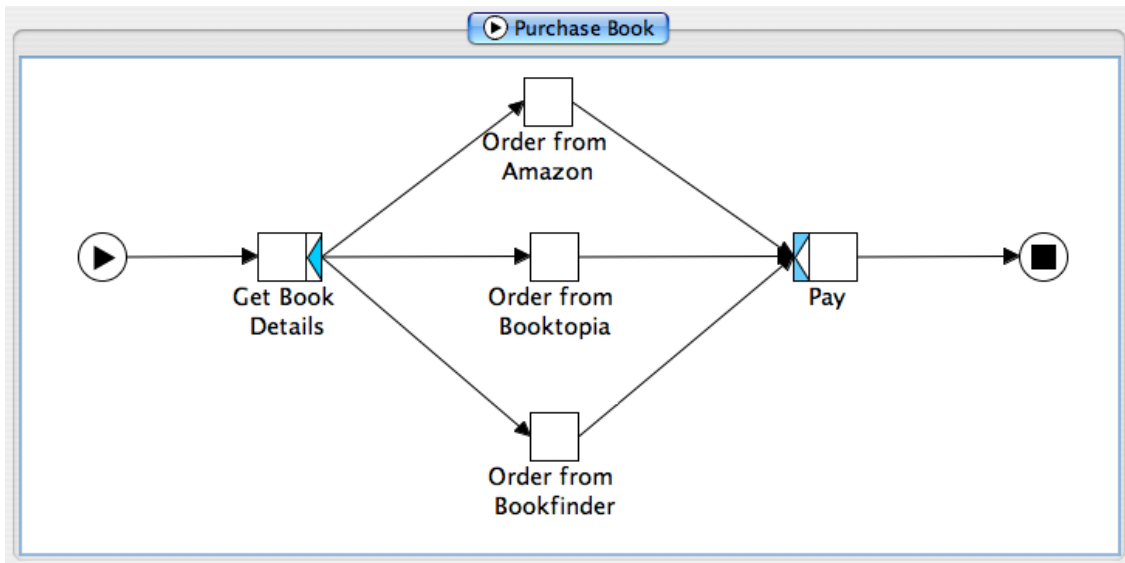

Figure 4.19: The Purchase Book specification

   Right-click on "Order from Amazon" and choose **<u>V</u>iew Cancellation Set**.

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, [icon], on the Menu Toolbar. Items will be given a red border to indicate they belong to the cancellation set of the grey-filled task (see Figure 4.20).

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

5. Once you have established the cancellation set, you can right-click on the cancellable task and reselect **View Cancellation Set** to toggle off the cancellation set view.
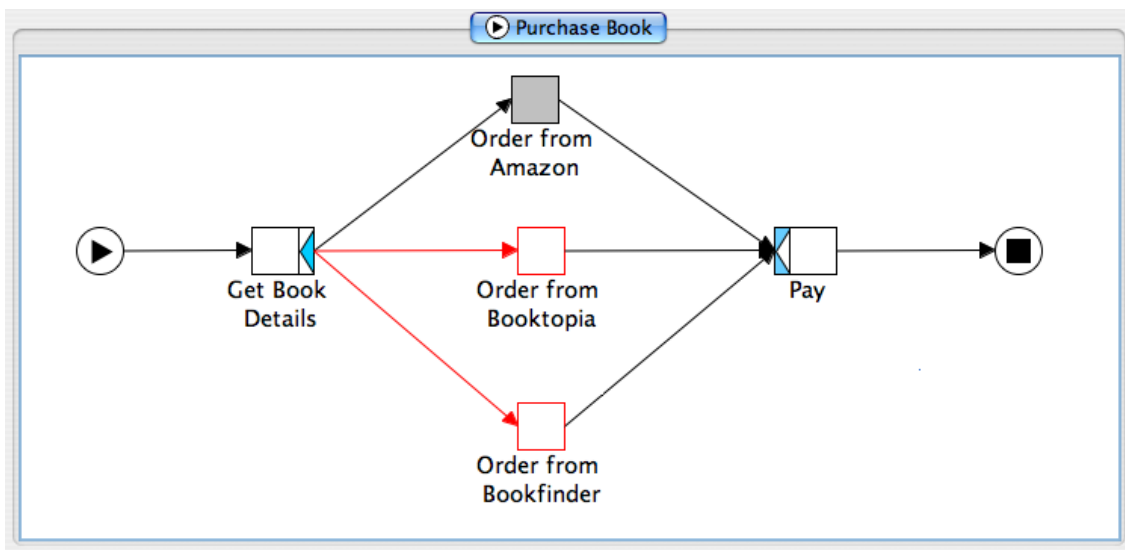

Figure 4.20: A Cancellation Set specified for the 'Order from Amazon' task

Notes about Cancellation Sets:

- In the example in Figure 4.20, notice that there is an AND-split decorating the "Get Book Details" task, but the "Pay" task has an XOR-join. This is because we know that when one of the "Order" tasks completes, the other two will be cancelled, so 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 the *deadlock* of the workflow instance.

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

- 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 4.16), 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 or not.

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

1. First, make sure you have the **View Cancellation Set** option ticked for the task. If it isn't ticked, select the task that has the Cancellation Set, right-click, then choose **View Cancellation Set**.

2. Select the element for removal.

3. Click on the Remove Selected Items from Visible Cancellation Set button, [icon], on the Menu Toolbar.

4.5.2   Data Type Definitions

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.

User-defined data types are also supported, by allowing for the definition of XML Schema complex types, which are added to a specification and then may be used to define variables based on those types. To define a new complex type for a specification:

1. Select **Update Data Type Definitions** from the **Specification** Menu.

2. Enter your XML Schema Data Type Definition into the dialog box. (See Figure 4.21).

3. If the definition text is green, your new data type is a valid definition and may be used for defining Net or Task variables in your specification. If the text is red, there is something wrong with your data type definition, and the data type will not be available. When the text is red, the split-pane will reveal a table listing parse errors that were collected when determining the validity of the text supplied. An example of this is shown in Figure 4.22.

Open the Data Type Definitions dialog and type in the XML text that appears in Figure 4.21.

The above example 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 4.23, the new data type "Geek" is available to choose from the list of available types when creating a task or net variable. 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, parse errors will be listed when the initial value text is red.

4.5.3   Net Decomposition Detail / Net 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.
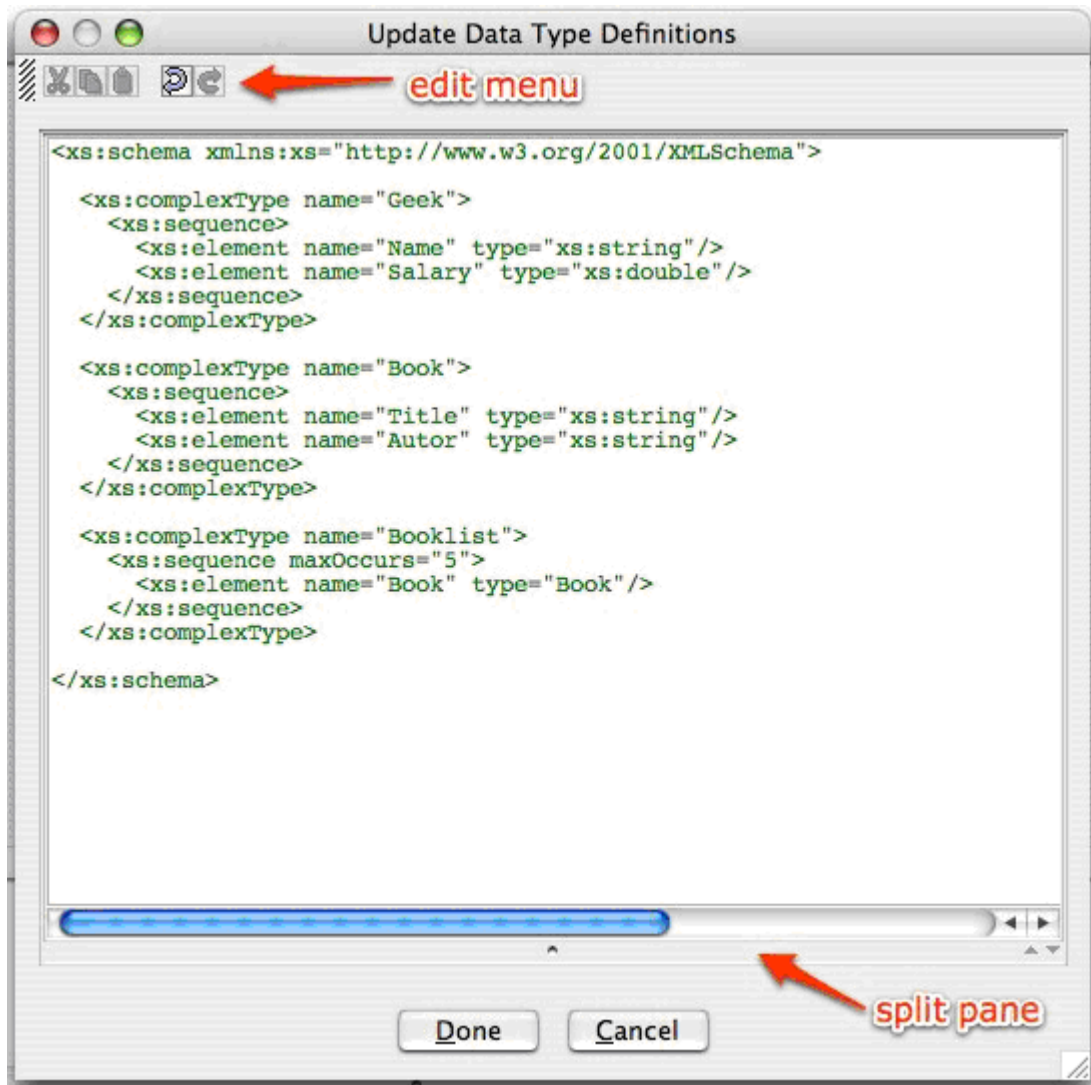
To add a variable to a Net:

Figure 4.21: Adding the "Geek" complex data type

1. Choose **Up̲date Net Detail** from the **Net** Menu.

   We will be setting up Net variables in the "Attend University" net. Go to the "Attend University Net" and choose **Up̲date Net Detail** from the **Net** Menu.

2. An Update Net Decomposition dialog box will appear (see Figure 4.24). Click on the **Create** button. This will show an Update Net Variable dialog box (see Figure 4.25).

3. Enter the Name of your variable, choose the Type and intended Usage of the variable from those listed, then click **Done**, then **Done** again to close the Net Decomposition dialog.

   Enter "StudentNumber" for the name of the variable, leave the type as "string", and set the Usage type to "Local". Leave the initial value blank. Click **Done** (see Figure 4.25). Create another Net Variable with the name "SubjectCode" and Type "string". Leave the Initial Value blank and set the Usage to "Local" (usage types will be explained a little later). Click **Done**.

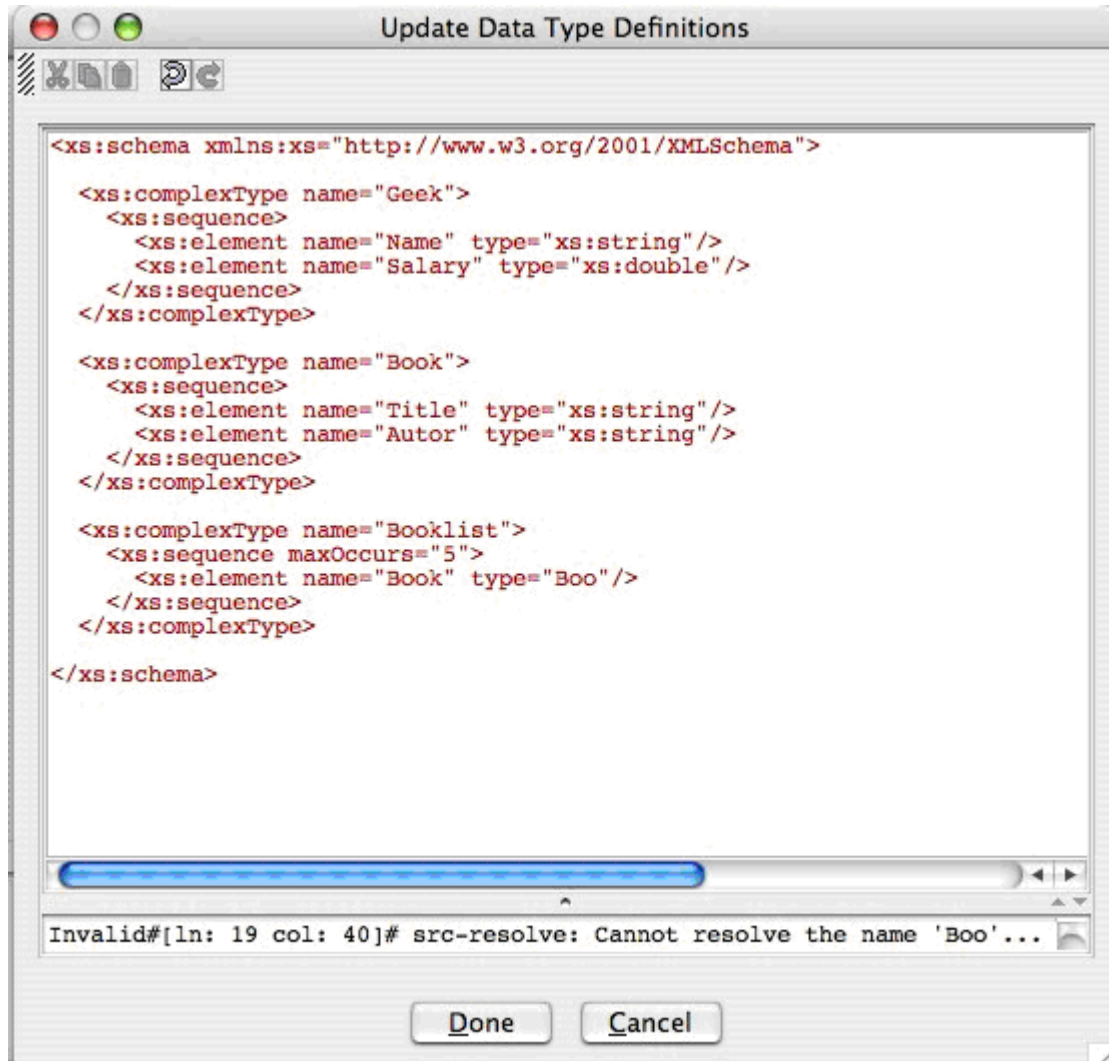4. The Net Variables should now appear in the Update Net Decomposition of Net "Attend University"

Figure 4.22: When the data type definition is invalid

dialog box (Figure 4.26).

4.5.4   Task Decomposition

By choosing the **Select Task Decomposition. . .** option when you right click on a task, you have the ability to identify which decomposition is attached to the task. A decomposition describes the variables 'handled' by the task, and the YAWL Service that will be responsible for performing the work the task represents at runtime.

Like nets, tasks have decompositions where you can specify variables and a label to associate with the task. Unlike nets, which cannot share net decompositions, 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 label, task decompositions also allow the workflow designer to identify which web service the decomposition should invoke in a running workflow engine, 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
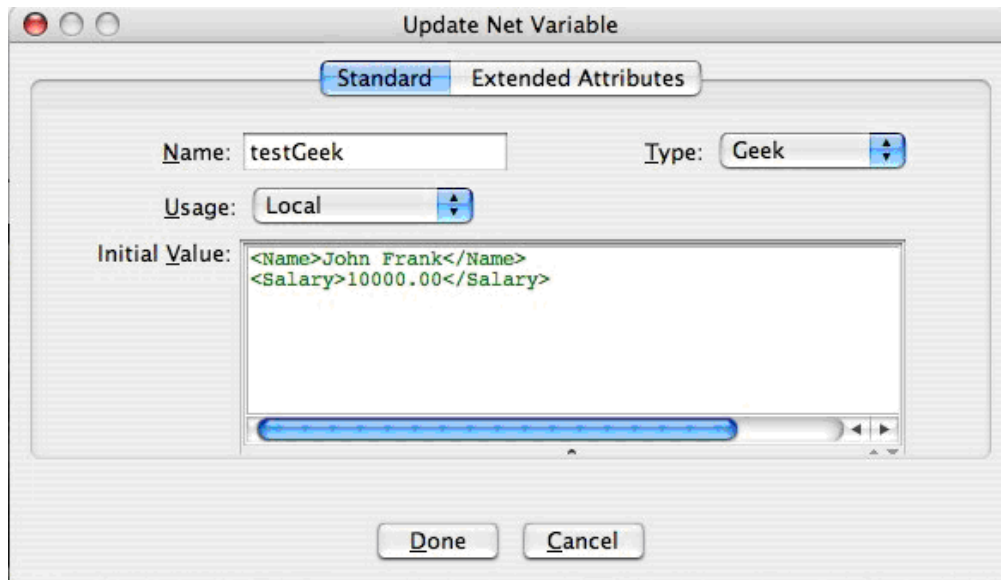
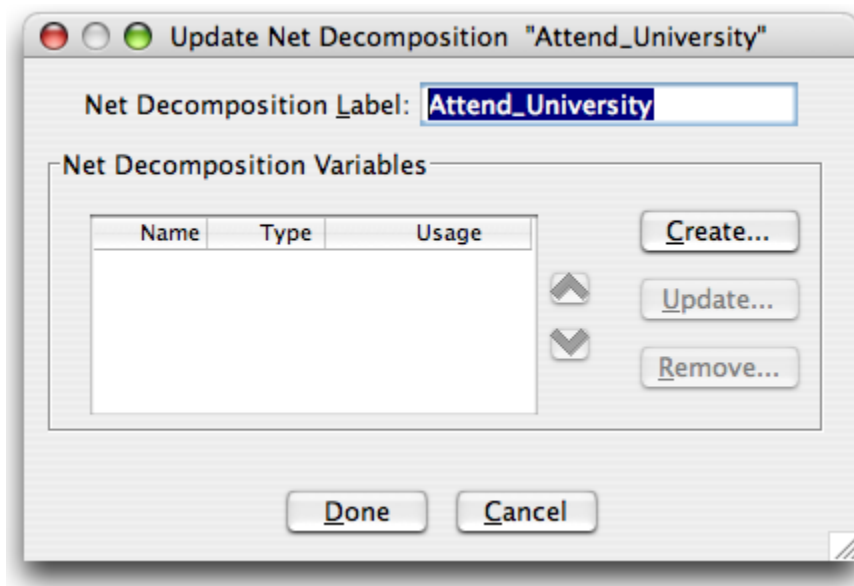**Figure 4.23:** A "Geek" net variable with a valid initial value

Figure 4.24: Updating "Attend University" Net Variables

(the two tasks may be named the same or differently, but they will share the same underlying definition of work).

From the Select Task Decomposition dialog, you can use the drop-down list to select an existing decomposition, or alternately you can click the **Create...** button to generate a new one that will become the task's decomposition (Figure 4.27).

4.5.5   Task Decomposition Detail / Task Variables

You can add variables to a task to store specific information relating to that task, in a similar way to adding variables to a net. Task variables have several uses. One use is for transferring information between workflow users and the workflow engine. A second use is for passing data between web services and/or external code
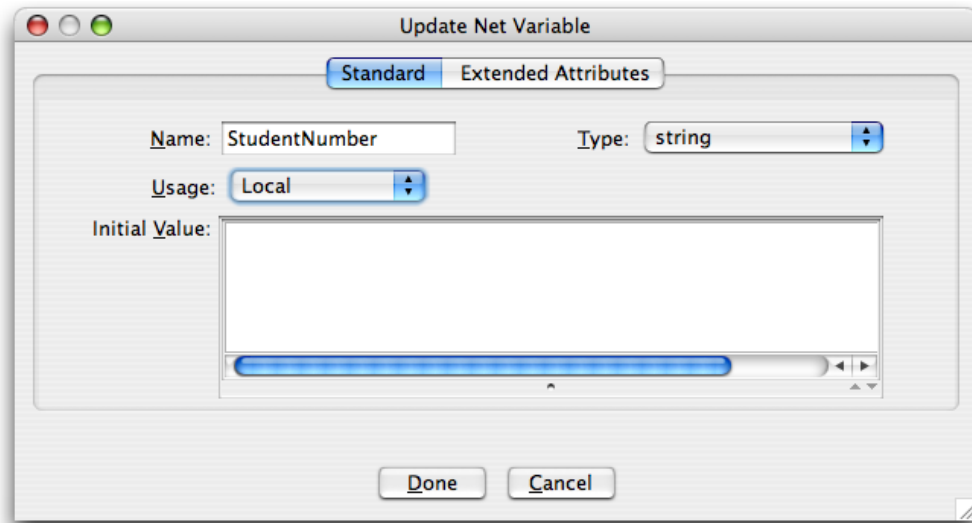
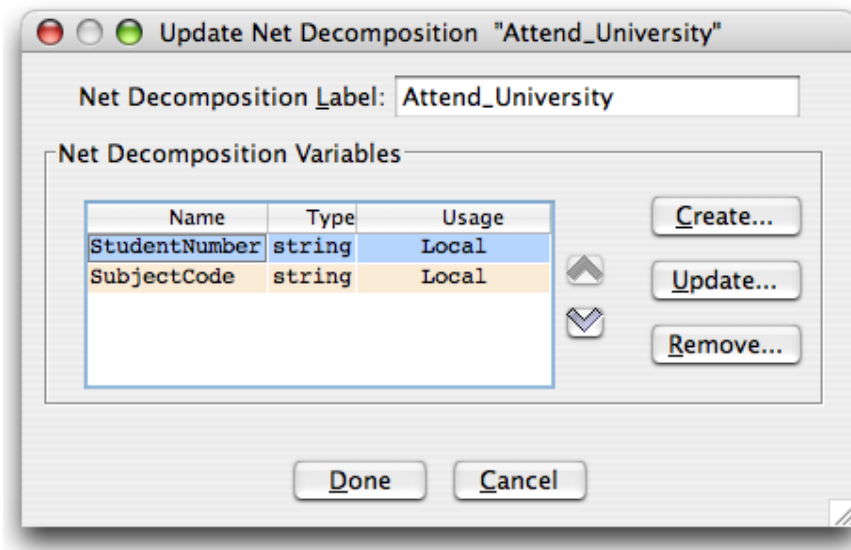**Figure 4.25:** The Net Variable "Student Number"

Figure 4.26: Updated "Attend University" Net Variables

and/or applications that the running workflow 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.

4.5.6   Adding a Variable to a Task

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.

2. Right-click on the task and choose **Task Decomposition Detail. . .** . An Update Task Decomposition
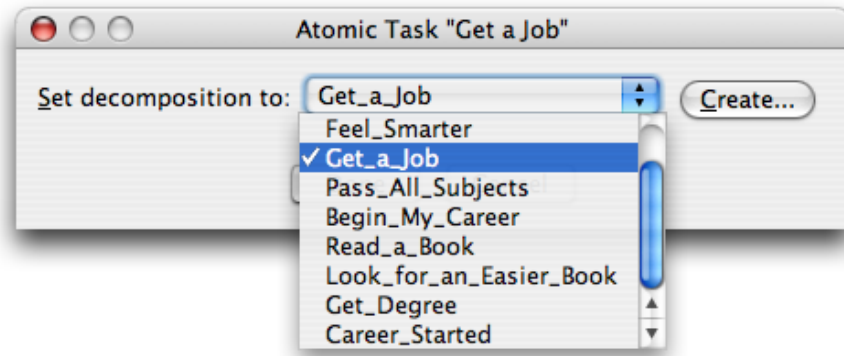
Figure 4.27: Select Task Decomposition (example 'Get a Job' task)

dialog box will appear.

Retrieve the decomposition detail for the "Enrol" task.

3. Click on the **Create...** button. An Update Task Variable dialog box will appear.
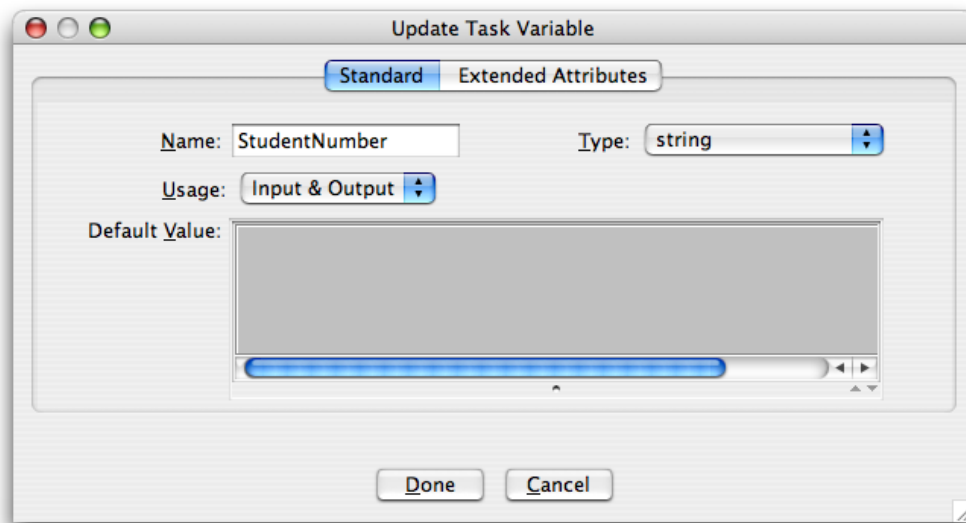
Figure 4.28: Updating the Task Variable

4. Enter the Name of your variable, choose the Type of the variable and its Usage from those listed, click **Done**, then **Done** again to exit the task decomposition detail dialog.

Enter "StudentNumber" for the name of the variable, leave the type as **string**, and set the Usage to "Input & Output". (Figure 4.28). Click **Done**. Create another variable for the same task, called "SubjectCode" with type **string** and usage of "Input & Output". Click **Done**.

5. The "Enrol" task now has two variables, "StudentNumber" and "SubjectCode" (Figure 4.29).

Figure 4.29: The Update Task Decomposition dialog for the Enrol task

4.5.7   Task Parameters

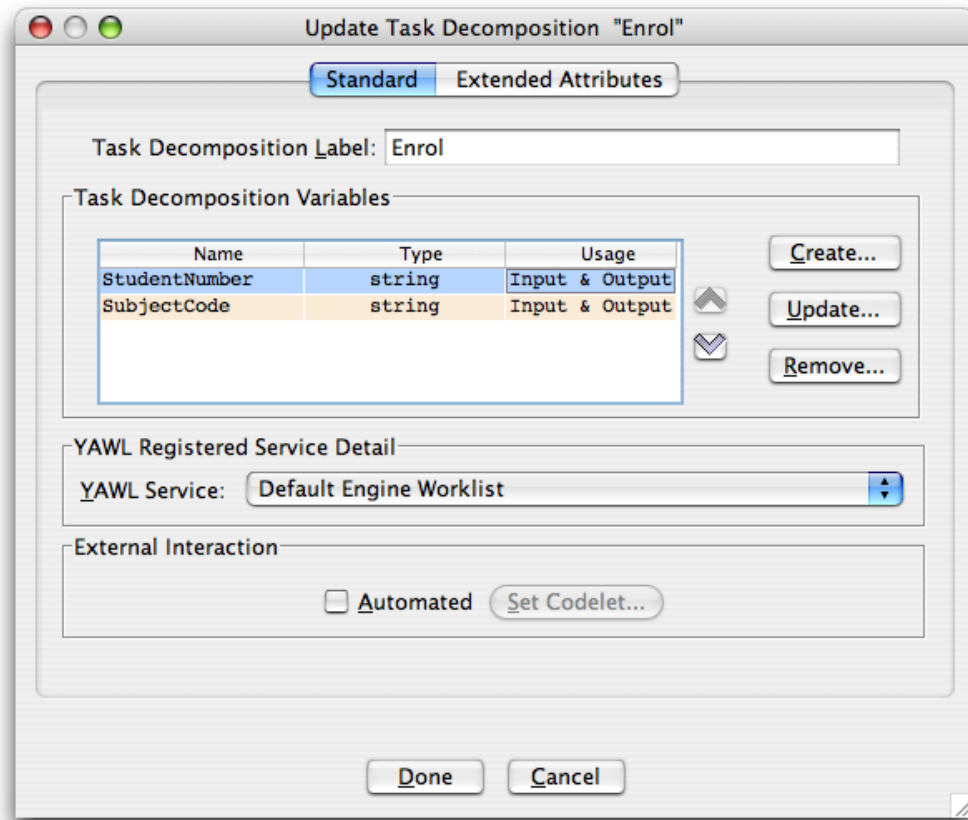A *parameter* defines how a value is assigned to a variable, and how a value is passed between net-level and task-level variables and vice-versa. Both Input and Output Parameters can be assigned to any tasks (depending on their usage type) to allow the passing of state between nets and their tasks, and between tasks and workflow engine, users and web services.

Parameters are defined using *XQuery* expressions[2]. Input Parameters use an XQuery expression to specify a value (possibly drawing on a number of static and/or net-level variable values) that can be passed to a single selected task variable. Output parameters use an XQuery expression to specify a value that can be passed to a single selected net variable.

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

To add an Input Parameter:

1. Select the task to add the parameter to.

   We will be setting up Input Parameters for the variables that we created in the Adding / Updating Task Variables section previously. Go to the "Attend University" Net and select the "Enrol" task.

2. Right-click on the task and choose **Update Parameters Mappings. . . .** An Update Parameter dialog

---

[2]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/
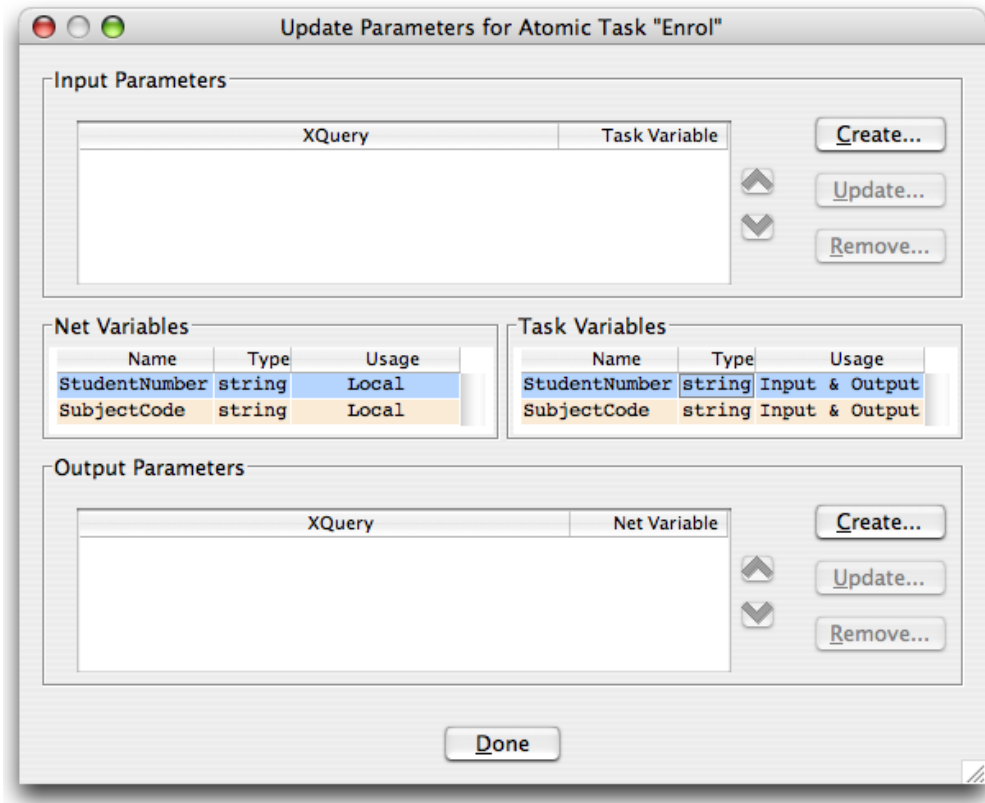
Figure 4.30: Updating Parameters for a Task

box will appear (Figure 4.30).

Update the Parameters for the "Enrol" task. Notice that the dialog in Figure 4.30 lists both this task's variables, and the variables of its containing net.

3. In the Input Parameters section, click on **Create...**. An Update Task Parameter dialog box will appear. If you have already set up a Task Variable for this task, then the Existing Task Variable option will be activated and there will be a list of task variables to choose from. Choose a variable from the list and click on **Done**, then **Done** again to close the task parameters dialog.

   If you haven't set up Task Variables, then click on **Create...** and return to the previous section dealing with Task Variables (Section 4.5.7).

   If you are familiar with XQuery syntax, then you can add an XQuery expression to allow manipulation of the Input Parameters. "Syntactically well formed" XQueries will be green, and badly formed ones will be red. Again, red text will be accompanied by a split-pane table, returning the parse errors that cause the text to be badly formed.

   For workflow designer convenience, two XQuery buttons are supplied to automatically generate XQuery expressions from available net variables.

   The **add XQuery of element's content** button will return just the content of the XML element for this variable, which is useful for simple value transfer between two variables of the same XML Schema type, and is expected will be the button used in most cases.

   The other button, **add XQuery of entire element**, will return the entire XML element of the selected variable, which is useful for times when you want to create a complex type expression from individual variable elements. Experience with XMLSchema and XQuery are necessary to understand the effects this button will have on runtime YAWL engine state.

Select the "StudentNumber" variable from the list of Existing Task Variables. Select the "Student-Number" net variable, then click **add XQuery of element's content** (Figure 4.31). Click **Done**. Create another Task Parameter and map the net "SubjectCode" to the task variable of the same name using this technique. Click **Done** and **Done** again.
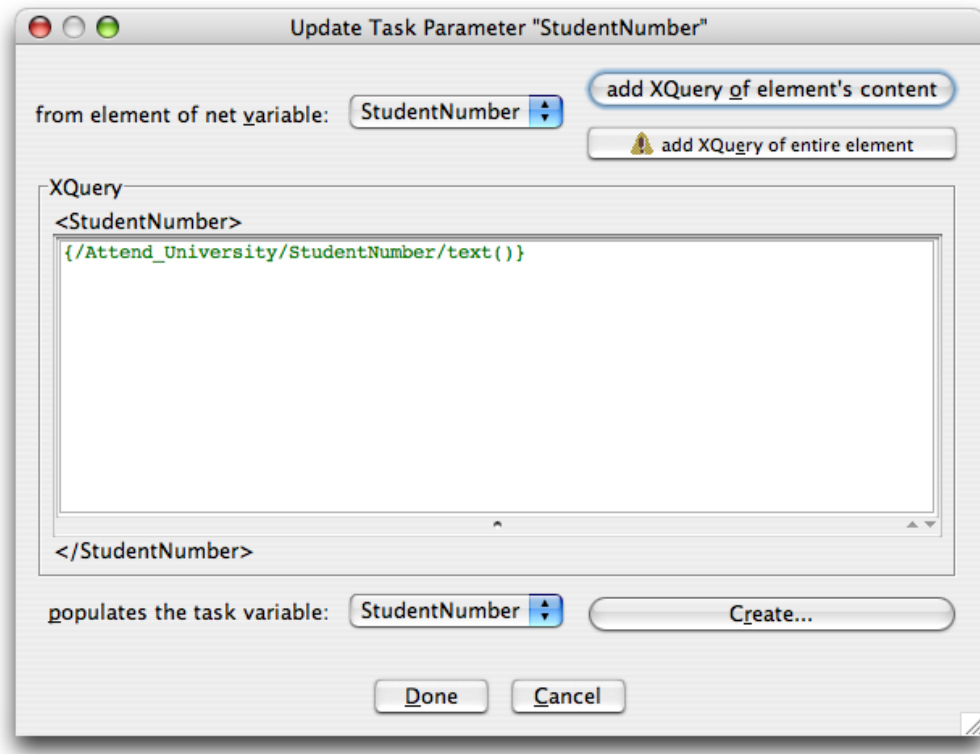

Figure 4.31: Passing a net's StudentNumber value to a Task variable

To add an Output Parameter:

1. First select the task to add the parameter to.

   We will be setting up Output Parameters for the variables that we created in the Adding / Updating Task Variables section. Go to the "Attend University" Net and select the "Enrol" task.

2. Right-click on the task and choose **Update Parameters...**. An Update Parameters dialog box will appear (Figure 4.31).

   Update the Parameters for the "Enrol" task.

3. In the Output Parameters section, click on **Create...**. An Update Net Parameter dialog box will appear (similar to Figure 4.31).

4. If you have already set up a Task Variable for this task, then the Existing Task Variable option will be activated and there will be a list of task variables to choose from. Choose a variable from the list and click on **Done**, then **Done** again to close the task parameters dialog. If you haven't set up Net

Variables, then click on **Create. . .** and return to the previous section (Section 4.5.3).

If you are familiar with XQuery syntax, then you can paste in an XQuery to allow manipulation of the Output Parameters. "Syntactically well formed" XQueries will be green, and badly formed ones will be red. For workflow designer convenience, two XQuery buttons are supplied to generate XQuery expressions from available task variables.

The **add XQuery of element's content** button will return just the content of the XML element for this variable, which is useful for simple state transfer between two variables of the same XML Schema type, and expected to be the typical button that users will start out with.

The other button, **add XQuery of entire element**, will return the entire XML element of the selected variable, which is useful for times when you want to create a complex type expression from individual variable elements. Experience with XMSchema and XQuery are necessary to understand the effects this button will have on runtime YAWL engine state.

From the list of task variables, select the "SubjectCode" task variable and click **add XQuery of element's content**. From the list of net variables, select the "SubjectCode" variable. Click **Done**. Create another Task Parameter and map the task "StudentNumber" to the net variable of the same name using this technique. Click **Done** and **Done** again.

These Parameters were set up to demonstrate a simple transfer of state from a net to a task and back to the net. Perhaps the task would allow a user to change the values of one of the variables which would eventuate in the net's values changing.

The Update Parameters dialog box should appear as in Figure 4.32.

Notes about parameters:

- For simple assignments, such as those in Figure 4.32, the XQuery expressions for the input parameters follow the form {/name_of_net/name_of_variable/text()} and are mapped to a task variable, while those for output parameters follow the form {/name_of_task/name_of_variable/text()} and are mapped to a net variable.

- An *Input* Usage mode means that the variable requires a value to be mapped into it when its task starts (via an input parameter). An *Output* Usage mode means that the variable is required to map a value from it (typically to a net-level variable) when its task completes (via an output parameter). An *Input & Output* Usage mode combines both requirements.

- Only Net-level variables may have a Usage mode of *Local*, which signifies a scope within the net but not external to it. Thus, sub-nets require net-level variables with modes other than *Local* to support data passing to and from their parent nets. A root (or top-level) net with variables of type *Input Only* or *Input & Output* 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 modes set for root net variables.

- Input parameters may only be created for variables of mode types *Input Only* or *Input & Output*.

- Output parameters may only be created for variables of mode types *Output Only* or *Input & Output*.

4.5.8   Flow Detail

When dealing with tasks that have XOR and OR splits, we need some way of defining which flows 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.
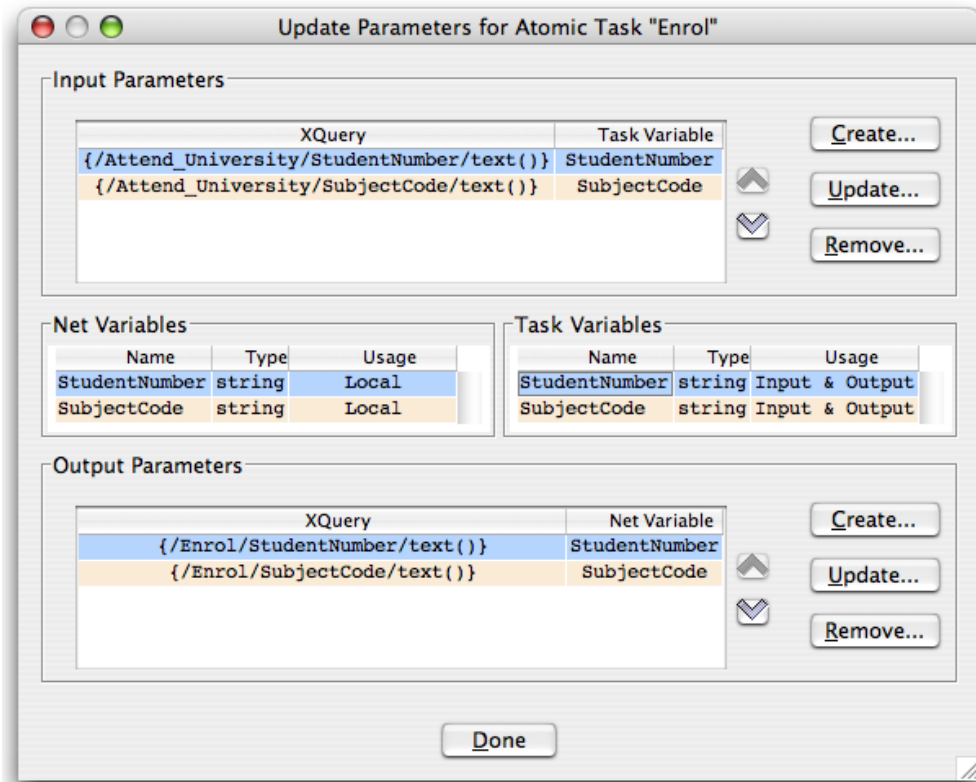
Figure 4.32: Established task parameters

- 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 evaluate to false, 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 flow detail of a task that has a split, right click on the task and choose **Update Flow Detail. . .**. The "Flow Detail" dialog appears, which list the flows coming out of the split and each flow's corresponding Predicate (or flow expression).

The arrowed buttons to the right of the list allow you to reorder the evaluation sequence of the predicates, so that the default predicate (the one you want to activate 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 any following flows will be ignored.

The currently selected flow in the dialog will be identified by being highlighted green in the Net (Figure 4.33).

To specify a predicate for a particular flow, select the flow from the list and click on **Predicate. . .**. Enter a predicate as a boolean XQuery expression and choose **Done**.
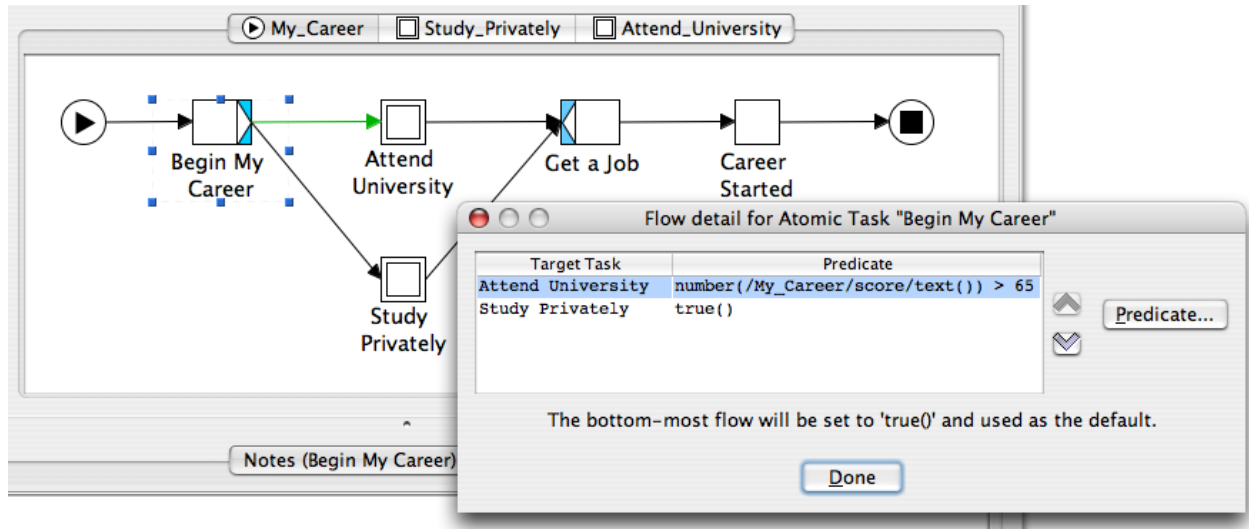
Figure 4.33: Updating Flow Detail

4.5.9 Multiple Instance Queries

Now that we have an understanding of parameter setting and XQueries, we can revisit, from a data perspective, parameter setting for the particular requirements of Multiple Instance (MI) Tasks. In general terms, an MI task receives, as input from a net-level variable mapping, a variable of complex type, typically a list of some other data type (but more complex constructions are of course supported too), which it then *splits* into a number of logically distinct data values, to form the data that is assigned to each task instance. When the MI task completes, it gathers all the individual pieces of data from the various task instances and reconstructs the complex type variable so that it can be mapped back to the net-level variable.

To illustrate the operation of MI tasks, with particular emphasis on the data perspective, we will use the "List Builder" specification shown in Figure 4.34, 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.
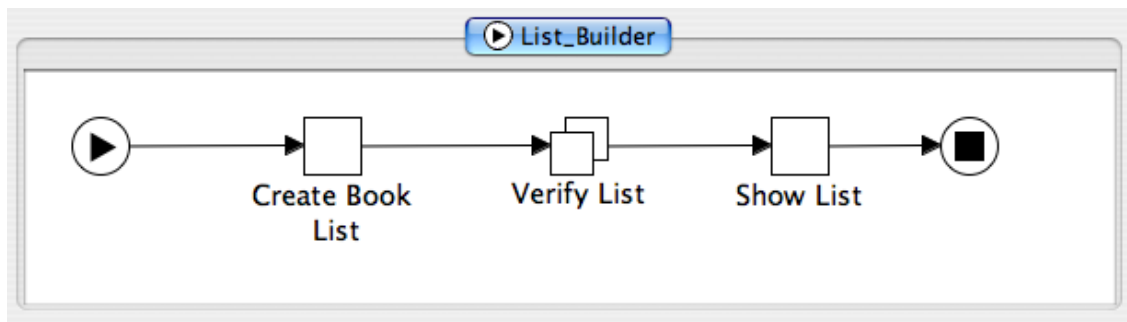

Figure 4.34: Example specification with a Multiple Instance Task

To prepare this specification, drag two atomic tasks and one MI task onto the canvas. We first need to define a complex data type to store the entire book order. Open the Update Data Type Definitions dialog from the Specification menu, then enter the following two 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 variable called **MasterList** of type **BookList** and usage **Local** (Tip: In the Update Variable dialog 'Type' dropdown, user-defined types are listed after all the built-in simple types). Give the variable this initial value:

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

  When entered correctly, the text will become green to show that it is a valid value to assign to the Masterlist variable of complex type 'BookList', since it defines values for the elements of one BookOrder. 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 will reject the specification.

- Add a decomposition to the first atomic task, and call it 'Create Book List'. Add to the decomposition a variable called **bookList** of type **BookList** and usage **Input & Output**. Then, open the Update Parameter Mappings dialog, add an input parameter mapping the net-level Masterlist variable to the task's bookList variable, and an output parameter to map it back again (see Figure 4.35). 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 the usage should be **Input Only**, and so only an input parameter mapping is required. 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').

- Add a decomposition to the MI task and call it 'Verify List'. To the decomposition, add a variable called **book** of type **BookOrder** and usage **Input & Output**. Note carefully that the task variable we have added to the decomposition is of BookOrder type – in our data definition we have defined the BookList type being comprised of a number of BookOrder type elements – so what has been defined in this decomposition is a mapping of a single BookOrder to each task instance that will be created when the MI task is executed.

- The mapping of input and output parameters for MI tasks is done a little differently to atomic (single instance) tasks, and involves a two stage process. Open the Update Parameter Mapping dialog for the MI task 'Verify List' and add the input and output parameter mappings as seen in Figure 4.36. Notice that the input parameter maps the entire net-level variable 'MasterList' to 'book', a single task-level BookOrder variable, while the output parameter maps the contents of the 'book' variable, wrapped in 'order' tags, back to the net-level 'MasterList'. These kinds of mappings would be invalid for a single-instance task, but here we need to define a mapping that on input takes a net-level list, assigns
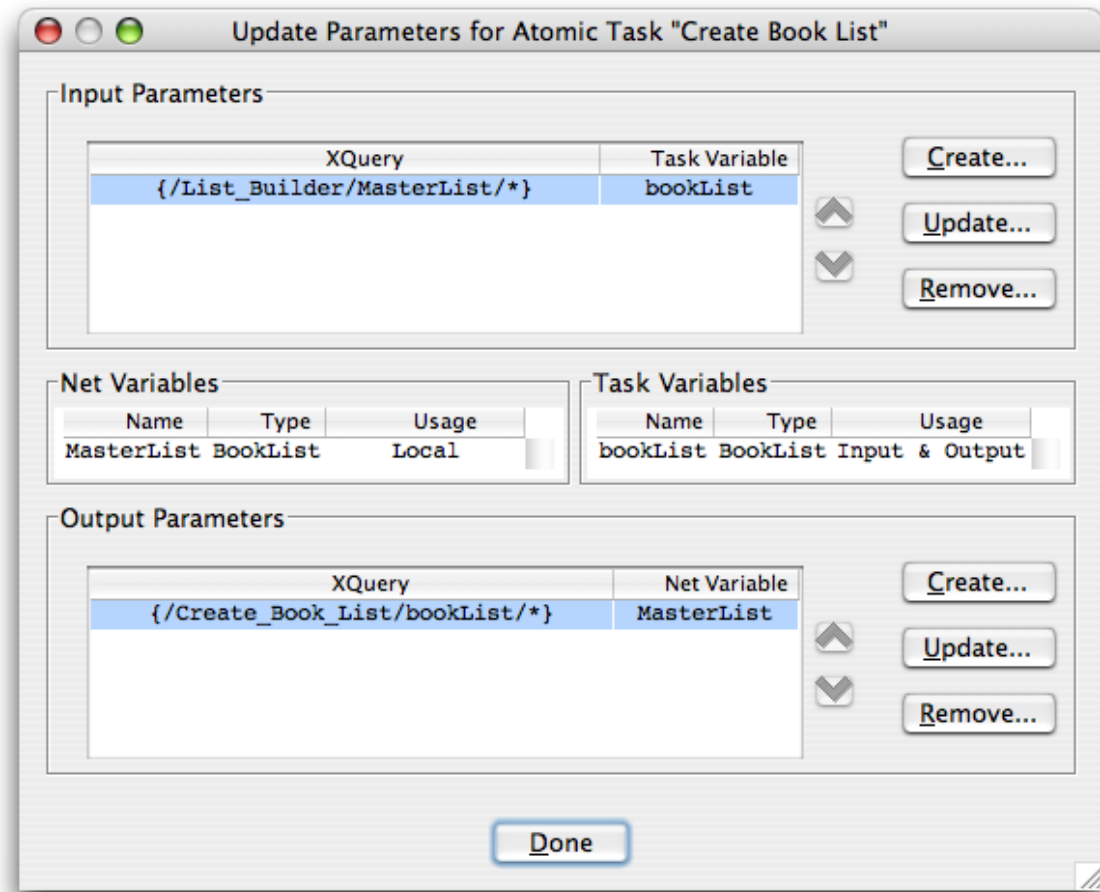
Figure 4.35: Update Parameter Mapping dialog for task 'Create Book List'

each member of the list to a task instance, and on output takes each task instance's variable and maps it back to the net-level list. The intermediary steps that allows this mapping to occur is done in the second stage of the mapping process.

- Right click on the 'Verify List' MI task and choose **Set Instance Detail. . .** from the popup menu. On the **Bounds** tab page, set minimum instance to 1, maximum instances to 20, continuation 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, the MI task will complete when 5 instances complete (or all complete if less than 5 were started) and new instances may not be dynamically started after task execution begins.

Now, click the **Queries** tab page. It is here that we will specify how the BookOrders contained in the MasterList variable will be 'split' into individual task variable instances and how those instance variables are 'aggregated' back into the MasterList on completion. Figure 4.37 shows the Queries tab page for the 'Verify List' MI task. There are six parts to this dialog, four requiring XQueries and two variable settings:

  - **Multiple Instance Variable**: this specifies the task instance variable that has be defined to accept a single item from the net-level 'list' variable – in our case, the 'book' variable.

  - **Accessor Query**: this query defines the source of the data that needs to be split into multiple task instances, in most cases (as in our example) it is a simple XPath expression specifying the net-level variable that is being mapped into the MI task, and exactly matches the input parameter set
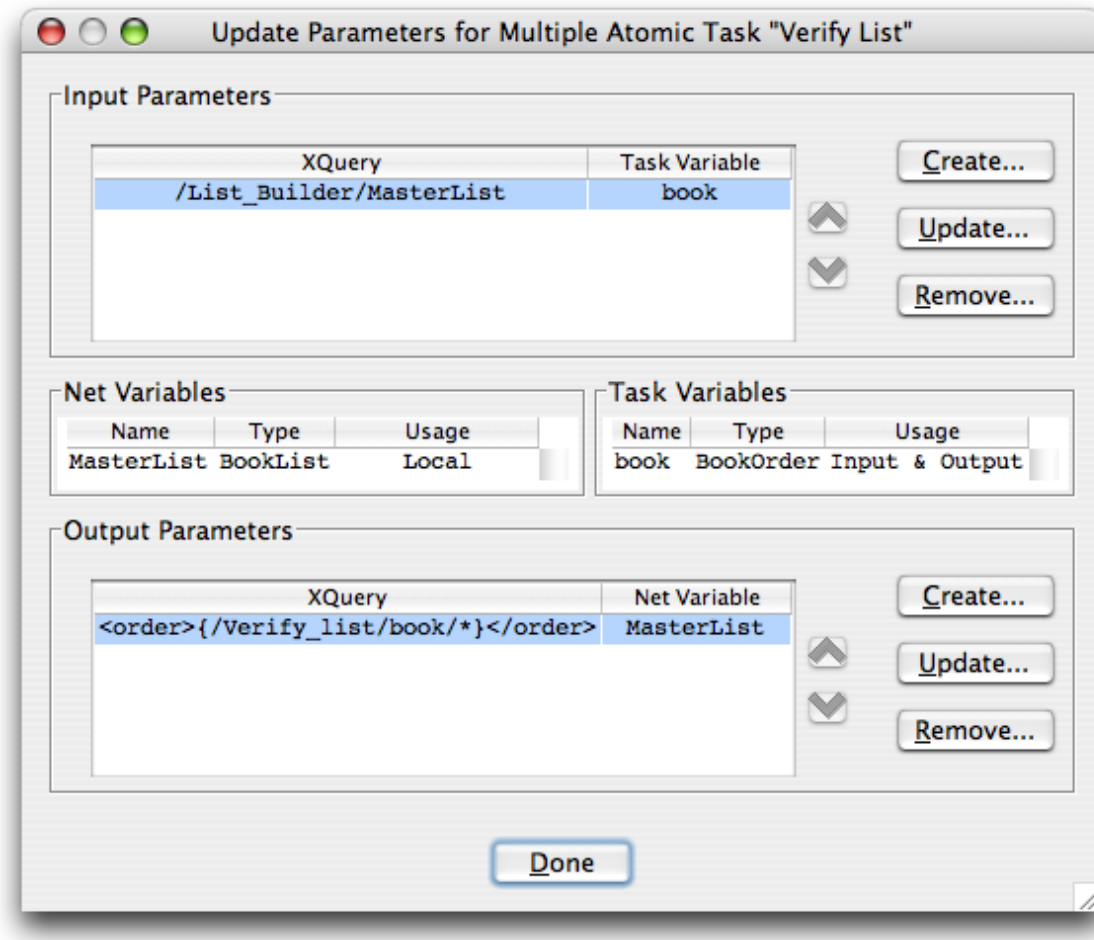
Figure 4.36: Update Parameter Mapping dialog for MI task 'Verify List'

for the variable in the Update Parameters dialog. In this case, it is /List_Builder/Masterlist –
note that it is the same XQuery expression as the task's input parameter as shown in Figure 4.36.

– **Splitter Query**: the splitter 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 element in the accessor variable, return that element".
In our example, the XQuery is: `for $e in /MasterList/* return <book>{$e/*}</book>`
The $e part is a query variable – all query variables start with a $ sign followed by one or more
characters ($e is a convention for a loop variable). The return value for each $e in our query would
start and end with 'order' tags; since our variable is called 'book', we have to replace the 'order'
tags with 'book' tags, so the return value of our XQuery is the *contents* of the order (`{$e/*}`),
but surrounded by 'book' tags. The final result is a number of 'book' values, each corresponding
to one 'order' element in MasterList.

– **Instance Query**: this query defines how the 'book' variable within each task instance should be
formatted prior to being mapped back to the net-level MasterList. Like the splitter query above,
we need to replace the surrounding 'book' tags with 'order' tags, the contents of the element being
the *contents* of the 'book' variable ({/Verify_list/book/*}). Notice also that this query exactly
matches the task's output parameter in Figure 4.36.

– **Aggregate 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
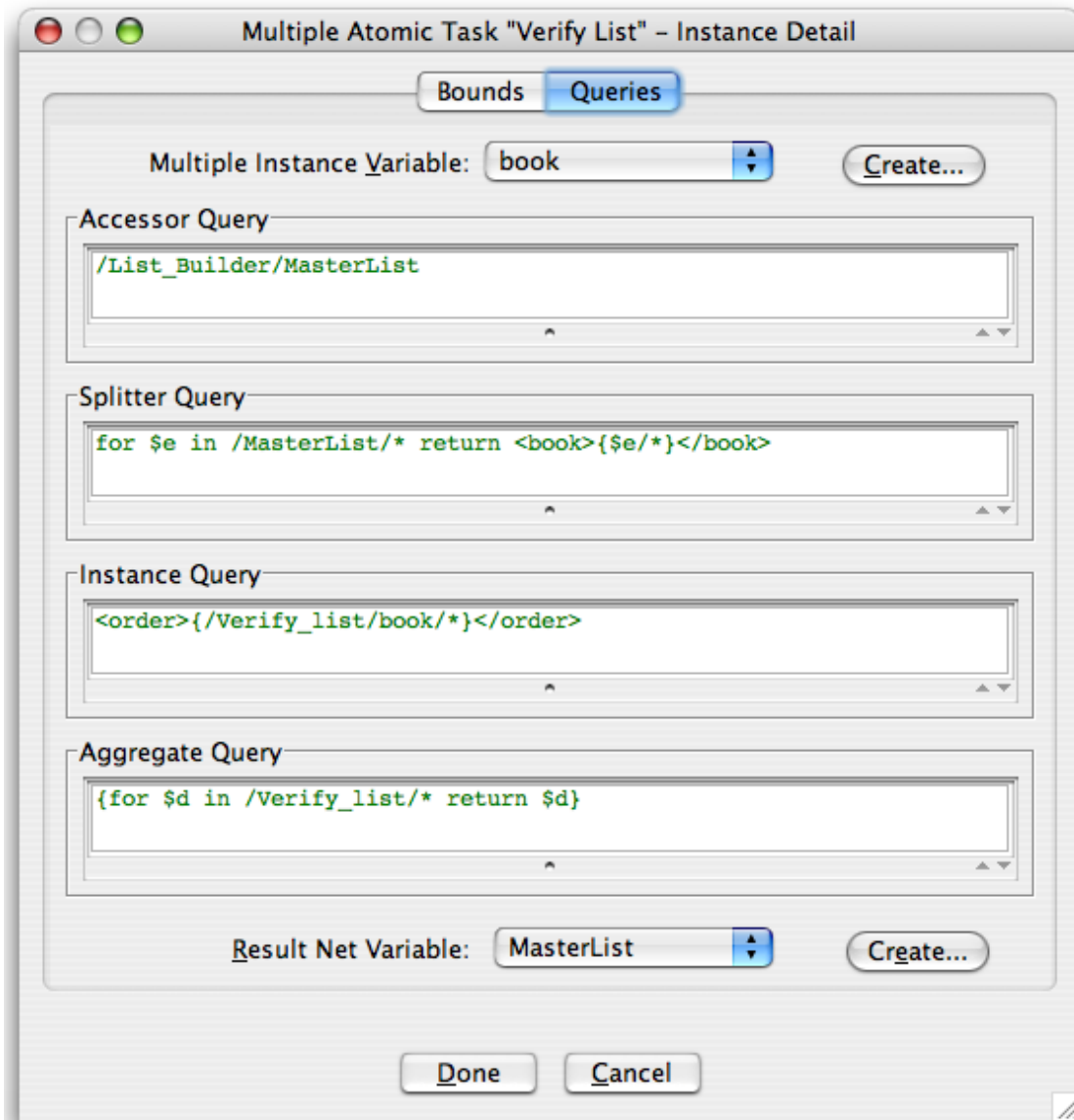like this in most cases ($d is used simply to differentiate it from $e in the splitter query, but its

Figure 4.37: Set Instance Detail dialog, Queries tab for MI task 'Verify List'

name is unimportant).

– **Result Net Variable**: this specifies the net-level variable to which the results of the aggregate query is to be mapped.

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

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.

4.5.10   Fast-Tracking Data Definition

If all of a task's input and output parameters will involve a simple mapping to and from net-level variables, a "fast-track" can be taken to do it all in a few mouse clicks. To achieve this, right-click on an atomic task that does not yet have a decomposition, and select **Decompose to Direct Data Transfer**. The dialog in Figure 4.38 will appear.
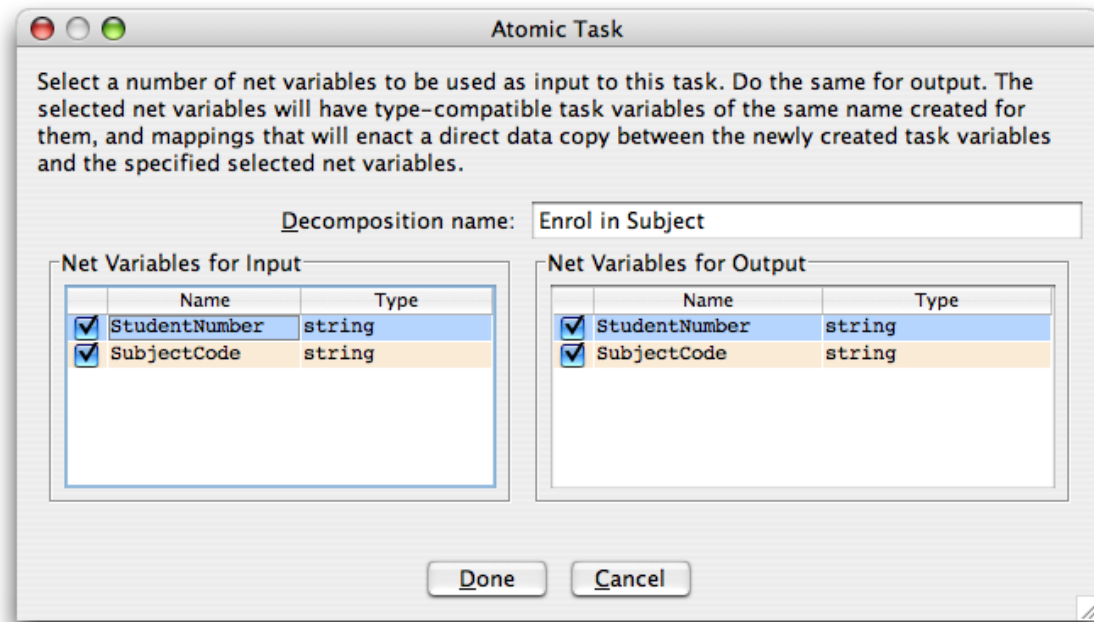

Figure 4.38: An example of the direct data transfer dialog

All of the available net-level variables for the task's containing net will be listed in the dialog. Simply check the boxes for the desired inputs and outputs. This dialog will automatically create a task decomposition and matching XQueries to directly transfer data from a selected net variable to a task variable of the same name and back again.

4.5.11   End of Scenario

This is where our scenario ends. The My Career Scenario was designed to outline the basic the functions of the YAWL Editor and to provide you with a understanding of designing a YAWL workflow specification.

But the functionality of the YAWL Editor does not stop here. If you are after something more challenging, try adjusting your version of the scenario to expand into more sub-Nets and more complex situations.

4.6   Connections

4.6.1   Connecting to the YAWL Engine

Each atomic task in a YAWL specification must be associated with a 'service' that will be responsible for performing the work of the task. If you do not explicitly specify a service for a task, by default the task will be assigned to the Resource Service to be displayed in its worklist handler. If you want to associate a task with a service other than the Resource Service, you can make the appropriate choice from a list in the task's decomposition dialog (see the following section). The Editor populates the list of available services by

making a call to a running Engine, which returns the list of services currently registered with it. Therefore, the Editor must first establish a connection to a running Engine.

On Editor startup, a connection to a running Engine is attempted. If successful, the Engine connection icon at the left of the status bar will turn green. If unsuccessful (perhaps because the Engine is not running, or the connection parameters are incorrect) the connection icon will show red. After the Editor has started, an Engine connection may be performed via the Tools menu.

To establish a connection with a running YAWL Engine:

1. Click the **Tools** Menu and choose the **Engine Connection Settings. . .** item.

2. From the resulting dialog (Figure 4.39), accept the default values or enter the following engine details:

   - YAWL Engine URI;
   - Administrator's User Name;
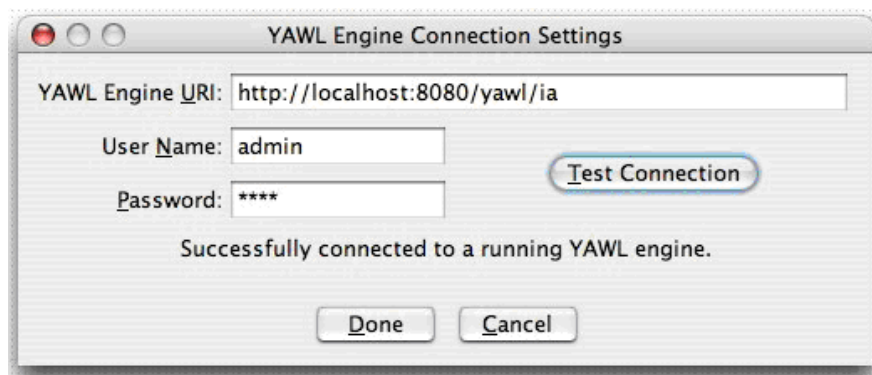   - Administrator's Password.

Figure 4.39: Specifying the YAWL Engine connection detail

The YAWL Engine URI value is set by default to a locally installed Engine (`http://localhost:8080/yawl/ia`). The User Name and Password are set to the default administrator user (name="admin", password="YAWL").

As a user convenience, a button called **Test Connection** is supplied, which will attempt to connect to a running engine with the detail supplied before the user commits to using those details for any further editor/engine interactions.

4.6.2 Connecting to the Resource Service

To use the organisational data managed by the Resource Service for assigning tasks to Participants or Roles, a connection between the editor and the Resource Service needs to be established.

Like the Engine connection (above), when the Editor starts a connection to a running Resource Service is attempted. If successful, the Resource Service connection icon at the left of the status bar will turn green. If unsuccessful (perhaps because the Service is not running, or the connection parameters are incorrect) the connection icon will show red. After the Editor has started, a Resource Service connection may be performed via the Tools menu.

To set the Resource Service connection details:

1. Click on the **Tools** menu and choose the **Resource Service Connection Settings. . .** item.

2. From the resulting dialog (Figure 4.40), accept the default values or enter the following engine details:

- YAWL Engine URI;
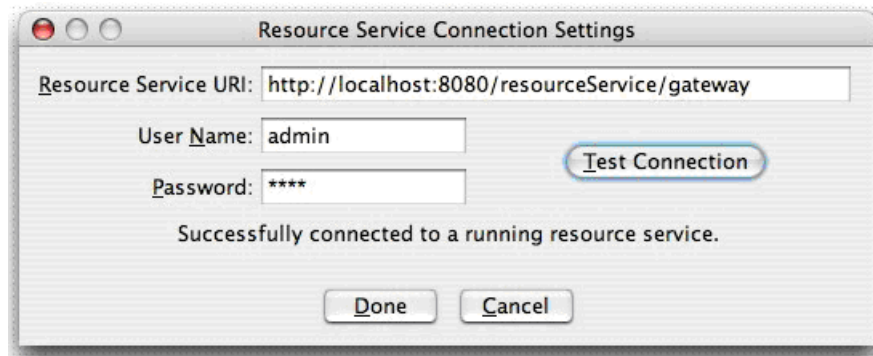- Administrator's User Name;
- Administrator's Password.

Figure 4.40: Specifying the Resource Service connection

The Resource Engine value is set by default to a locally installed Resource Service (http://localhost:8080/resourceService/gateway). The User Name and Passwords are set to the default administrator user (name="admin", password="YAWL").

4.6.3   Connecting a Decomposition to a registered YAWL Service

You can use task decompositions within your workflow to make a connection to custom YAWL services that have been registered with a running engine. By associating a task decomposition with a custom service, all task instances based on that decomposition will be passed to the 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, do the following:

1. Right-click on a task, the choose **Task Decomposition Detail. . .** . An Update Task Decomposition dialog box will appear (see Figure 4.29).

2. In the "YAWL Registered Service Detail" panel, Select the required Web Service from the dropdown list of those available in the Engine.

3. Click **Done** to finish.

If the Editor is connected to a valid running YAWL Engine instance, the YAWL Service Dropdown Box will contain entries for all custom YAWL services the engine has registered. Otherwise, only the Default Worklist (of the Resource Service) will be available.

When you select a YAWL Service, the Editor will query this 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.

4.7 Validating and Saving a Specification

At any stage you can validate and/or save your specification to a YAWL Engine file (.yawl).

To validate your specification:

1. Click the Validate button, ![validate button], on the Menu Toolbar or click **Specification** on the Menu and choose **Validate Specification...**.

2. If problems are detected, a table listing them will appear at the bottom of the Editor with details of any inconsistencies that would stop the specification from running in the YAWL Engine. Figure 4.41 show a specification with no validation problems. Figure 4.42 shows an example invalid specification, where the Validation Problems table shows that the 'Bad Task' task has no outgoing flow, and is not on a path between the input and output conditions.
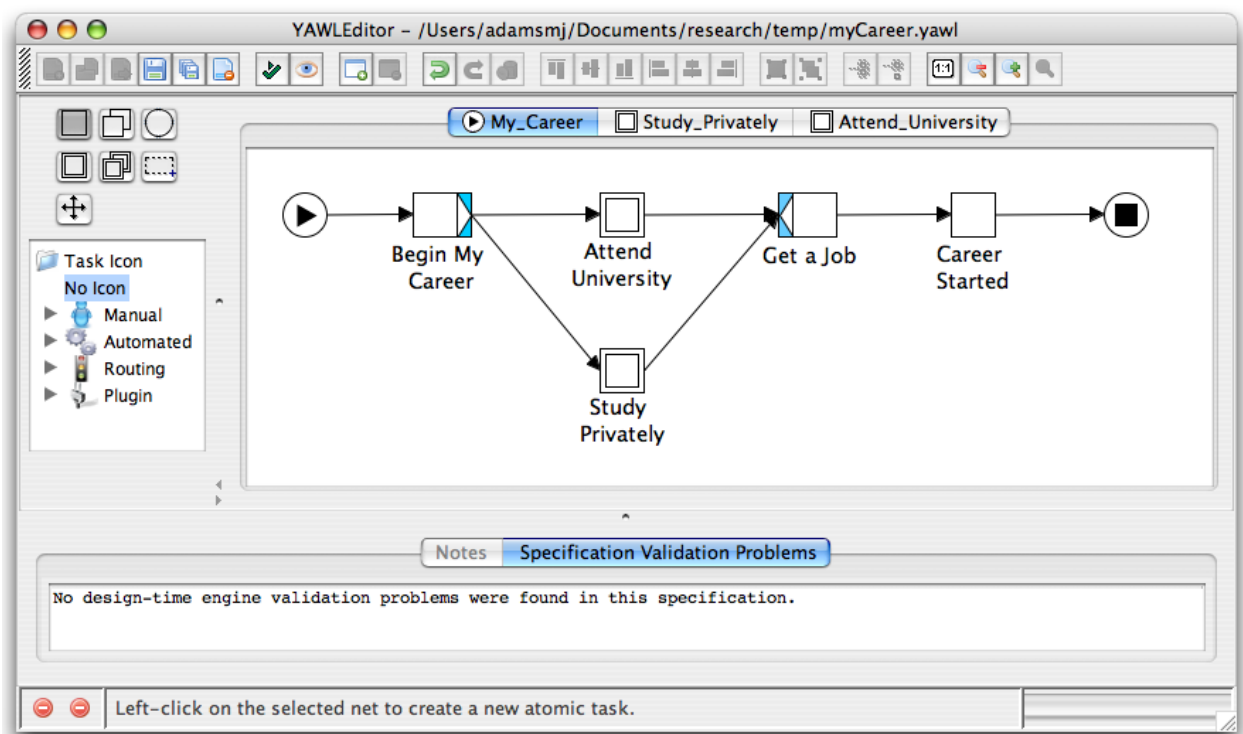

Figure 4.41: A valid specification

To save your specification to file:

1. Click on the Save button, ![save button], on the Menu Toolbar or click **Specification** on the Menu and choose **Save Specification...**. The 'Specification File Save Options' dialog appears (Figure 4.43).

   If this is the first time the specification has been saved, and a 'Specification ID' has not yet been entered via the 'Update Specification Settings' dialog, a 'Specification ID' field will appear in the dialog (all Specifications must be given a valid ID). The ID can be a combination of alphanumeric characters and the underscore, but cannot start with a numeric character. Once set, this field will not reappear in subsequent saves of the specification.

   The 'Version Number' field provides a version number for the specification. Version numbering allows different versions of the 'same' specification to co-exist in the Engine. YAWL specification version
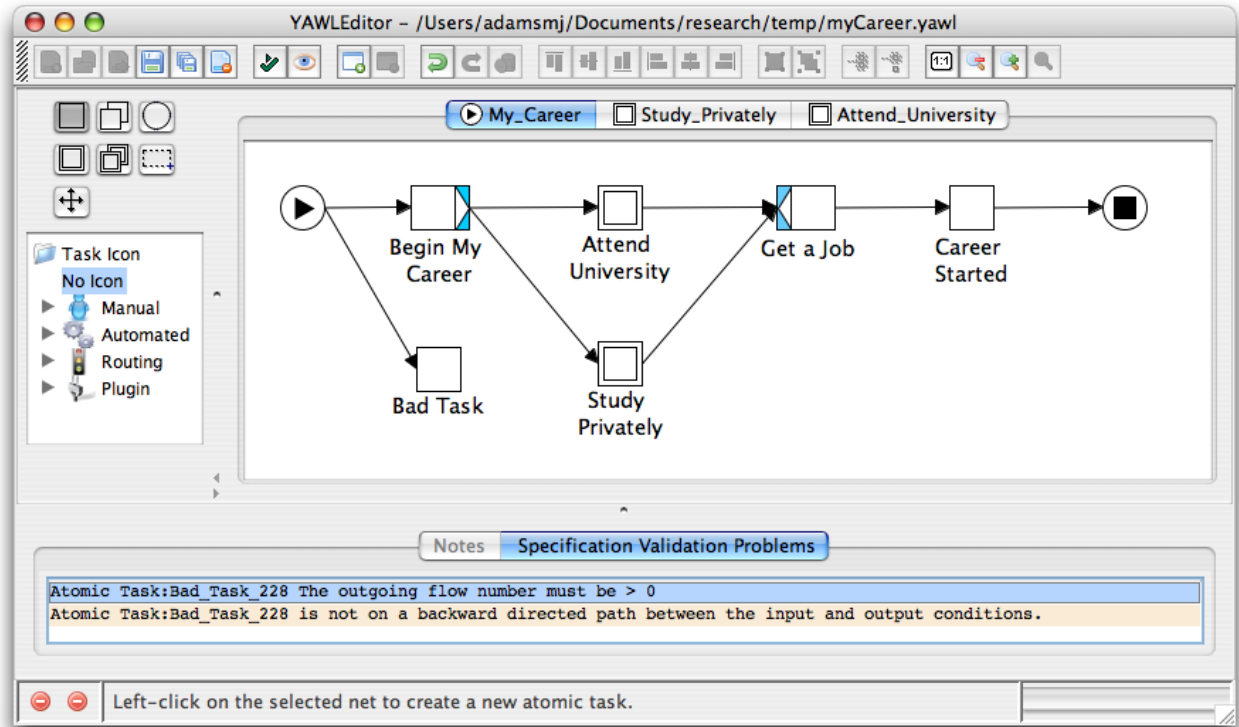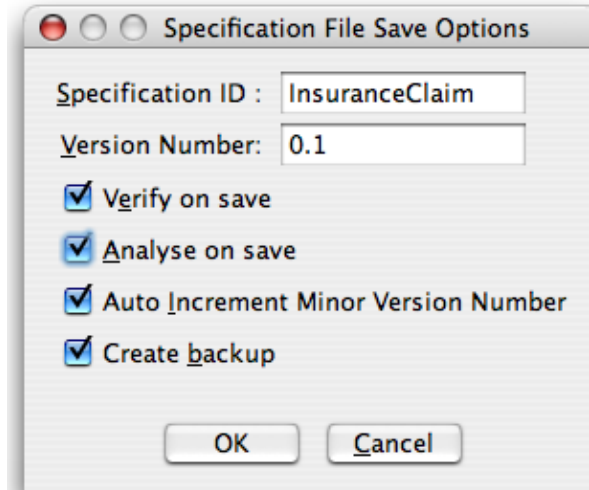
Figure 4.42: A invalid specification

Figure 4.43: Specification File Save Options dialog

numbers consist of a major part (to the left of the point) representing a major revision, and a minor part (to the right of the point) representing a minor revision. Version numbers begin at 0.1; you may edit the version number directly to a *higher* version number at any time. If the 'Auto Increment Minor Version Number' checkbox is checked, the minor revision will increase by one each time the specification is saved.

The 'Verify on save' and 'Analyse on save' checkboxes, if checked, will verify and and/or analyse the specification before it is saved, providing a final check of the specification (note: verification and

analysis may take some time for large and/or complex models, and analysis in particular should be unchecked for incremental saves of such models).

The 'Create backup' checkbox, if checked, will copy the previous version of the specification file to a file of the same name, but with a .bak extension. Checking this option is recommended.

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.

To import a specification file from a pre-2.0 version of the editor (i.e. a .ywl file):

1. Click on the "Import a YWL Specification file" button, , on the Menu Toolbar or click **Specification** on the Menu and choose **Import from YWL File. . . .**

   A window will appear asking you specify the YWL file to import.

2. Select the file and choose **Open**.

   The YWL file will be loaded and converted to a YAWL 2.0 version specification. The specification can now be edited and saved as a 2.0 specification in the usual way.

4.8 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 Specification** menu item under the **Specification** 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 via the **Tools** menu, **Configure Specification Analysis** (see Figure 4.44). 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 4.45). The dialog includes a checkbox that, when checked, will keep the dialog open after the analysis completes so that the messages may be noted. This functionality can also be controlled via a setting in the Configure Specification Analysis dialog (see Figure 4.44).

If the optional YAWL specification analysis utility **wofyawl.exe**, is supplied in the same directory as the Editor, an extra tab entitled **WofYAWL Analysis** 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.

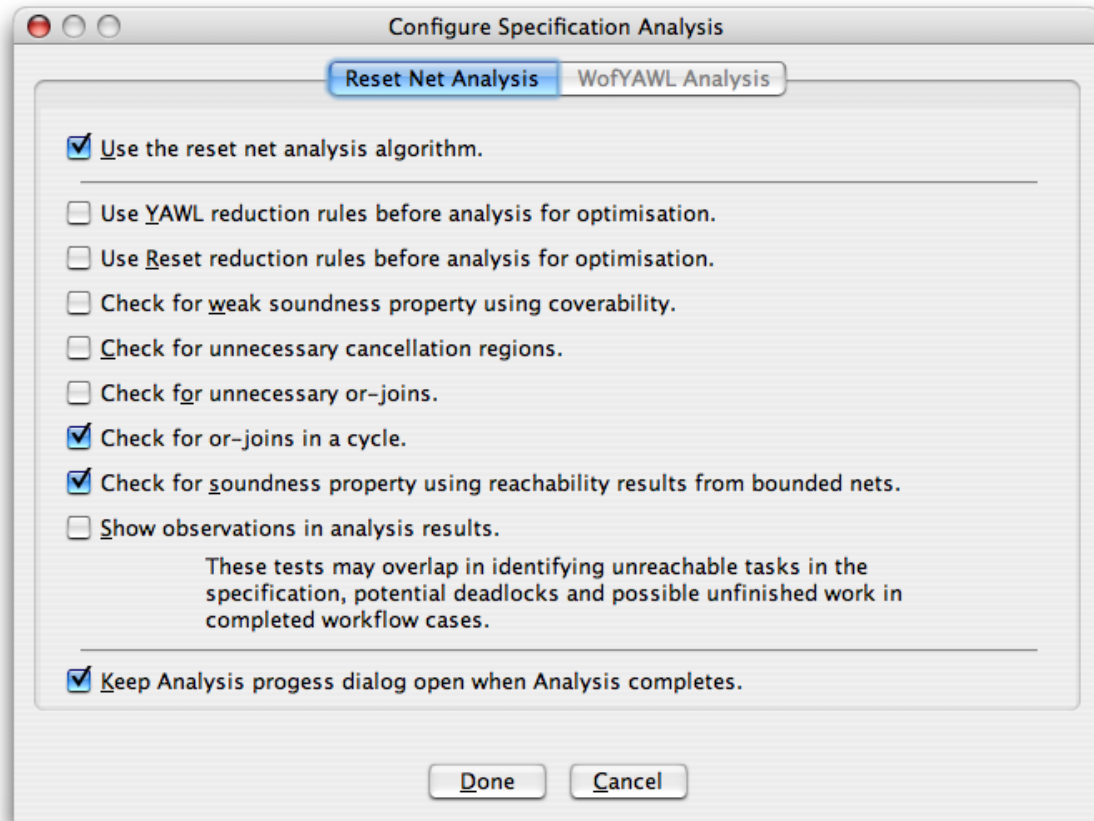
[3] WofYAWL is currently only available for Windows environments.
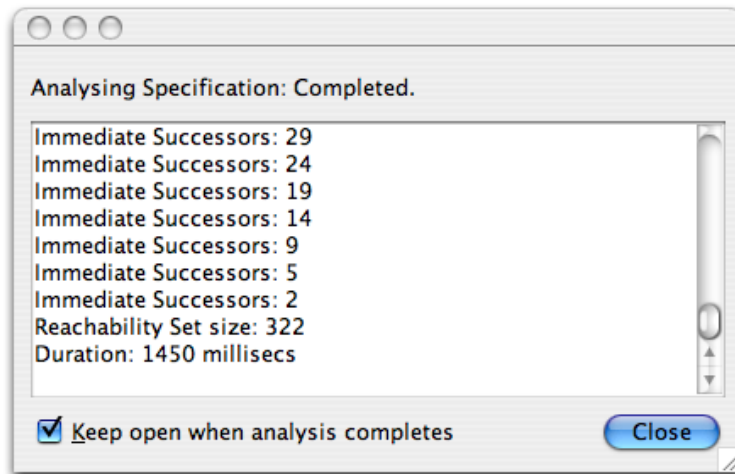
Figure 4.44: Configure Specification Analysis dialog

Figure 4.45: Configure Specification Analysis dialog

4.8.1   Verification and Analysis Explained

This section provides a brief overview of verification 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 forms a predecessor of YAWL, which does not support OR-splits/joins, Multiple Instance Tasks and cancellation regions. Informally speaking a workflow is sound iff [4]:

- 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.

- The net has *proper completion*. This means that when the output condition is marked 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 [23]. 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, 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 [23] is based on the theory of Reset nets (this is built into the Editor). The other approach [21] uses Petri net theory and in particular, the concept of transition invariants. For this latter approach the program wofYAWL is to be used (the executable of this program should be put in the same directory as the Editor under the name wofyawl0.4.exe). These approaches are different in that there are workflow specifications where one of them can pick up an error which the other approach cannot.

By choosing "Configure Specification Analysis" under the Tools menu in the Editor one 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 4.45 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 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.

The wofYAWL analysis tab-page of the 'Configure Specification analysis' dialog is shown in Figure 4.46. The reader is referred to [21] for a detailed explanation of the concepts behind this approach to verification.

4.9 Automated task

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
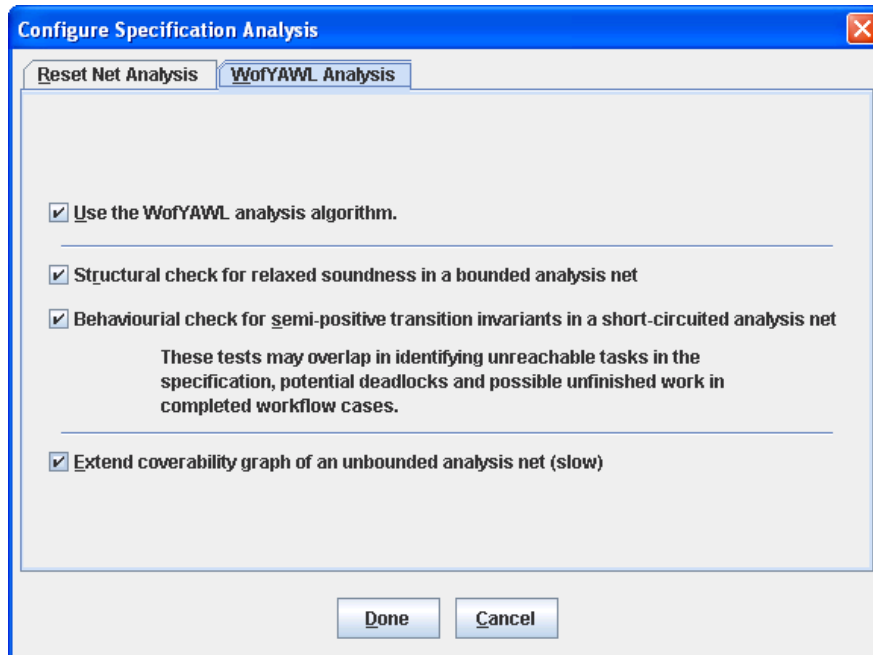
Figure 4.46: Verification using wofYAWL

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 it 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 right-clicking on it and selecting the **Task Decomposition Detail** dialog, then ticking the **Automated** checkbox in the **External Interaction** section (see Figure 4.29). When the Automated checkbox is checked, the option **Manage Resourcing** in the task's right click menu will be disabled (since human resources are not required for automated tasks).

Data manipulation can be achieved by using the task's variables and their Input and Output Parameters. Parameters are generally used to copy the content of a net variable to a task variable and back again. But parameters 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. Figure 4.47 shows an simple example of copying from expression, where the literal **true** is copied onto the net variable PO_timedout after choosing the from expression radio button. This choice is only available for automated tasks, and any task variables used in the XQuery expression itself must be declared as Input & Output usage types.

4.9.1   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
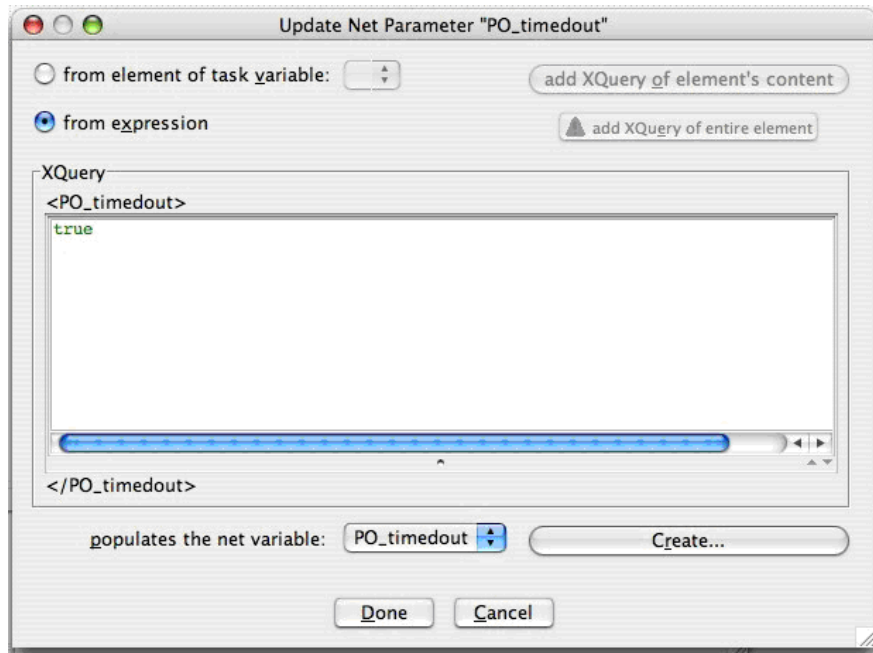
Figure 4.47: An output parameter set to copy from expression for an automated task

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 three codelets available in the Editor (see the *YAWL Technical Manual* for directions on how to add user-defined codelets):

- ExampleCodelet: 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.

- 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.

To associate a codelet with a task decomposition, first mark the decomposition as automated by ticking the **Automated** checkbox in the Task Decomposition dialog (see Figure 4.29). When checked, the **Set Codelet** button is enabled; clicking that button opens the *Set Codelet for Automated Decomposition* dialog, listing the available codelets (Figure 4.48). Note that a valid connection to the Resource Service is required for this list to be populated with codelets. 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 parameters 'a' and 'b', and output variable 'c' (all of type 'long'). These parameters must be added to the task in the normal manner – at runtime, if the required variables 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.
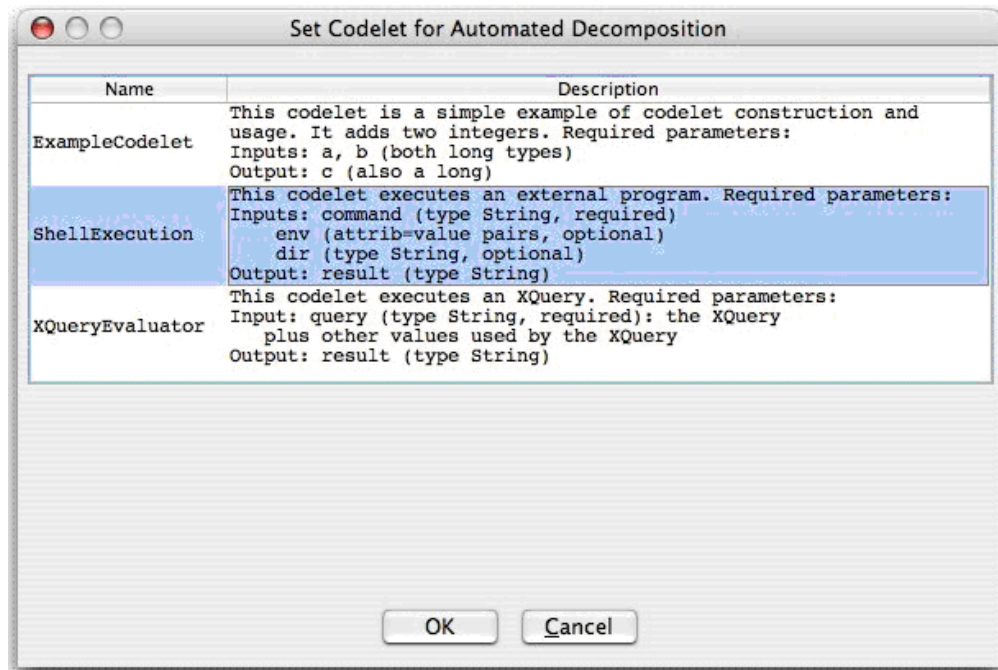
Figure 4.48: The Set Codelet dialog

4.10   Resource Management (Manual task)

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 Task Decomposition Dialog) can be allocated resources via a number of steps, by right-clicking on the task and selecting **Manage Resourcing**. This will launch the resource manager wizard. Figure 4.49 shows the dialog window for Step 1 of the wizard.

Step 1

In Step 1 we can specify the interaction strategy for work items of the selected task. There are three interaction points: offer, allocation, and start, each of which may be either User or System processed.

If we choose that work items are to be *Offered* by the User and not by the System, at runtime an Administrator will need to manually choose the resource(s) to offer the work item to. If System offering is chosen, then in Steps 2 and 3, we can specify the resources that will automatically be offered work items of the task at runtime.

If we choose that work items are to be *Allocated* by the User and not by the System, 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). If System allocation is chosen, in Step 4 we can specify how work items will be allocated to a participant by the System.

If we choose that work items are to be *Started* by the User and not by the System, then the user will choose to manually start working on the work item at a time of their choosing. If System starting is chosen, the work item, once it is allocated to the participant, will be immediately started.
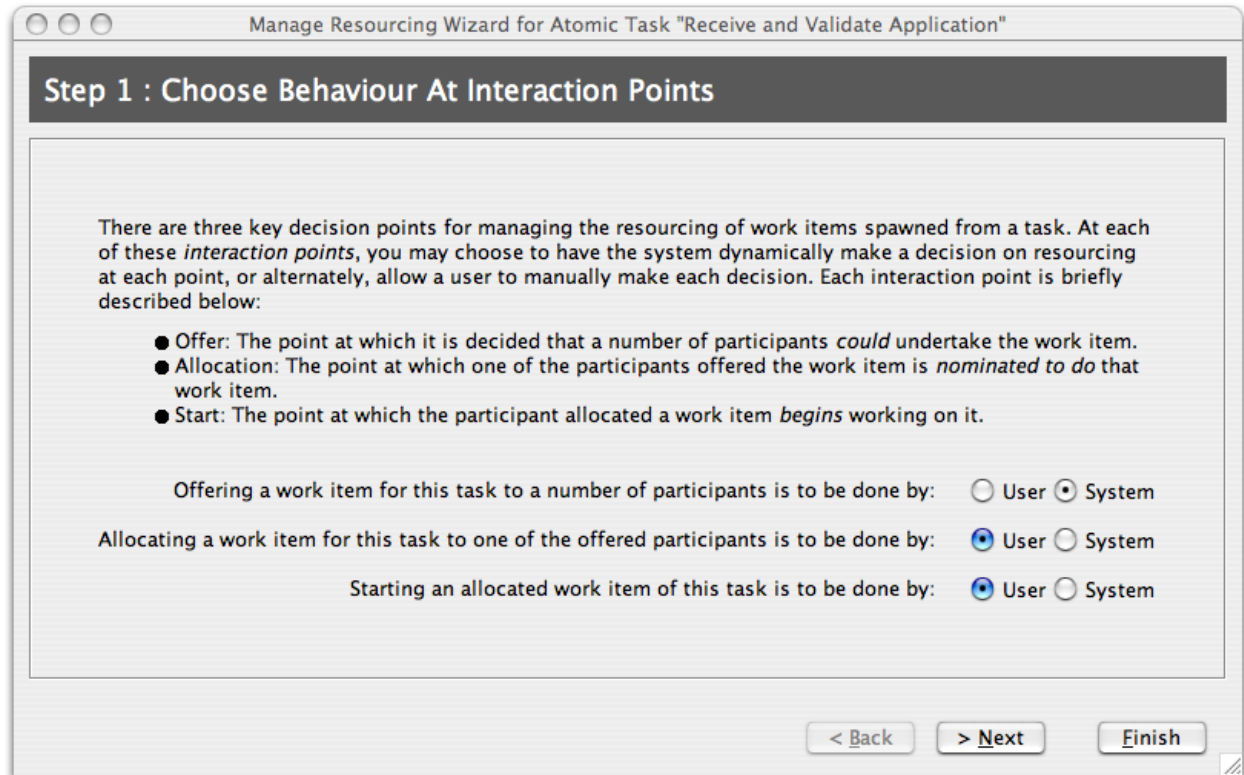
Figure 4.49: Step 1 of the resource management wizard

Step 2

In Step 2, shown in Figure 4.50, we can select an initial set of 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[4]. These can be picked from the relevant lists in this dialog (multiple selections are supported).

Additionally, in the Net Parameters panel you may nominate one or more net-level variables that at runtime will contain a value of the userid of a participant or the name of a role (that is, *deferred allocation*). For a variable to appear in the Net Parameters panel, it must be of type 'string' and usage 'Input & Output' or 'Input Only'. Note that *all* net-level variables that are of string type and have one of those usage types will be listed in the *Net Parameters* table, and will initially all be set as referring to 'Data'. For each of those that will contain resourcing information at runtime, the *Refers To* value must be changed to either 'Participant' or 'Role' via the drop down list on each row, depending on the type of resource the value will refer to at runtime.

Step 3

In Step 3, shown in Figure 4.51, we can filter the set of participants that have been described in Step 2. Filtering can be done over capabilities and/or over positions and organisational groups. Moreover, from this dialog it is possible to:

- allow the work items of the selected task to be offered only to participants who are members of the set specified in Step 2 and have previously completed work items of another task (as specified) in the current process instance;

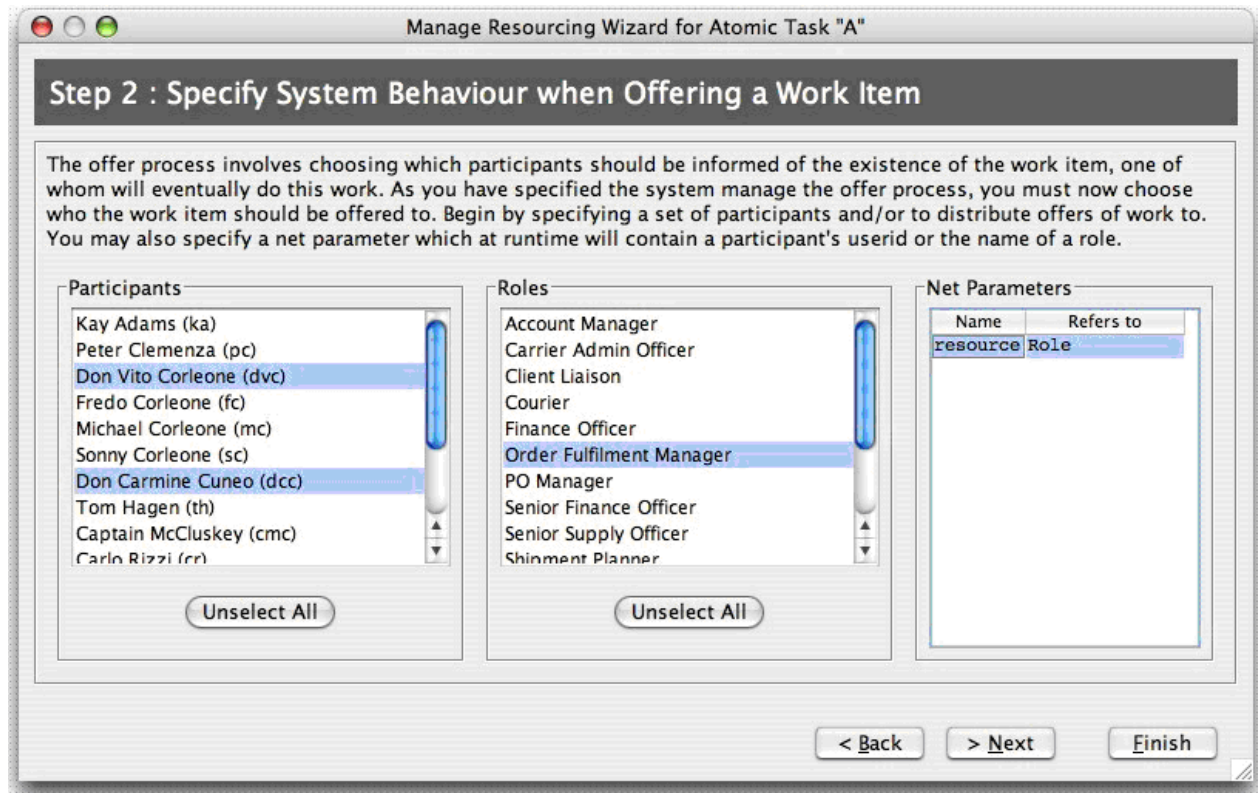[4]A role is essentially a set of participants.

Figure 4.50: Step 2 of the resource management wizard

- do *not* allow the work items of the selected task to be offered to participants who are members of the set specified in Step 2 and have previously completed work items of another task (as specified) in the current process instance;

- Allow all of the work items of the selected task to be *piled* to a particular participant, which means that, if a participant chooses to pile a work item of the task, that participant will be automatically allocated the work item *and all future instances of work items 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.

Step 4

In Step 4, shown in Figure 4.52, we can select the allocation mechanism that determines at runtime which participant, chosen from the members of the distribution set defined earlier, will actually be allocated the work item. The currently available mechanisms 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 this task instance the least number of times in the past;

- Round Robin (by experience): chooses the participant in the set who has been allocated this task instance the most number of times in the past;

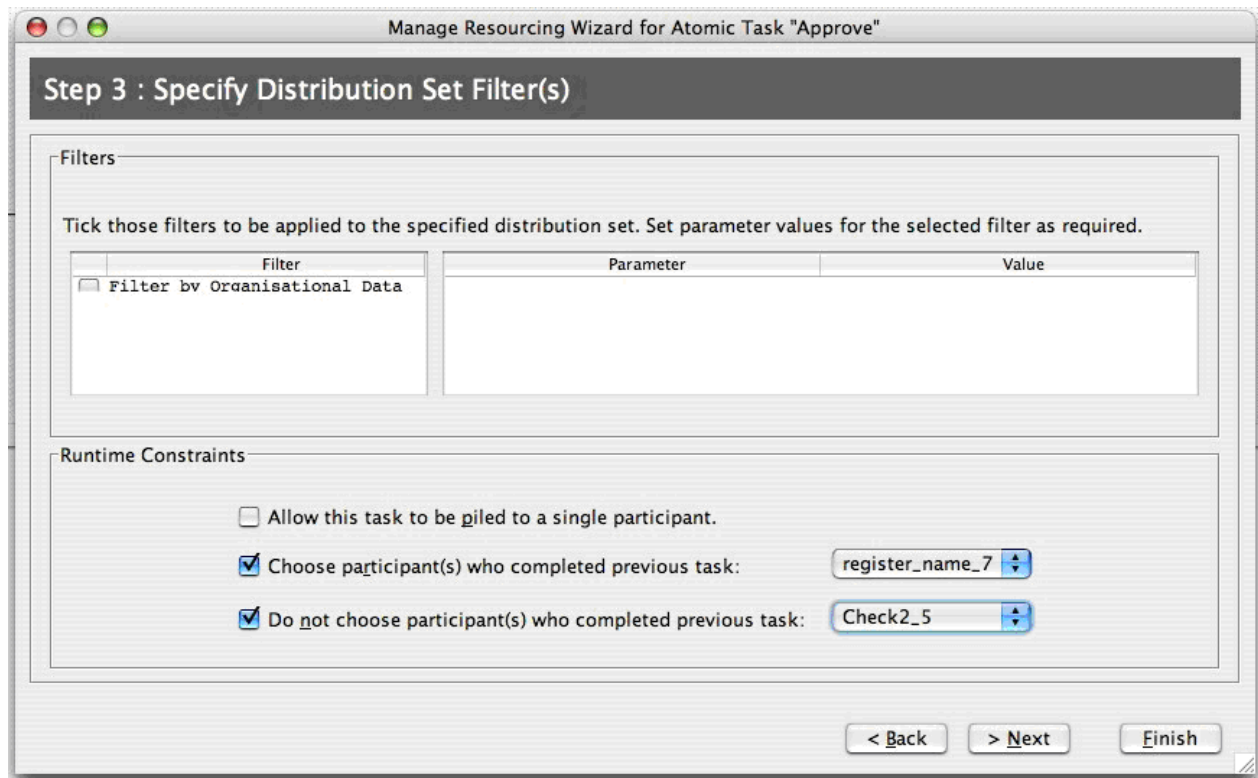- Random Choice: randomly chooses a participant from the set (the default); and

Figure 4.51: Step 3 of the resource management wizard

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

Step 5

In Step 5, shown in Figure 4.53, we can specify participant-task privileges (privileges that apply only for this task). For example, we can specify whether participants are allowed to suspend the execution of work items of the selected task.

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

4.11 Task Timer

Any atomic task can be assigned a timer behaviour by right-clicking on it and selecting the **Set Task Timer. . .** . The dialog in Figure 4.54 will appear.

From this dialog it is possible to set an activation type and an expiry 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.

4.11.1 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
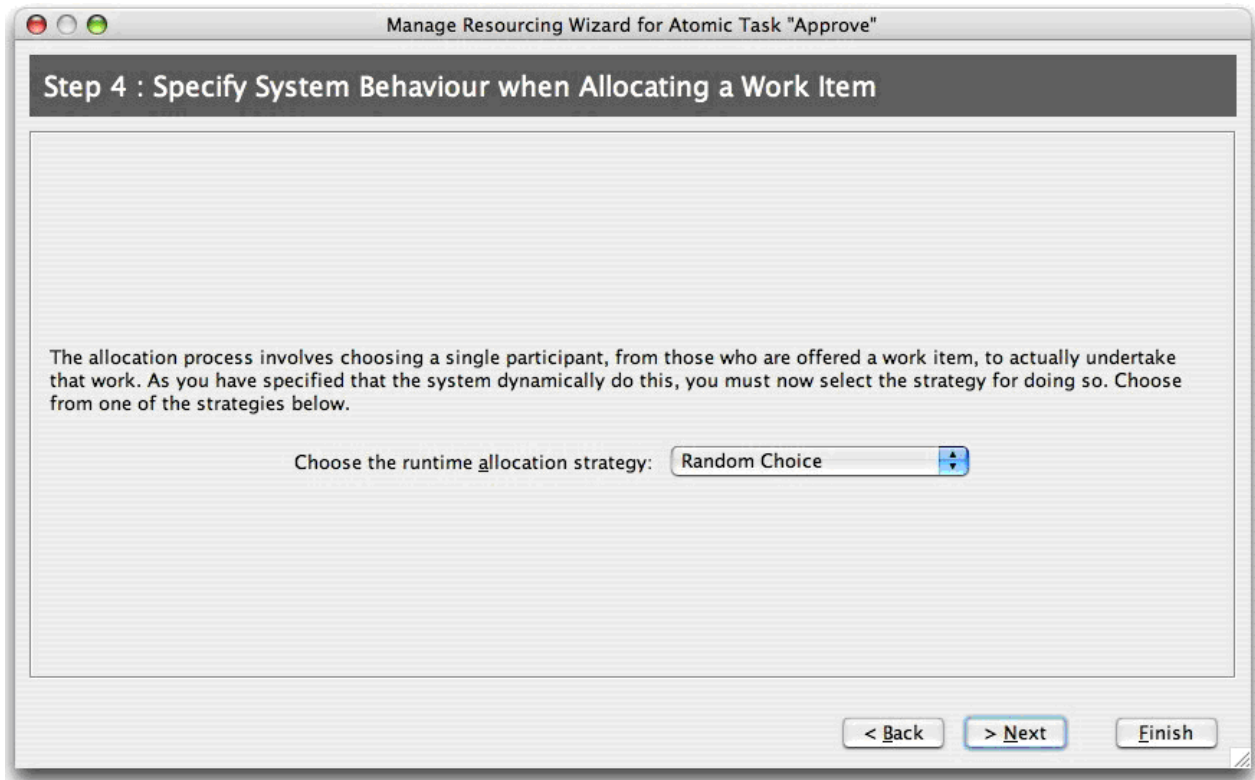
Figure 4.52: Step 4 of the resource management wizard

timer expires, the task instance will complete no matter what its current status is (offered, allocated, started). The possible danger of this behaviour is that a work item might be timed out while being edited by a user, in which case any modifications the user makes after that time are lost.

- 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 not reached. Once the timer expires, the task is immediately executed and completed.

4.11.2   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 starts. Again, the timer may expire while the task is being edited by a user.

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

4.11.3   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 the time of' radio button on the Set Timeout Detail dialog (Figure 4.54) 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
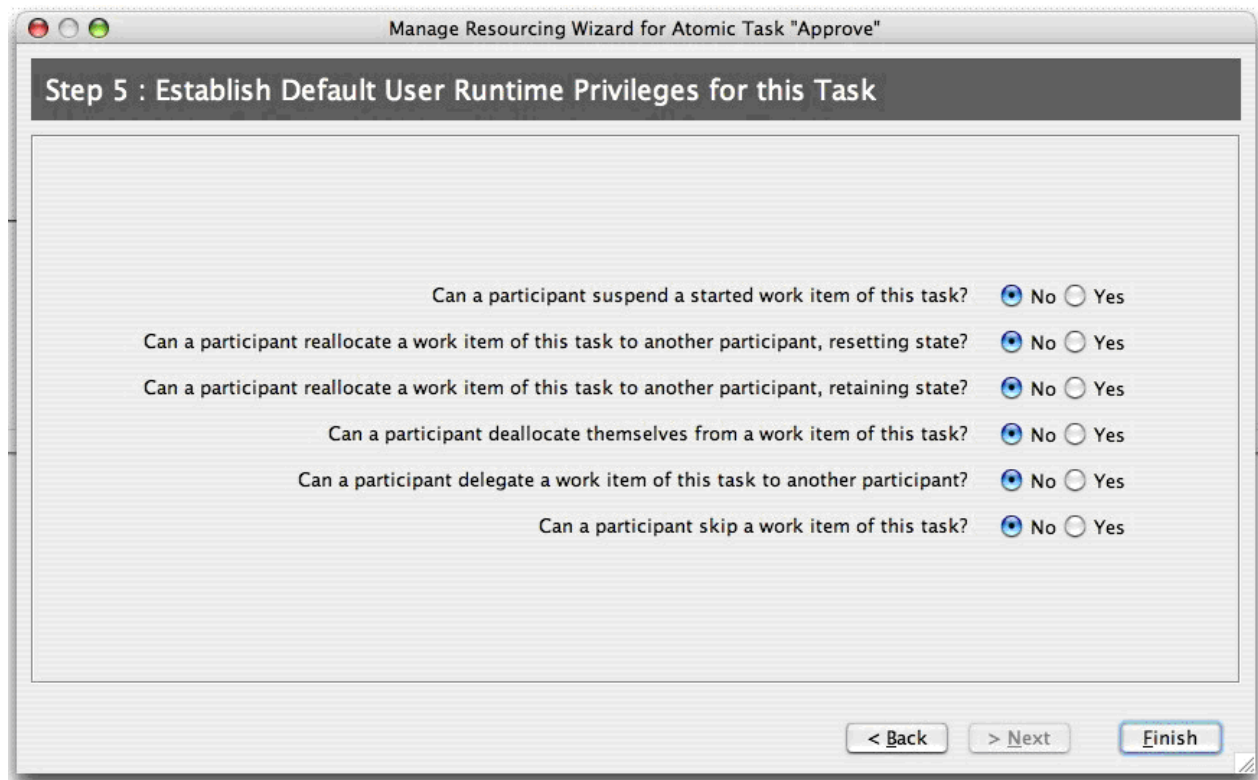
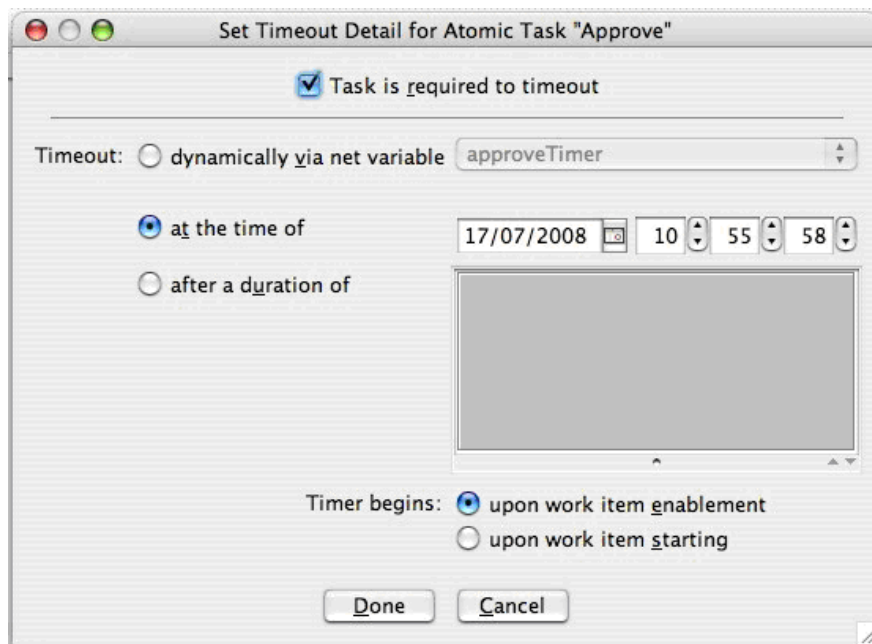**Figure 4.53:** Step 5 of the resource management wizard

Figure 4.54: The timer dialog for an atomic task

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 a duration of' radio button on the Set Timeout Detail dialog (Figure 4.54) and enter the required value. A Duration value is expressed in the following form:

$$P n Y n M n D T n H n M n S$$

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. Care should be taken when specifying a duration, for example P2M 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 variable of type YTimerType, 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 parameter of type YTimerType.

2. Choose a task, right click on it and choose **Set Task Timer...** to open the Set Timeout Detail dialog (Figure 4.54), click the 'dynamically via net variable' option and select the net-level parameter created in Step 1.

3. Map the net-level parameter to another (previous) task in the flow, 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 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.

An example of how a variable of YTimerType appears in a dynamic form at runtime can be seen in Figure 4.55.

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

4.12 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 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, right click and select **Set Custom Form...**, then 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).
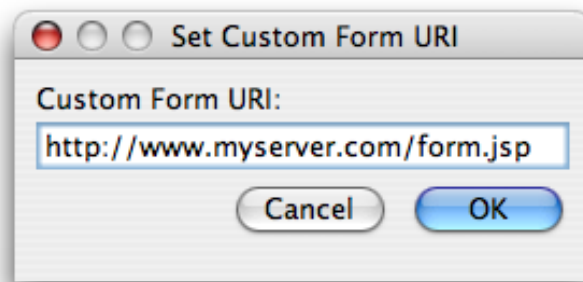
Figure 4.56: Custom form dialog

See the YAWL technical manual for information regarding the creation and configuration of custom forms.

4.13 Extended Attributes

The Editor offers a means for defining extended attributes to be associated with task decompositions, net variables and task variables. Currently, the implementation and the semantics of these attributes is left to the developer.

Extended attributes can be defined in property files, which need to be placed in the folder:

```
<editor_install_path>/YAWLEditorPlugins/ExtendedAttributeProperties
```

Attributes referring to net and task variables must be defined in a file named VariableProperties (no extension), while attributes referring to task decompositions must be defined in a file named DecompositionProperties (no extension). The files are read at the Editor's next restart.

Attributes can be of type string, boolean, enumeration and XQuery. The following is an example of a property file that defines the attributes description, help, mode, refresh and skipSchemaValidation (comments are indicated by a line starting with #).

```
#Decomposition Attributes
#Wed May 14 17:35:42 AET 2008
```
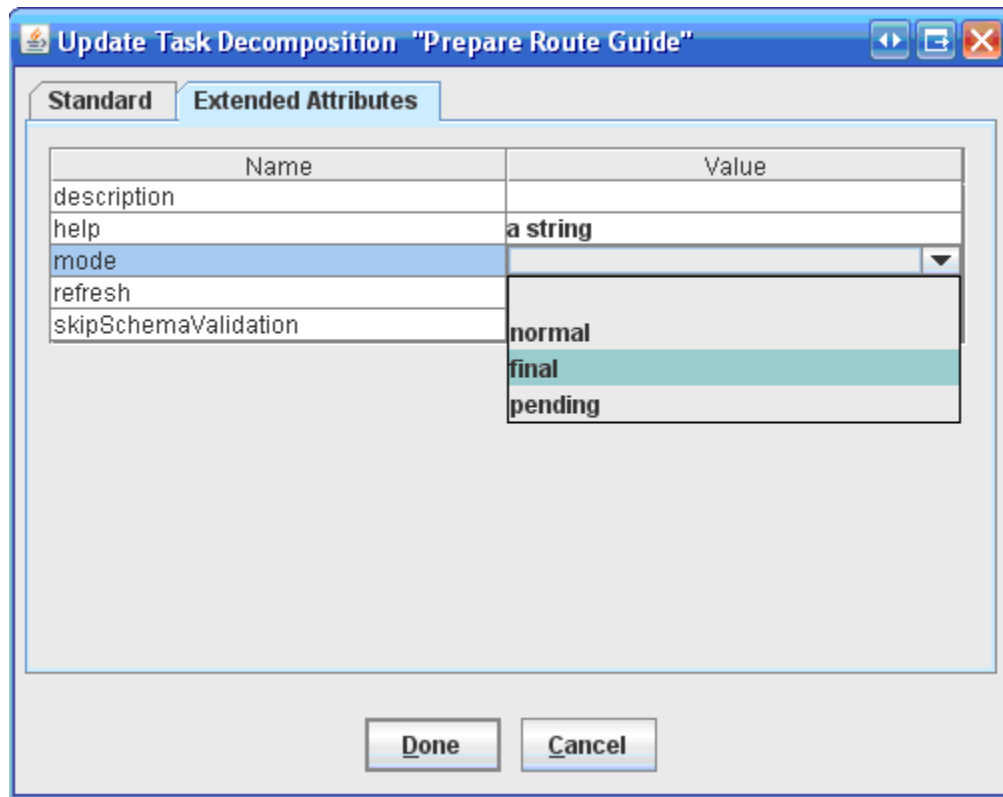
Figure 4.57: The dialog for editing the extended attributes associated to task decompositions

```
description=xquery
help=string
mode=enumeration{normal,final,pending}
refresh=xquery
skipSchemaValidation=boolean
```

From the Editor, the user can visualise and edit the list of attributes by selecting the tab **Extended Attributes** from one of the following dialog windows:

- Update Task Decomposition,

- Update Net Variable,

- Update Task Variable.

Figure 4.57 shows the rendered list of attributes for the above property file.

Note that attributes can only be specified through property files and not via the above dialogs.

Editor Troubleshooting

The YAWL Editor is not responding to my mouse clicks and it is beeping every time I click the mouse.

Check to see if you have any Editor dialogs open. To do so, in Windows hold down the ALT key (for OSX, hold down the Command key) and press the TAB key until you reach the dialog window, then let go of the

keys. Close the offending dialog window to reenable use of the YAWL Editor.

I can't connect two elements with a flow. Why?

The Editor will not allow you to connect one element to another where such a connection is invalid. Typical examples include:

- Trying to connect a second flow to an undecorated task.

- Trying to point an incoming flow to a split decorator (or an outgoing flow from a join decorator).

See the troubleshooting entry "How do I find out more about elements and principles....." for more details.

When I validate my Net, I get the following validation message, 'The net (. . .) may complete without any generated work. Should all atomic tasks in the net be unlabelled?'

This message appears if none of your atomic tasks are associated with a decomposition. To add a decomposition to a task (and thus ensure that the engine will get a user to handle the task at this point in the workflow), right click on the task and choose **Select Task Decomposition. . . .** **Choose Create. . .** and in the following window, "Update Task Decomposition", and enter the name of your task under Decomposition Label, then click **Done**.

My specification won't validate without any errors and I don't know what to do.

First, check the logic of your specification and exercise every branch of your workflow for correctness.

If you are sure of your workflow, consult the YAWL website for the proper use of the YAWL elements:

http://yawlfoundation.org.

How do I find out more about the elements and principles of the YAWL workflow specifications?

For more information about the mechanics of the YAWL workflow elements, please consult the YAWL website:

http://yawlfoundation.org.

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 a list of all 46 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, the users can also supply 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 [6].

Data usage is also part of the 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 net variable may be assigned an initial value at design time. Further details will be given in the next section on data transfer.

---

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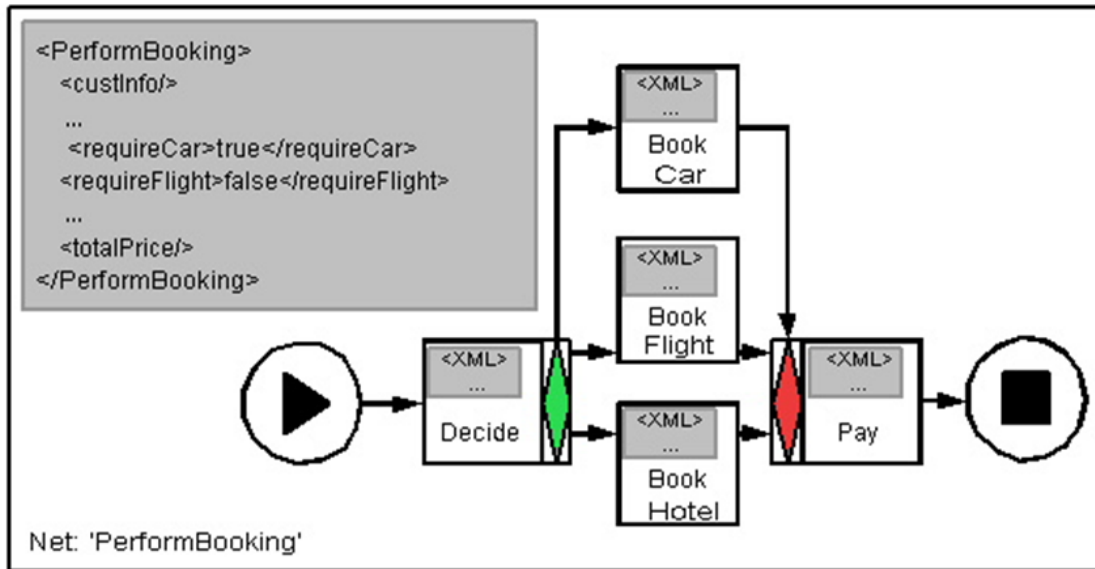[2]XQuery 1.0: An XML Query Language. W3C Working Draft, 4 April 2005.

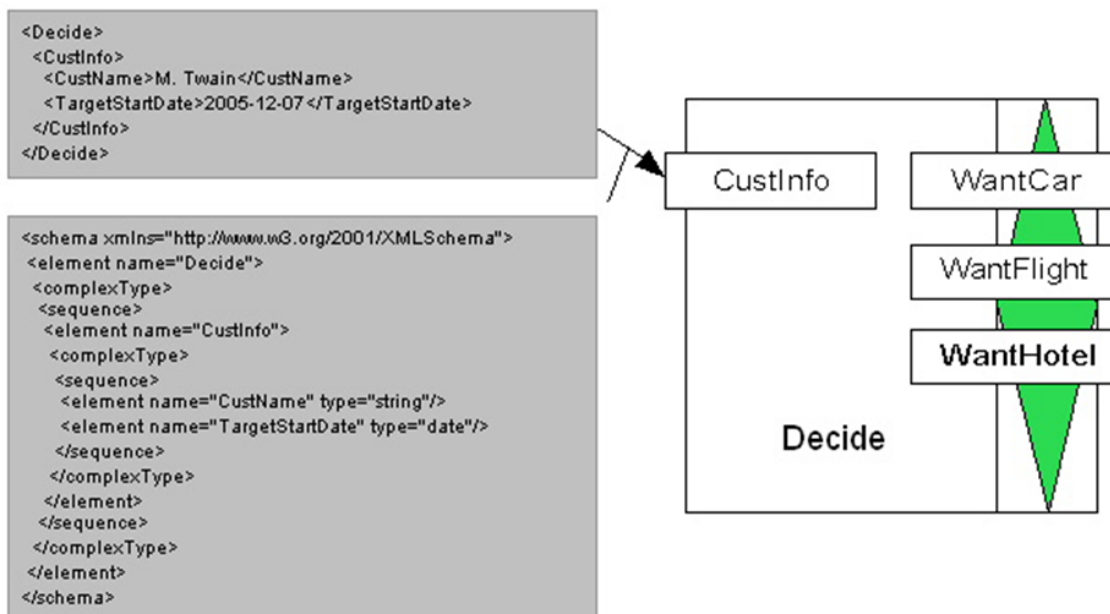**Figure 5.1:** A YAWL net "PerformBooking" and its data representation

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

Note In Chapter 4, Section 4.5.3 to Section 4.5.6 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 Va) to task B (e.g. variable Vb), an appropriate net variable of N (e.g. Vn) must be available to convey data from Va to Vb. In the YAWL editor, each task can be assigned an *input parameter* and/or an *output parameter* (depending on the specified 'Usage' 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. Considering 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").
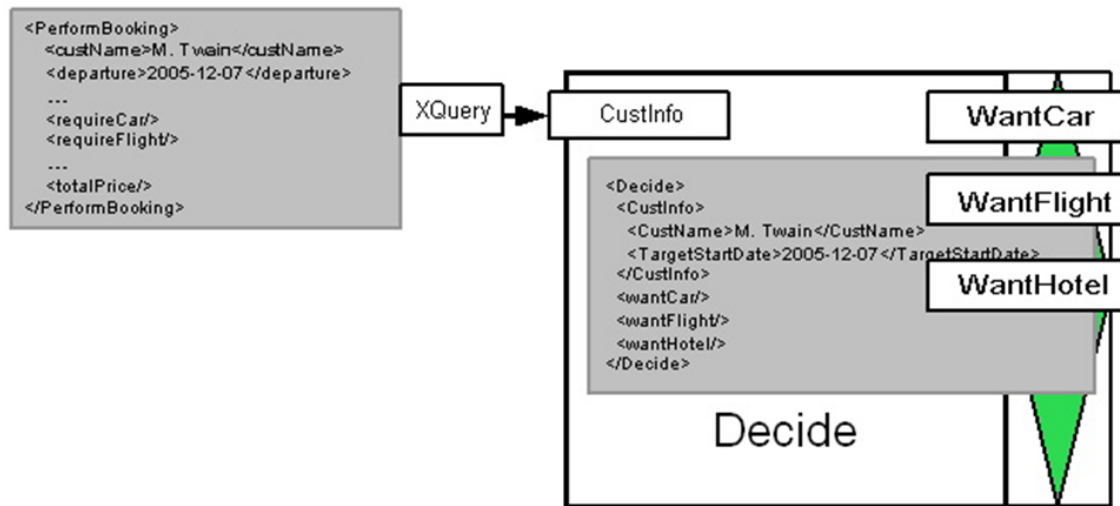

Figure 5.3: An example of data transfer from net "PerformBooking" to task "Decide"

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 web 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 usage. 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 usages. 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 4.5.7 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, the validation result will indicate that "there exists an input parameter (V-in) in (task) T that is not mapped to by this task".

3. When specifying XQueries in the parameter definitions of a task, the YAWL editor may automatically generate an engine-compatible XQuery or indicate whether a user-defined XQuery has valid or invalid syntax (by displaying the query in green or red respectively). 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 *flow detail* 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 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 at task "Decide", for choosing the branch of "Book Car" in the "PerformBooking" process depicted in Figure 5.1.

For a task with XOR split, all (conditional) flows are specified in a list, and their predicates are evaluated in the same order as they are present in the list. Since an XOR split allows only one flow to be chosen, once the engine reaches a 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 present 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 4.5.8 illustrates how to specify flow predicates for tasks with XOR or OR splits in the YAWL Editor. Two things are worth noting:

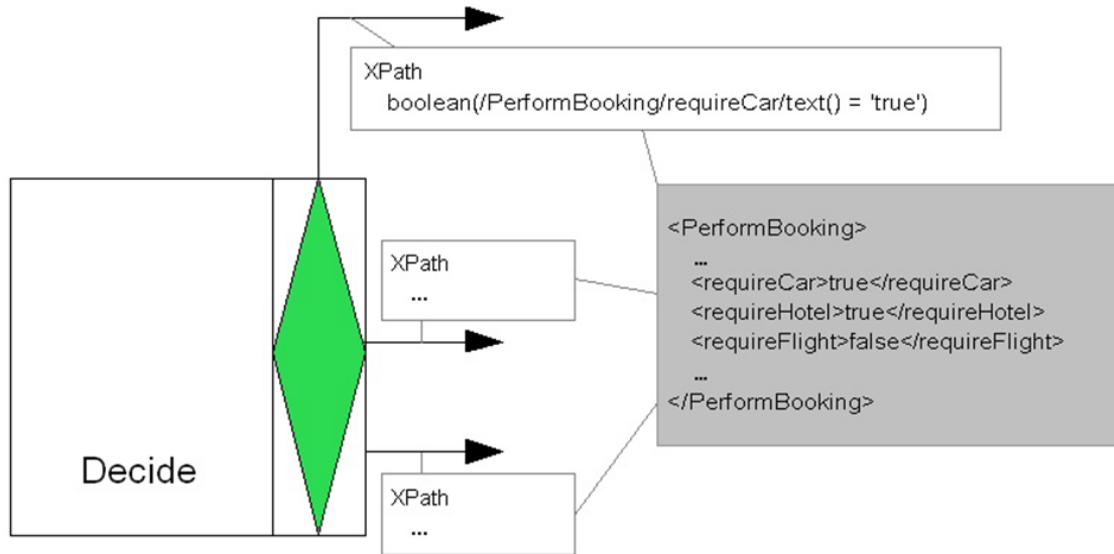[4]XML Path Language (XPath) 1.0. W3C Recommendation, 16 November 1999.

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

1. Only net variables are allowed to be used in specifying flow predicates. This is because the flow 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. As YAWL applies only XPath boolean expressions for specifying flow detail, an XPath boolean expression with invalid semantics may evaluate to false, and as a result the execution of the process will still continue. In such a case, designers need to validate their YAWL specification against the desired system behaviour by themselves.

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 [18]. 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 [2]. A set of four types of XQueries may be 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 4.5.9 illustrates how to specify multiple instance data manipulation, e.g., the four types of XQueries, in the YAWL Editor.

5.5   Illustrative Examples

Below are three illustrative examples which cover the aforementioned 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 [2].

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'.
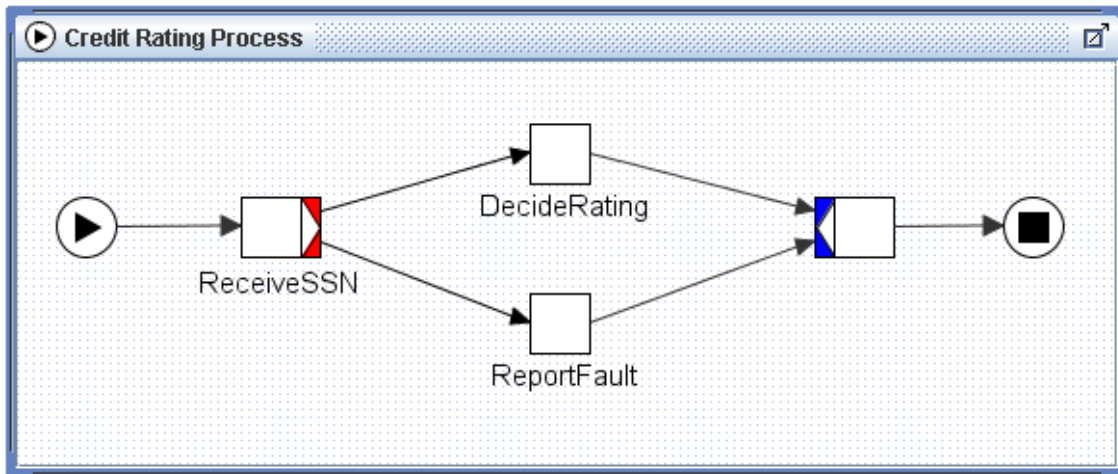

Figure 5.5: The "CreditRatingProcess" net

Figure 5.6 shows three net variable definitions for the "CreditRatingProcess" 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 5.7 shows the parameter definition of task ReceiveSSN. It can be observed that this task has an output only variable called ssn. This variable requests social security number from the client, and as specified in the output parameter definition, it then passes the data to net via XQuery "{/ReceivedSSN/ssn/text()}", which can be generated automatically by clicking the **add XQuery o͟f element's content** button in the *Update Net Parameter* dialog.

Figure 5.8 shows the flow definition at task ReceiveSSN. The predicate for the flow leading to task ReportFault is "starts-with(/CreditRatingProcess/ssn/text(),'0')"[5]. It returns true if the string conveyed by variable ssn

---

[5]**starts-with**(*string*, *string*) is an inbuilt XPath boolean function. It returns true if the first argument string starts with the
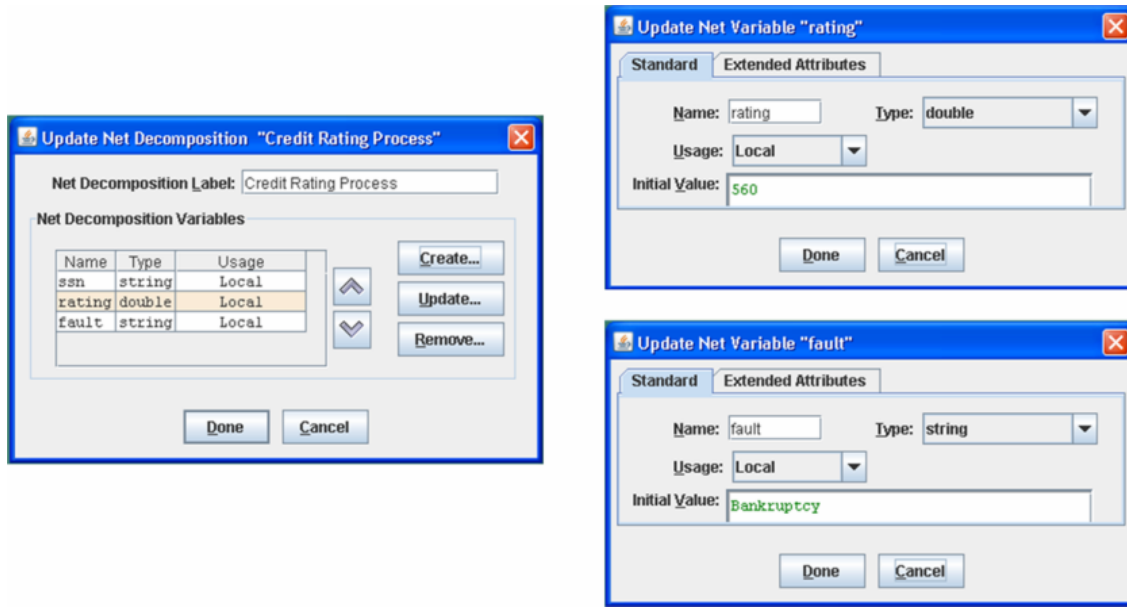
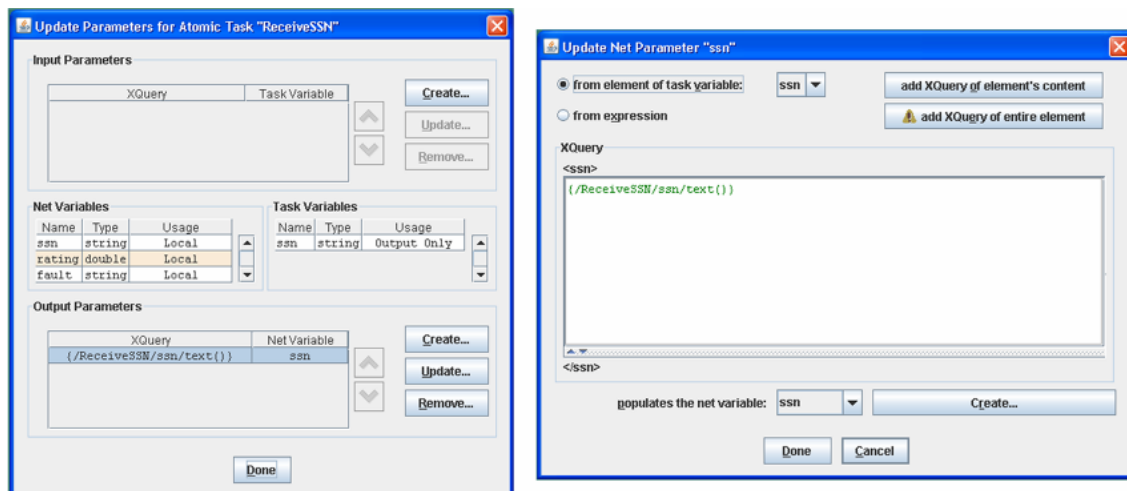**Figure 5.6:** Net variable definitions

Figure 5.7: Parameter definition for task Receive 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 "/CreditRatingProcess/ssn/text()" can be generated by clicking the **XPath Expression** button in the *Update Flow Predicate* dialog.

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

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 shows that "No design-time engine validation problems

---

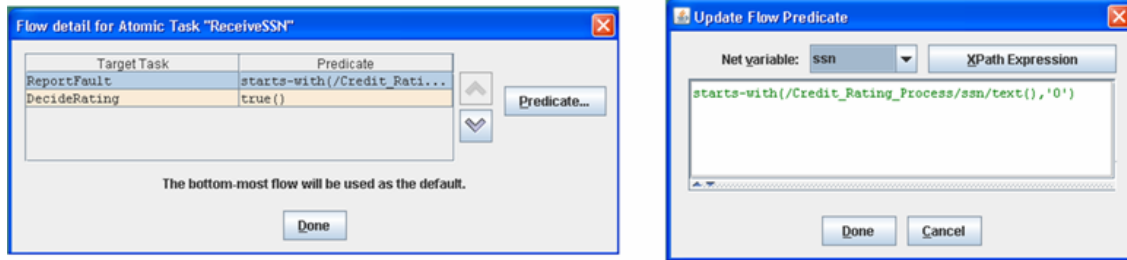second argument string, and otherwise returns false.

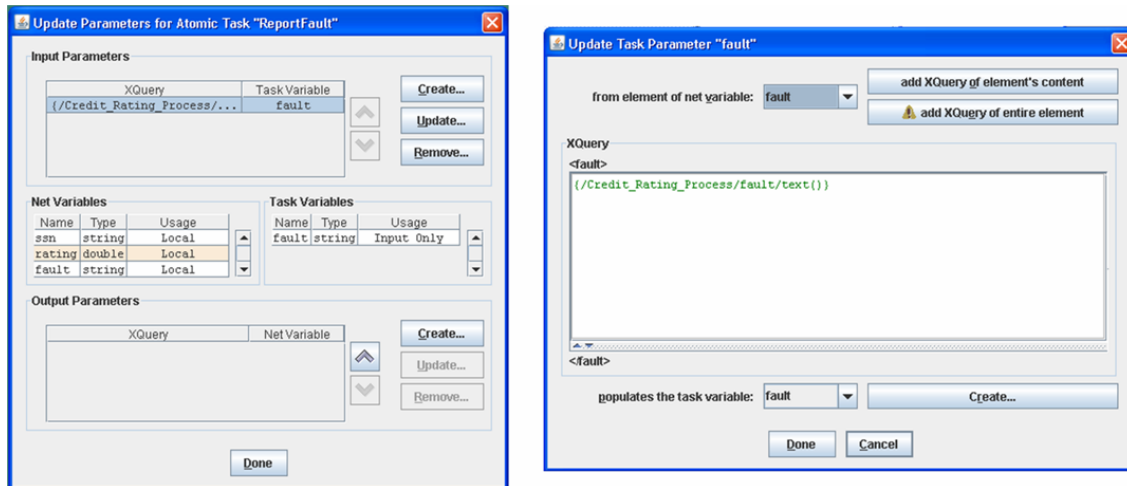**Figure 5.8:** Flow definition at task Receive SSN

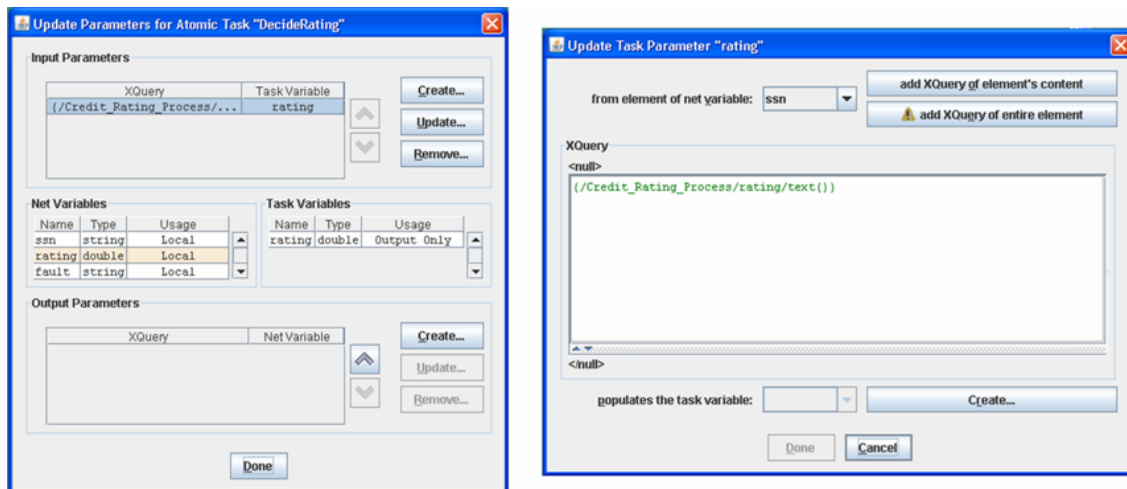**Figure 5.9:** Parameter definition for task ReportFault

Figure 5.10: Parameter definition for task Decide Rating

were founded in this specification". The last step before deployment is to specify the resource details for each of the manual tasks (i.e. ReceiveSSN, DecideRating, and ReportFault). In Chapter 4, Section 4.10 illustrates how to allocate resources to a manual task via the resource manager wizard. To keep things simple, we will choose to not specify any resourcing details for those task; in doing so, each of those tasks will be assigned the default settings: that offering, allocating, and starting a work item are all to be done by the User (at run time).

Finally, we save the "CreditRatingProcess" 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 (especially large) YAWL specifications.

Error 1: Missing Data Assignment for Input Variable

Assume that we forget to specify the mapping from net variable *fault* to task variable *fault* in the input parameter definition for task ReportFault (see Figure 5.11). This results in two syntax errors after validation of the "CreatRatingProcess" specification in the Editor, as shown in Figure 5.11.
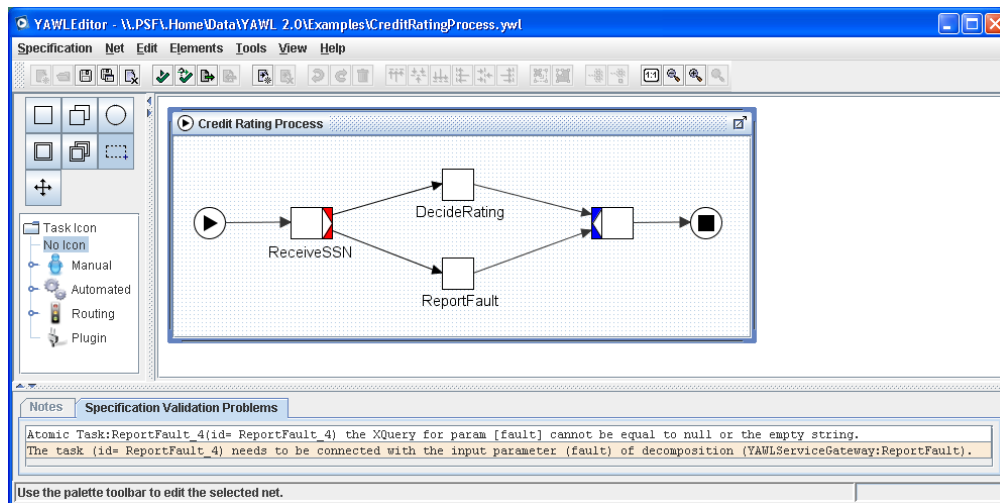

Figure 5.11: An invalid specification with missing data supply for an input variable

Error 2: XQuery with Invalid Semantics

Assume that we use XQuery "/ReceiveSSN/ssn" but not "/ReceiveSSN/ssn/text()" in the output parameter definition for task ReceiveSSN (see Figure 5.12). This modified mapping is incorrect because it passes "<ssn> some social security number </ssn>" from task variable *ssn* to net variable *ssn*. However, the mapping has a valid syntax (because both variables are strings) 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 5.12 will appear, indicating a failure has occurred validating the XQuery. Also, the executed credit rating process is halted.

5.5.2   Example 2: Simple Make Trip Process

This process provides 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 may be executed in parallel according to the user's requirements, and the booking details are also provided in each executed task. After all the booking tasks complete, the user is then asked to make payment for the trip, and the process ends.
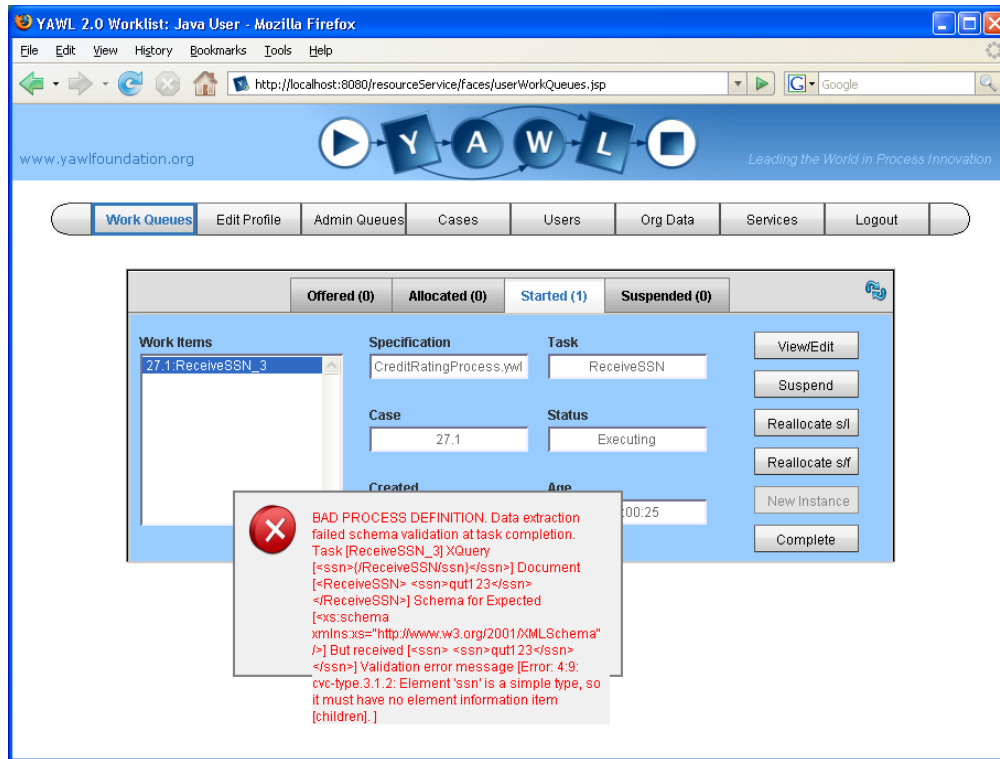
Figure 5.12: An error indicating invalid semantics of an XQuery used in the specification

YAWL Specification

Figure 5.13 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.
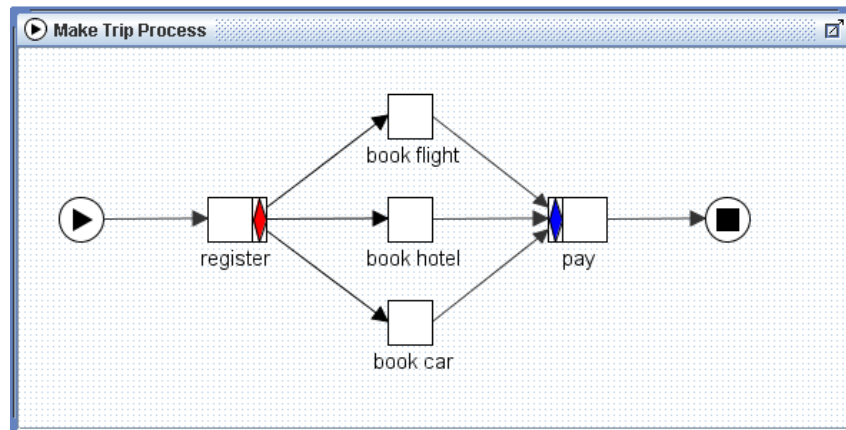

Figure 5.13: The simple "Make Trip Process" net

Figure 5.14 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 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

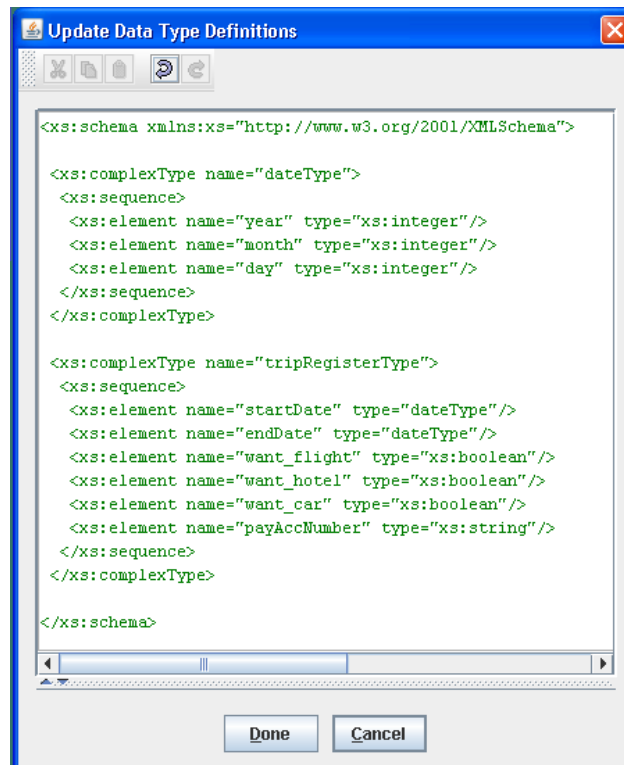*endDate* are of *dateType*.

Figure 5.14: Updated data type definitions

Figure 5.15 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).
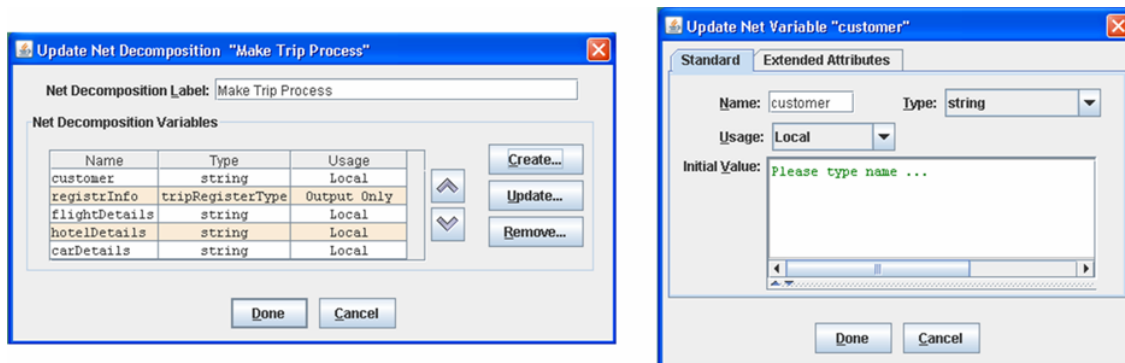

Figure 5.15: Net variable definitions

Figure 5.16 shows the parameter definition of task "register" with two task variables. The input & output variable *customer* gets the customer's name, and then passes it to the net. The output only variable *registrInfo* requests registration information from the customer, and then passes the data to the corresponding net variable via the XQuery "{/register/registrInfo/*}".

Figure 5.17 shows the flow definition at task "register". The predicate for the flow leading to task "book flight" is "/Make_Trip_Process/registrInfo/want_flight='true'". Similarly, the predicate associated with the flow leading to task "book hotel" or "book car" is to determine whether the variable *want_hotel* or *want_car*

Figure 5.16: Parameter definition for task "register"

has a boolean value of "true" or "false". Note that this flow definition is defined with an OR split, so that any number of the flows to booking tasks can be taken.

Figure 5.17: Flow definition at task "register"

Figure 5.18 shows the parameter definition of task "book flight" with four task variables. The three input only variables get data from the net via appropriate XQueries. Note that both *startDate* and *endDate* get data from the net variable *registrInfo*. There is one output only variable called *flightDetails*, which requires the information from the customer, such as his/her preferable airlines or even an exact flight number. The parameter definitions of tasks "book hotel" and "book car" are specified in a similar way, except that the output only variable is *hotelDetails* or *carDetails*.

Figure 5.19 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.

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 in the previous example will be executed. As a result, we obtain a more complex Make Trip

Figure 5.18: Parameter definition for task "book flight"

Figure 5.19: Parameter definition for task "pay"

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 5.20 shows the YAWL net specifying the above make trip process. There are two nets: the root net called "make trip" and the subnet specifying details for the composite task "do itinerary segment", which is basically the simple "make trip" net in the previous example.
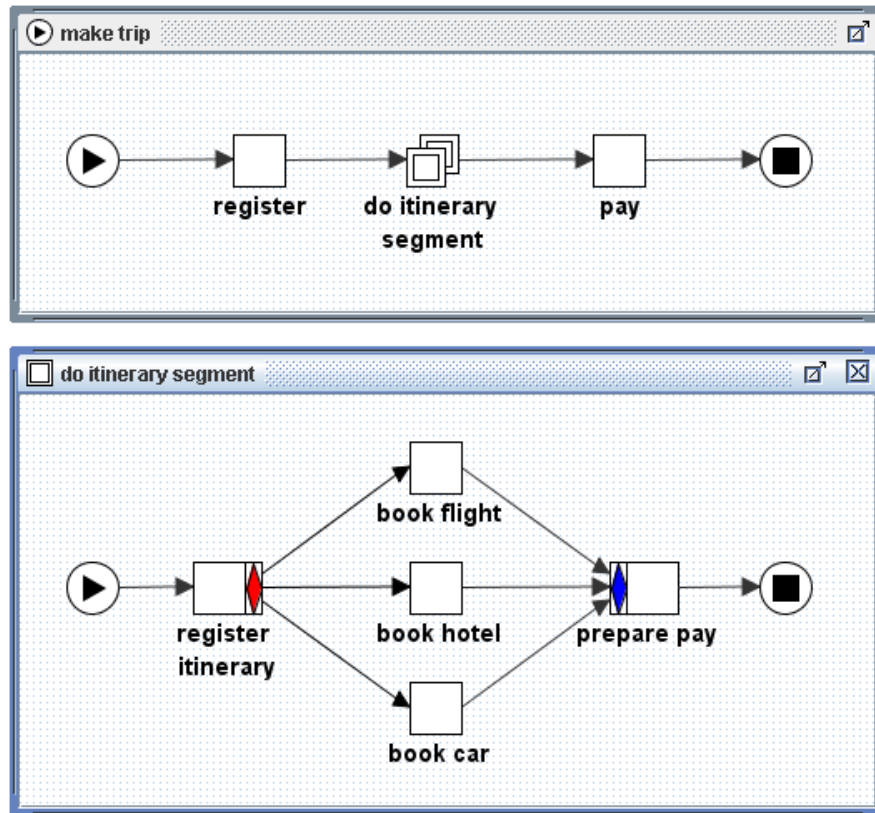

Figure 5.20: The "make trip" net with a multiple instance composite task "do itinerary segment" and the corresponding subnet

Figure 5.21 lists the data type definitions in the "make trip" net. 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 5.22 shows the net variable definitions for the "make trip" net. There are three new variables, *itinerary*, *legs*, and *subTotals*, as compared to the previous simple "Make Trip Process" net in Section 5.5.2.

Figure 5.23 shows the parameter definition of task "register". There are three task variables, *customer*, *legs*, and *payAccNumber*, with appropriate input or output parameter definition.

Figure 5.24 shows the parameter definition of task "pay". There are four variables which are all input only 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 5.25 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*,
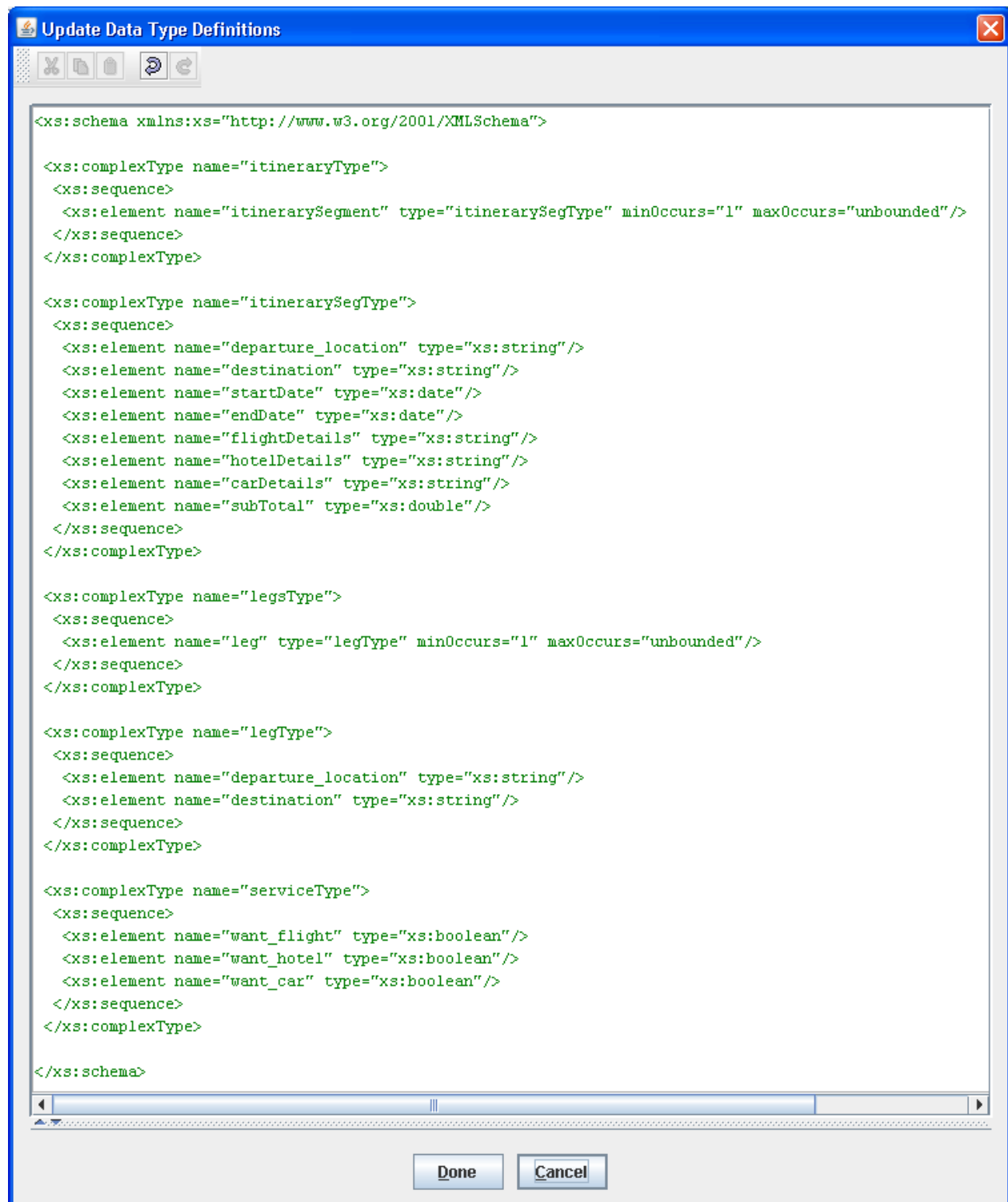
Update Data Type Definitions

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

 <xs:complexType name="itineraryType">
  <xs:sequence>
   <xs:element name="itinerarySegment" type="itinerarySegType" minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name="itinerarySegType">
  <xs:sequence>
   <xs:element name="departure_location" type="xs:string"/>
   <xs:element name="destination" type="xs:string"/>
   <xs:element name="startDate" type="xs:date"/>
   <xs:element name="endDate" type="xs:date"/>
   <xs:element name="flightDetails" type="xs:string"/>
   <xs:element name="hotelDetails" type="xs:string"/>
   <xs:element name="carDetails" type="xs:string"/>
   <xs:element name="subTotal" type="xs:double"/>
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name="legsType">
  <xs:sequence>
   <xs:element name="leg" type="legType" minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name="legType">
  <xs:sequence>
   <xs:element name="departure_location" type="xs:string"/>
   <xs:element name="destination" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>

 <xs:complexType name="serviceType">
  <xs:sequence>
   <xs:element name="want_flight" type="xs:boolean"/>
   <xs:element name="want_hotel" type="xs:boolean"/>
   <xs:element name="want_car" type="xs:boolean"/>
  </xs:sequence>
 </xs:complexType>

</xs:schema>
```

Done   Cancel

Figure 5.21: Updated data type definition for the "make trip" net
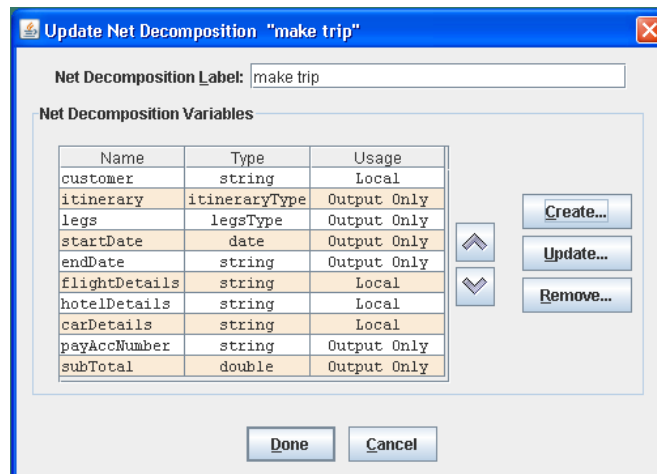
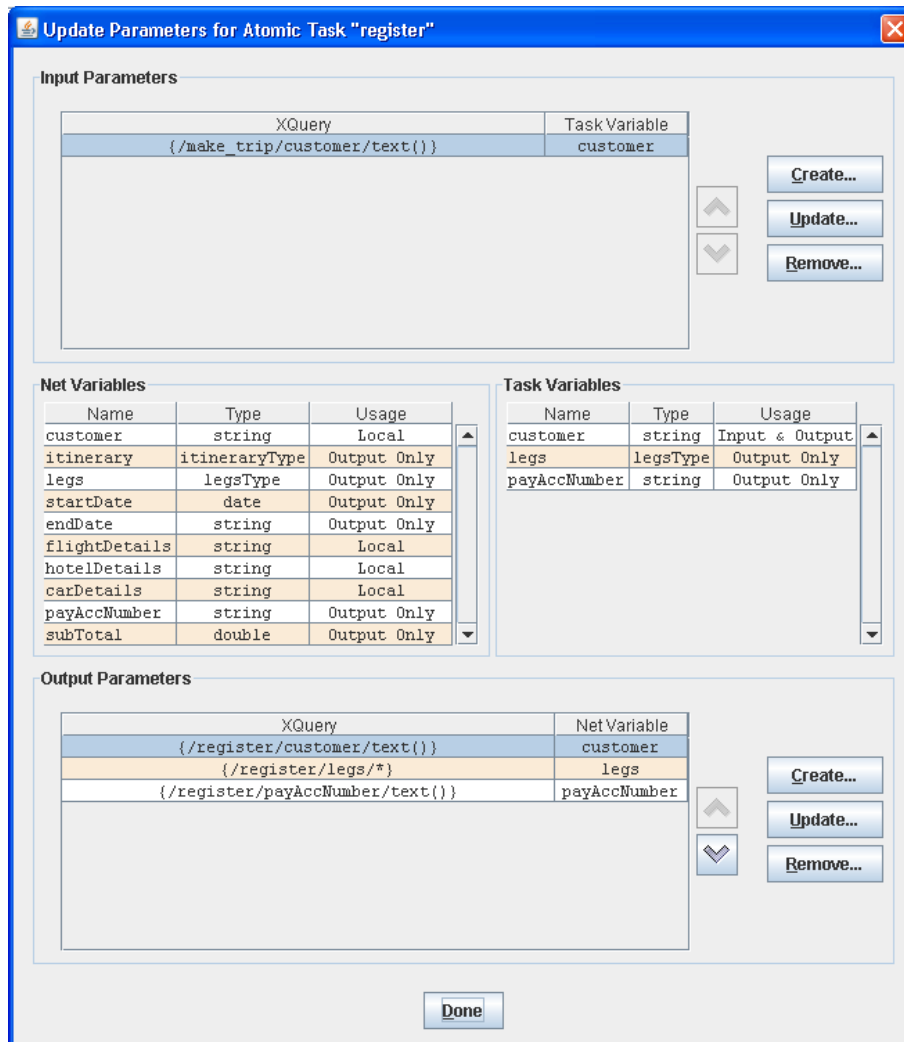**Figure 5.22:** Net variable definitions for the "make trip" net

Figure 5.23: Parameter definition for task "register"

Figure 5.24: Parameter definition for task "pay"

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 *leg* and the output parameter definition for variable *itinerary* are both determined by the task instance queries (see below).

Figure 5.25: Parameter definition of task "do itinerary segment"

Figure 5.26 shows the instance details for task "do itinerary segment". The left tab window titled **Bounds** is for specifying the task attribute data, i.e. the maximum and minimum number of instances allowed and the threshold value. The right tab window titled **Queries** is for defining the multiple instance data specific to individual execution instances of task "do itinerary segment" for each *leg* within one itinerary (i.e. a single process instance). There are four queries. Firstly, an *accessor query* manipulates the overall data carried by root net variable *legs* before the data is split out to each individual *leg*. This query determines the input parameter definition for variable *leg*. Secondly, a *splitter query* separates the unique values from the overall data carried by variable *legs*, and passes a unique value to variable *leg* 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 5.26: Instance details for task "do itinerary segment"

The subnet of task "do itinerary segment" specifies the simple "make trip" process that is similar to the one in Section 5.5.2. Thus, we do not go into every detail, but provide the parameter definitions for tasks "register itinerary", "book flight" and "prepare pay" in Figure 5.27 to Figure 5.29, 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".

Figure 5.27: Parameter definition for task "register itinerary"

Figure 5.28: Parameter definition for task "book flight"

Figure 5.29: Parameter definition for task "prepare pay"

Note Since the variable *subTotal* is of *double* type, the XQuery function *number*() is used to extract data from the variable.

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 provides 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 [19] and Nick Russell's PhD thesis [20][1]. This work formed the basis basis for the design and implementation of the resource perspective in YAWL 2.0.

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 [17] 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 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

---

[1]For a discussion of how the workflow resource patterns have been realised in newYAWL, see appendix A.3 starting on page 373 of [20].

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 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   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.

- **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 web forms of the Resource Service. The parameter value specifies the number of minutes to wait between refreshes.

- **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.

- **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.

- **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.

6.2   Logging On

To log on to YAWL:

1. Start the YAWL engine by choosing the option "Start Engine" from the YAWL program menu;

2. Start the Resource Service 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 logon (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 buttons, as shown in Figure 6.1; ordinary participants and participants with some extra privileges will see a subset of those buttons when they log on. The "admin" userid is a 'special' logon, which can be used for administrative tasks only – 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.

6.3   Administration

A workflow administrator can load new workflow specifications, can start cases for them, can register or remove custom services, and add and manage participants, roles, positions and organisational groupings. In this section we will explore how these functions are achieved.

6.3.1   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 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 at the bottom of the screen.

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.

Note that trying 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,
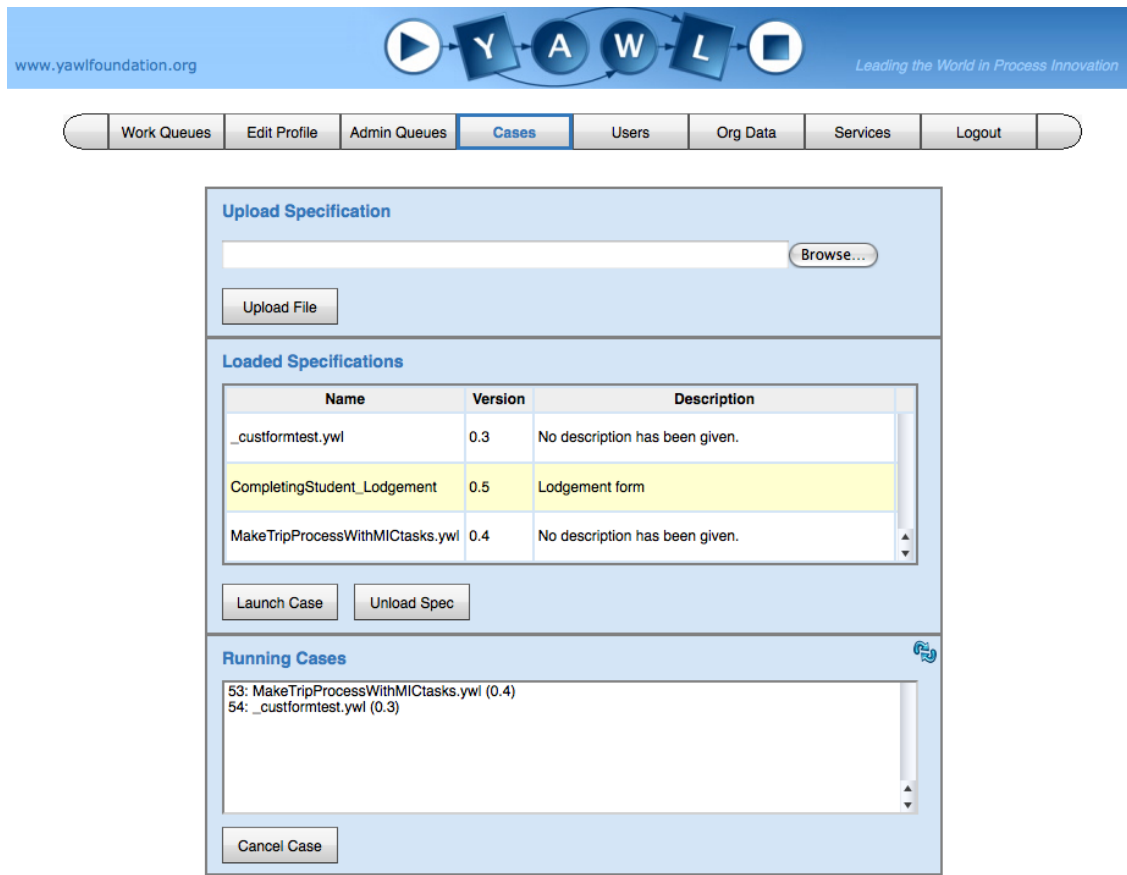
Figure 6.1: Case Management

but there are still cases running against the older version), but new cases may only be launched for the latest version uploaded.

To unload a specification from the Engine, simply 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.

Finally, the *Running Cases* panel show 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.

6.3.2   Admin Queues

An administrator can view all of the various work items that are currently active and their status in the Admin Queues Screen (see Figure 6.2). 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.

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
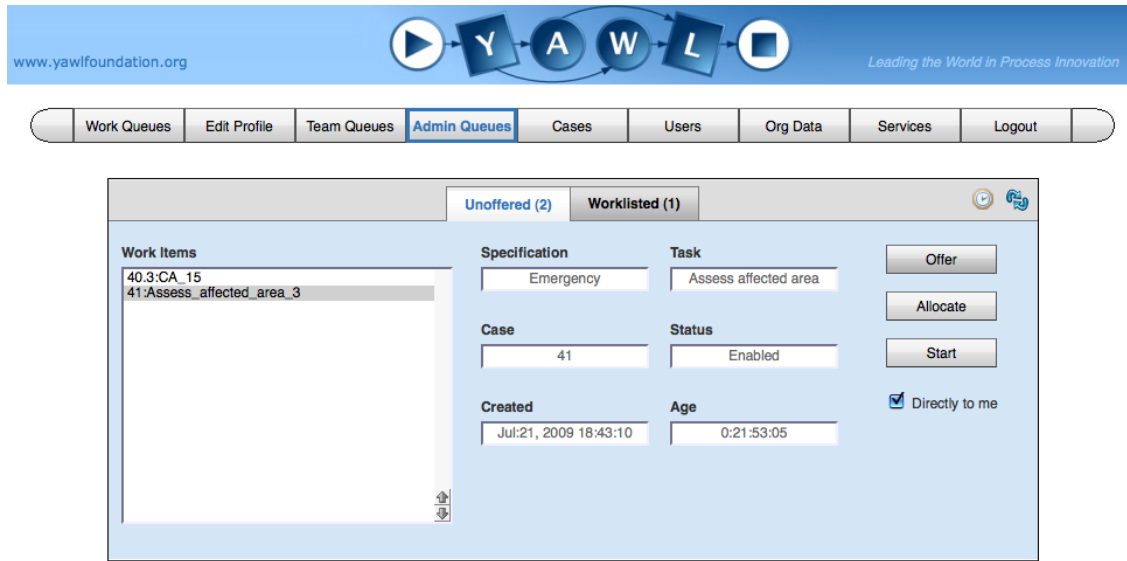
Figure 6.2: Administration Queues

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 the 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, so may occasionally be of some benefit to long-running instances.
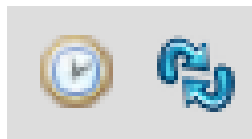
Figure 6.3: Admin Queue Tool Buttons: Synch (left), Refresh (right)

6.3.3   Service Management
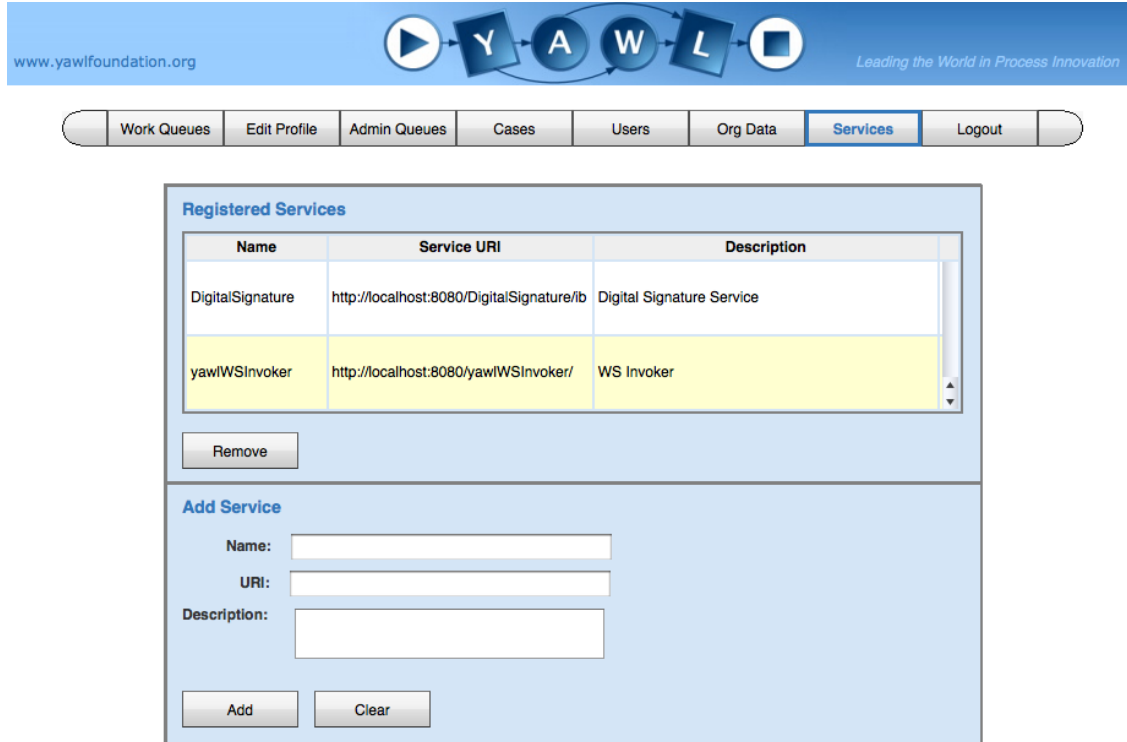
Figure 6.4: Service Management

The Service Management Screen can be used to add to to remove registered services. In the screen shot of Figure 6.4, two services are already registered, the *Digital Signature Service* and the *Web Service Invoker Service*.

A new service can be added by providing a Name, a URI and a Description. While the URI is validated by contacting it and waiting for an appropriate response, care should be taken that the URI provided exactly matches that of the specified service. By using the Editor, when connected to the Resource Service (see Section 4.6.2), such services can then be assigned to tasks (see Section 4.6.3).

To deregister a service from the Engine, select it in the list, then click the *Remove* button.

6.3.4   Managing Organisational Data

Roles, capabilities, positions and organisational groupings can be defined through the Org Data Mgt Screen, see Figure 6.5:

- **Role**: Generally, a role is a duty or set of duties that are performed by one or more participants. For example, 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.
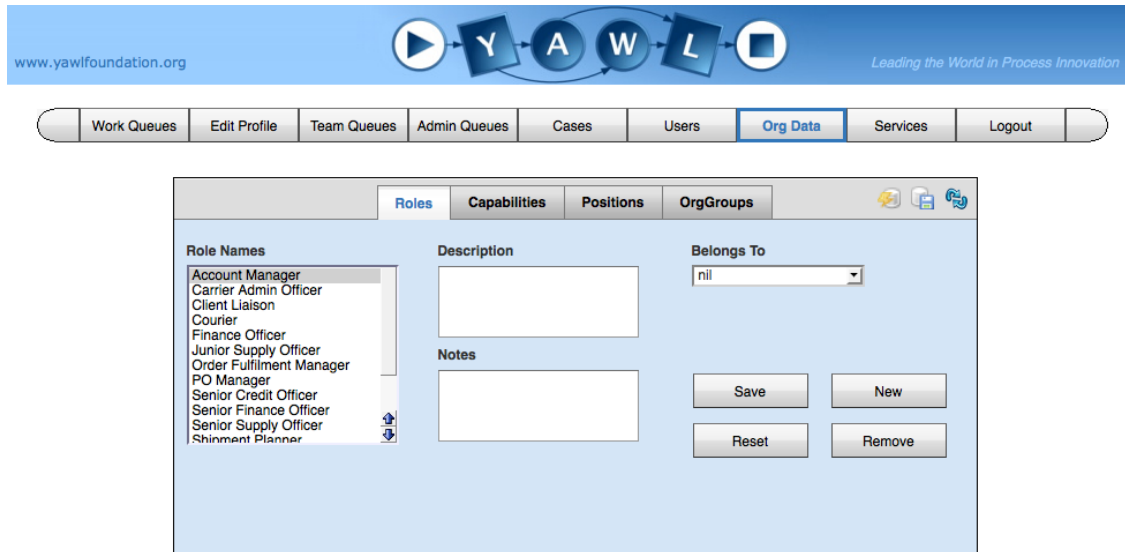
Figure 6.5: Organisational Data Management

- **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 6.4 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 (see Figure 6.6

Figure 6.6: Adding a New Role

for an example on the Role tab-page). 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 description, note, belongs to and/or reports to, then click the *Save* button.

- To **Delete** an entity, select it from the list and click the *Remove* button.

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 6.7).

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 6.8.

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 in XML format (passwords are encrypted).

Figure 6.7: The Positions tab-page

Figure 6.8: Org Data Form Tool Buttons

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.

6.3.5 Managing Users

Though the User Mgt Screen (see Figure 6.9), an administrator can add participants and change details and privileges for existing participants.

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.

Figure 6.9: User Management

- 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 6.4 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.

6.3.6 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 4.10) *on an individual task basis*. The relevant wizard panel is re-shown in Figure 6.10, and a description of each task privilege is included here for completeness.

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:

- **Can Suspend**: When granted, allows a participant to suspend the execution of a work item after it has been started.

- **Can Reallocate Stateless**: 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 reset to the values held when the work item was first started.

- **Can Reallocate Stateful**: 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.

- **Can Deallocate**: 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.

- **Can Delegate**: 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.
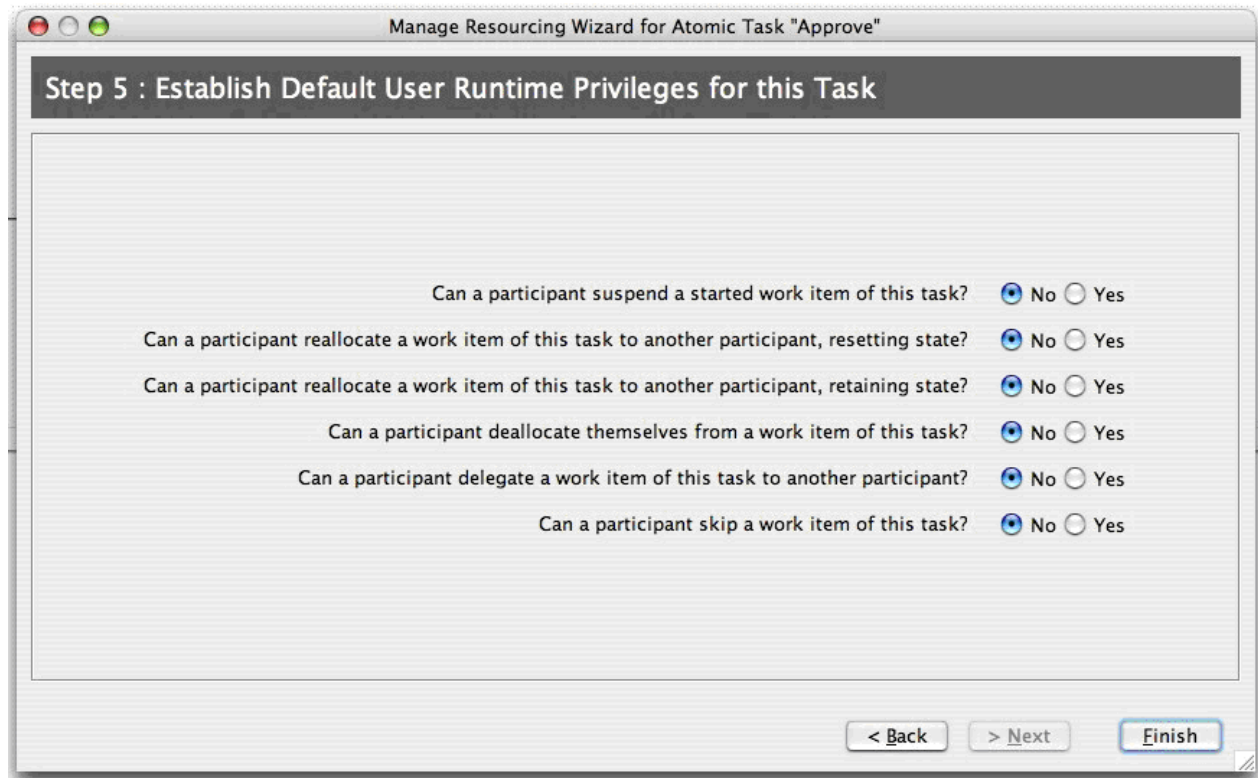
Figure 6.10: Step 5 of the resource management wizard

- **Can Skip**: When granted, allows a participant to have the execution of a work item skipped – that is, immediately completed without performing its work.

- **Can Pile**: 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.

6.4   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 6.11. 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.

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 6.12). 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. 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. Each work queue may be selected by clicking on the appropriate tab.
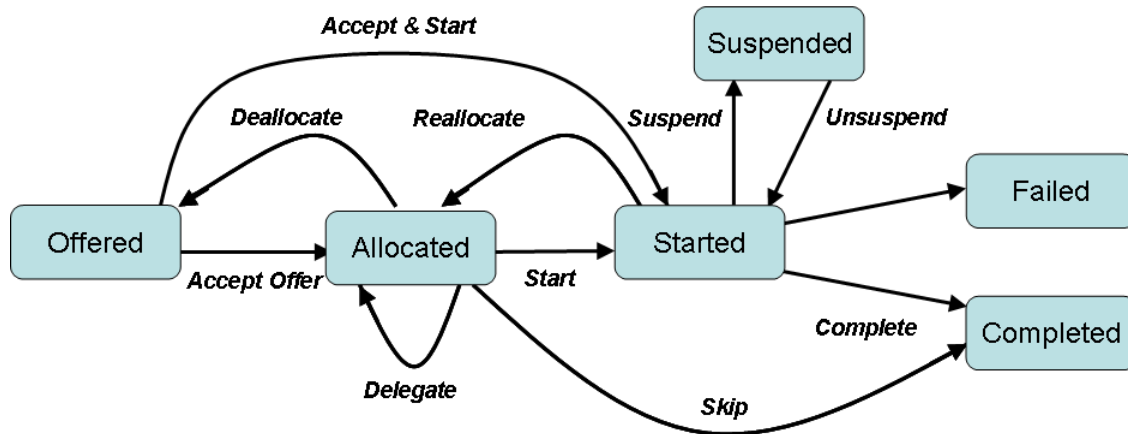
Figure 6.11: Part of the Life-cycle of a Work Item

6.4.1   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.
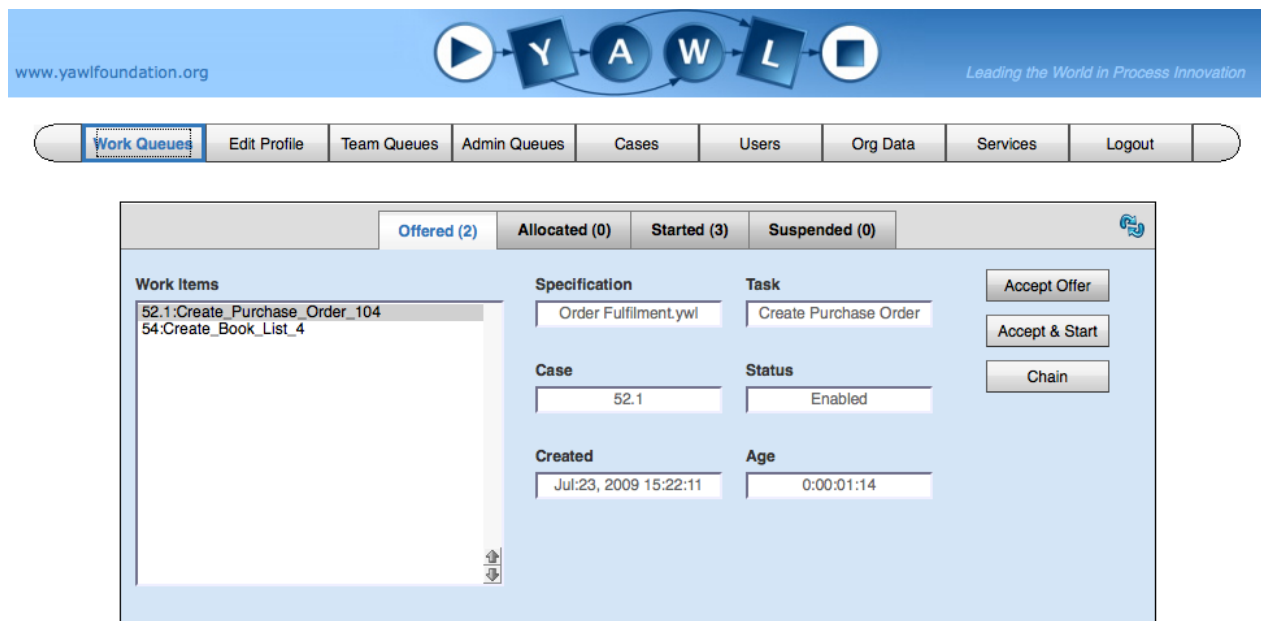
Figure 6.12: The Offered Work Queue

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.

6.4.2   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.
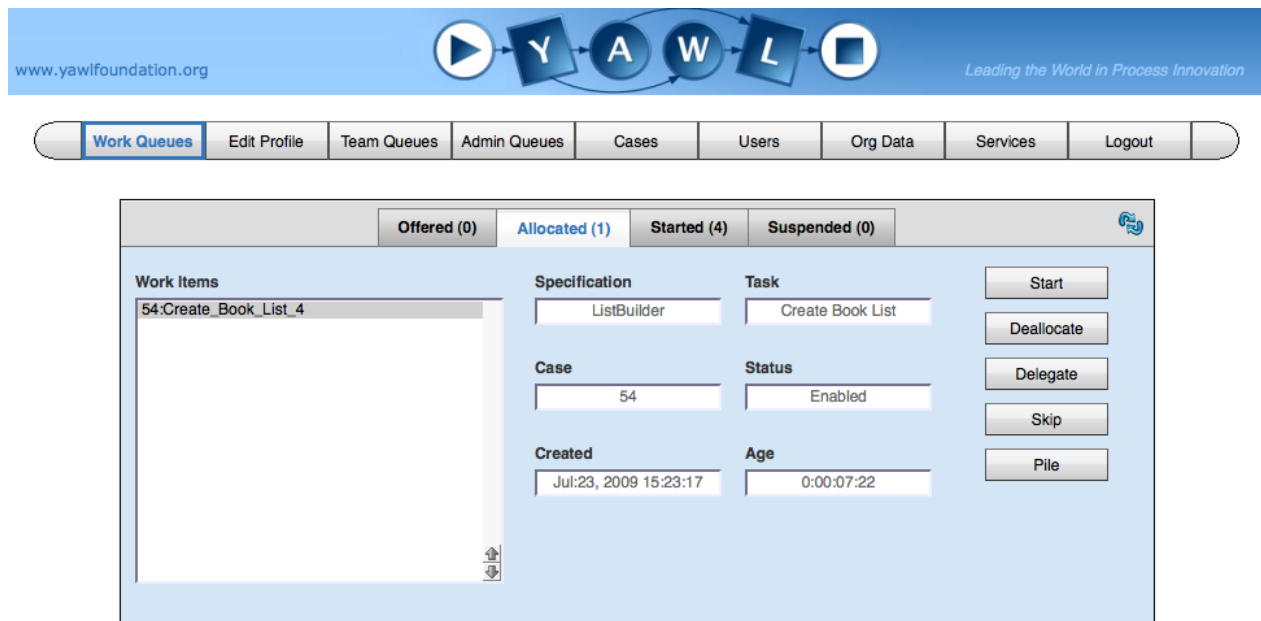
Figure 6.13: The Allocated Work Queue

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.

6.4.3 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 6.14: 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 6.15) 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

Figure 6.15: An Example of a Dynamically Generated Form

the 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 6.14), and becomes enabled after the first view/edit of the work item.

6.4.4   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.
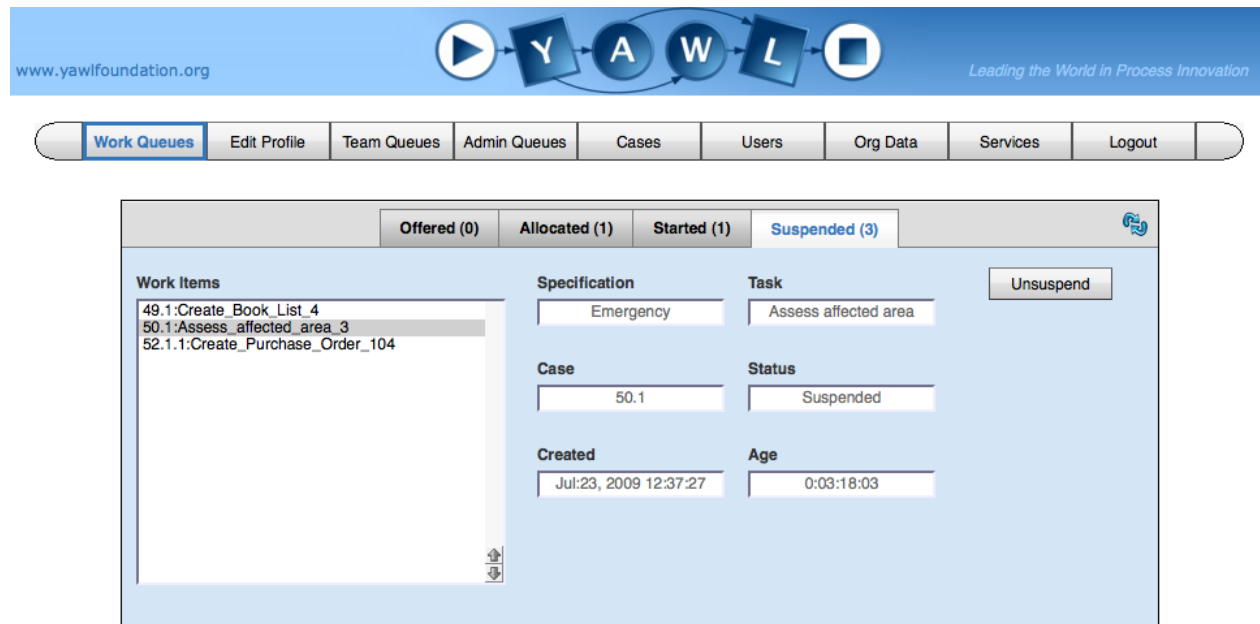
Figure 6.16: The Suspended Work Queue

6.5 User Profiles
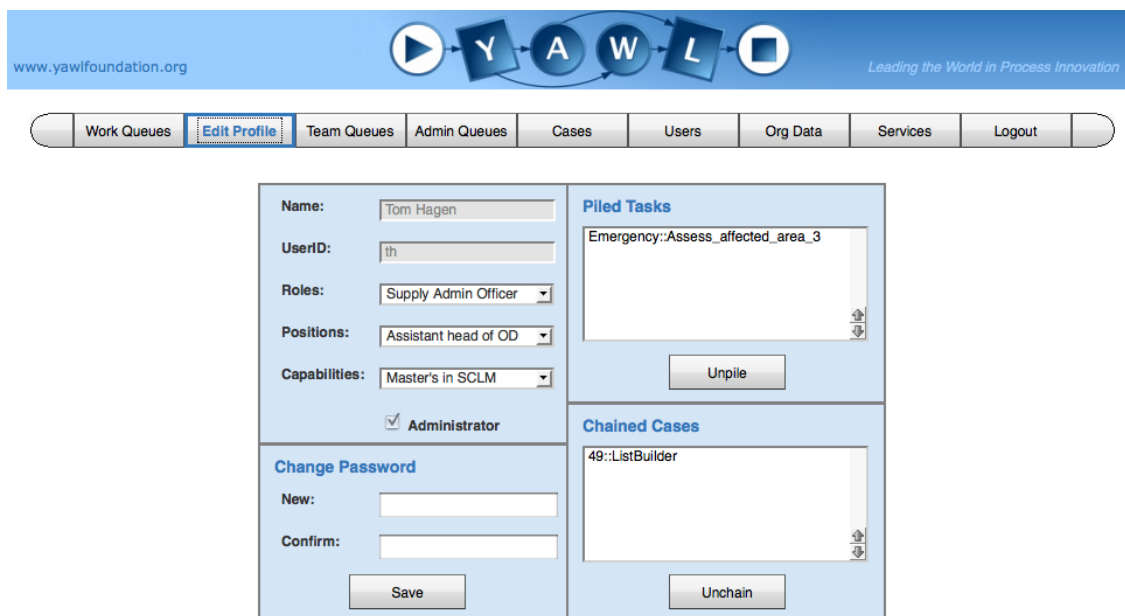
Figure 6.17: The Edit Profile Screen

The *Edit Profile* screen consists of four panels (Figure 6.17):

- 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.

6.6   Team Queues
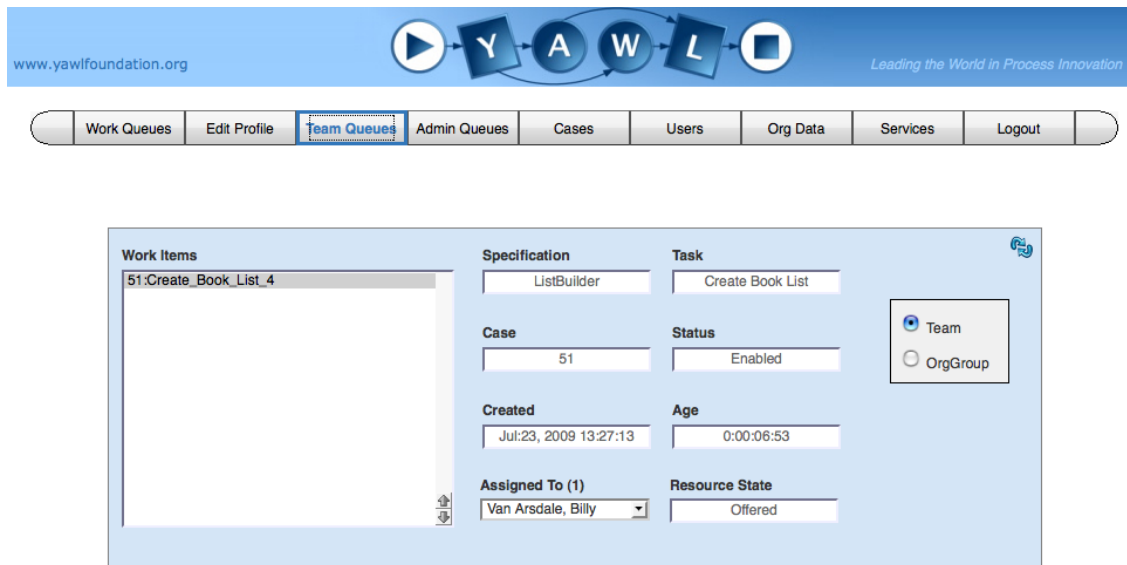
Figure 6.18: The Team Queues Screen

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 6.18 the participant listed holds the position 'CD clerk', which reports to the position of the currently logged on participant, 'Head of CD'.

- 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.

6.7   YAWL Worklist iGoogle Gadget

A simplified version of the YAWL worklist is offered as an iGoogle Gadget, so that users can see the workitems currently in their work queues as a 'live feed', along with the information in the other gadgets they may have

enabled on their iGoogle page.

6.7.1 Setting up the Worklist Gadget

The first step in setting up the gadget is to create a personal iGoogle page. To do so, browse to Google, then click the *iGoogle* link in the top right corner of the page. If you have not yet done so, you will be asked to create an account.

When you have your iGoogle page loaded, click on the *Add Stuff* link on the right of the page header. You will see a page with a list of popular gadgets to choose for your page (you must be connected to the internet to access an iGoogle gadget). To load the 'YAWL Worklist' gadget:

1. Click the *Add feed or gadget* link on the bottom of the list of links on the left frame of the page (under 'Sort By' and 'Narrow by Category' link lists).

2. In the input box that appears (Figure 6.19) enter the URL: http://www.yawlfoundation.org/gadgets/igYWorklist.xml

3. Click the *Add* button.

4. Click *OK* on the warning dialog that appears.

Figure 6.19: Adding the Worklist Gadget to an iGoogle Page

When you go back to the iGoogle home page, the YAWL Worklist should appear (Figure 6.20), but before it can be used, some settings must be entered for it:

- **UserID** *(required)*: A valid YAWL user account id;

- **Password** *(required)*: The password for the nominated YAWL account. Note that the password is retained as plain text, so please ensure you log out of iGoogle before anyone else uses the same computer.

- **YAWL Base URI** *(required)*: The URI of the Tomcat container running the YAWL Resource Service that will recognise the userid and password entered. The YAWL URI may be local (on the current computer) or remote (accessible via the internet).

- **Name** *(optional)*: Any text entered here will display on the titlebar of the gadget, after the 'YAWL Worklist:' title.

- **Alt Tab Names** *(optional)*: By default, the names shown in the worklist's tabs for each queue are 'Offered', 'Allocated', 'Started' and 'Suspended'. Choosing the *Alt Tab Names* option will instead show them as 'Available', 'Assigned', 'In Progress' and 'Suspended'.

- **Show Suspended** *(optional)*: If this option is unselected the 'Suspended' queue will be hidden. Note that, due to the width limits of iGoogle gadgets, when the 'Suspended' queue is shown, the number of work items in each queue cannot be displayed wth the tab title.
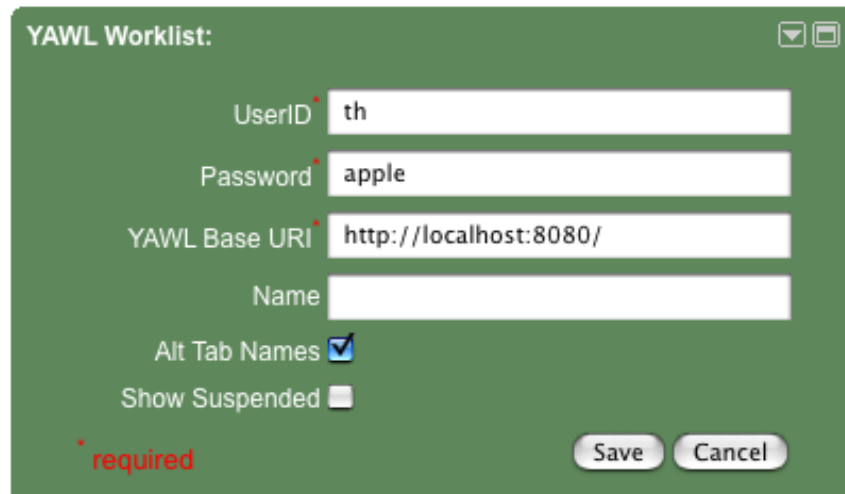
Figure 6.20: The Worklist Gadget's Settings dialog

Click the *Save* button to save your settings. If they are valid, you will see a worklist something like the one in Figure 6.21. You can modify these settings at any time by clicking on the dropdown menu on the top-right of the gadget's titlebar (the small button with the dropdown arrow) and choosing *Edit Settings*.
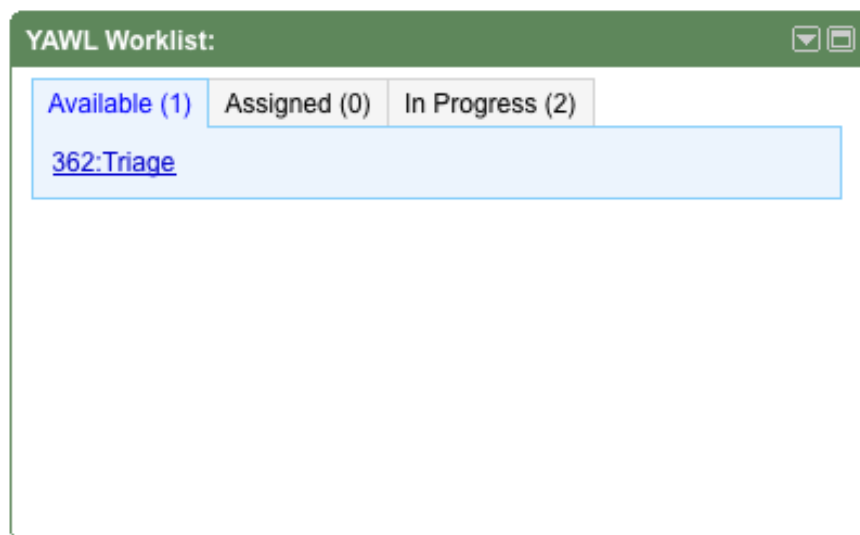
Figure 6.21: The Worklist Gadget

6.7.2   Using the Worklist Gadget

Each queue shows the work items available to the user specified as hyperlinks. When you click on a work item link in *any* queue, the data form for the work item will display on a new tab or window (depending on browser settings). That is, when a work item link is clicked:

- If the work item is on the Offered (Available) queue, it is immediately allocated to you and started, then its form is displayed;

- If the work item is on the Allocated (Assigned) queue, it is immediately started, then its form is displayed;

- If the work item is on the Started (In Progress) queue, its form is displayed;

- If the work item is on the Suspended queue, it is immediately unsuspended, then its form is displayed.

Just as it is for the default worklist handler of the Resource Service, if the work item was associated with a custom form at design time, then the custom form will display; if not, then a dynamically generated form will appear. You do not have to be logged on to YAWL before you click on a work item link in the gadget.
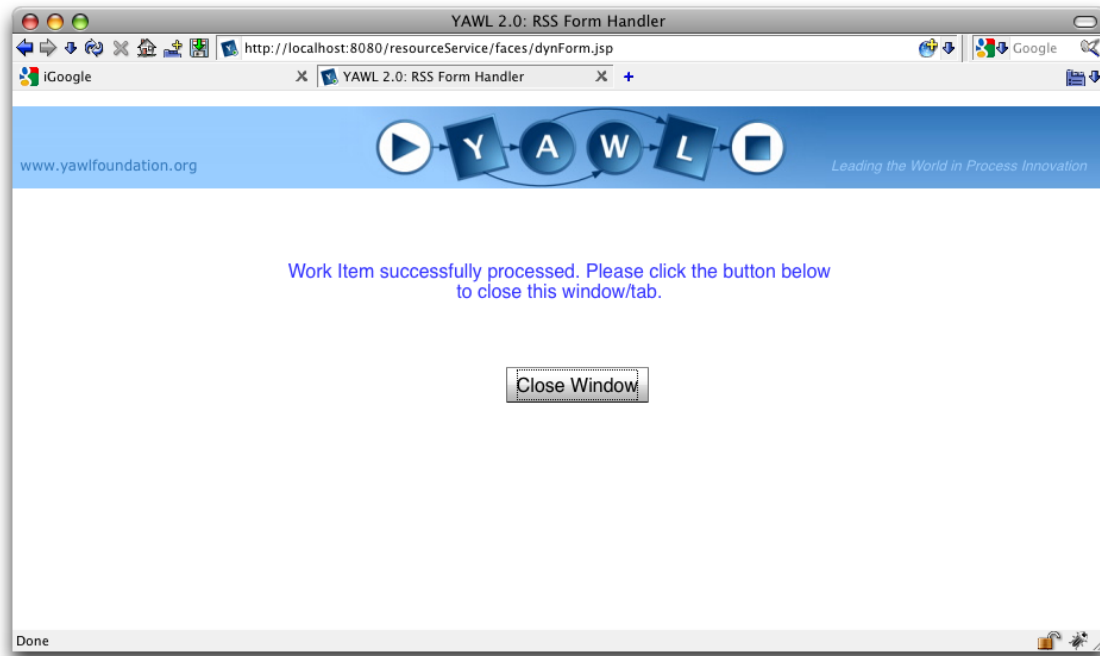
Figure 6.22: The Worklist Gadget's Post-Form-Display Message

When you leave the form (whether by a Cancel, Save or Complete action) a message will appear on the page informing of the success or otherwise of the action taken (Figure 6.22). Note that your current logon status is unaffected by actions you take on the gadget – if you were already logged on to YAWL in another tab or window, that session will remain active after the action completes; if you were not already logged on, you will automatically be logged on to enable the processing of the work item, then logged out again when the processing completes. Close this page to return to the iGoogle tab/window, and refresh the iGoogle page to show any newly available work items or queue changes caused by your actioning of the work item.

Chapter 7

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 manual from start to finish the first time it is read. This manual 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 technical manual. A more concise version of the worklets approach can be found in [9], while a more concise version of the exlets approach can be found in [8]. The ultimate reference on worklets and exlets is Michael Adams' PhD thesis [7]. All these publications can be downloaded from www.yawlfoundation.org.

All of the example specifications, rule sets, and so on referred to in this manual can be found in the "worklet repository" distributed with the service as part of the YAWL 2.0 release.

This icon indicates a hands-on method or instruction.

7.1 What is a Custom YAWL 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 to be completed, and proceed with the execution.

7.1.1 What is the YAWL Worklet Service?

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.

7.1.2   The Selection Service

The Worklet Dynamic Process Selection Service (or *Selection Service*) enables flexibility by providing a process designer with the ability to substitute a workitem in a YAWL process at runtime with a dynamically selected "worklet" - a discrete YAWL process that acts as a sub-net for 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 base 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.

7.1.3   The Exception Service

During every instance of a workflow process, certain things happen "off-plan". That is, it doesn't matter 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, 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 but 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.

7.2 Installation

7.2.1 Worklet Installation Package

The Worklet Service is distributed as a standard component of the YAWL environment, and so is included in each of the various installer packages, and the **YAWL2.0_CoreWebServices.zip** file used for manual installations. Specifically, the service is contained in the file **workletService.war** - this is the worklet web application component. When unpacked, *workletService.war* also contains the **worklet repository**, a required set of directories where worklets and rules files are stored, logs are written to and so on, and a number of sample worklets and rule sets (including all those discussed in this manual) are located.

The structure of the repository is shown in Figure 7.1.
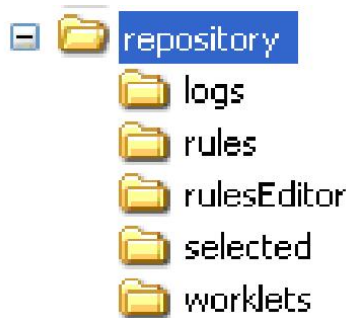
Figure 7.1: Worklet Repository Folder Structure

- The *logs* folder is where the Service writes certain log files during its operation. In particular, the *eventLog.csv* file logs the key events of the service (when database persistence is disabled). All are plain text files.

- The *rules* folder contains the sets of rules used during the selection and exception handling processes. Rules files have an XML format. Each YAWL specification that uses the service will have a corresponding rules file of the same name, except with an extension of *xrs* (XML rule set).

- The *rulesEditor* folder contains a tool that is used to manage and modify rule sets (see Section 7.5.2 for a complete description of the Rules Editor).

- The *selected folder* contains a set of files, in XML format, that are essentially log files that capture the results of each selection process (either via the Selection Service, or through the selection of a compensating worklet via the Exception Service). These files have an *xws* extension (XML Worklet Selection). These files are used by the Rules Editor to enable the addition of new rules for a specification.

- The *worklets* folder contains the worklet specification files. These files are YAWL specifications that are run as required by the service.

Several of the folders contain examples.

7.2.2 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 2.5 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.

Once the service is fully extracted and installed, there are a couple of minor configuration tasks to complete.

Open the Worklet Service's *web.xml* file (in folder \webapps\workletService\WEB-INF\) in any text editor.

By default, the worklet repository is located within the service's own file structures. However, you may change the location of the repository if you wish. Locate the parameter named *Repository*, and change its *param-value* to the path where you have relocated the worklet repository. The value should include the folder 'repository' and end with a slash, as shown in Figure 7.2[1].

```
<context-param>
    <param-name>Repository</param-name>
    <param-value>${catalina.base}\webapps\workletService\repository\</param-value>
    <description>
        The path where the worklet repository is installed.
    </description>
</context-param>

<context-param>
    <param-name>EnablePersistence</param-name>
    <param-value>false</param-value>
    <description>
        'true' to enable persistence and logging
        'false' to disable
    </description>
</context-param>
```

Figure 7.2: The Worklet Service's *web.xml* file (detail)

If you have enabled database persistence for the YAWL Engine (if necessary, see Chapter 2, Section 2.5 for details about enabling persistence for the YAWL Engine), then persistence should also be enabled for the Worklet Service (so that case data for running processes can be persisted across Tomcat sessions). To enable persistence, change the *param-value* to **true** for the *EnablePersistence* parameter.

Save and close *web.xml*. Worklet Service configuration is now complete. However, by default the Selection Service is enabled within the YAWL Engine, but the Exception Service is not. If you wish to enable the Exception Service, proceed to Step 3 below.

Enabling the Worklet Exception Service

When YAWL 2.0 is installed, the Exception Service is configured as disabled in the YAWL Engine. To enable the Exception Service, a parameter has to be set in the YAWL Engine's *web.xml* file.

Open the engine's *web.xml* file (in folder \webapps\yawl\WEB-INF\). Locate the parameter named *Enable-ExceptionService*; to enable the Exception Service, change the *param-value* to **true** (see Figure 7.3). Save and close *web.xml*.

The Exception Service uses extensions (or 'hooks') in the YAWL default worklist handler (a component of the Resource Service) to provide methods for interacting with the Worklet Service, so if you have enabled the service in the Engine as above, you also need to enable the extensions in the Resource Service.

To do so, locate and open the Resource Service's *web.xml* file (in folder \webapps\resourceService\WEB-INF\).

Locate the context parameter named *InterfaceX_BackEnd*. By default, the entire parameter block is commented out. To enable the Exception Service extensions to the worklist, simply remove the comment tags (the "<!–" and "–>" surrounding the *context-param* block – see Figure 7.4 below). Save and close *web.xml*.

---

[1]${catalina.base} is an environment variable pointing to the path of your Tomcat installation.

```
<!-- PARAMS FOR EXCEPTION SERVICE -->

<context-param>
   <param-name>EnableExceptionService</param-name>
   <param-value>true</param-value>
   <description>
  Set this value to 'true' to enable monitoring by an
        Exception Service (specified by the URI param below).
  Set it to 'false' to disable the Exception Service.
   </description>
</context-param>
```

Figure 7.3: The YAWL Engine's *web.xml* file (detail)

```
<!-- 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 7.4: The Resource Service's *web.xml* file (detail)

The Exception Service is now fully enabled and operational. For information on how the Exception Service works and how to use it, see Sections 7.3 and 7.6.

Installing the Rules Editor

The Worklet Rules Editor allows you to browse the rule sets of specifications, add new rules to existing rule sets, and add complete, new rules trees to rule sets. It is a .NET based application, so has the following requirements:

- Operating System: Windows OSs 98SE or better

- The Microsoft .NET framework (any version). If you don't have the framework installed, it can be downloaded for free from Microsoft: http://www.microsoft.com/downloads.

The Rules Editor tool is found in the *rulesEditor* folder of the worklet repository. It can be executed directly from there - no further installation is required.

For more details on the Rules Editor and how to use it, refer to Section 7.5.2.
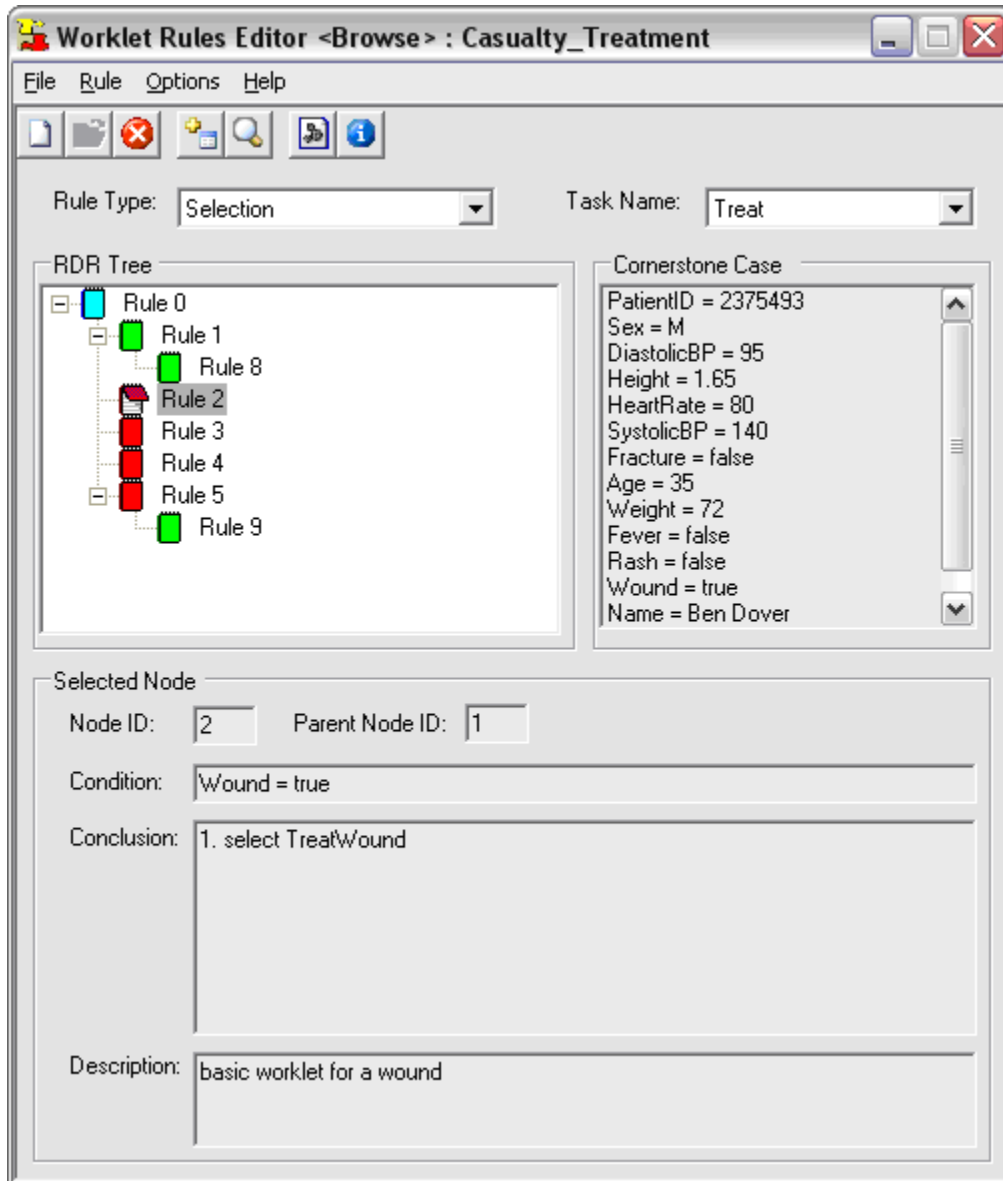
Figure 7.5: The Worklet Rules Editor

7.3   Using the Worklet Selection Service

Fundamentally, a worklet is nothing more than a workflow specification that has been designed to perform one part of 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 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 which will act as the top-level (or manager or parent) specification, and one or more worklets which will be dynamically substituted for particular top-level tasks at runtime.

The YAWL Editor is used to create both top-level 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 7.2 of this chapter and/or Chapter 2 for more information).

First, a top-level specification needs to be defined.

7.3.1 Top-level or Parent Specifications

To define a top-level 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.

For example, Figure 7.6 shows a simple specification for a *Casualty Treatment* process. In this process, we want the *Treat* task to be substituted at runtime with the appropriate worklet based on the patient data collected in 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.

Worklets may be associated with an atomic task, or a multiple-instance atomic task. Any number of worklets can be associated with (i.e. comprise the repertoire of) an individual task, and any number of tasks in a particular specification can be associated with the Worklet Service.
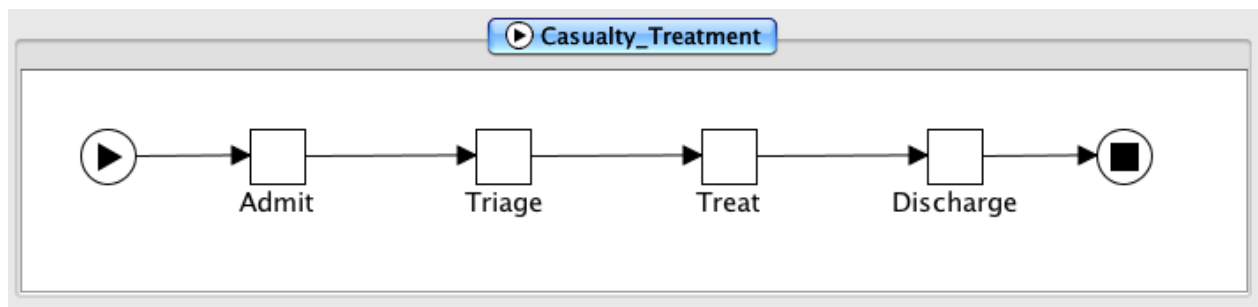
Figure 7.6: Example Top-level Specification

Here, we want to associate the Treat task with the Worklet Service. To do so, right click on the task, then select *Task Decomposition Detail* from the popup menu. The *Update Task Decomposition* dialog is shown (Figure 7.7). This dialog shows the variables defined for the task - each one of these 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 this task. The result is that 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 manual.

The list of task variables in Figure 7.7 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. The last three variables are defined as 'Input & Output', so that the worklet can "return", or map back to these variables, data values that are captured during the worklets execution.

The dialog has a section at the bottom called *YAWL Registered Service Detail*. It is here that the task is associated with the Worklet Service by choosing the Worklet Service from the list of available services. Note that list of services will only be seen if the Engine is currently running and it has services installed.

Select the Worklet Service from the list. That's all that is required to make the top-level specification worklet-enabled. Next, we need to create one or more worklet specifications to execute as substitutes for the worklet-enabled task.
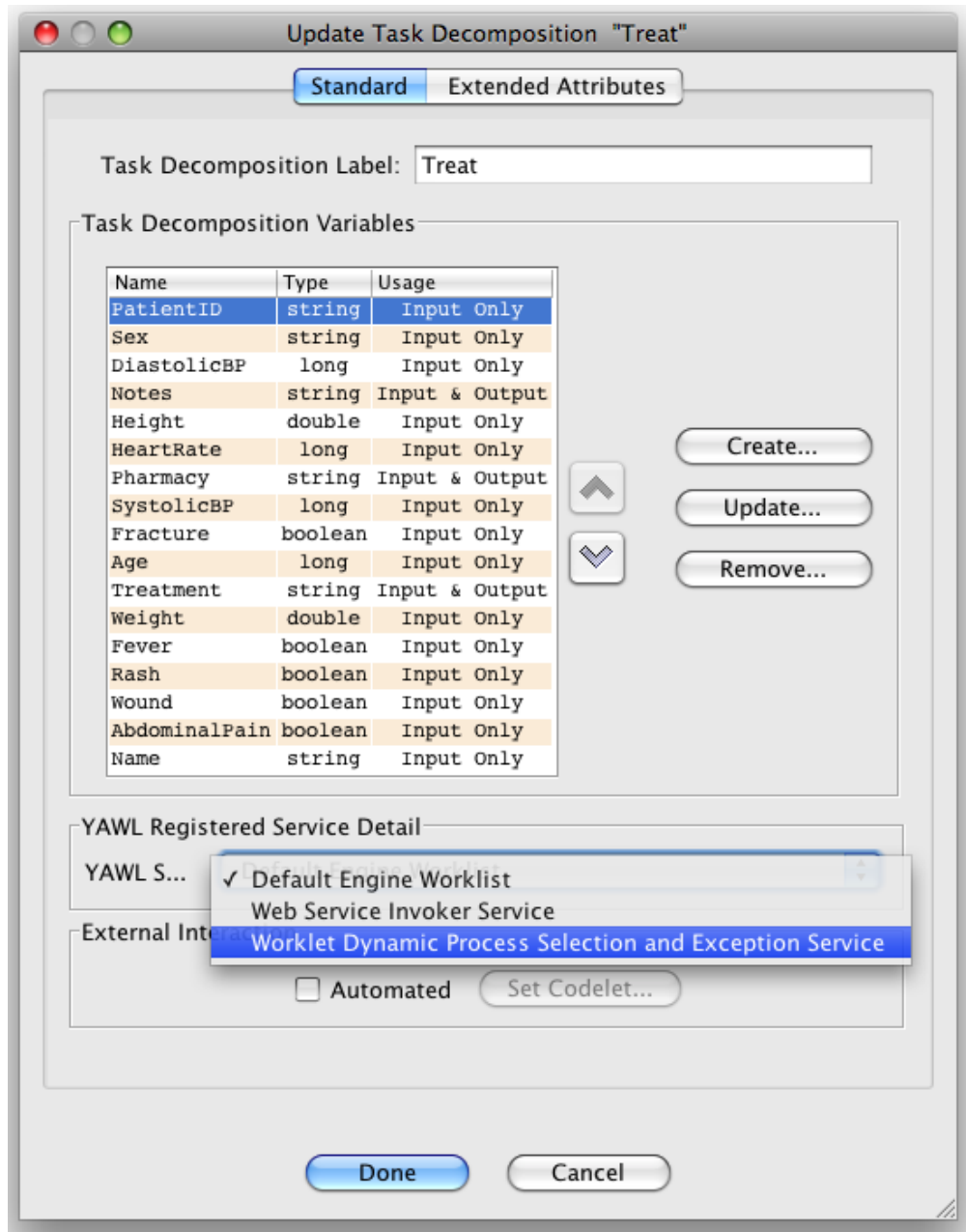
Figure 7.7: Associating a task with the Worklet Service

7.3.2   Worklet Specifications

When the *Casualty Treatment* top-level specification is executed, the YAWL Engine will notify the Worklet Service when the worklet-enabled *Treat* task becomes enabled. 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 top-level 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. Each of the data variables that are required to be passed from the parent task to the

worklet specification need to be defined as net-level variables in the worklet specification.

Figure 7.8 shows a simple example worklet to be substituted for the Treat top-level task when a patient complains of a fever.
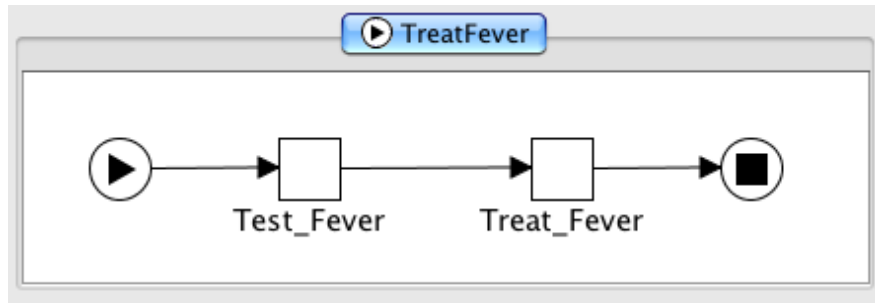
Figure 7.8: The *TreatFever* Worklet

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 is a standard YAWL specification and as such may be executed directly by the YAWL engine without any reference to the Worklet Service.

As mentioned previously, those data values that are required to be mapped from the parent task to the worklet need to be defined as net-level variables in the worklet specification. Figure 7.9 shows the net-level variables for the *TreatFever* task.
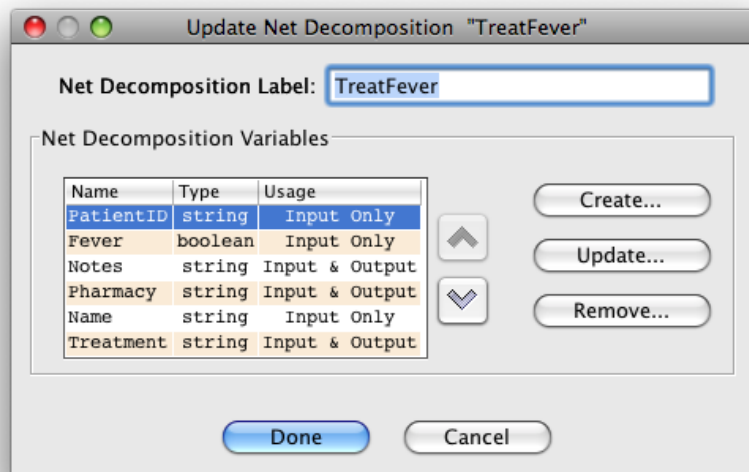
Figure 7.9: Net-level Variables for the *TreatFever* Specification

Note the following:

- Only a sub-set of the variables defined in the parent *Treat* task (see Figure 7.7) 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 *Usage* 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 *Usage* of 'Output Only' or 'Input & Output' will pass their data values back to the parent task when the worklet completes.

In Figure 7.9, 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; the *Notes*, *Pharmacy* and *Treatment* variables will receive values during the execution of the worklet and will map those values back to the top-level *Treat* task when the worklet completes.

The association of tasks with the Worklet Service is not restricted to top-level 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.

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 7.5.2.

7.4 Using the Worklet Exception Service

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 Exception Service leverages off the worklet framework to also provide support for the myriad exceptions that may occur during the execution of any process instance.

Every process instance, no matter how rigidly structured, 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 may be never even imagined could occur). In any case, when they do happen, if they are not part of the process model, 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 aborted.

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 when the next unexpected exception occurs.

The Exception Service addresses these problems by allowing you to define exception handling processes (that may include worklets as compensation handlers) for parent workflow instances when certain events occur. Rules are defined in much the same way as for the Selection Service, 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 the Selection Service is linked explicitly to tasks as defined in the YAWL Editor, and thus available whenever a worklet-enabled task is executed, the Exception Service is either enabled or disabled (on or off); 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. Also, the Selection and Exception Services can be used in combination within particular case instances to achieve dynamic flexibility and exception handling simultaneously.

7.4.1 Exception Types

This section introduces the ten different types of exception that have been identified, eight of which are supported by this version of the Worklet Service. Some are related, while others are more distinct. Later sections will show examples of each of these.

When the Exception Service is enabled, it is notified whenever any of these exception types occur for every process instance executed by the YAWL Engine (by various means). The Exception Service maintains a set of rules (described in detail in Section 7.5.2) 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 or after execution of that workitem or case begins. Thus, there are four types of constraint exception:

- **CasePreConstraint** - case-level pre-constraint rules are checked when each case (i.e. instance) begins execution;

- **ItemPreConstraint** - item-level pre-constraint rules are checked when each workitem in a case becomes enabled (i.e. ready to be checked out);

- **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 Exception Service receives notification from the YAWL Engine when each of these events occur, then checks the rule set for the case 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, an exception process is invoked.

Note that for each of the constraint events, an exception process is invoked for a rule when that rules condition evaluates to **true**. So, for example, if the condition of an *ItemPreConstraint* rule for a *Triage* task was "PrivateInsurance=false", and that value of that attribute in the workitem was also false, then the exception process 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 (or is linked to the deprecated Time Service) 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 itself and each of the other workitems that were running in parallel with the timed-out item. 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 program (as opposed to a human user) reports that the program has aborted before completion. This event is not supported by this version of Exception Service.

ConstraintViolation

This event occurs when a data constraint has been violated for a workitem during execution (as opposed to pre or post execution). This event is not currently supported by the Exception Service.

7.4.2   Exception Handling Primitives

For any exception event that occurs, a handling process may be invoked. Each handling process, called an **exlet**, contains a number of steps, or *primitives*, in sequence, and is defined graphically using the Worklet Rules Editor (see Section 7.5.2). 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 - un-suspends (or continues) 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 workitem execution back to start. Resets the workitems 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).

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).

An example of a definition of an exception handing process in the Rules Editor is below:
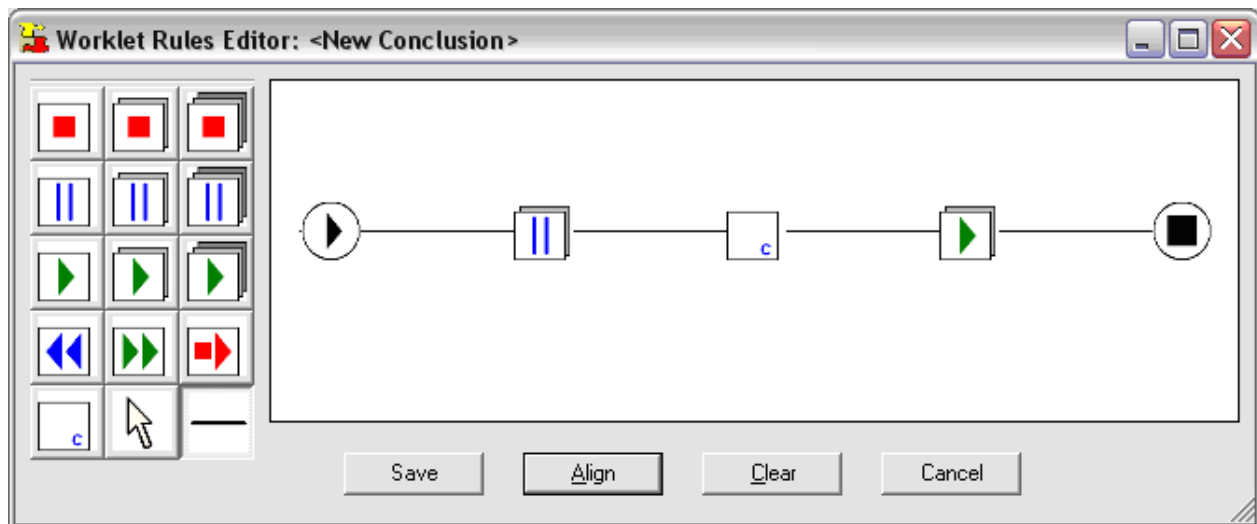
Figure 7.10: Example Handler Process in the Rules Editor

When invoked, this handler will suspend the current case, then run a compensating worklet, then continue execution of the case.

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 7.6. But before we consider the walkthroughs, we must first look at exactly how the rule sets are formed and how

they operate, and how to use the Worklet Rules Editor to manage rule sets for specifications. These topics are discussed in the next section.

7.5   Worklet Rule Sets and the Rules Editor

This section describes the structure and operation of worklet rule sets. A tool has been designed to manage the creation and modification of rule sets for specifications, called the *Worklet Rules Editor*. The structure and operation of rule sets is best described by using the Rules Editor to display and manipulate them. So a description of how to use the Rules Editor is interspersed in this section with the description of the rule sets themselves.

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 indicated.

7.5.1   Worklet Rule Sets

Any YAWL specification may have an associated rule set. The rule set for each specification is stored as XML data in a disk file that has the same name as the specification, but with an ".xrs" extension (XML Rule Set). All rule set files are stored in the *rules* folder of the worklet repository. For example, the file *Casualty_Treatment.xrs* contains the worklet rule set for the *Casualty_Treatment.yawl* YAWL process specification. Figure 7.11 shows an excerpt from that file.

A rule set for a specification consists of a collection of *rule trees*. Each rule tree represents a set of modified *Ripple-Down Rules (RDR)*, which 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 nodes condition is also evaluated. If a nodes condition evaluates to *False*, and there is a false child, then that child nodes 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 (Rule 0) of the tree is always a default node with a default True condition and conclusion, 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 parent node is an "else" rule to its parent (or an alternate to the parent rule). For example, see the selection rule tree for the *Casualty Treatment* specification (Figure 7.12). 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.

One worklet rule set file is associated with each specification, and may contain up to eleven sets of rule trees (or *tree sets*), one for selection rules 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 workitem 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 types for each specification. You only need to define rules for those types that you want to handle - any exception types that arent 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

```
        </ruleNode>
      </pre>
    </case>
  </constraints>
  <selection>
    <task name="Treat">
      <ruleNode>
        <id>0</id>
        <parent>-1</parent>
        <trueChild>1</trueChild>
        <falseChild>-1</falseChild>
        <condition>True</condition>
        <conclusion>null</conclusion>
        <cornerstone> </cornerstone>
        <description>root level default node</description>
      </ruleNode>
      <ruleNode>
        <id>1</id>
        <parent>0</parent>
        <trueChild>8</trueChild>
        <falseChild>2</falseChild>
        <condition>Fever = true</condition>
        <conclusion>
          <_1>
            <action>select</action>
            <target>TreatFever</target>
          </_1>
        </conclusion>
        <cornerstone>
```

Figure 7.11: Excerpt of Rule Set file Casualty_Treatment.xrs

added later if required (as could any of the other types not yet defined in the rule set).

Referring back to Figure 7.11, notice that the file specifies a *Selection* rule tree for the Treat task. The second *ruleNode* contains a condition "Fever = True" and a conclusion of "TreatFever". Thus, when the condition "Fever = True" evaluates to true, the worklet *TreatFever* is chosen as a substitute for the *Treat* task. Notice also that each rule node (except the first) has a parent, and may have two child nodes, a *true* child and a *false* child.

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 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. A rule set will contain one or more tree sets - one
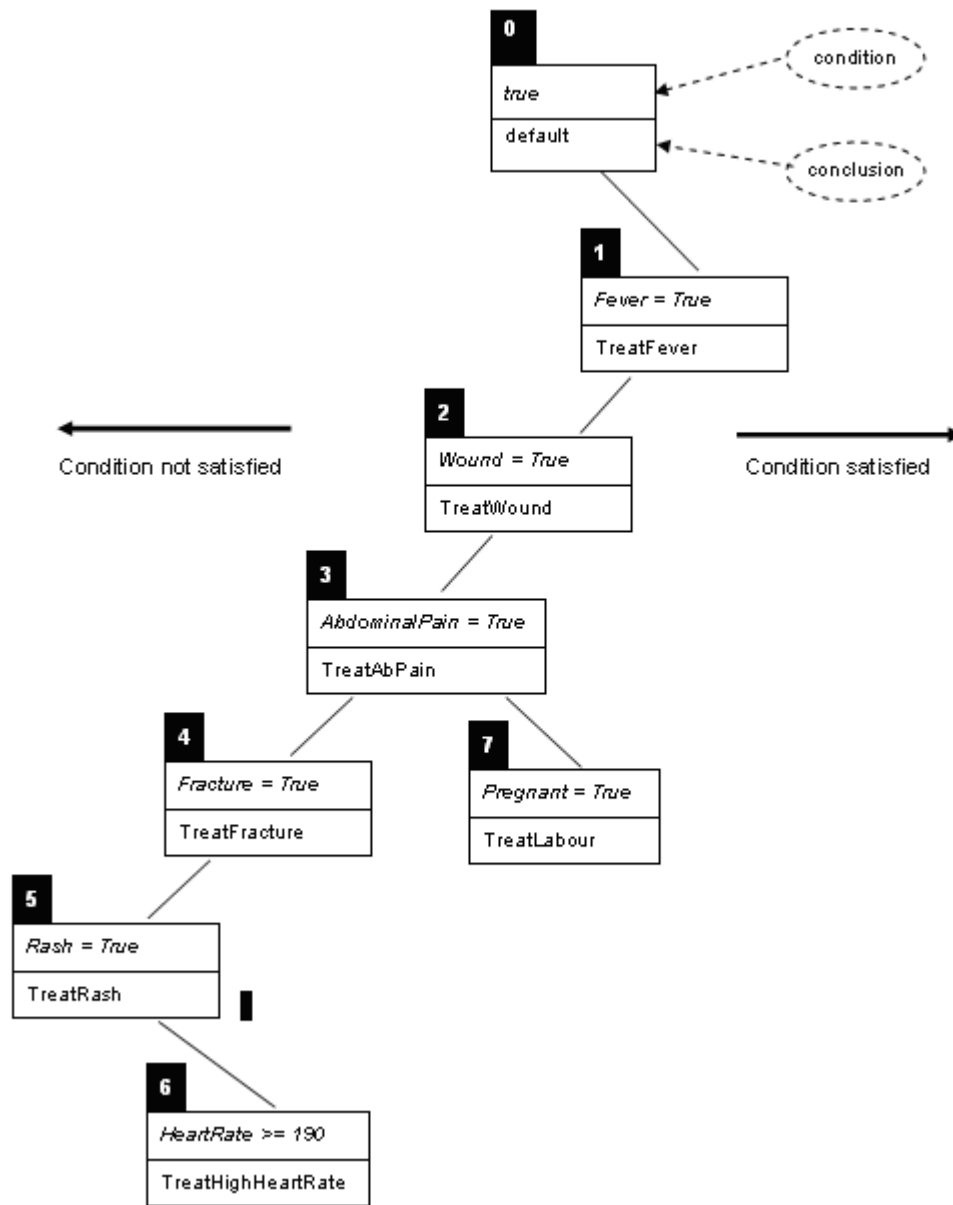
Figure 7.12: Example Rule Tree (Casualty Treatment spec)

for each rule type for which rules have been defined.

Of course, to maintain a rule set of any complexity by directly editing the XML in a rule set file would be daunting, to say the least. To make things much easier, a *Rules Editor tool* has been developed, and can be found in the *rulesEditor* folder of the worklet repository. It can be run directly from there - no further installation is required (depending on the requirements below).

7.5.2   The Rules Editor

The Worklet Rules Editor allows for the addition of new rules to existing rule sets of specifications, and the creation of new rule sets. It is a .NET based application, so has the following requirements:

- Operating System: Windows 98SE or later.

- The Microsoft .NET framework (any version). If you dont have the framework installed, it can be downloaded free from Microsoft.

When the Rules Editor is run for the first time, the following dialog shown in Figure 7.13 is displayed.
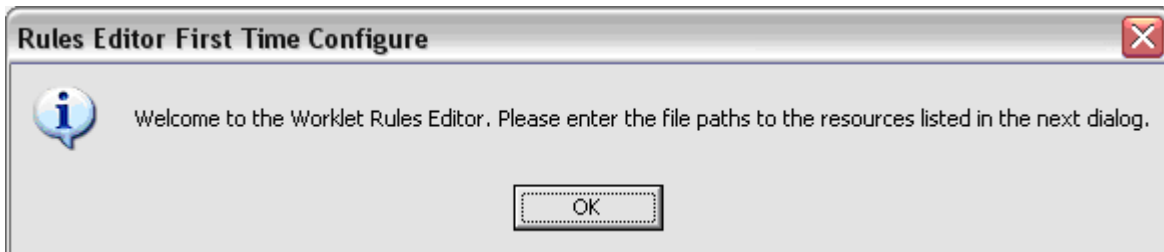
Figure 7.13: Rules Editor First Time Use Message

Clicking OK shows the *Configuration* dialog (Figure 7.14), where the paths to resources the Rules Editor uses are to be specified. Some default paths are shown, but can be modified directly or by using the browse buttons where available. he following paths must be specified:

- **Worklet Service URI:** the URI to the Worklet Service. The default URI assumes it is installed locally. If it is remote to the computer running the Rules Editor, then that URI should be entered, ensuring it ends with "/workletService".

- **Worklet Repository:** the path where the worklet repository was installed. The default path shown assumes the Rules Editor was started from the *rulesEditor* folder of the repository. If it was started from another location, specify the actual path to the repository by editing the path or browsing to the correct location.

- **Specification Paths:** the path or paths to locations on the local computer where YAWL specification files may be found. The rules editor will search each of the paths provided for specifications for which rule sets may be created. Multiple paths can be provided, separated by semicolons ';'

- **YAWL Editor:** the path and filename for the YAWL Editor.
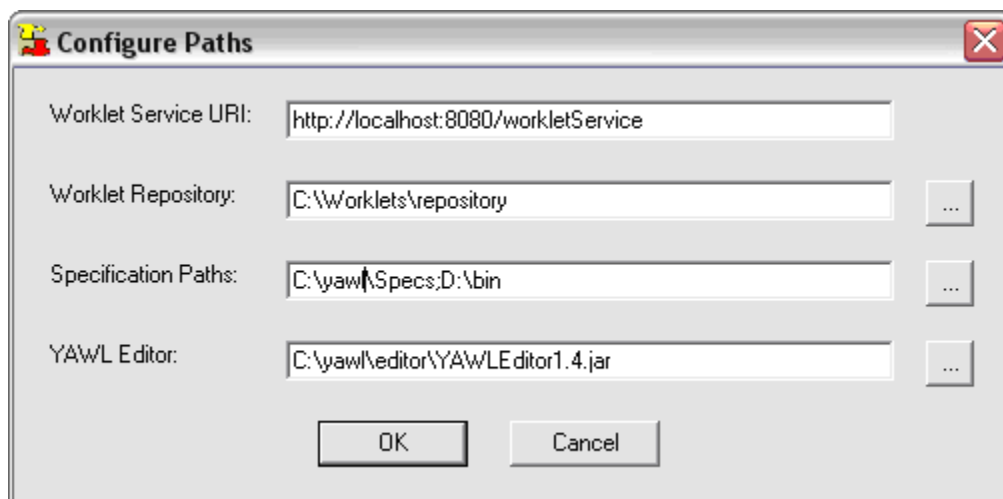
Figure 7.14: The Rules Editor Configuration Dialog

Some checks will take place to make sure the paths are valid and you will be asked to correct any that are not. Once the configuration is complete, the main screen will appear. This screen allows you to view each node of each rule tree defined for a particular specification. From this screen you can also add new rules to the current rule set, create new tree sets for an existing rule set, and create entirely new rule sets for new specifications.

7.5.3   Browsing an Existing Rule Set

To load a rule set into the Rules Editor, click on the *File* menu, then select *Open...*, or click on the *Open* toolbar button. The File Open Dialog will open with the *rules* folder of the repository selected. Select the file you wish to open, and then click OK.

Figure 7.15 shows the main screen with the rule set for the *Casualty Treatment* specification loaded. On this screen, you may browse through each node of a rule tree set and view the various parts of each node. The main features of the screen are explained below.
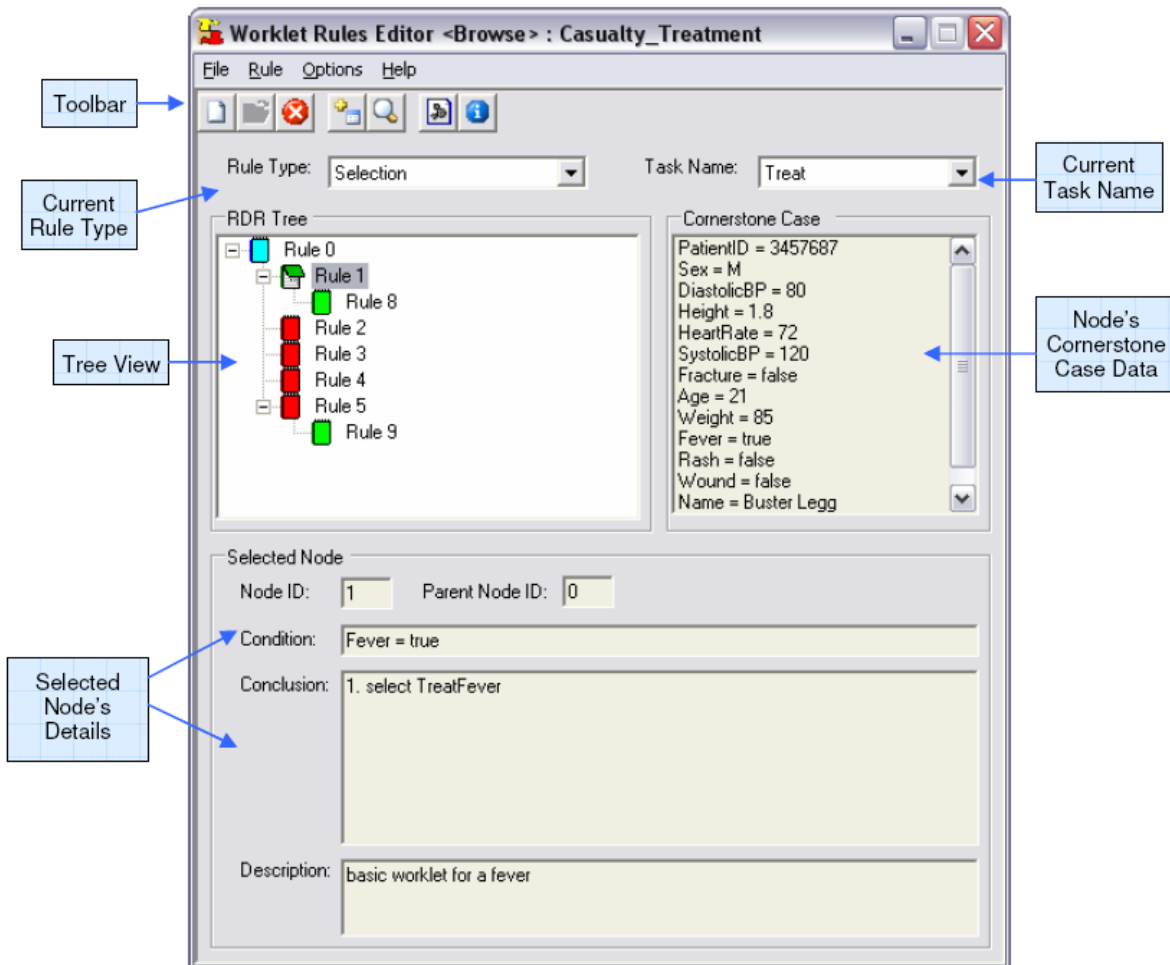
Figure 7.15: Rules Editor Main Screen

The Toolbar

The toolbar buttons replicate the functions available from the main menu see Figure 7.16.
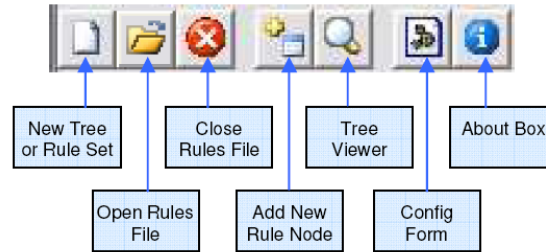
Figure 7.16: Toolbar

- **New Tree or Rule Set:** If there is no rules file currently open in the Editor, this button displays the *New Rule* form to allow the creation of an entirely new rule set file for a specification (i.e. one that does not yet have a rule set file defined). If there is a rules file currently open in the Editor, displays the *New Rule* form to allow the addition of new tree sets to the opened rule set file (for rule types that have not yet been defined for that specification). See the *Creating a New Rule Set and/or Tree Set* section below for more details.

- **Open Rules File:** Opens an existing rules file for browsing and/or editing. The title bar shows the name of the specification associated with the currently loaded rule set.

- **Close Rules File:** Closes the currently opened rules file. Only one rules file may be open at any one time.

- **Add New Rule Node:** Displays the *Add Rule* form to allow the addition of a new rule node to an existing tree to refine a worklet selection. See the section below on the *Adding a New Rule* for more details.

- **Tree Viewer:** Displays the *Tree Viewer* form, which provides the ability to view large trees in full-screen mode.

- **Config Form:** Displays the configuration form discussed above.

- **About Box:** Displays some information about the rules editor (version number, date and so on).

Other Features

- **Current Rule Type:** This drop-down list displays each rule type that has a tree set defined in the opened rules file. Selecting a rule type from the list displays in the *Tree View* an associated rules tree from the tree set. Works in conjunction with the *Task Name* drop-down list.

- **Current Task Name:** This drop-down list displays the name of each task that has a rules tree associated with it for the rule type selected in the *Rule Type* list. Selecting a task name will display the rules tree for that task in the *Tree View*. This drop-down list is disabled for case level rules types.

- *Tree View:* This area displays the currently selected rules tree in a graphical tree structure. Selecting a node in the tree will display the details of that node in the *Selected Node* and *Cornerstone Case panels*. Nodes are colour coded for easier identification:

    - Blue nodes represent the root node of the tree
    - Green nodes are **true** (**exception**) nodes (i.e. they are on a true branch from their parent node)
    - Red nodes are **false** (**else**) nodes (i.e. they are on a false branch from their parent node)

- **Selected Node:** Displays the details of the node currently selected in the *Tree View*.

- **Cornerstone Case:** displays the complete set of case data that, in effect, caused the creation of the currently selected rule (see *Adding a new rule* below for more details). In Figure 7.15, the Cornerstone Case data shows that, amongst other things, the variable *Fever* had a value of true, while the variables *Rash*, *Wound* and *Fracture* each have value of false.

7.5.4   Adding a New Rule

There are occasions when the worklet returned for a particular case, while the correct choice based on the current rule set, is an inappropriate choice for the case. For example, if 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, it may be desirable to treat the racing heart rate before the rash is attended to. In such a case, as the Worklet Service begins execution of an instance of the *TreatRash* process, it is obvious that a new rule needs to be added to the rule set so that cases that have such data (both now and in the future) will be handled correctly.

To add a new rule to a particular tree of a rule set, it is first necessary to open the rule set in the Rules Editor (as described above). Then, click *Rules* on the top menu, then *Add...*, or click the *Add Rule* toolbar button, to open the (initially blank) *Add Rule* form.

Notice that the name of the opened rule set is shown in the title bar of the form, and the rule type and task name that are currently selected on the main form have been transferred to the *Add Rule* form. Thus, to add a new rule to a rule tree, that rule tree must first be selected on the main Rules Editor form before the *Add Rule* form is opened.

Every time the Worklet Service selects a worklet to execute for a specification instance, a log file is created that contains certain descriptive data about the worklet selection process. These files are stored in the *selected* folder of the worklet repository. The data stored in these files are again in XML format, and the files are named according to the following format:

> *CaseID_SpecificationID_RuleType_WorkItemID.xws*

For example: *12_CasualtyTreatment_Selection_Treat.xws* (xws for Xml Worklet Selection). The identifiers in each part of the filename refer to the parent specification instance, not the worklet case instance. Also, the *WorkItemID* identifier will not appear for case-level rule types.

So, to add a new rule after an inappropriate worklet choice, the particular *selected* log file for the case that was the catalyst for the rule addition must be located and loaded into the Rules Editor.

From the *Add Rule* screen, click the *Open...* button to load the selection information from the relevant *selected* log file. The File Open dialog that displays will open in the *selected* folder of the repository. Select the appropriate file for the case in question then click OK. Note that the *selected* file chosen must be for an instance of the specification that matches the specification rule set loaded on the main screen (in other words, you cant attempt to add a new rule to a rule set that has no relation to the *xws* file opened here). If the specifications dont match, an error message will display.

Figure 7.17 shows the *Add Rule* form with the *selected* file *12_CasualtyTreatment_Selection_Treat.xws* loaded. The *Cornerstone Case* panel shows the case data that existed for the creation of the original rule that resulted in the selection. The *Current Case* panel shows the case data for the current case - that is, the case that is the catalyst for the addition of the new rule.

The *New Rule Node* panel is where the details of the new rule may be added. Notice that the ids of the parent node and the new node are shown as read only - the Rules Editor takes care of where in the rule tree the new rule node is to be placed, and whether it is to be added as a true child or false child node.

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 require 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 7.17, such as *PatientID*, *Name*, *Sex*, *Blood Pressure readings*, *Height*, *Weight* 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.

Figure 7.17: Add New Rule Form

Clicking on the line "HeartRate = 190" in the *Current Case* panel copies that line to the *Condition* input in the *New Rule Node* panel. 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 branch of the *TreatRash* node. By doing so, 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 return 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 175. While selecting data items from the *Current Case* panel is fast and easy, it is often the case that the condition needs to be further modified to correctly define the relevant rule. The *Condition* input allows direct editing of the condition.

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 7.18.

Precedence	Operators	Type
1	* /	Arithmetic
2	+ −	
3	= > < != >= <=	Comparison
4	**&**	Logical AND
	|	Logical OR
	!	Logical NOT

Figure 7.18: Operators Supported

All conditions must finally evaluate to a Boolean value (i.e. true or false).

TIP: In addition to conditions expressed in the above mentioned format, conditional expressions may also take the form of XQuery expressions that evaluate to a boolean value.

To make the condition for the new rule more appropriate, the condition "HeartRate = 190" should be edited to read "HeartRate > 175".

After defining a condition for the new rule, the name of the worklet to be executed when this condition evaluates to true must be entered in the read-only *Conclusion* field of the *New Rule Node* panel (refer Figure 7.17). To select or create an appropriate worklet, click the *New...* button.

What happens next depends on whether the rule type for the tree you are adding the new rule to is of Selection type, or one of the exception types. Adding a conclusion for a Selection rule is explained below. Refer to the *Creating a New Rule Set and/or Tree Set* section below for details on adding a conclusion for the exception types.

7.5.5   Adding a Conclusion - Selection Rule Type

For a Selection rule tree, when the *New..* button is clicked, a dialog is displayed that comprises a drop-down list containing the names of all the worklets in the worklets folder of the worklet repository (refer Figure 7.19). An appropriate worklet for this rule may be chosen from the list, or, if none of the existing worklets are suitable, a new worklet specification may be created.

Clicking the *New...* button on this dialog will open the YAWL Editor so that a new worklet specification can be created. When defining the new worklet, bear in mind that to pass data from the original work item to the worklet, the names and data types of the variables to be passed must match those of the work item and be created as net-level variables in the worklet specification. Also, all new worklets must be saved to the *worklets* folder of the repository so that the Worklet Service can access it.

TIP: You may choose more than one worklet in this dialog simultaneously by holding down the Ctrl or Alt keys while clicking the mouse (in the usual Windows way). If you choose several worklets, when this rule is invoked for a process at runtime **all** of the worklets chosen will be launched concurrently and the process will continue only after **all** the worklets launched have completed.
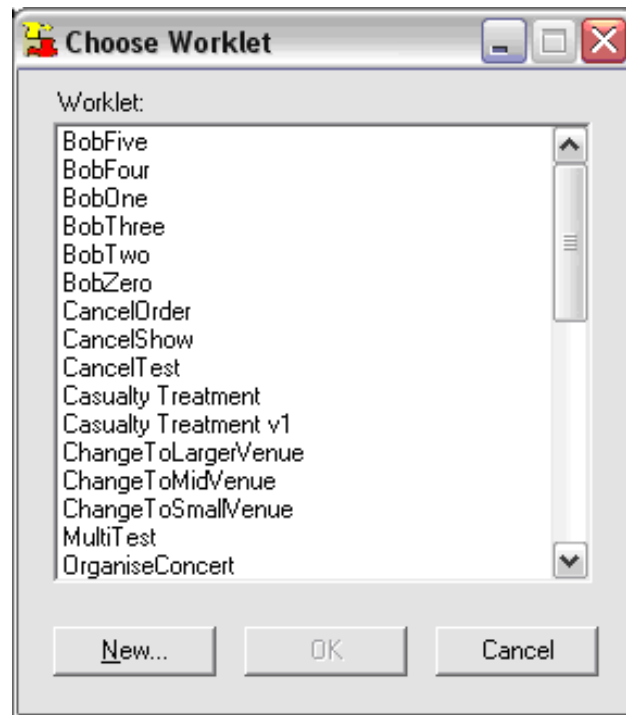
Figure 7.19: The Choose Worklet dialog

When the new worklet is saved and the YAWL Editor is closed, the name of the newly created worklet will be displayed and selected in the worklet drop-down list. Click the OK button to confirm the selection and close the dialog. Figure 7.20 shows the *New Rule Node* panel after the definition of the example new rule has been completed. A value in the *Description* field is optional, but recommended.

Figure 7.20: The *New Rule Node* Panel after a New Rule has been Defined

Once all the fields for the new rule are complete and valid, click the *Save* button to add the new rule to the rule tree.

7.5.6 Dynamic Replacement of an Executing Worklet

Remember that the creation of this new rule was triggered by the selection and execution of a worklet that was deemed an inappropriate choice for the current case. So, when a new rule is added, you are given the

choice of replacing the executing (inappropriate) worklet instance with an instance of the worklet defined in the new rule.

After the *Save* button is clicked, a message similar to the Figure 7.21 is shown, providing the option to replace the executing worklet, using the new rule. The message also lists the specification and case ids of the original work item, and the name and case id of the running worklet instance.
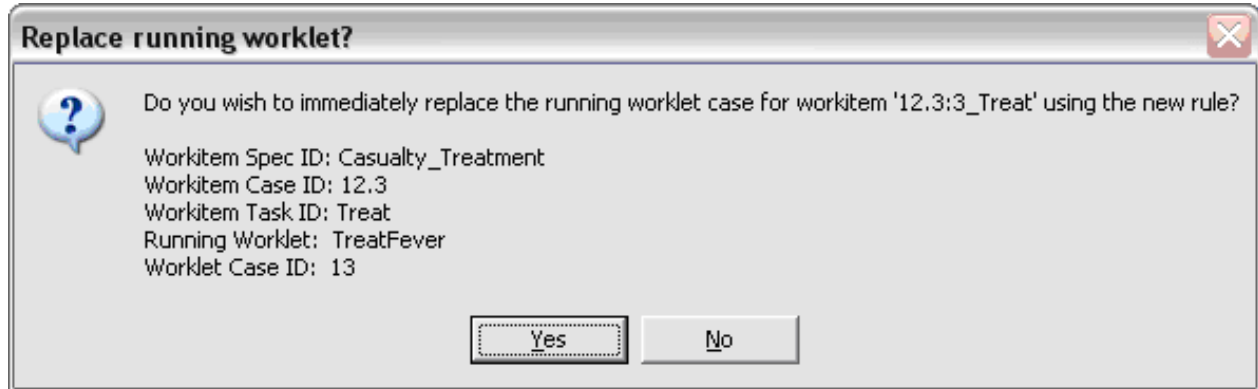
Figure 7.21: Message Dialog Offering to Replace the Running Worklet

If the *Yes* button is clicked, then in addition to adding the new rule to the rule set, the Rules Editor will contact the Worklet Service and request the change. For this process to succeed, the following must apply:

- Tomcat is currently running and the Worklet Service is correctly installed;

- The Service URI specified in the Rules Editor configuration dialog is valid; and

- The worklet originally chosen is currently running.

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 7.22.

If the *No* button is clicked, then the new rule is simply added to the rule set.
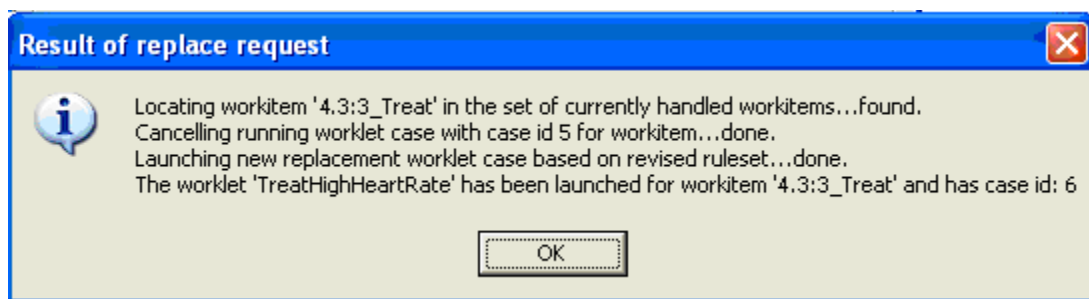
Figure 7.22: Result of Replace Request Dialog

Figure 7.23 shows the main Rules Editor screen with the new rule added in the correct place in the tree, with the current case data becoming the *Cornerstone Case* for the new rule.

7.5.7   Creating a New Rule Set and/or Tree Set

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 encompassing every task in a specification. So, most typically, rule sets will have
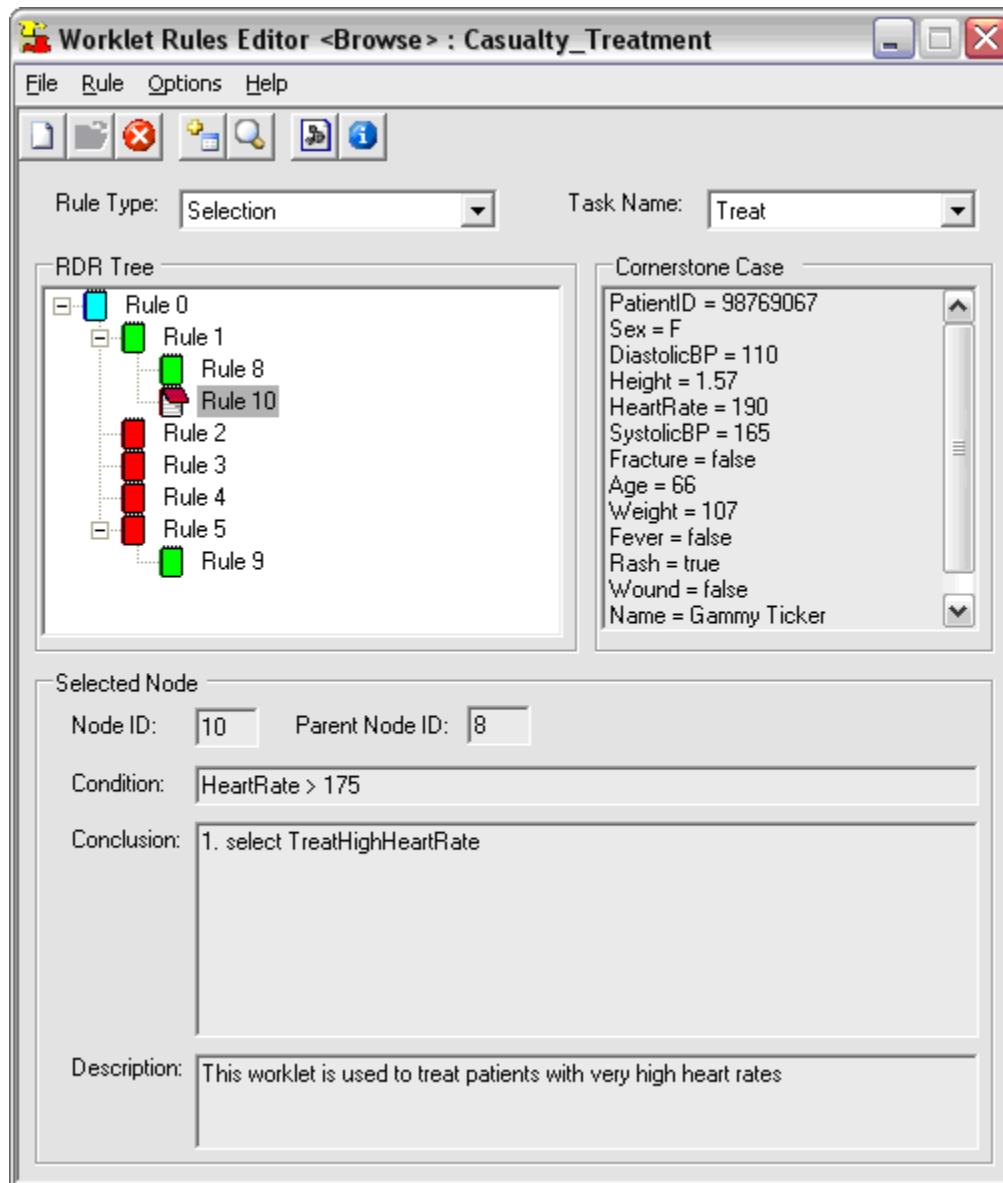
Figure 7.23: Main Screen after Addition of New Rule

rules defined for a few rule types, with some 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 where you will 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. Also, when a new specification has been created, a corresponding base rule set will also need to be created (if you want to handle selections and exceptions for the new specification).

For each of these situations, the Rules Editor provides a *New Rule* form, which allows 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.

The use of the *New Rule* form varies slightly depending on whether it is working with a new rule set or an existing rule set. This section will describe the features of the *New Rule* form for adding a new rule set, and

describe how the process differs for existing rule sets as necessary.

To create a new rule set, click the *File* menu then select *New...*, or click the New Rule toolbar button. To add a new rule set, make sure there is no rule set file currently open in the Editor. If you are creating a new rule set, a dialog will display asking for the path and file name of the specification for which the rule set is being created (Figure 7.24).
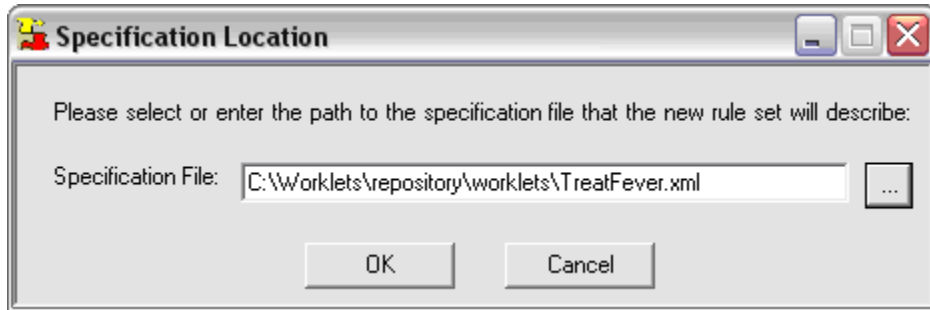

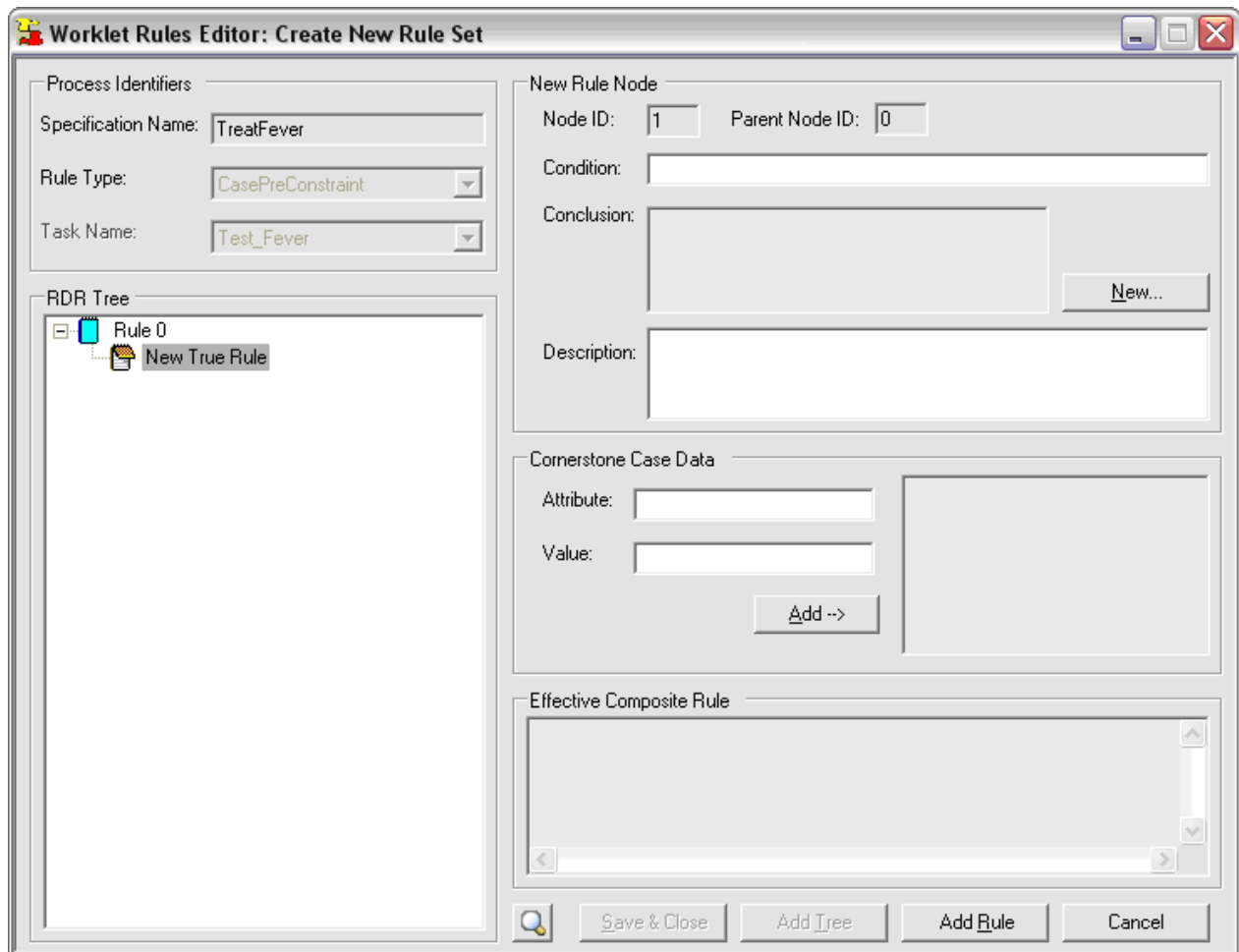**Figure 7.24:** The Specification Location Dialog

Figure 7.25: The Create New Rule Set Screen

Figure 7.25 shows the *Create New Rule Set* form. The form allows you create a rule set, one rule tree at a

time (for the selected specification, rule type and task name). On this form:

- The *Process Identifiers* panel is where the names of the specification, rule type and task name for the new tree are defined. The *Specification Name* input is read-only - for new rule sets it is the specification chosen via the *Specification Location* dialog (Figure 7.24); for existing rule sets it is the specification for the rule set currently loaded into the Rules Editor. The *Rule Type* drop-down list contains all of the available rule types (i.e. all the rule types for which no or incomplete tree sets exist). For new rule sets, all rule types are available. The *Task Name* drop-down list contains all the available tasks for the selected rule type (i.e. tasks for which no tree exists in the tree set for this rule type). The *Task Name* list is disabled for case-level rule types.

- The *New Rule Node* panel is identical to the panel on the Add New Rule form. Here a condition and optional description can be entered, and the conclusion for the new rule created or selected from the list (depending on the rule type - see below).

- The *Cornerstone Case Data* panel allows a set of cornerstone data for the new rule to be defined. Add a variable name to the *Attribute* input, and give it a value in the *Value* input, then click the *Add* button to add it to the set of Cornerstone Case data. Usual naming rules apply to the data attributes: the attribute name must begin with an alpha character or underscore and contain no spaces or special characters.

- The *Effective Composite Rule* panel displays a properly indented text equivalent of the composite condition comprising the conditions of the selected node and all ancestor nodes back to the root node - in other words, the entire composite condition that must evaluate to true for the conclusion of the selected node to be returned.

- The *RDR Tree* panel dynamically displays graphically the new rule tree as it is being created.

New rule nodes can be added wherever there is a node on the tree that does not have a child node on both its true and false branches (except the root node which can have a true branch only). To identify possible locations to add a rule node, special 'potential nodes can be seen in the *RDR Tree* panel, called "New True Node" or "New False Node". These potential nodes are coloured yellow for easy identification.

To add a new rule, select the yellow new rule node where you would like the rule to be added. When you select a new rule node, the various inputs for the new rule become enabled. Refer to the *Adding a New Rule* section above for details of the types of operands and operators you can add as the condition of the new rule.

To add a conclusion to the new rule, click the *New...* button. If the currently selected rule type is *Selection*, a worklet can be added as a conclusion in the way described in the *Adding a New Rule* section. If it is one of the exception rule types, the *New...* button will display the graphical *Draw Conclusion* dialog, allowing you to build a sequence of tasks (or primitives) in an exception handling process (explained in detail below). When the conclusion sequence has been defined and the dialog closed, a text-based version of it will display in the *Conclusion* panel.

Once the new rule node has a valid condition and conclusion, and optionally some cornerstone data and a description, click the *Add Rule* button to add the rule to the tree. The new node will be displayed at the selected position on the tree with the relevant coloured node icon indicating whether it is a true or false node of its parent. New potential node add-points will also be displayed. See Figure 7.26 for an example of a newly created tree that has several nodes added.

Repeat the add rule procedure for however many rule nodes you wish to add by clicking on the appropriate yellow node (note that when you click on a node other than yellow, its stored details are displayed in the various form inputs).

TIPS:

- If the number of nodes starts to get a little confusing, you can check which node is the parent of the selected node by looking at its id in the Selected Node panel.
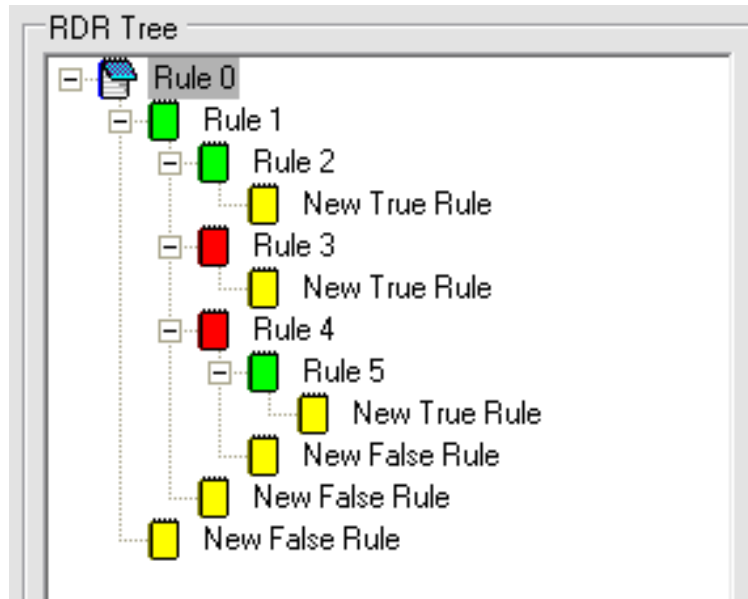
Figure 7.26: Creating a New Rule Tree

- If you start to add details for a new node then change your mind about adding it, simply click on any other node (rather than clicking on the *Add Rule* button) - doing that will immediately discard any input data values. **Don't** click Cancel - that closes the entire form without saving anything (see below).

When you are done adding nodes, click the *Add Tree* button to add the tree you have just created to the tree set selected (via the selected *Rule Type* and *Task Name* lists).

IMPORTANT: Once you have added the newly created tree to the selected tree set using the *Add Tree* button, you will no longer be able to add nodes to the tree via the *New Rule Set* form. This is to protect the integrity of the rule set. Since each subsequent rule will be added because of an exceptional case or where the selected worklet does not fit the context of a case, the preferred method is to create the base rule set and then add rules as they become necessary via the *Add Rule* form as described earlier. In this way, added rules are based on real case data and so are guaranteed to be valid. In a similar vein, there is no option to modify or delete a rule node within a tree once the tree has been added to the rule set, since to allow it would destroy the integrity of the rule set, because the validity of child rule nodes depend on the conditions of their parents.

When a tree is added to the tree set:

- If it is a case-level tree, the rule type that the tree represents will be removed from the *Rule Type* list. That is, the rule type now has a tree defined for it and so is no longer available for selection on the *New Rule* form.

- If it is an item-level tree, the task name that the tree represents will be removed from the *Task Name* list. That is, the task now has a rule tree defined for it (for the selected rule type) and so is no longer available.

- If it is an item-level tree, and all tasks now have trees defined for them for the selected rule type (i.e. this was the final task of the specification for which a tree has been defined), the rule type that the tree represents will be removed from the *Rule Type* list.

This approach ensures that rule trees can only be added where there are currently no trees defined for the selected specification.

Once the tree is added, the form resets to allow the addition of another new tree as required, by repeating the process above for a new rule type (or rule type/task name for item-level trees).

After you have completed adding trees, click the *Save & Close* button to save all the additions to the rule set file. The Rules Editor will return to the main form where the additional trees will immediately be able to be browsed.

IMPORTANT: No additions will be actually saved until the *Save & Close* button is clicked - this is to allow you the option to discard all additions, if you wish, by clicking the *Cancel* button. That is, cancelling returns to the main Editor form and discards **ALL** additions for the session; *Save & Close* returns to the main form and saves all additions.

7.5.8 Drawing a Conclusion Sequence

As mentioned in the *Adding a New Rule* section, adding a conclusion to a Selection rule is simply a matter of choosing a worklet from the list or creating a new worklet in the YAWL editor. However, when adding a conclusion for a rule type other than *Selection* (i.e. an exception rule type), an exception handling sequence needs to be defined that will manage the handling process invoked by the rule. The earlier section on the Exception Service detailed the various actions that make up the available set of exception handling 'primitives or tasks that may be sequenced to form an entire handling process.

The *Draw Conclusion* dialog makes the process of defining an exception handling sequence easier by allowing you to create the sequence graphically. Simply select the appropriate primitive from the toolbox on the left, and then click on the drawing canvas to place the selected primitive. Figure 7.27 shows an example of the *Draw Conclusion* dialog.
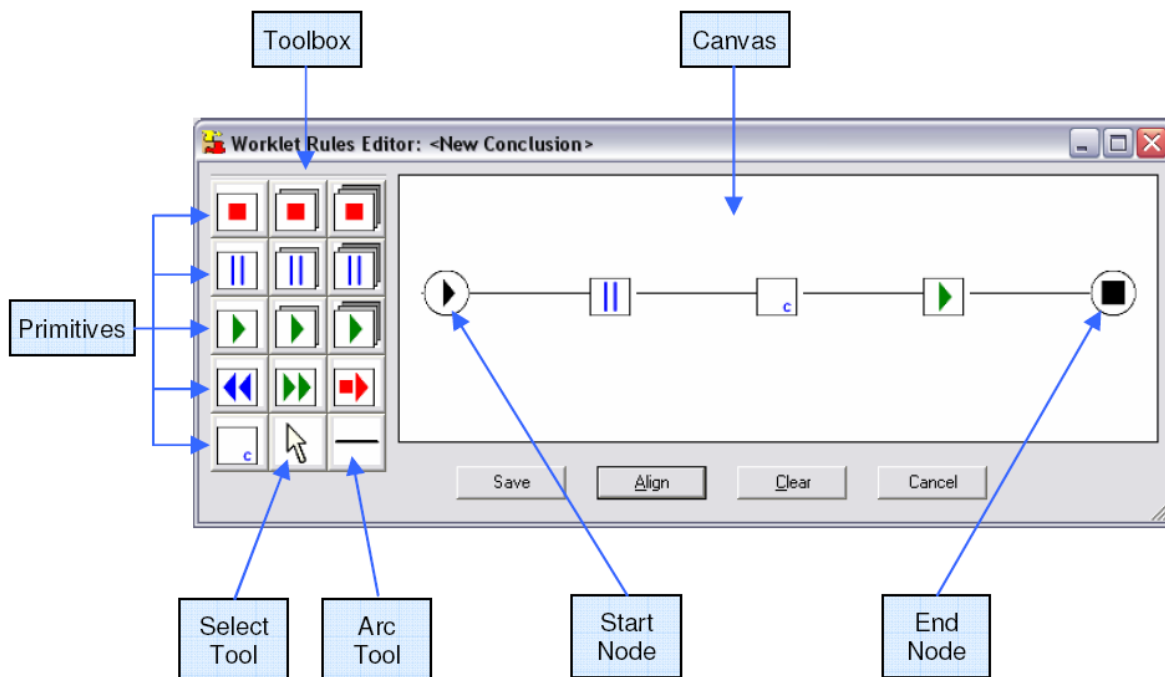
Figure 7.27: The Draw Conclusion Dialog

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 a conclusion 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 conclusion will be considered invalid if there are any nodes on the canvas that are not attached to the sequence when *Save* is attempted.

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 7.27).

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 form.

The *Save* button will save the conclusion and return to the previous form (as long as the sequence is valid).

To delete a primitive from the canvas, right click on the primitive and select *Delete* from the popup menu.

The *Compensate* primitive will, when invoked at runtime, execute a worklet as a compensation process as part of the handling process. To specify which worklet to run for this sequence, right click on the *Compensate* primitive and select *Define Worklet* from the popup menu. The *Choose Worklet* dialog will appear (identically to the *Selection* conclusion process) allowing the selection of an existing worklet or the definition of a new worklet to run as a compensatory process. Select the appropriate worklet to add it to the compensatory primitive. Note that a sequence will be considered invalid if it contains a *Compensate* primitive for which a worklet has not yet been defined.

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 primitives 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.

IMPORTANT: No validation is done for the defined sequence, besides that described above. It is up to the designer of the sequence to ensure it makes sense (for example, that it doesnt try to continue a case it has previously removed).

When a valid sequence is saved, you will be returned to the previous form (i.e. either the *Add Rule* or *New Rule* form depending on where you are in the Editor). The conclusion will be displayed textually as a sequential list of tasks (Figure 7.28, for example).

Conclusion:
1. suspend workitem
2. run worklet SendReminder
3. continue workitem
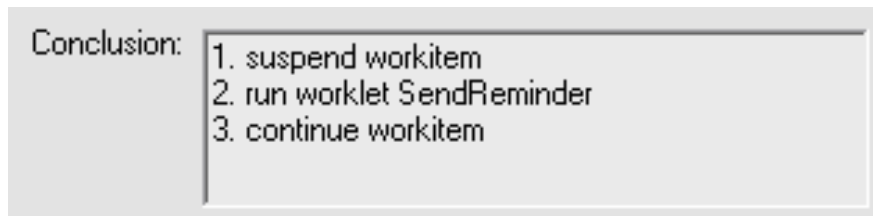
Figure 7.28: A Conclusion Sequence shown as Text (detail)

7.6   Walkthrough - Using the Worklet Service

The worklet repository that comes with the Worklet Service release 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, and then 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 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 *worklets* folder of the worklet repository. 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 *Submit* (Figure 7.29).

Figure 7.29: Launching a Casualty Treatment Case (detail)

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.

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 7.30). 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*, the others to *False*.

Lets assume the patient has a fever. Set the *Fever* field to *True*, the rest to *False*, and then click the *Complete* button (Figure 7.31).

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 as a conditional branch into the process model.

The Worklet Service greatly simplifies this problem, by providing an extensible repertoire of discrete workflow

Edit Work Item: 1.1

Admit

Weight:	85
DiastolicBP:	80
Sex:	M
Height:	1.8
HeartRate:	72
SystolicBP:	120
Age:	21
PatientID:	123456
Name:	Iva Payne

[Cancel]   [Save]   [Complete]

Figure 7.30: Editing the *Admit* Workitem (detail)

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 submitted, 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 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 7.32). The *TreatFever* process has been chosen by the Worklet Service to replace the *Treat* workitem based on the data passed to the service.

Note that the case id for the Test Fever workitem is different to the case id of the parent process. Worklets

Figure 7.31: Editing the *Triage* Workitem (detail)

Figure 7.32: New Case Launched by the Worklet Service

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 7.33).

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 7.34). Enter some information into the *Notes* field, and then complete it.
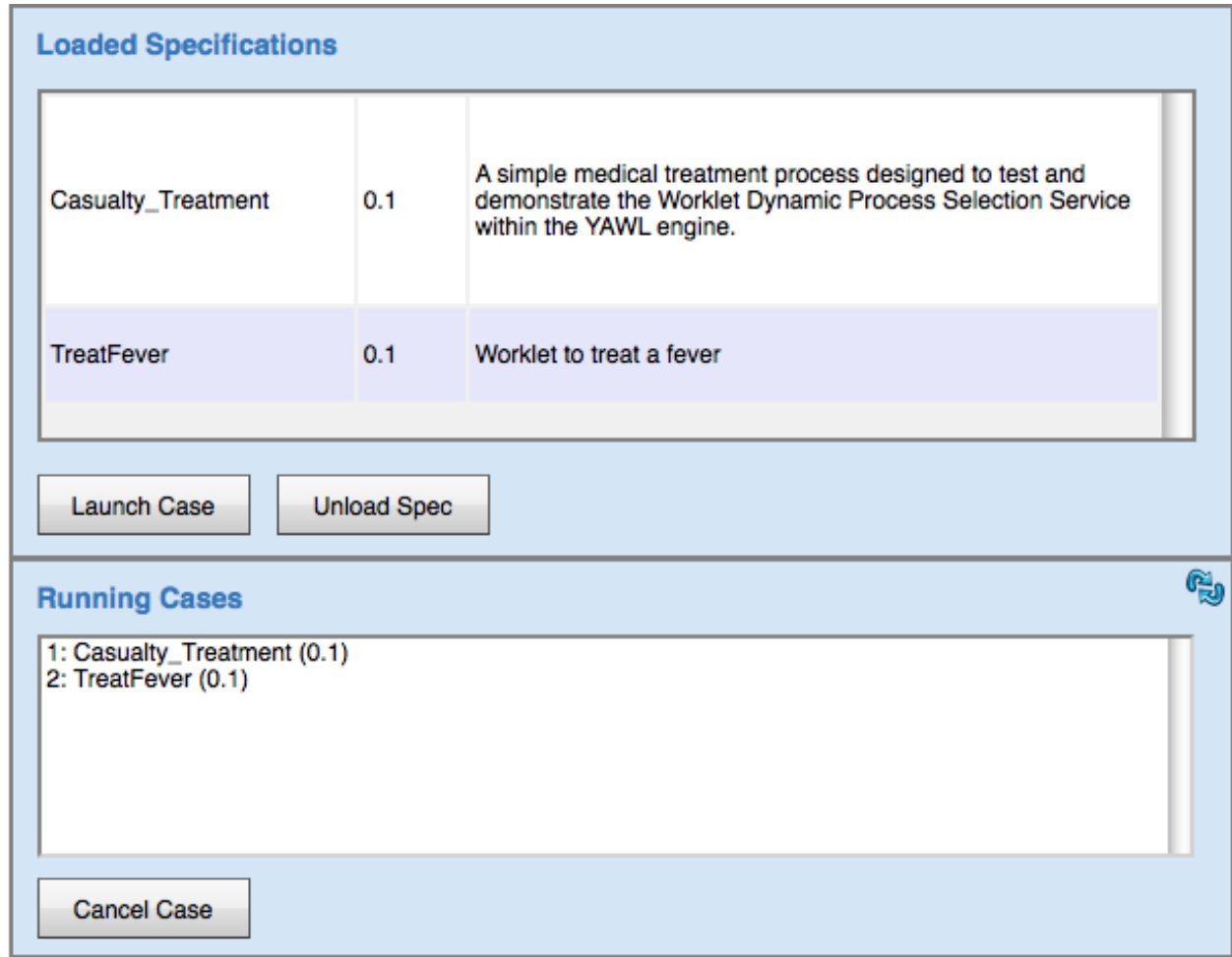
Figure 7.33: TreatFever Specification Uploaded and Launched

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 7.35). Enter some data here, and then complete it.

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.

Go to the *Work Queues* page, and you'll see that the *Discharge* workitem is available (Figure 7.36). 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 the YAWL Editor User Manual and worklet-enables the *Verify List* task to show how multiple instance atomic tasks are handled by the Worklet Selection Service.

Edit Work Item: 2.1

Test Fever

Notes:	bitten by spider
PatientID:	123456
Fever:	☑
Name:	Iva Payne

Cancel Save Complete
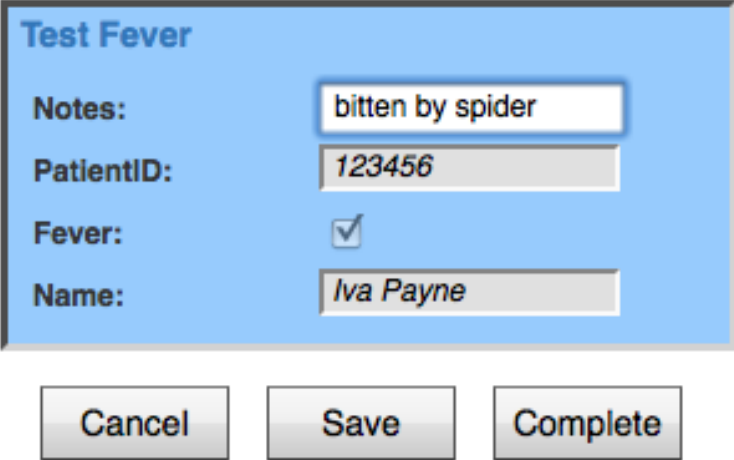
Figure 7.34: Test Fever Workitem (detail)

Edit Work Item: 2.2

Treat Fever

Notes:	bitten by spider
Pharmacy:	aspirin - twice a day
Treatment:	anti-venene
PatientID:	123456
Name:	Iva Payne

Cancel Save Complete
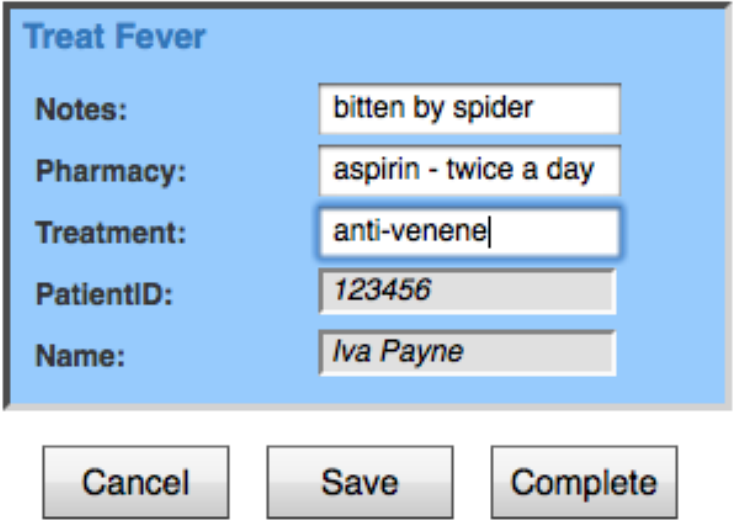
Figure 7.35: Treat Fever Workitem (detail)

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 7.37 shows the specification.

Go to the *Case Mgt* page and upload the *wListMaker* specification from the worklets folder of the worklet repository. Then, launch an instance of *wListMaker*.

When the case begins, enter three values for the *Bob* variable, as shown in Figure 7.38 - 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 7.36: Discharge Workitem with Data Mapped from *TreatFever* Worklet

Figure 7.37: The *wListMaker* Specification

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 ta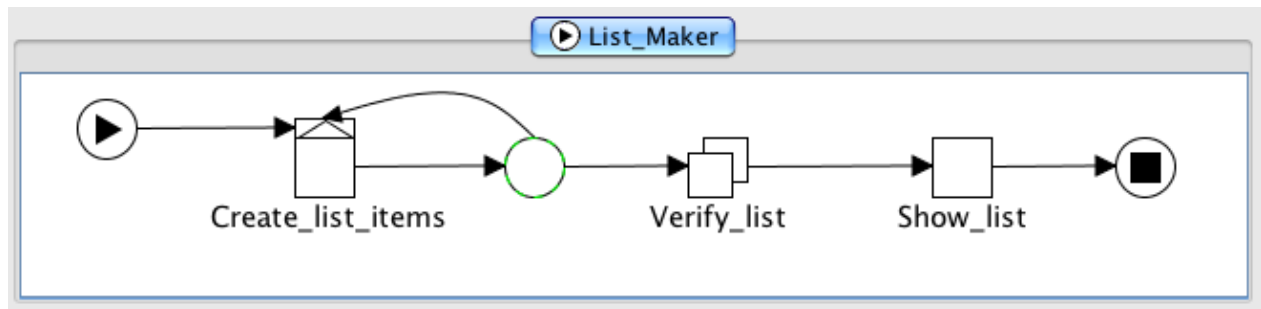sk, three child workitem instances of the task are created, one for each of the *Bob* values entered previously. The Worklet Service will determine that it is a multiple instance atomic task and will treat each child workitem instance separately, and will launch the appropriate worklet for each based 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 one task.

Go to the *Work Queues* page. There are three workitems listed, each one the first workitem of a separate case (see Figure 7.39).

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 7.40 - 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"

Figure 7.38: Start of *wListMaker* Case with Three 'Bob' Values Entered (detail)

Figure 7.39: Workitems from each of the three Launched Worklet Cases

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.
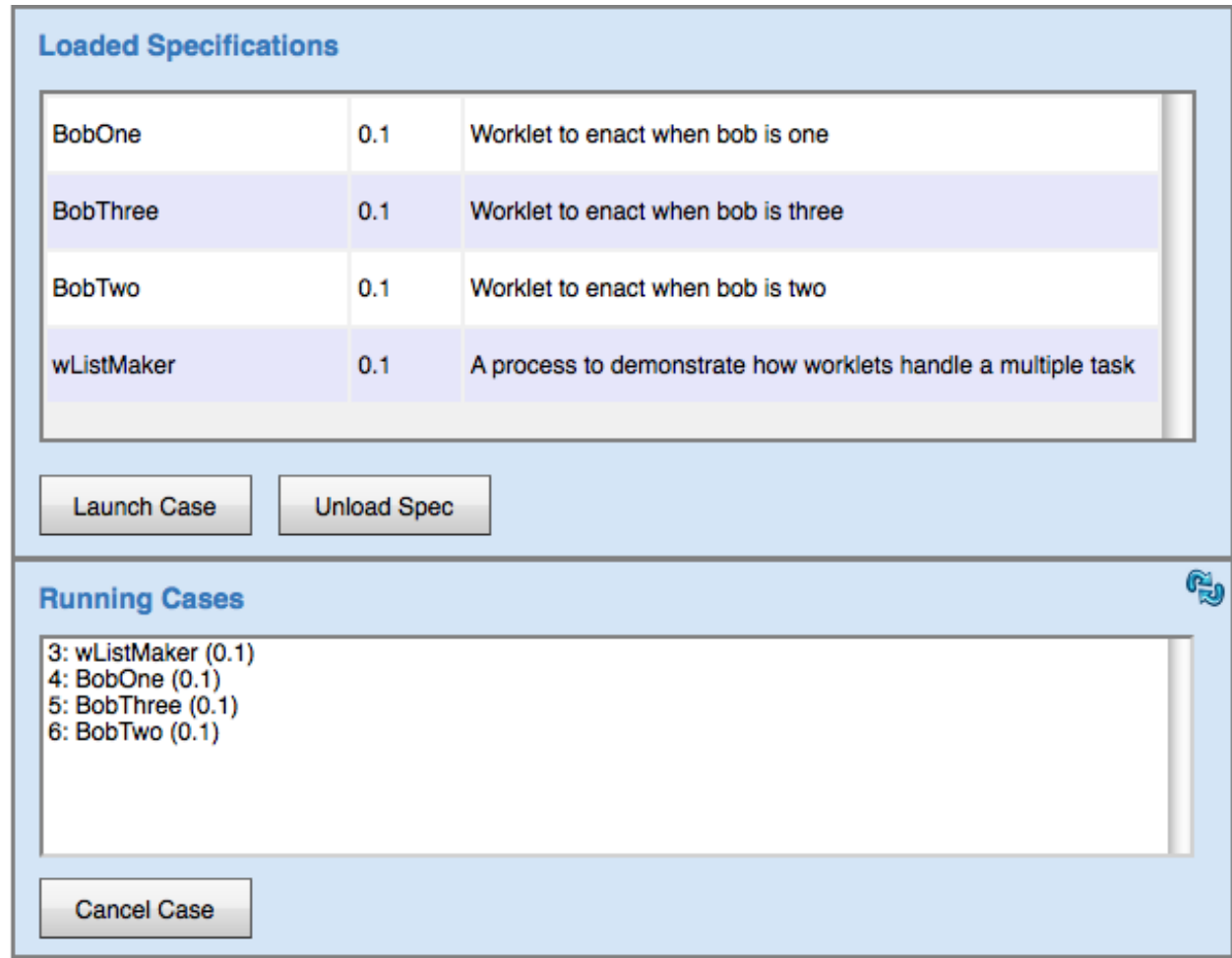
Figure 7.40: 'Bob' Specifications Loaded and Launched by the Worklet Service

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* workitem has completed and so the original 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 7.41).

C. Exception: Constraints Example

This walkthrough uses a specification called *OrganiseConcert* 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 7.42 shows the specification as it appears in the YAWL Editor.

First, ensure the Exception Service is enabled (see Section 7.1.3 for details). Navigate to the YAWL *Case Mgt* page and upload the *OrganiseConcert* specification from the worklets folder of the worklet repository. 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,

Figure 7.41: The Show List Workitem Showing the Changes to the Data Values
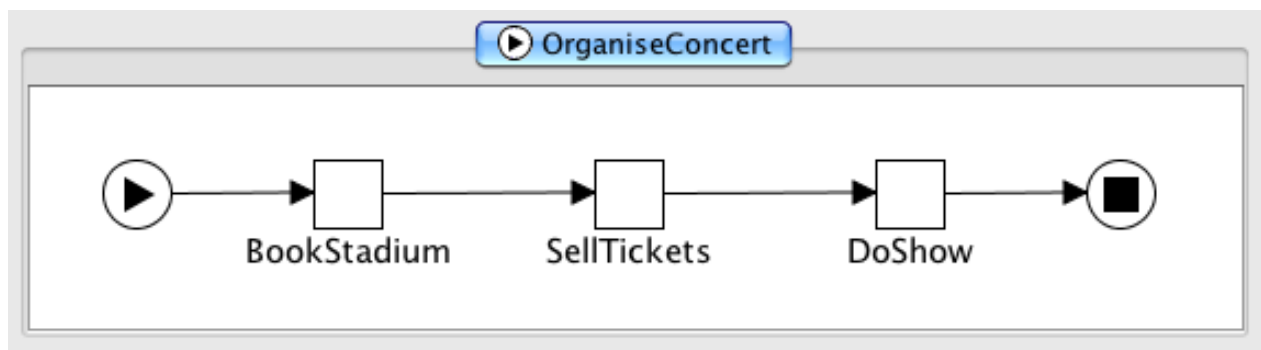
Figure 7.42: The *OrganiseConcert* Specification

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 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 (found in the logs folder of your Tomcat installation). See Appendix 7.8 of this manual for a sample log output for this walkthrough.

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, 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 7.43).

Edit Work Item: 8.1

BookStadium	
VenueCost:	100000.00
Seating:	25000
VenueName:	ANZ Stadium

Cancel   Save   Complete

Figure 7.43: 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 7.44).

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:

In other words, when *Do Show* is enabled and the value of the case data attribute "TicketsSold" is less 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.

Figure 7.44: The Sell Tickets Workitem (detail)

Figure 7.45: Effective Composite Rule for Do Shows Pre-Item Constraint Tree

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 7.46, then complete the workitem.

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 that every case instance involves some deviation from the standard process model. Sometimes, events occur completely removed from the actual process model itself, but affect the way the process

Edit Work Item: 9.2

Book Ent Centre

VenueCost: 50000.00

Seating: 15000

VenueName: Ent Centre

Cancel   Save   Complete

Figure 7.46: The Book Ent Centre Workitem (detail)

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, 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 7.47).

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 cases specification. See Figure 7.48 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, lets 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 7.49).

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

---

[2]If the two buttons don't appear, the exception service has not been correctly enabled for the Resource Service; review the installation section at the beginning of this chapter for details on how to enable the service.
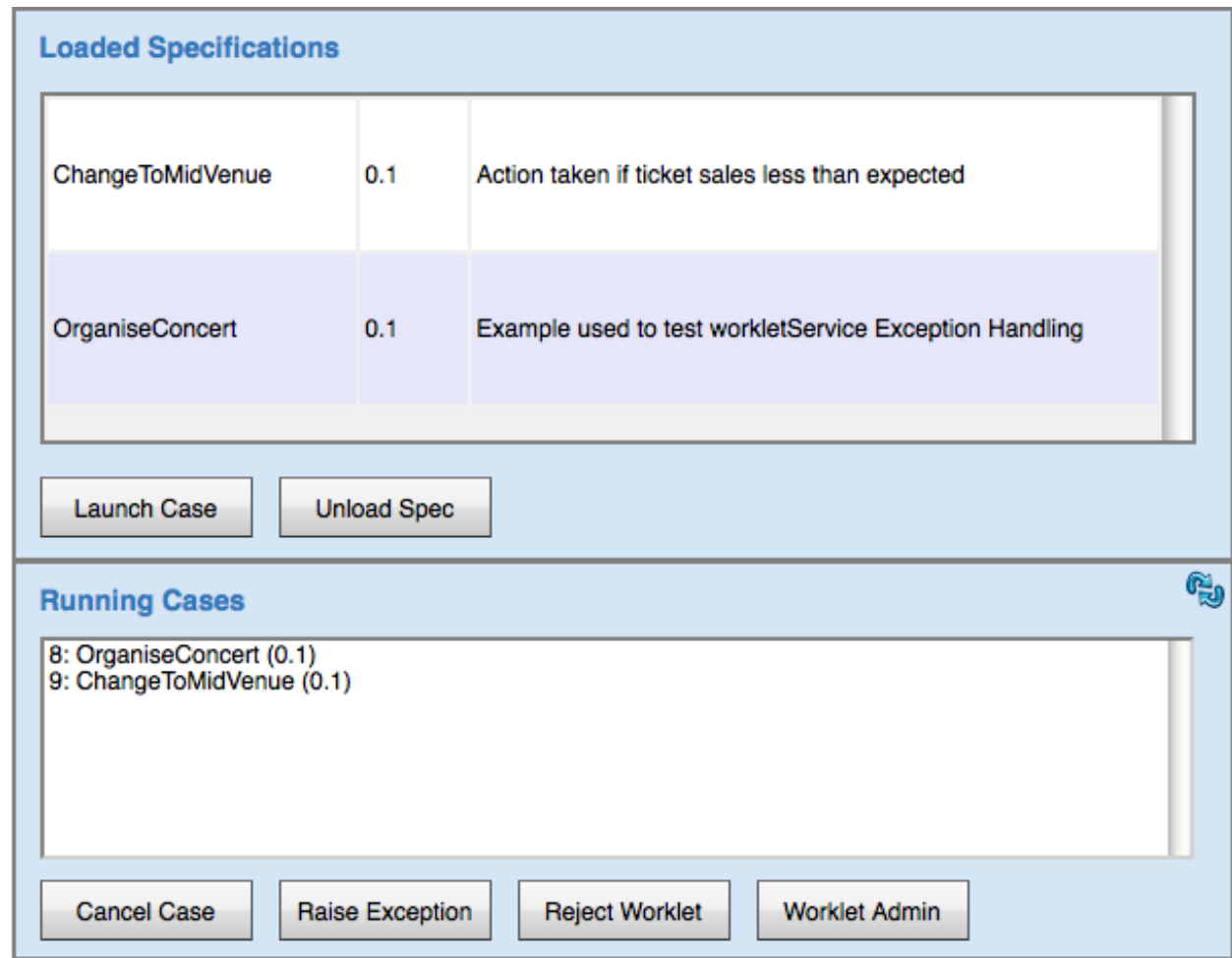
Figure 7.47: Case Mgt Screen, *OrganiseConcert* case running

from the list, then clicking the *Raise Exception* button at the top-right toolbar (the green 'forked arrow' within the tabbed area – see Figure 7.49). 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).

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 *TimeoutTest3* gives an simple example of how a timeout exception may be handled (Figure 7.50). 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 7.51).
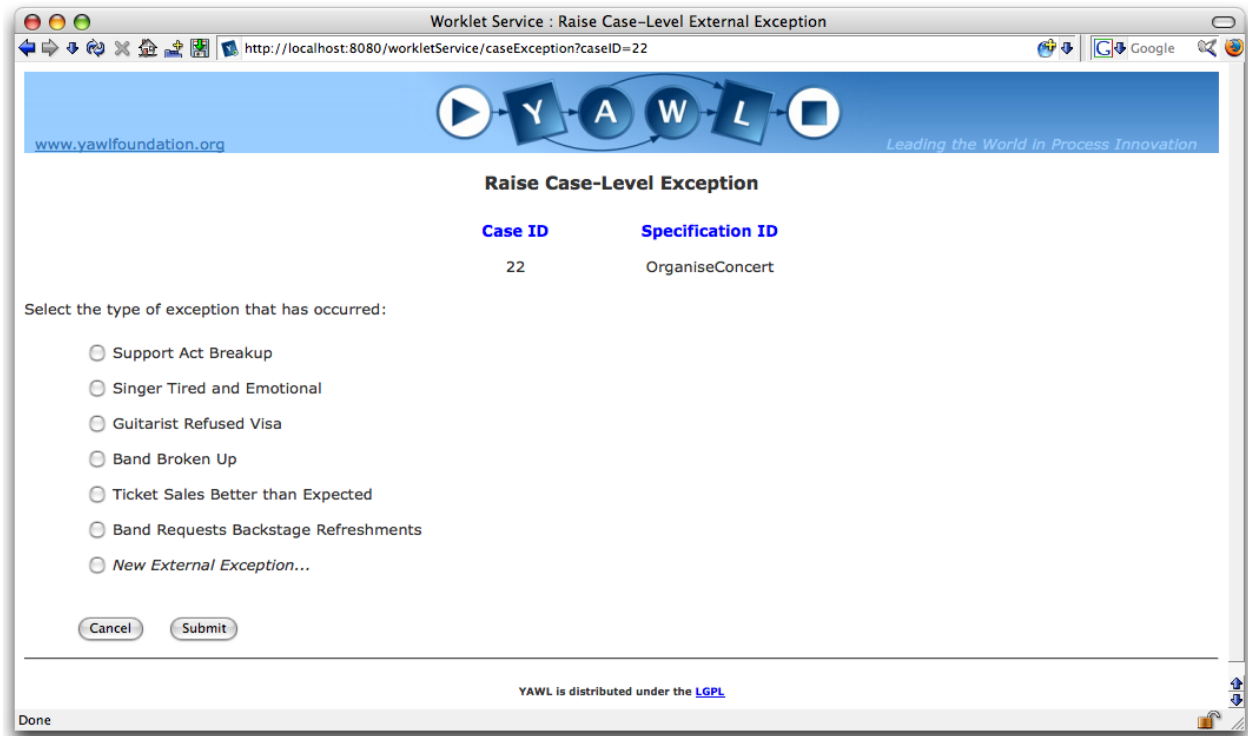
Figure 7.48: Raise Case-Level Exception Screen (Organise Concert example)
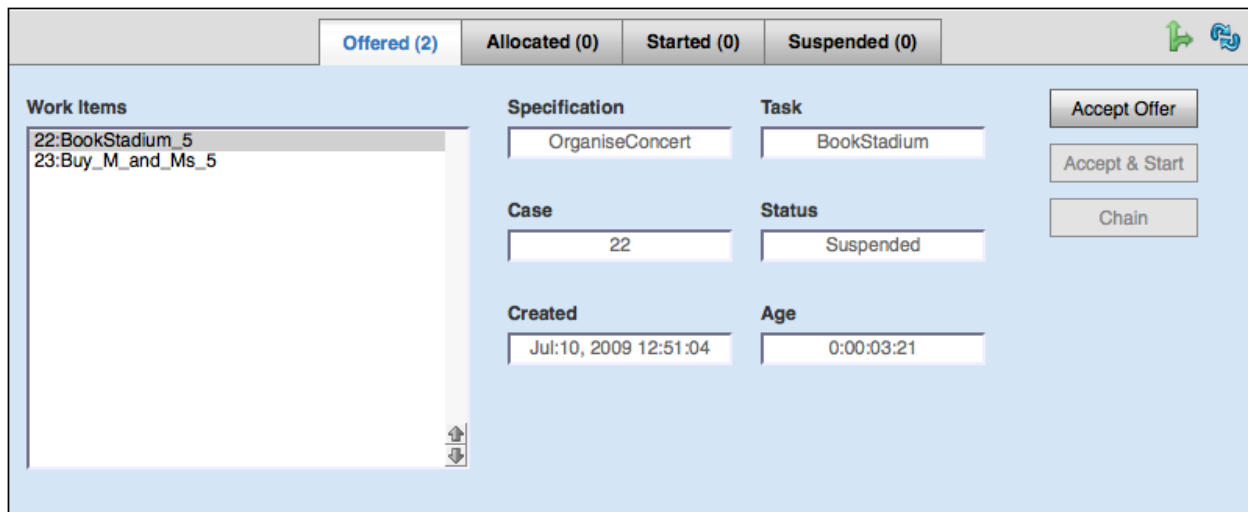
Figure 7.49: Available Work Items after External Exception Raised

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 7.52).

Notice the rules condition: isNotCompleted(this):

- *isNotCompleted* is an example of a defined function that may be used as (or as part of) conditional expressions in rule nodes.

Figure 7.50: The Timeout Test 3 Specification

Figure 7.51: The Set Timer Detail dialog for the Receive Payment task

- *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 Appendix 7.7 for more information about defining functions.

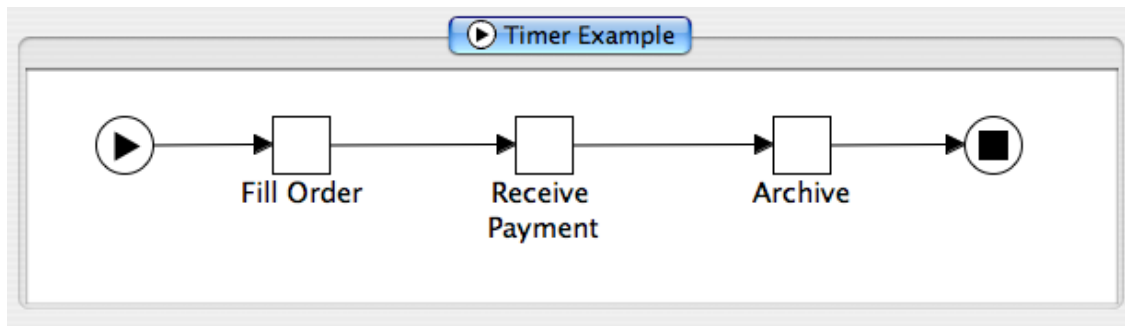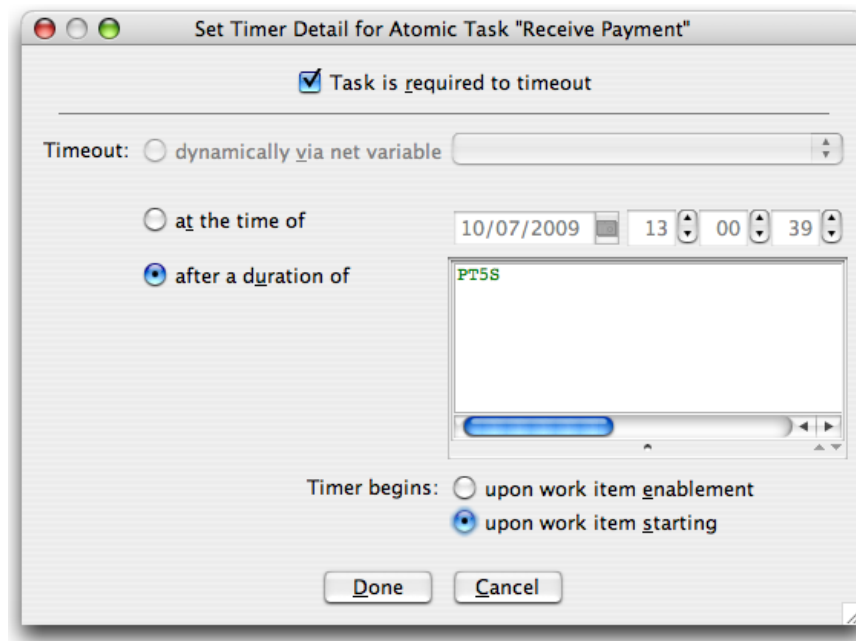In this case, the condition tests if the workitem *Receive_Payment* has not yet completed (i.e. if it has a status of *Fired*, *Enabled*, *Executing* or *Suspended*). If it hasn't completed (thus payment for the order has not yet been received) then the conclusion will be executed as an exception handling process, including the launching of the worklet *SendReminder*.

The *SendReminder* worklet consists of two tasks: *Send Request*, and timer-enabled *Receive Reply* – again, for the purposes of the example, this task waits for 5 seconds before timing out. When the task times out, the Exception Service is notified. There is a single timeout rule for the *Receive Reply* task - its condition is again *isNotCompleted(this)* but this time, the rules conclusion looks like this:

File Cancellation is the first task of the *Cancel Order* worklet. What we now have is a hierarchy of worklets: case (*TimeoutTest3*) 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
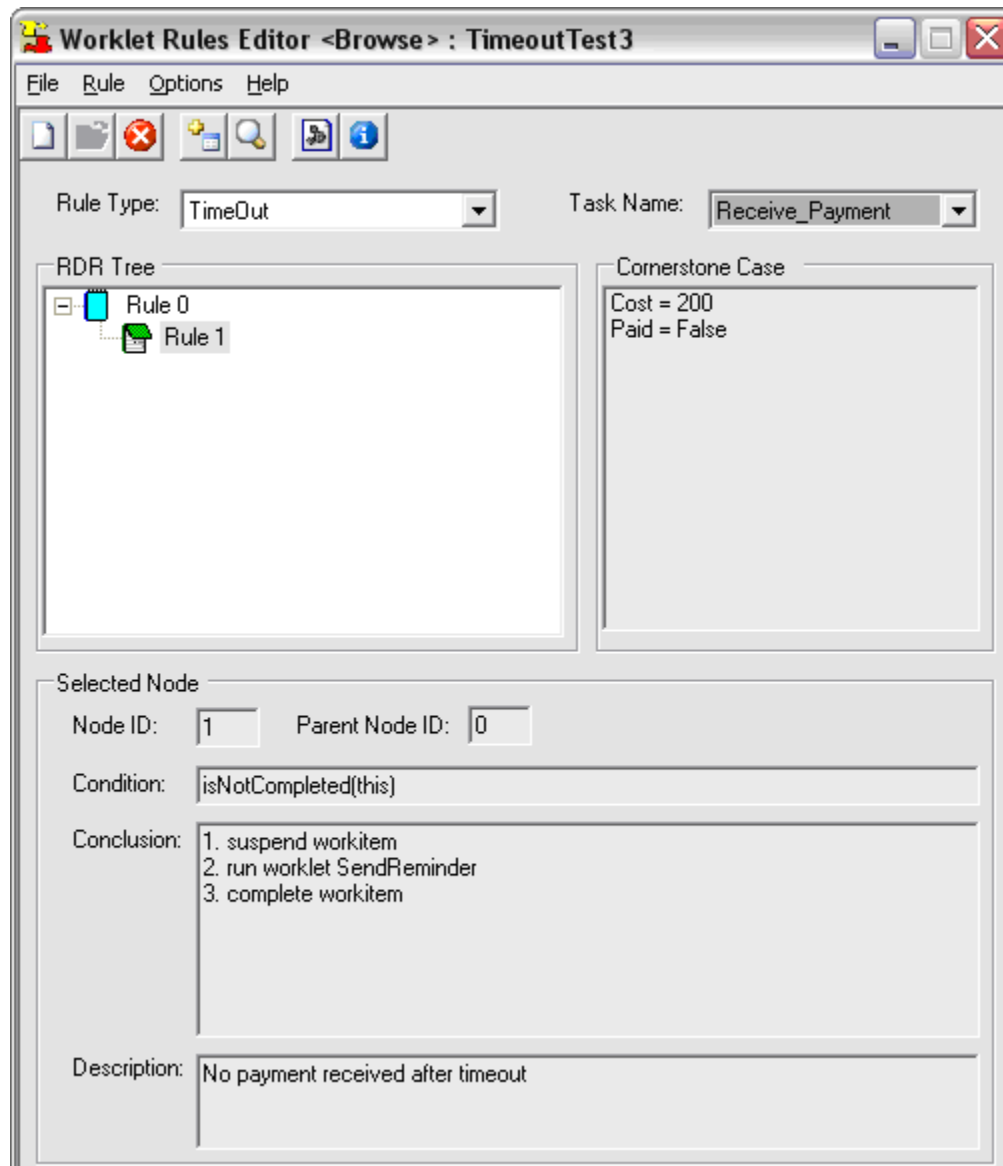
Figure 7.52: Rules Editor Showing Single Timeout Rule for Receive_Payment Task
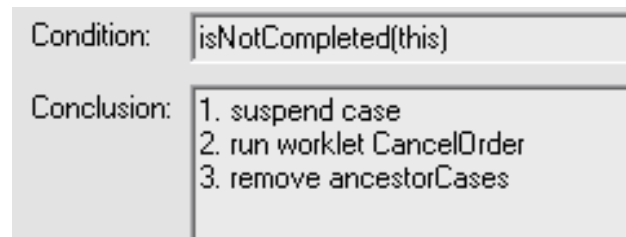
Figure 7.53: Rule detail for Receive Reply

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 *Timeout Test 3* 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 7.47). You will be redirected to the *Reject Worklet Selection* screen, another Worklet Service add-in screen (Figure 7.54). 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.

Figure 7.54: Reject Worklet Selection Screen

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 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 7.55 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 7.5.2 on the Rules Editor for more detail). The process requires a user with administrator privileges to action the rejection request, rather than allow 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.

Figure 7.55: Example of a New Case-Level Exception Definition

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 7.47). 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 7.56).

7.7   Defining New Functions for Rule Node Conditions

In Section 7.5, 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 D*, an example of a defined function was given (*isNotCompleted*), 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 *au.edu.qut.yawl.worklet.support* package. Currently, this class 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.
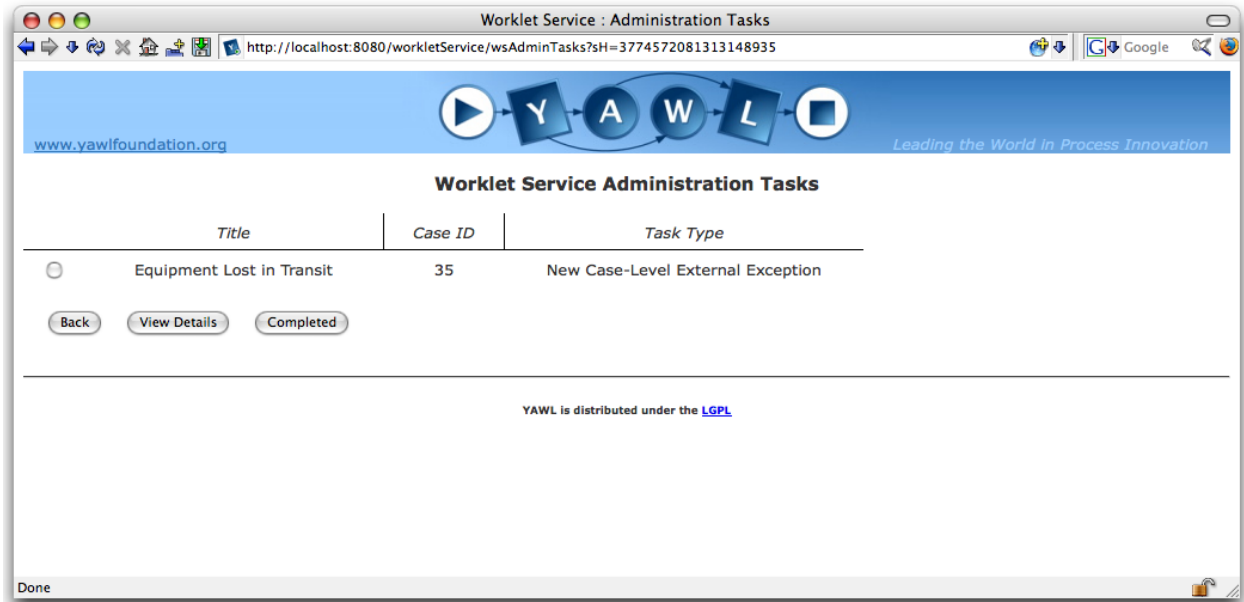
Figure 7.56: Administration Tasks Screen (detail)

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 7.57.

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 7.58.

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).

```
public static String max(int x, int y) {

        if (x >= y) return String.valueOf(x) ;

        else return String.valueOf(y) ;

    }
```

Figure 7.57: Max function

```
public static final String[] _functionNames = { "max",

                                                 "min",

                                                 "isNotCompleted",

                                                 "today"} ;
```

Figure 7.58: Adding the name

```
else if (name.equalsIgnoreCase("max")) {

    int x = getArgAsInt(args, "x");

    int y = getArgAsInt(args, "y");

    return max(x, y);

}
```

Figure 7.59: Execute method for the max function

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 7.59.

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 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 7.60.

The block gets *this* variable as a *String* from the "args" *HashMap* and then calls the actual *isNotCompleted* method (see Figure 7.61).

```
if (name.equalsIgnoreCase("isNotCompleted")) {

    String taskInfo = (String) args.get("this");

    return isNotCompleted(taskInfo);

}
```

Figure 7.60: Execute method's sub-block for isNotCompleted function

```
public static String isNotCompleted(String itemInfo) {

    Element eItem = JDOMConversionTools.stringToElement(itemInfo);

    String status = eItem.getChildText("status");

    return String.valueOf(! isFinishedStatus(status) );

}
```

Figure 7.61: 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 7.62).

```
private static boolean isFinishedStatus(String status) {
    return status.equals(WorkItemRecord.statusComplete) ||

        status.equals(WorkItemRecord.statusForcedComplete) ||

        status.equals(WorkItemRecord.statusFailed) ;

}
```

Figure 7.62: Definition of method isFinishedStatus

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.

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.

7.8   Sample Log (generated by Walkthrough C)

In the following every line started with the date, which has been omitted for readabilty reasons. Similary, "ExceptionService" has been abbreviated to "ES" and "WorkletService" to "WS".

12:16:31,875	[INFO] ES	:-	HANDLE CHECK CASE CONSTRAINT EVENT
12:16:31,984	[INFO] ES	:-	Checking constraints for start of case 20
			(of specification: OrganiseConcert)
12:16:32,093	[INFO] ES	:-	No pre-case constraints defined for spec: OrganiseConcert
12:16:32,109	[INFO] ES	:-	HANDLE CHECK WORKITEM CONSTRAINT EVENT
12:16:32,156	[INFO] ES	:-	Checking pre-constraints for workitem: 20:BookStadium_5
12:16:32,281	[INFO] ES	:-	No pre-task constraints defined for task: BookStadium
12:28:17,968	[INFO] ES	:-	HANDLE CHECK WORKITEM CONSTRAINT EVENT
12:28:18,000	[INFO] ES	:-	Checking pre-constraints for workitem: 20:SellTickets_3
12:28:18,015	[INFO] ES	:-	No pre-task constraints defined for task: SellTickets
12:28:18,078	[INFO] ES	:-	HANDLE CHECK WORKITEM CONSTRAINT EVENT
12:28:18,093	[INFO] ES	:-	Checking post-constraints for workitem: 20.1:BookStadium_5
12:28:18,093	[INFO] ES	:-	No post-task constraints defined for task: BookStadium
12:56:08,000	[INFO] ES	:-	HANDLE CHECK WORKITEM CONSTRAINT EVENT
12:56:08,015	[INFO] ES	:-	Checking pre-constraints for workitem: 20:DoShow_4
12:56:08,140	[INFO] ES	:-	Workitem 20:DoShow_4 failed pre-task constraints
12:56:08,140	[DEBUG] ES	:-	Invoking exception handling process for item: 20:DoShow_4
12:56:08,156	[DEBUG] ES	:-	Exception process step 1. Action = suspend, Target = workitem
12:56:08,171	[DEBUG] ES	:-	Successful work item suspend: 20:DoShow_4
12:56:08,203	[DEBUG] ES	:-	Exception process step 2. Action = compensate,
			Target = ChangeToMidVenue
12:56:08,343	[DEBUG] WS	:-	Worklet specification 'ChangeToMidVenue'
			is already loaded in Engine
12:56:08,546	[DEBUG] WS	:-	Launched case for worklet ChangeToMidVenue with ID: 21
12:56:08,578	[INFO] ES	:-	HANDLE CHECK WORKITEM CONSTRAINT EVENT
12:56:08,593	[INFO] ES	:-	Checking post-constraints for workitem: 20.2:SellTickets_3
12:56:08,593	[INFO] ES	:-	No post-task constraints defined for task: SellTickets
12:56:08,593	[INFO] ES	:-	HANDLE CHECK CASE CONSTRAINT EVENT
12:56:08,593	[INFO] ES	:-	Checking constraints for start of case 21
			(of specification: ChangeToMidVenue)
12:56:08,609	[INFO] ES	:-	No pre-case constraints defined for spec: ChangeToMidVenue
12:56:08,609	[INFO] ES	:-	HANDLE CHECK WORKITEM CONSTRAINT EVENT
12:56:08,640	[INFO] ES	:-	Checking pre-constraints for workitem: 21:CancelStadium_3
12:56:08,656	[INFO] ES	:-	No pre-task constraints defined fortask: CancelStadium
13:02:48,171	[INFO] ES	:-	HANDLE CHECK WORKITEM CONSTRAINT EVENT
13:02:48,187	[INFO] ES	:-	Checking pre-constraints for workitem: 21:Book_Ent_Centre_5
13:02:48,234	[INFO] ES	:-	No pre-task constraints defined for task: Book_Ent_Centre
13:02:48,250	[INFO] ES	:-	HANDLE CHECK WORKITEM CONSTRAINT EVENT
13:02:48,265	[INFO] ES	:-	Checking post-constraints for workitem: 21.1:CancelStadium_3
13:02:48,265	[INFO] ES	:-	No post-task constraints defined for task: CancelStadium
13:10:10,468	[INFO] ES	:-	HANDLE CHECK WORKITEM CONSTRAINT EVENT
13:10:10,484	[INFO] ES	:-	Checking pre-constraints for workitem: 21:Tell_Punters_4
13:10:10,500	[INFO] ES	:-	No pre-task constraints defined for task: Tell_Punters
13:10:10,500	[INFO] ES	:-	HANDLE CHECK WORKITEM CONSTRAINT EVENT
13:10:10,515	[INFO] ES	:-	Checking post-constraints for workitem:
			21.2:Book_Ent_Centre_5
13:10:10,515	[INFO] ES	:-	No post-task constraints defined for task: Book_Ent_Centre

13:13:59,281	[INFO] ES	:-	HANDLE CHECK CASE CONSTRAINT EVENT
13:13:59,281	[DEBUG] ES	:-	Checking constraints for end of case 21
13:13:59,281	[INFO] ES	:-	No post-case constraints defined for spec: ChangeToMidVenue
13:13:59,296	[DEBUG] ES	:-	Worklet ran as exception handler for case: 20
13:13:59,437	[DEBUG] ES	:-	Exception process step 3.
			Action = continue, Target = workitem
13:13:59,468	[DEBUG] ES	:-	Successful work item unsuspend: 20:DoShow_4
13:13:59,515	[INFO] ES	:-	HANDLE CHECK WORKITEM CONSTRAINT EVENT
13:13:59,531	[INFO] ES	:-	Checking post-constraints fo workitem: 21.3:Tell_Punters_4
13:13:59,531	[INFO] ES	:-	No post-task constraints defined fortask: Tell_Punters
13:13:59,546	[INFO] ES	:-	Exception monitoring complete for case 21
13:13:59,750	[INFO] ES	:-	HANDLE CHECK CASE CONSTRAINT EVENT
13:13:59,875	[DEBUG] ES	:-	Checking constraints for end of case 20
13:13:59,953	[INFO] ES	:-	No post-case constraints defined for spec: OrganiseConcert
13:14:00,046	[INFO] ES	:-	HANDLE CHECK WORKITEM CONSTRAINT EVENT
13:14:00,046	[INFO] ES	:-	Checking post-constraints for workitem: 20.3:DoShow_4
13:14:00,156	[INFO] ES	:-	No post-task constraints defined for task: DoShow
13:14:00,171	[INFO] ES	:-	Exception monitoring complete for case 20

Chapter 8

Other Services

Because of the open design of the YAWL interfaces, it is possible to design a YAWL Custom Service to perform the work of a task instance using a wide variety of techniques to meet particular needs. Previous chapters have described the Resource and Worklet 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, their varied purposes and their use. Please consult the YAWL Technical Manual for detailed information on the development of YAWL Custom Services.

8.1 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 to 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 opening the Task's Decomposition dialog in the Editor and choosing the WSInvoker from the list of available services (Figure 8.1. 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 decomposition; 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 parameters. Note that when a task is registered with the Invoker Service at design time, the Editor automatically populates the task's input parameters 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).
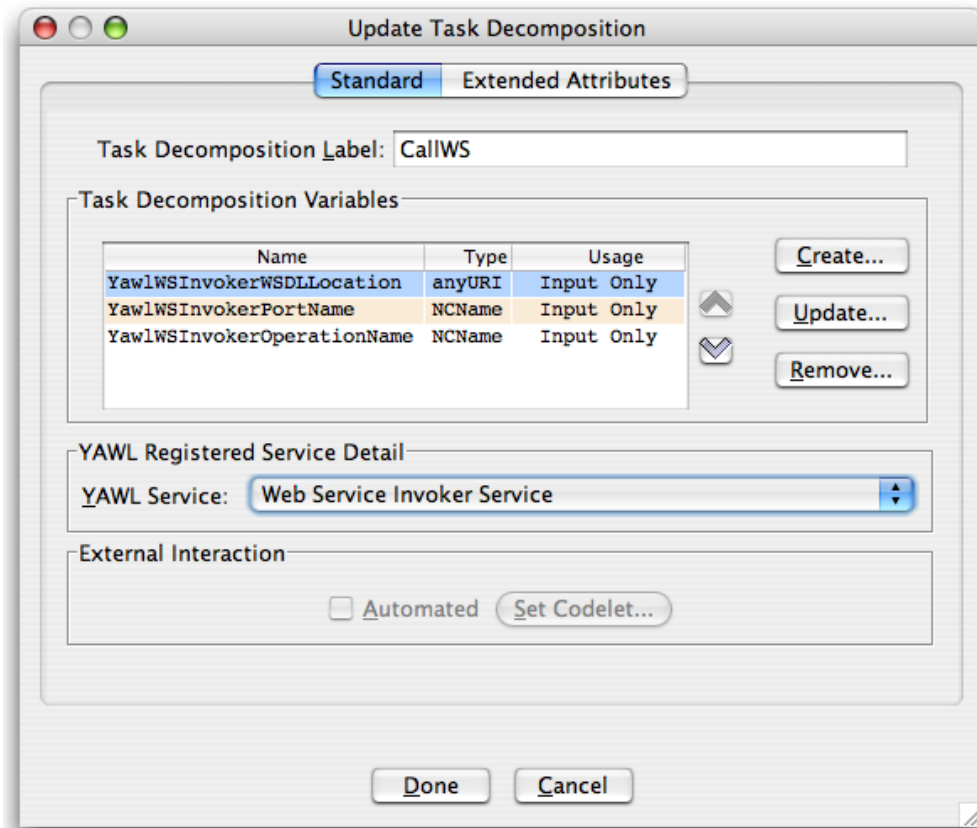
Figure 8.1: Task Decomposition Dialog for WSInvoker-associated task

8.2   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 by opening the Task's Decomposition dialog in the Editor and choosing the SMS Service from the list of available services (Figure 8.2), 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.
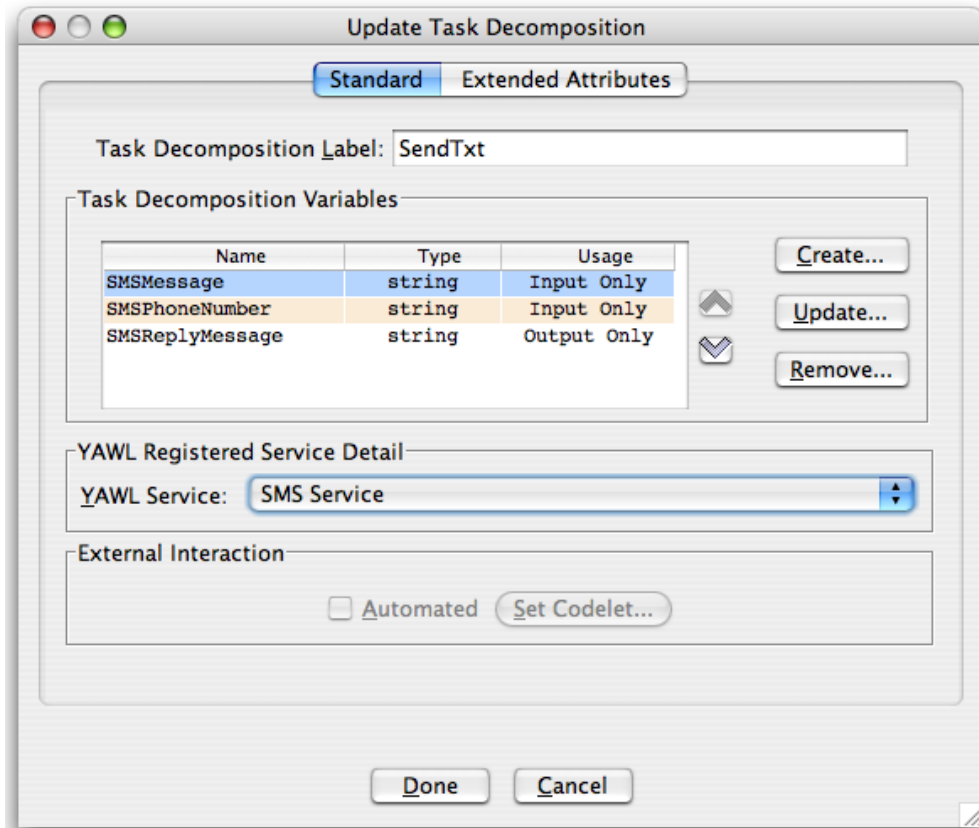
Figure 8.2: Task Decomposition Dialog for SMS Service-associated task

8.3 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 four required variables:

- **status**: The message text to send to Twitter;

- **userid**: A valid Twitter user account name;

- **password**: The password for the nominated Twitter account; and

- **result**: 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, through its API using the specified userid and password, post the status update (if connection was successful) and put Twitter's response text in the result variable.

8.4 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.

8.4.1   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.
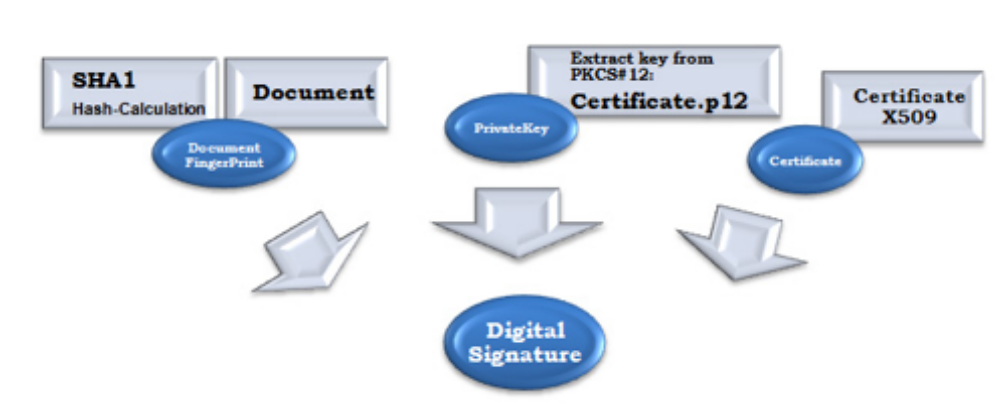

Figure 8.3: An overview of the production of a digital signature

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.

8.4.2   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.

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.
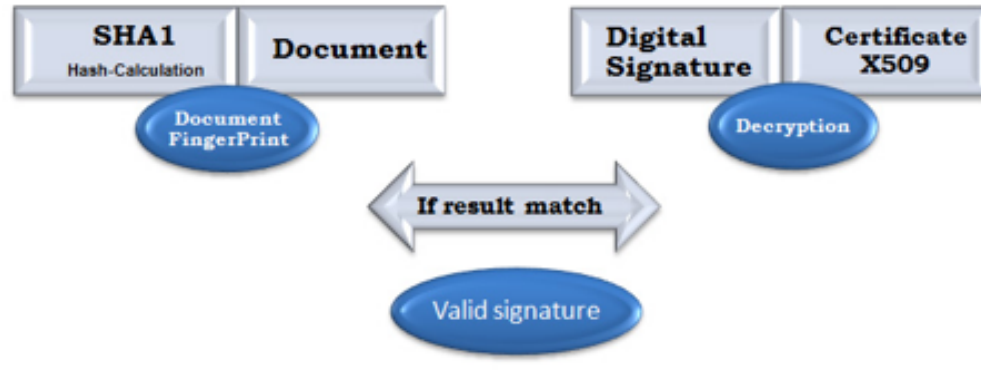
Figure 8.4: An overview of the verification of a digital signature

8.4.3 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.

8.4.4 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 8.5).

Then use this key Store to generate the key pair and the X.509 certificate by filling in the owner information (Figure 8.6).

When the key pair is created you can review the certificate information produced (Figure 8.7).

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 8.8).

### 8.4.5 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 8.9).

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 'Update Data Type Definition' dialog of the Editor, as the example in Figure 8.10:

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 right-clicking on the task in the Editor, choosing 'Set Custom Form' from the menu, and then entering the URI of the form in the dialog shown (Figure 8.11).

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 usage type of the 'Signature' is Input & Output.

You may also have to change the paths in the 'upload.jsp' file: where the default values are 'localhost', they
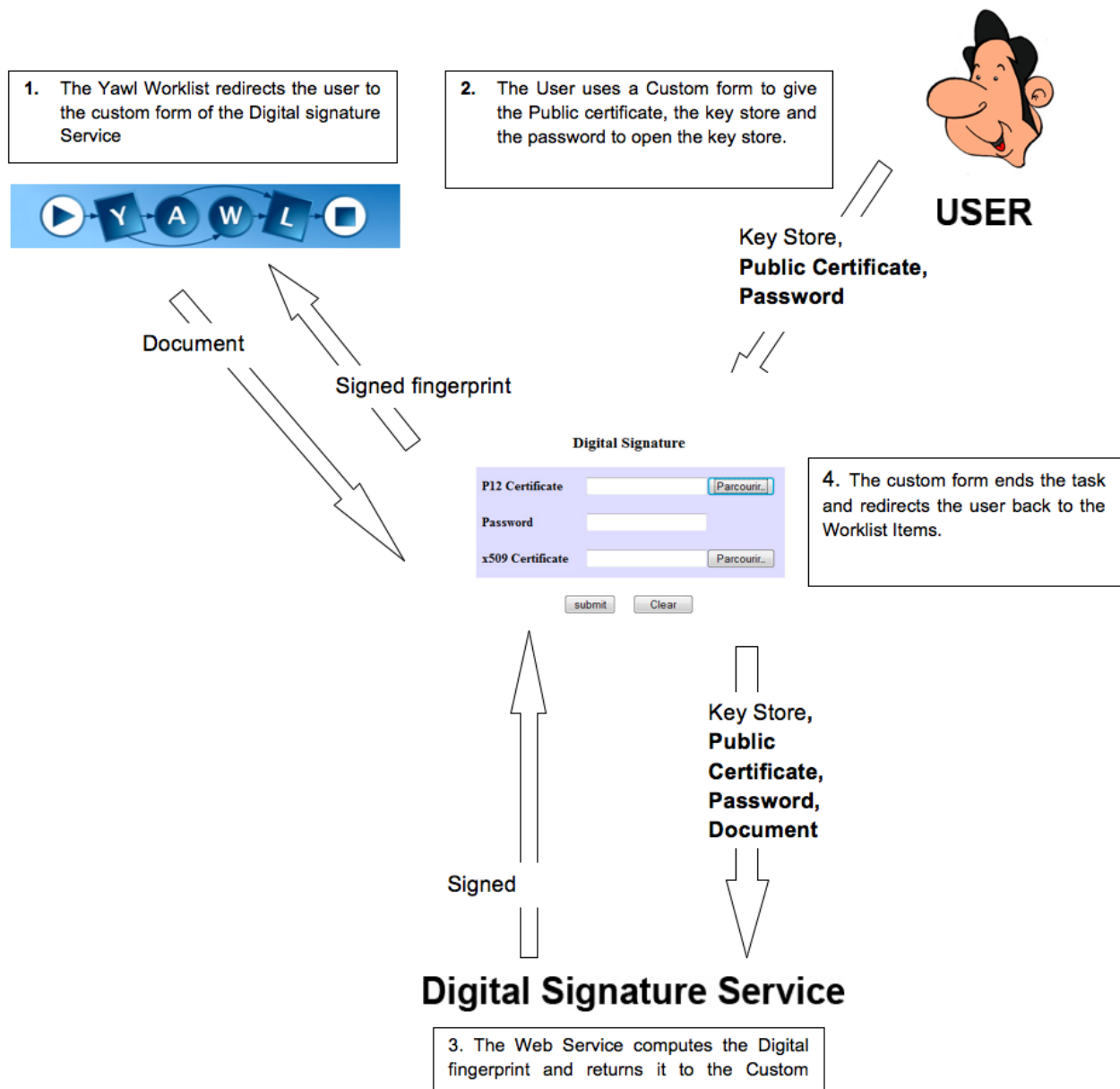
may have to be changed to the actual address where the service runs (if it is not running locally):

```
String Path = "http://localhost:8080/DigitalSignature/files/";
String redirectURL = "http://localhost:8080/resourceService/" +
                     "faces/userWorkQueues.jsp?workitem=" +
                     wir.toXML();
```

The last task is the custom service part; it has to be specified in the 'YAWL Registered Service Details' panel of the Task Decomposition dialog (Figure 8.13).

The Signature will be checked with the certificate loaded in the jsp page and it will provide the document 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.
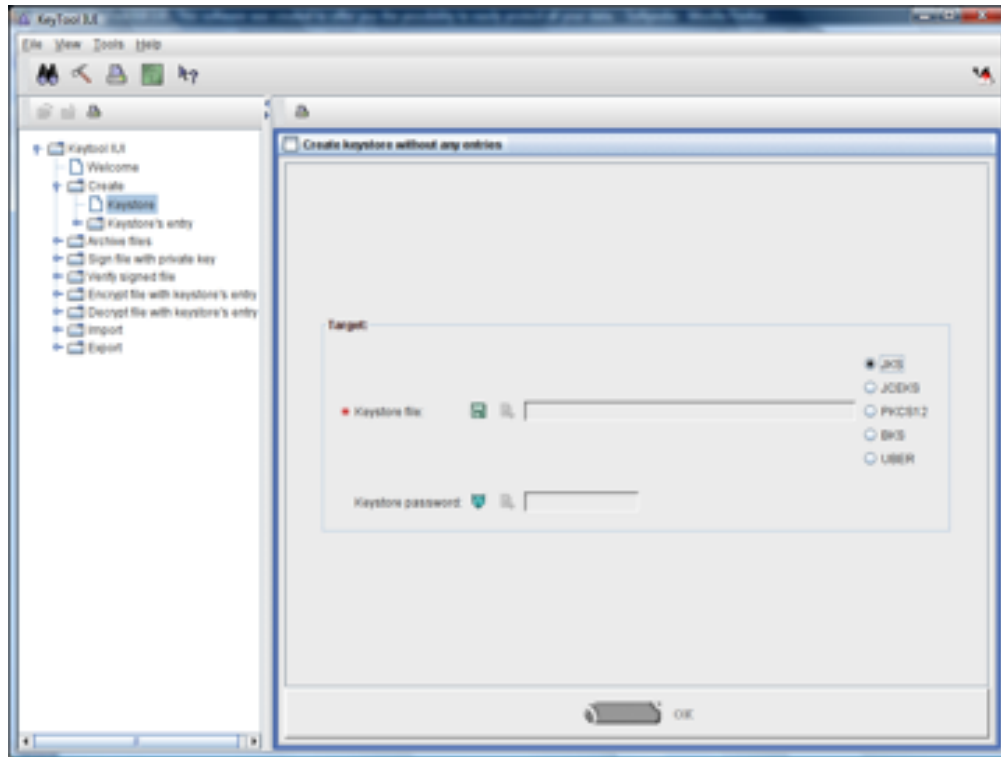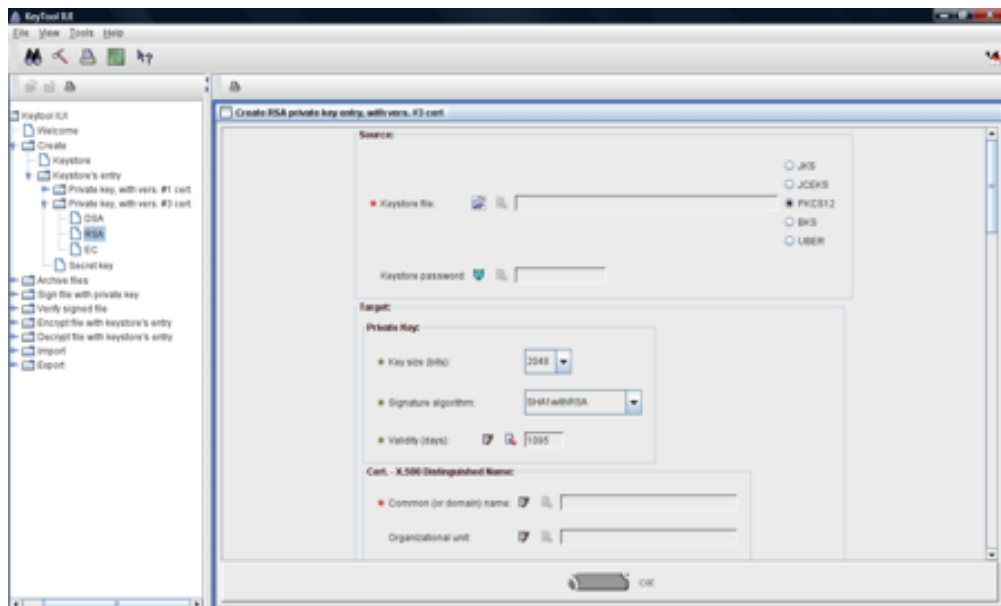
Figure 8.5: Creating an Empty Keystore

Figure 8.6: Generating the Key Pair

8.5   Email Sender Service

The Email Sender Service allows users to send simple emails from within the workflow process instance.
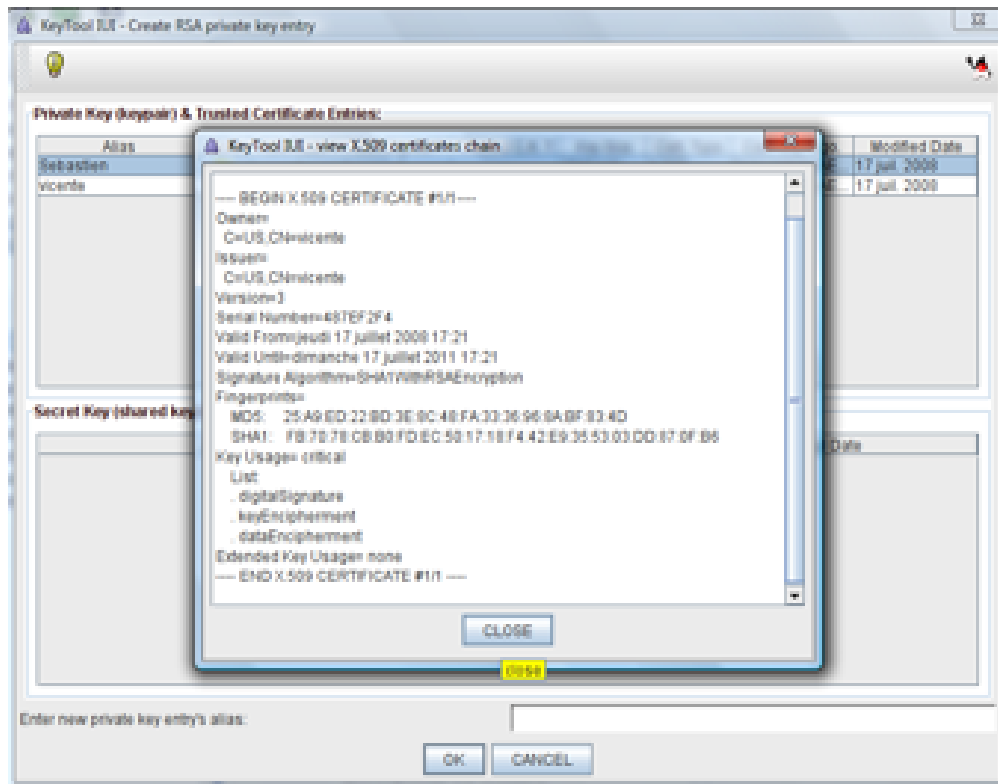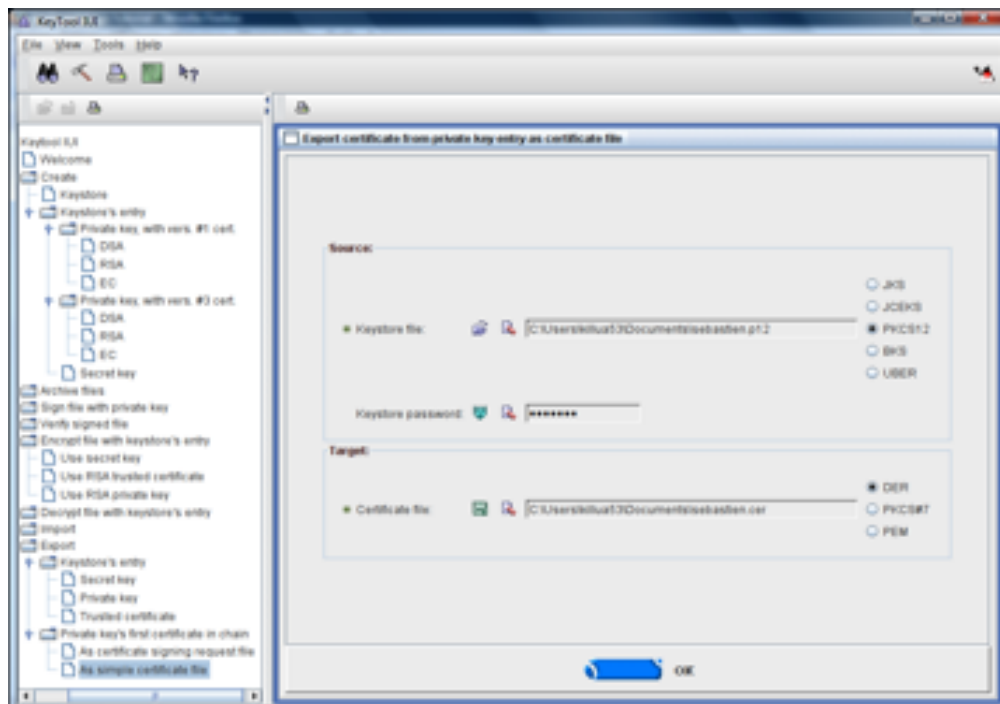
Figure 8.7: The Created Certificate
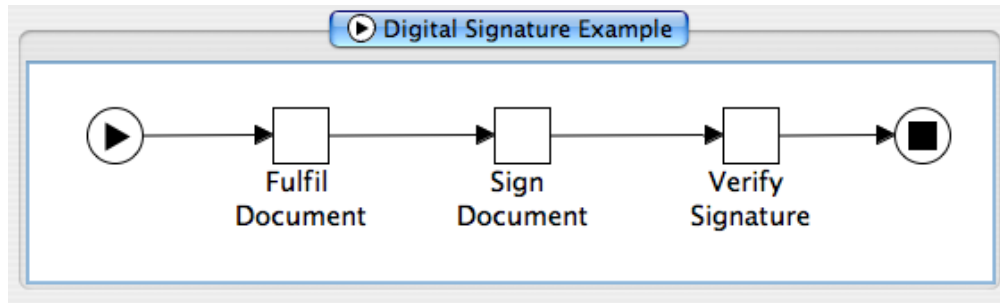

Figure 8.8: Exporting the Certificate
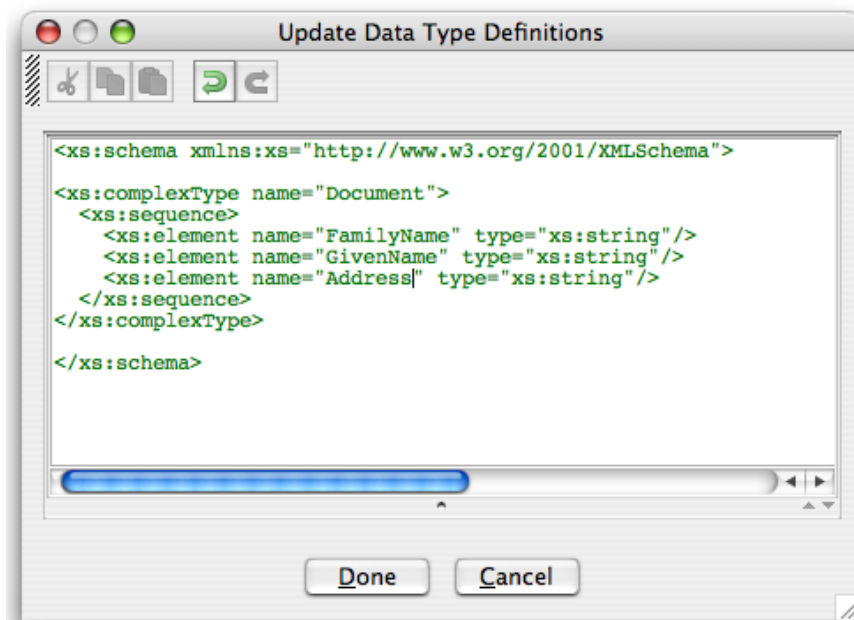
Figure 8.9: Example YAWL Process

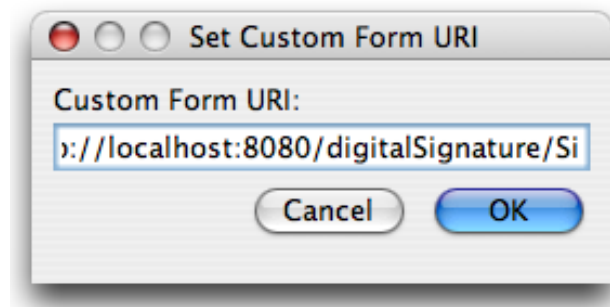**Figure 8.10:** Data Definition for 'Document' type

Figure 8.11: Custom Form dialog in YAWL Editor

8.5.1 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
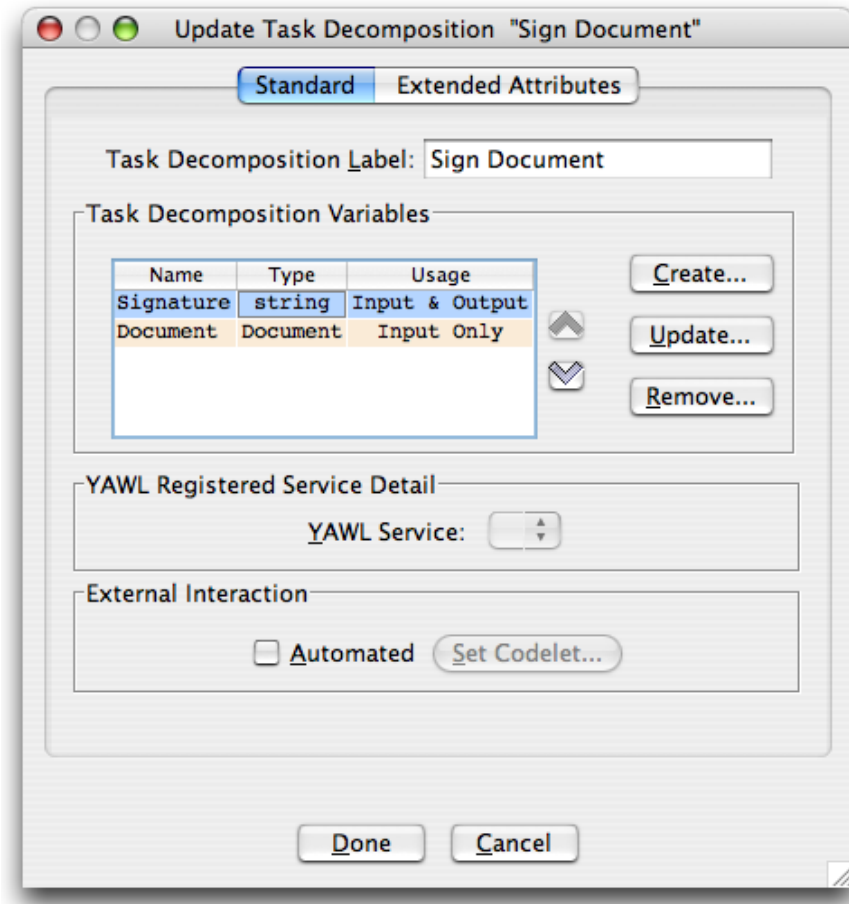
Figure 8.12: Task variables for the 'Sign Document' task

Mail Sender Service (Figure 8.14).

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.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 8.15.

You can see that the form in Figure 8.15 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>
```

Figure 8.13: Task variables and service details for the 'Verify Signature' task
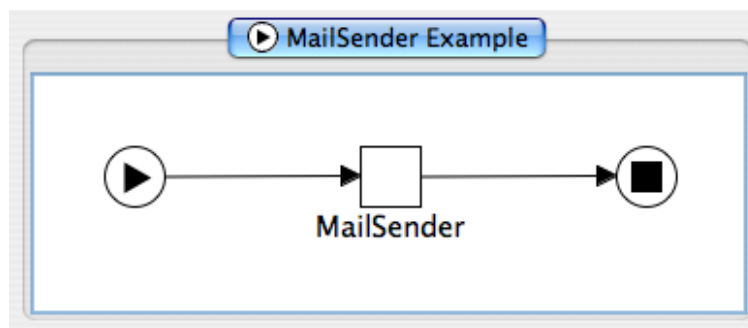
Figure 8.14: Example Mail Sender Process

```
    <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.

Figure 8.15: Example Mail Sender Custom Form

Chapter 9

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 discover 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 sourceforge site (http://sourceforge.net/projects/yawl). 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, listing messages produced in the console window and/or log files, providing a backup of the database, and so on.

The YAWL Issues Tracker (code.google.com/p/yawl) can be used for reporting bugs or proposing enhancements. 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, carefully search 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. The application of Petri nets to workflow management. *Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[2] 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.

[3] 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, 2007.

[4] W.M.P van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods and Systems*. MIT Press, Cambridge, MA, USA, 2002.

[5] 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.

[6] W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet another workflow language. *Information Systems*, 30(4):245–275, 2005.

[7] Michael Adams. *Facilitating Dynamic Flexibility and Exception Handling for Workflows*. PhD Thesis, Queensland University of Technology, Brisbane, Australia, 2007. Available through http://www.yawl-system.com.

[8] Michael Adams, Arthur H.M. ter Hofstede, Wil M.P. van der Aalst, and David 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, 2007.

[9] Michael Adams, Arthur H.M. ter Hofstede, David 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.

[10] 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, 2008.

[11] 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.

[12] T. Murata. Petri nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[13] 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.

[14] J.L. Peterson. *Petri net theory and the modeling of systems.* Prentice-Hall, Englewood Cliffs, USA, 1981.

[15] 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, 2008.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] N.C. Russell. *Foundations of Process-Aware Information Systems.* PhD Thesis, Queensland University of Technology, Brisbane, Australia, 2007. Available through http://www.yawl-system.com.

[21] 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.

[22] M. Weske. *Business Process Management: Concepts, Languages, Architectures .* Springer, 2007.

[23] Moe Thandar 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://www.yawl-system.com.

[24] 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.