REuse and Migration of legacy applications to Interoperable Cloud Services

REMICS

Small or Medium-scale Focused Research Project (STREP)

Project No. 257793

Deliverable D4.3
REMICS Migrate Toolkit, Interim Release

Work Package 4

Leading partner: SOFTEAM

Author(s): Antonin Abhervé
Andrey Sadovykh, Pelle Jakovits, Sébastien Mosser

Dissemination level: Public

Delivery Date: 31-08-2012

Final Version: 1.0

Copyright © REMICS Consortium 2010-2013
# Versioning and contribution history

<table>
<thead>
<tr>
<th>Version</th>
<th>Description</th>
<th>Contributors</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Initial Layout</td>
<td>Softeam</td>
</tr>
<tr>
<td>0.2</td>
<td>Modernization Process Descriptions</td>
<td>Softeam</td>
</tr>
<tr>
<td>0.3</td>
<td>Softeam Tool Descriptions</td>
<td>Softeam</td>
</tr>
<tr>
<td>0.4</td>
<td>UT Tool Description</td>
<td>UT</td>
</tr>
<tr>
<td>0.5</td>
<td>SINTEF Tool Description</td>
<td>SINTEF</td>
</tr>
<tr>
<td>0.6</td>
<td>Delivery Integration</td>
<td>Softeam</td>
</tr>
<tr>
<td>1.0</td>
<td>Review</td>
<td>Softeam</td>
</tr>
</tbody>
</table>
Executive Summary

This document, D4.3 – MigrateToolkit, Interim Release, is a public deliverable of the Project “Reuse and Migration of legacy applications to Interoperable Cloud Services (REMICS)” in “Small or medium-scale focused research project (STREP)” within the European seventh framework program for the ICT Call 5 (FP7-ICT-2009-5) challenge 1 Pervasive and Trusted Network and Service Infrastructures.

This report complements the WP4 prototypes. It gives an overview of WP4 approach to migrating legacy applications to SOA application deployed on cloud computing platforms. It describes the migration process and the tools developed in this project by Softeam, UT and Sintef.

For each tool, this document provides explanations for the role of the tools in the process, an overview of the tool architecture, user documentation, as well as the theoretical foundations on which this tool was built.
# Table of contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXECUTIVE SUMMARY</td>
<td>3</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>4</td>
</tr>
<tr>
<td>1 INTRODUCTION</td>
<td>6</td>
</tr>
<tr>
<td>1.1 INTENDED AUDIENCE</td>
<td>6</td>
</tr>
<tr>
<td>1.2 MIGRATE TOOLKIT IN THE REMICS CONTEXT</td>
<td>6</td>
</tr>
<tr>
<td>1.3 REMICS MIGRATE TOOLKIT OVERVIEW</td>
<td>7</td>
</tr>
<tr>
<td>2 MODERNISATION PROCESS</td>
<td>9</td>
</tr>
<tr>
<td>2.1 MODERNISATION PROCESS OVERVIEW</td>
<td>9</td>
</tr>
<tr>
<td>2.2 RECOVERY PHASE</td>
<td>12</td>
</tr>
<tr>
<td>2.2.1 Recovery Process</td>
<td>12</td>
</tr>
<tr>
<td>2.2.2 Component Model</td>
<td>12</td>
</tr>
<tr>
<td>2.2.3 Component Recovery mechanism</td>
<td>13</td>
</tr>
<tr>
<td>2.3 MIGRATION PHASE</td>
<td>14</td>
</tr>
<tr>
<td>2.3.1 Migration Process</td>
<td>14</td>
</tr>
<tr>
<td>2.3.2 Architecture Model</td>
<td>15</td>
</tr>
<tr>
<td>2.3.3 Design by pattern composition mechanism</td>
<td>19</td>
</tr>
<tr>
<td>2.3.4 Implementation model</td>
<td>20</td>
</tr>
<tr>
<td>2.4 DEPLOYMENT PHASE</td>
<td>21</td>
</tr>
<tr>
<td>2.4.1 Deployment Process</td>
<td>21</td>
</tr>
<tr>
<td>2.4.2 Deployment Model</td>
<td>21</td>
</tr>
<tr>
<td>2.4.3 Domain-Specific Language to Deploy Applications in the Clouds</td>
<td>22</td>
</tr>
<tr>
<td>3 TOOLS CATALOGUE</td>
<td>24</td>
</tr>
<tr>
<td>3.1 MODELIO TOOLS</td>
<td>24</td>
</tr>
<tr>
<td>3.1.1 Migration Process module</td>
<td>24</td>
</tr>
<tr>
<td>MODELIO OFFICIAL MODULE</td>
<td>29</td>
</tr>
<tr>
<td>3.1.2 Component Recovery module</td>
<td>34</td>
</tr>
<tr>
<td>SOAMLDESIGNER</td>
<td>40</td>
</tr>
</tbody>
</table>
3.1.3 Pattern Composition modules ................................................................. 43
3.1.4 Web Designer module .................................................................................. 49
3.1.5 PIM4Cloud Designer module ...................................................................... 52
3.2 CLOUDSCRIPT ................................................................................................ 54
  3.2.1 Motivating example: SensApp ..................................................................... 54
  3.2.2 Deploying at the Software as a Service level .............................................. 55
  3.2.3 Deploying at the Platform as a Service level ............................................ 57
  3.2.4 Infrastructure as a Service ........................................................................... 60
  3.2.5 Off-the-shelf Components (CloudScript StdLib) ....................................... 61

3.3 DESKTOP TO CLOUD MIGRATION TOOL (D2CM) .................................. 64
  3.3.1 Tool approach .............................................................................................. 64
  3.3.2 Migration process ......................................................................................... 64
  3.3.3 Deployment process ..................................................................................... 70
  3.3.4 Performance monitoring ........................................................................... 74
  3.3.5 Getting started ............................................................................................ 76

4 ANNEXES ............................................................................................................ 78
  4.1 TOOLS REFERENCE CARD ........................................................................... 78
  4.2 LIST OF ABBREVIATIONS ............................................................................... 81
  4.3 LIST OF REFERENCES ..................................................................................... 81
1 Introduction
This document presents important information about the REMICS Migrate Toolkit. The document is structured in the following way:

- In the introduction we present the toolkit as a whole and identify its place in the global REMICS process. In addition, we outline the details of the migration process.
- In the getting started chapter we provide detailed guidelines of the individual tools, which are the part of the toolkit.
- Finally we provide the technical information of the current release, such as REMICS requirements coverage, progress report and the roadmap.
- In annexes, we provide the details about methods implemented by the tools.

This chapter intends to situate the toolkit in the global REMICS context and provide a global picture of the toolkit.

1.1 Intended Audience
The first part of the document is general information about the toolkit and thus it is dedicated for a broad community of specialists interested to learn about REMICS architecture and process.

The second part is technical and is dedicated to Migrate Toolkit users since the chapter provides getting started information.

The final part is an administrative overview dedicated to REMICS managers and EC.

1.2 Migrate Toolkit in the REMICS Context
Figure 1 depicts the whole process in the REMICS global context. The steps involved in the Migrate stage are highlighted.

![Diagram of Migrate Toolkit in Global Process](image)

Figure 1 Migrate Toolkit in Global Process

The migrate toolkit is involved from the stage when UML models are available. Through componentization and refactoring SOA models are created. From these models a Web application
is generated. In parallel, the cloud environment is modelled with PIM4Cloud. This process includes specification of the deployment artefacts issued from the Web application generation process. On the next stage the Cloud Script is used from specification of the cloud deployment. Finally, Desktop2Cloud Migration tool is used for deployment and configuration.

1.3 REMICS Migrate Toolkit Overview

Figure 2 depicts various components of the Migrate Toolkit.

Four partners are involved in the development of the Migrate Toolkit. Netfective provides BluAge, which extracts UML models from the legacy code. These UML models are transferred to the SOFTEAM's tools based on Modelio environment. Modelio toolkit regroups refactoring, application generation and cloud environment modelling tools. Through continuous refactoring and refinement process, the model of the Web application is created. This model is used to generate application code. In a parallel process a cloud deployment environment is modelled with PIM4Cloud.

SINTEF's CloudScript is used to define the deployment and specify the configuration scripts. Finally, the Desktop2Cloud Migration tool is used to deploy the cloud images to the infrastructure, configure the installation and manage the application life-cycle.

The flow of the information between the tools is insured with various model transformations.

**At the current stage the link between BluAge and Modelio is fully functional. Links between Modelio and CloudScript as well as between CloudScript and Desktop2Cloud Migration tool are currently manual. The continuous flow will be fulfilled in the Year 3.**

The description of various tool components is provided in Table 1 below.
### Table 1 Migrate Toolkit Tools

<table>
<thead>
<tr>
<th>Model</th>
<th>Migration Process tool</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The MigrationProcess module is the orchestrator of migration process. It offers a support for the approach of migrating legacy applications to SOA application deployed on cloud computing platforms. This module provides: Profile dedicated to the structuring migration project Model transformations functionality between different phases of the migration process. Service of code generation from behaviours expressed as activity diagrams. Integration of model analysis tools, transformations tools and code generation tools involved in migration process.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model</th>
<th>Component Recovery tool</th>
</tr>
</thead>
</table>
|       | The Component Recovery tool covers methods for components identification, discovery of their services, interfaces and dependencies. This module provides:  
- Tools for analysing dependency between elements of the application architecture.  
- Representation of dependencies in the form of dependency diagrams or matrices dedicated to human analysis.  
- Service achieving breakdown of architecture into coarse-grain components providing business services |

<table>
<thead>
<tr>
<th>Model</th>
<th>Pattern Composition tool</th>
</tr>
</thead>
</table>
|       | The Pattern Composition tool is generic solution to involve application of SOA and Cloud Computing design patterns to source architecture. It allows implementing the iterative refactoring of legacy architecture to the targeted Service Clouds architecture expressed with PIM4Cloud SoaML metamodel. The PatternComposition tool is an extension of PatternDesigner module that allows to defining and packaging new composed patterns. This module provides:  
- Option to define a new pattern from a model part  
- Application of existing pattern to legacy architecture model.  
- Creating a complex pattern by composing simple patterns |

<table>
<thead>
<tr>
<th>Model</th>
<th>Web Designer tool</th>
</tr>
</thead>
</table>
|       | The Web Designer module provides modelling services for web interface including:  
- Modelling front-end and backend of web interface  
- Modelling backend behaviour as Activity Diagram  
- Support dedicated to the integration of code generation tools from the web model.  
- Implementation of a JSF (JAVA –J2EE) code generation tool. |
The PIM4Cloud module provides implementation of PIM4Cloud profile, an extension for OMG SoaML in order to address peculiarities of SaaS applications modelling on the platform independent level.

This module provides:
- Modelling deployment of application to cloud computing platform.
- Integration of architectures components into deployment model.

The CloudScript Domain-Specific language supports the modelling of the cloud applications to be deployed using a component-based approach. It allows us to model using high level concepts the different software components to be deployed (e.g., a web server, a virtual machine), as well as their installation dependencies (e.g., this web server will be deployed on this virtual machine).

The main goal of D2CM tool is to support the migration of scientific applications to the cloud and provide a proof of concept for complete migration and deployment of applications to both private and public clouds.

The D2CM tool is developed in Python and provides two main facilities:
- Programmatical transformation of user-supplied virtual machine images to the target cloud image format. Currently it has support for the Amazon Machine Image format and Eucalyptus compatible XEN image.
- Defining the deployment configuration and life-cycle management of distributed applications, hosted on Amazon’s Elastic Cloud Computing (EC2) platform (or compatible infrastructure, such as Eucalyptus).

2 Modernisation Process

This chapter describes the modernisation process developed in context of REMICS project for a legacy application to a service cloud application. The deliverable described in this document consists of a set of tools involved during the migration process. Understanding of the migration process is a prerequisite to tools analysis.

2.1 Modernisation Process Overview

The REMICS Migrate Toolkit provides a generic process for building service cloud applications starting from recovered business models.

This process consists of three distinct phases: the Recovery Phase which is to continue the identification of high level services components and modelling their internal structure, the Migration Phase which includes the refactoring operations of the application architecture and production model implementation of the new application and the Deployment Phase which addresses the deployment of this application on a cloud computing platform.

Any of these phases is composed of several models representing each the application in a different state of migration process. The migration process of application requires two mechanisms:
- **Refactoring into a model** assisted by tools. This step includes the refactoring of the application architecture, the addition of new services, and the implementation of SOA and PIM4Cloud patterns.
- **Automatic model transformation** to move from one model to another, such as the passage of the architecture model to an implementation model.

![Migration process phases](image)

**Figure 3 : Migration process phases**

The recovery phase is based on two models, the recovered model and the component model:

- **A Recovered Model** corresponds to the initial model, resulting from the work of WP3. This model imported into the migration toolkit using XMI format.
- **A Component Model** is the result of the breakdown of architecture into high grain components providing business services. This model is generated from the initial model by model transformation and involves the Component Recovery Tool.

The migration phase involves the architecture model, implementation model, and test model:

- **The Architecture Model** highlights the organisation of the applications, services provided, data types exchanged by these services, data model, and presentation aspects. It is on this model that most of refactoring work is made. At this layer, the SoaML profile is used to model application architecture.
- **The Implementation Model** representing the code of the target application. This model is specific platform of execution. This model is obtained by model transformation from model architecture model. During this transformation, it is necessary to choose the target execution platform of the application. An executable application can be generated from this model but at this level, the context of application deployment is not taken into consideration.
- **The test model** is intended to record the modelling of integration tests and unit tests. These tests will be the results of the integration of the results of work package 6.

The deployment phase involves a deployment model, a distributed implementation model and distributed test mode:

- **The Deployment Model** allows modelling the deployment of the application on a cloud computing platform. Services component identified at architecture layer are used to make the link between the architecture model and the deployment model. Modelling application deployment is based on usage of PIM4Cloud profile.
- **The Distributed Implementation Model** is an improved version of implementation model taking account of deployment constraints of migrated application. This model is obtained by model transformation from architecture model and deployment model.

At this horizontal division of migration process, we add a vertical division thus allowing isolation the major aspects of a service-oriented application. Starting from Model Component of recovery phase, we isolate Data Model, Behaviour and Presentation aspects of implementation. Thereafter, each of these aspects is processed independently during migration process. These are represented in more detailed way in the remainder of this document.

**Figure 4: Separation of Entity, Behaviour and Presentations Aspects**
2.2 Recovery Phase

2.2.1 Recovery Process
The service cloud platforms impose the development paradigm through highly decoupled reconfigurable SOA components. Architecture recovery activities usually result in monolithic and closely coupled architectures. Migration of legacy applications to a cloud computing platform implies deep refactoring of their architecture. To facilitate interaction between the different modules, services or systems while maintaining their individual independence, the main goal is to have several independent and reusable services. The objective of the Recovery Phase is the identification of weakly coupled Model Components extracted from initial recovered model.

This phase starts with the import of recovered (UML) models into Modelio. Thereafter we use the Components Discovery tool on initial recovered model to identify potential Architecture Component by dependency analysis. Once these components are discovered, we identify manually type of each component by separating components of type Presentation, Behaviour and Entity.

This Component Mode is later then used as input to the Architecture Model of migration Phase.

![Figure 5: Recovery Phase](image)

2.2.2 Component Model
The migration process defined in REMICS includes the task of refactoring the architecture of existing legacy applications using design patterns for SOA and cloud computing. The Component Model is resulting of this breakdown activity.

**Input:** The component model is created from initial recovered model using Component Recovery Tool. Initial recovered Model is the model resulting from recovery phases of global REMICS process (Outcome of work performed by WP3). Stored in XML format (EMF UML2), the model is imported into Modelio using standard import / export functions.

The **Component Recovery Tool** is used to extract Component Model from initial model. The usage of this tool is describing more precisely in next chapter of this document.

Model Structure: The Component Model is structured by set of components and each of these contains a sub part of the initial model. As the objective of the breakdown process is to obtain components that can be deployed on distributed architectures, they are weakly coupled. During component discovery process, we identify three types of component: component belonging to Presentation layer, component belonging to Behaviour layer and component belonging to Entity layer.

Presentations Components includes model element representing the modelling of graphical user interface application.
The Behaviour Component includes the modelling of services provided by the application grouped into interfaces. This layer contains also the type definition of data exchanged by services. Services are represented by UML operations and their behaviour is expressed as Activity diagrams.

Entity components contain Data Model of the application. They are represented by means of UML class diagrams.

### 2.2.3 Component Recovery mechanism

The SOA paradigm implies the breakdown of architecture into coarse-grain components providing business services. The breakdown of architecture into components enables scalability and thus makes possible the multiplication of a number of instances of the same component. On the other hand, when creating a cloud application, it is necessary to ensure that the application can run on a set of low performance resources, to cope with network latencies in a loosely coupled environment. This requires the decomposition and decoupling of the application architecture, which is basically very similar to the SOA paradigm.

Our goal is to develop a tool that covers methods for component identification, and discovery of their services, interfaces and dependencies.

To achieve this result through the REMICS migration process, we apply the following methodology inspired by Neeraj Sangal’s approach to managing dependencies:

**Understand the Application**: Obtain a working knowledge of the function and use of the application. As the model does not contain all the information required to make the division into components, the user must obtain an understanding of business processes implemented by the application. This approach is accompanied by documentation generated from the UML model.
Create Business Architecture: The user can expand the model with information on the business aspects of objects handled. These relationships are modelled as business dependencies.

Extract dependencies from UML model: Identify the dependencies of the application architecture elements. These dependencies are extracted from the UML model and from business dependencies added by the user.

Dependency Analysis with DSM: Perform a dependency analysis to determine the groups of model elements with the strongest links. To conduct this analysis, we use the Design Structure Matrix applied to the dependencies listed in steps 2 and 3.

Architectural Remediation: Following the dependency analysis, a distribution of architectural elements into model components is proposed to the user.

Architecture Validation: Once the components have been identified, the algorithm for extracting dependency and dependency analysis is re-applied to these components. The results presented as a dependency matrix are used to validate the division.

2.3 Migration Phase

2.3.1 Migration Process

The purpose of Migration Phase is to define methods and tools for architecture migration of a legacy application to a Service Oriented Application that might be deployed on service clouds platforms.

This phase take as input a Component Model provided by the recovery phase. Using a transformation service provide by Process management tool, the Component Model is transformed into an Architecture Model that involves the separation of "Data", "Behaviour" and "Presentation" aspects of application architecture. The Architecture Model is divided into three sub models that use different representation profile: Behaviour Components are transformed into a SoaML Architecture Model, Data Components into an Entity Model, and Presentation Components into a Presentation Model.

It is on this model that most of architecture refactoring work is made. Change in application architecture, definition of new services, changes to behaviour of existing services are carried out at the SoaML model. This refactoring is done either manually or through tools such as Composition Pattern tools. The application data model can be modified through the SQLDesigner module, a pre-existing tool delivered in commercial version Modelio.
Once the architectural refactoring is completed, The Process Management module offers the possibility to generate the implementation model from architecture model. As the Implementation model is specific to an execution platform, it is necessary to select this platform before starting the transformation. The current version of toolkit provides service transformations for Java J2EE and .NET platform.

The Implementation Model is a model representing the source code of the refactored application. Using pre-existing Modelio tools such as SQLDesigner and JavaDesigner and tools developed in context of REMICS project such as WebDesigner, it is possible to build an executable application from this model. At the Migration layer, the context of the application deployment is not taken into consideration, therefore the generated application is used to test the result of refactoring operations.

From the architecture model, it is also possible to produce a basic template for the Deployment of model (deployment phase).

2.3.2 Architecture Model

The Architecture Model is used to model the organisation of the applications, services provided, data types exchanged by these services, data model and presentation aspects.

The model is divided into three sub models that use different representation profile:

Service Architecture > SoaML profile: The Service Architecture of application is based on SoaML\(^1\), an OMG profile that provides a standard way to architect and model SOA solutions using the Unified Modelling Language.

Presentation Model > BasicWeb profile: The Presentation Model is based on BasicWeb profile, a profile for modelling web aspect of enterprise application and developed by Softeam in the context of REMICS project.

Entity Model > Persistent profile: The Entity Model is based on the Persistent profile, a pre-existing profile provided by Softeam and dedicated to Data modelling and object-relational mapping.

\(^{1}\) http://www.omg.org/spec/SoaML/
2.3.2.1 Service Architecture model

**Input:** The Service Architecture model is created from Behaviour Components extracted from the Component Model by a model transformation service provided by the Migration Process tool.

Transformations Rules applied:

<table>
<thead>
<tr>
<th>Component Model</th>
<th>SoaML Model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td>Participant</td>
<td>Provider and/or consumer of services. In the business domain a participant may be a person, organization or system. In the systems domain a participant may be a system, application or component.</td>
</tr>
<tr>
<td>Interface</td>
<td>Service Provider</td>
<td>The Provider stereotype specifies an interface and/or a part as playing the role of a provider in a provider/consumer service.</td>
</tr>
<tr>
<td></td>
<td>Service Interface</td>
<td>Defines the interface to a Service Point or Request Point and is the type of a role in a service contract</td>
</tr>
<tr>
<td></td>
<td>Service Point</td>
<td>The offer of a service by one participant to others using well defined terms, conditions and interfaces.</td>
</tr>
<tr>
<td>Operation</td>
<td>Operation</td>
<td>Service operation</td>
</tr>
<tr>
<td>Behaviour</td>
<td>Activity Diagram</td>
<td>Activity Diagram use to modelling operation behaviour</td>
</tr>
<tr>
<td>Class (exchange data type)</td>
<td>Message Type</td>
<td>The specification of information exchanged between service consumers and providers</td>
</tr>
</tbody>
</table>

**Model structure overview:** The Service Architecture model is structured following OMG recommendations relating to the organisation of a SoaML project. Applications services are organised around Participant. For each service, we defined a Service Provider and a derived Service Interface. A Participant exposes a public service to others participant using Ports. Data exchange by service is modelled with MessageType.
2.3.2.2 Entity Model

**Input:** The Entity Architecture model is created from Data Components extracted from the Component Model by a model transformation service provided by the PersistentProfile module. You can refer to the module documentation Persistence Profile for more details about this transformation.

Model structure overview: The Entity Model is based on a class diagram containing Entity (Class), attributes (property of entity) and relations (Aggregation, Composition) between Entities. Among the properties of entities, some are reported as Identifiers of the entity.
2.3.2.3  Presentation Model

**Input:** The Presentation Architecture Model is created from the Presentation model extracted from the Component Model by a model transformation service provided by Migration Process tool.

In the Recovered Model, the Presentation layer of the application is represented by set of Activity Diagram structured in order to represent user’s interactions. The transformation process is used to extract an MVC model from these diagrams.

Transformations Rules applied:

<table>
<thead>
<tr>
<th>Presentation Component</th>
<th>Presentation Model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity</td>
<td>Page</td>
<td>Representing a Web Page with associated data model and controller</td>
</tr>
<tr>
<td></td>
<td>View</td>
<td>Root for View modelling</td>
</tr>
<tr>
<td></td>
<td>Model</td>
<td>Data Model associated with view</td>
</tr>
<tr>
<td></td>
<td>Controller</td>
<td>Modelling of behaviour of the view</td>
</tr>
<tr>
<td></td>
<td>Utility</td>
<td>Modelling of utility services used by view</td>
</tr>
<tr>
<td>CallOperation (Server Side)</td>
<td>Controller Operation</td>
<td>Behaviour operation</td>
</tr>
<tr>
<td>ControlFlow (from server to screen)</td>
<td>View Transition</td>
<td>Modelling transition for a view to another</td>
</tr>
</tbody>
</table>

The Recovery Model contains no data on the visual aspect of the user interface; these elements must therefore be modelled manually by the user.

**Model structure overview:** The model is organised around the notion of a page. Each of these pages includes a view, a model, a controller and a utility component.

- **View Package:** Modelling the graphical organisation of the page.
- **Model Package:** Modelling the Data Model associated with view. The model of a page is represented by set of typed attributes.
- **Controller Package:** Modelling the Behaviour functions called by the view as interface providing operations. Activity Diagram is used to model the behaviour of operations.
- **Utility Package:** Utility services used by the view.
2.3.3 Design by pattern composition mechanism

Service-oriented computing has a specific set of strategic goals and benefits associated with it. Most of these goals, such as increasing agility, are well known, as is the fact that to attain these goals, there is a need to design solutions by following service orientation, a distinct design approach tailored to support service-oriented computing. For the same reasons, the deployment of an application on a cloud environment means respecting a number of predetermined schemas.

The migration phase of a legacy application therefore requires a deep refactoring of the application architecture. In order to facilitate this migration phase, we are building on a design pattern catalogue oriented around the SOA and PIM4Cloud patterns.

The methodology developed is based on two main axes, the notion of pattern catalogue and the concept of pattern composition.

**Pattern catalogue**: A pattern catalogue contains a set of problem-solving techniques and provides invaluable insights as to how and when those techniques should be used to help us attain design goals.

SOA Design patterns can be broken down into four types, each of which represents a common scope of implementation:

- **Service Architecture**: The architecture of a single service.
- **Service Composition Architecture**: The architecture of a set of services assembled by a service composition.
- **Service Inventory Architecture**: The architecture that supports a collection of related services that are independently standardised.
- **Service-Oriented Enterprise Architecture**: The architecture of the enterprise itself, to whatever extent it is service-oriented.

**Pattern composition**: By pattern composition, we mean the methods developed in the REMICS project to integrate a given pattern to an existing architecture and patterns for assembling many of them in order to build the architecture of the target application as a set of interconnected patterns.
This method relies on the graphical modelling of patterns:

A. In specialised diagrams, unmask existing architecture elements which will be integrated into the pattern.
B. Select one or more patterns from a pattern catalogue and add it/them to the diagram. This pattern is represented graphically as a component and has a number of provided and required services.
C. Model the dependencies between the elements of the existing architecture and the services required and provided by the patterns. It is also possible to establish dependencies between the provided and required services of two patterns.
D. Apply the transformation service delivered by the Pattern Composition tool to apply the pattern to the current architecture.

2.3.4 Implementation model

The Implementation Model representing the code of the migrated application. This model is specific to the platform of execution and use a formalism specific to the concerned execution platform. For example, the implementation model of a migrated application in the Java EE platform uses formalism made available by the Java Designer (a solution for java code generator and the reverse of Modelio).

From this implementation model, it is possible to generate an executable application.

**Input:** The Implementation Model is result of an automatic model transformation form the Architecture Model. The applied transformation depends on the platform execution of the target application, so it is mandatory to choose this platform before carrying out the transformation. The Migration Process module provides a configurable service transformation. The architecture of this tool is designed to facilitate the addition of new target processing. For more details about transformation engine, see the chapter dedicated to the MigrationProcess tool.

**Model structure overview:** The structure of the model implementation depends on the execution platform of the migrated application. Here is an example of implementation model for Java target.

![Implementation Model](image)

**Figure 12 : Implementation Model**
2.4 Deployment Phases

2.4.1 Deployment Process

The purpose of Deployment Phase is to define methods and tools to manage deployment of migrated application to Cloud Computing architecture. This process is based on the use of PIM4Cloud profile, an extension for the OMG's SoaML, which addresses peculiarities of SaaS applications modelling at the platform independent level.

Modelling the application deployment platform is an independent process which does not depend directly on the work of the migration phase. However, it is necessary to integrate the components identified during implementation phases of migration to models deployment.

The deployment process can start as an independent way or be initialised from the architecture model by a model transformation. In the latter case, the model would be initialised with the deployment units extracted architecture model. From this deployment model it would then be possible to generate a script that allows easy deployment and an initialised infrastructure on a cloud computing platform.

![Deployment Process Diagram](image)

**Figure 13: Deployment Process**

Deploying an application on the cloud leads to a number of constraints related especially to the distributed aspect of the application architecture and to the necessity to manage the communication between these independent components. The Distributed Implementation model aims to address these constraints.

Created from the architecture model and the deployment model by a model transformation, the Distributed Implementation model is a specialised version of the implementation model for an application deployed on Cloud Computing platforms.

In the current version of the toolkit, the issue of generating code from the model of deployment is not yet covered. Softeam's work for the coming year, as part of the REMICS project, will focus on these aspects.

2.4.2 Deployment Model

The Deployment model is an implementation of PIM4Cloud profile performed under the Modelio PIM4Cloud profile and addresses three issues related to deploying an application on the Cloud: modelling the technical architecture of the application, modelling the deployment of this technical architecture on public and private cloud computing platforms, and modelling the physical infrastructure used in conjunction with cloud computing platforms.

**Input:** As explained above, the deployment model is implemented in parallel with the architecture model. It is not the result of the recovery process and therefore it cannot be created by model transformation from previous models. To ease the modelling deployment, we extended the
catalogue provided by Pattern Composition tools, a group of template dedicated to profile templates PIM4Cloud.

**Model structure overview:** The deployment model is structured in two separate packages addressing the Application domain and the Cloud Provider domain:

- The Application Domain allows the modelling of the software infrastructure required for deploying and managing a particular type of an architecture component. This domain will be used for example to model the application framework or operating systems which are deployed with the applications.
- The Cloud Provider domain allows to model and configure services provided by a cloud computing platform.

![Deployment Diagram](image)

**Figure 14 Deployment Model**

### 2.4.3 Domain-Specific Language to Deploy Applications in the Clouds

We named the language *CloudScript*, as it is a Platform-Independent Model dedicated to Clouds. The key idea of the *CloudScript* is to support the deployment of concrete pieces of software in the cloud. Using the Domain-Specific Language (DSL), the application designer models the software to be deployed. In parallel, the infrastructure provider describes the available infrastructure to be used by the application. From a coarse-grained point of view, it means that the designer requires "computation nodes" (e.g., virtual machines) from the cloud, and the infrastructure provider describes such nodes (based on its own catalogue). An interpreter is then used to identify which resources have to be used in the infrastructure to support the requirements expressed by the application designer. The interpreter then does the provisioning, and actually deploys the modelled application. It returns as feedback to the designer a live model of its application, annotated with runtime property bound to each modelled artefacts (e.g., the public IP address associated to a given virtual machine).

To achieve this goal, we use a reduced component meta-model, described in Figure 15. This meta-model is expressive enough to support the modelling of both infrastructure and applicative artefacts in an endogenous way. Components can be scalars or composite, i.e., containing sub-components inside their boundaries. A Component may offer one or more deployment Services, i.e., deployment protocols one can used to deploy other components onto this one (e.g., a servlet container will offer a WAR service to support the deployment of java-based web applications). Obviously, it may require one Service if it aims to be deployed on another one (e.g., a WAR artefact will require a
Components are connected through Connectors. A component can offer and expect a Property, e.g., a database component may expect both username and password, and provide a URL to be remotely accessed. These elements are used at run-time (asked in a deployment descriptor, or filled using the feedback obtained from the underlying cloud infrastructure). In a Composite model, one can express bindings between properties, that is, a formal link between an expected and an offered property. These links (RuntimeBinding) are used at run-time to properly transfer the expected information.

Implementation: This meta-model is intended to be specialised according to the user’s needs, as its intrinsic simplicity makes it easy to introduce it in user’s code. We provide a reference implementation of this approach using the Scala language, exposed as an internal domain-specific language to support the usage of this meta-model in JVM-based languages. The DSL is designed in a modular way, and implements several constructions (e.g., “offering a service”, “containing a component”) as independent modules, implemented as traits. This design supports the evolution of the DSL, as adding a new syntactic construct is assimilated as the mix of a new trait.

Figure 15 CloudScript meta-model
3 Tools Catalogue

3.1 Modelio tools

This section of the document provides details on the tools developed by Softeam for REMICS project. Development tools constituting the Migration Toolkit are based on the Modelio framework. Modelio is an enterprise-class open source modelling environment (UML2, BPMN2, ...) delivering functionality for business, software and infrastructure architects.

Modelio's wide range of modules enables the developer to configure a bespoke solution for specific requirements. A project can be extended by installing "modules" that package MDA extensions and services. The installation of such modules customises the Modelio tool, adding new services, applying dedicated profiles and customizing Modelio's GUI.

Modelio provides a Rich Java API to program new model capacities: Whatever can be done manually within Modelio can also be done programmatically. Model transformation, diagram creation and pattern application can, for example, easily be programmed using the API. New generators can be developed using predefined services for roundtrip engineering. The Modelio editor can be customized by adding new diagram types, a new GUI or new couplings to other tools.

The tools developed in REMICS project depend on the extension capabilities of Modelio. They are delivered as Modelio modules. Developed in Java, these modules provide UML profiles, model transformation algorithms, and other modelling support necessary for implementation of the migration process. These tools are developed in a Modelio open source version 2.2.002.

As part of this deliverable, we have developed five modules that constitute the migration process associated with other services provided by the open source version of Modelio.

3.1.1 Migration Process module

3.1.1.1 Tool Approach

The Migration Process module is the orchestrator of the migration process. It offers support for the migration of legacy applications starting from Business models recovered in WP3 down to SOA, and technical design models, and finally implementation. The module allows integrating the results of the work of others tasks of the project into a coherent whole, and the development of a unified user interface.

The approach is based SOA components composition practices by developing advanced methods for architecture layering, traceability and model-to-model transformations for integrating business models down to the implementation.

---

2 http://www.modelio.org/
Figure 16 Migration Process Model

The migration process is carried out through a number of services including the following:

- **Define a profile dedicated to the structuring migration project**: Develop graphical interfaces and data structures allowing the implementation of the migration phase of the REMICS process. These tools highlight the different stages of the process and the mechanisms used to move from one to another.

- **Recovery**: component identification

- **Migration**: architecture refactoring and implementation generation

- **Deployment**: deployment modelling and distributed implementation generation

- **Integrate functionality delivered by others tools**: The migration process involves several external tools. Some of these were pre-existing and delivered with the open source version of Modelio, or and some are developed in project REMICS context (*ComponentDiscovery, PatternComposition, WebDesigner, DynamicGenerator, SoaMLDesigner and others code generation tools*). The Migration Process module allows you to centralise and combine the functionality provided by these tools.

- **Define model transformations functionality between different phases of the migration process**: The transition between the different phases of the process involves setting up complex model transformations. This tool centralizes model transformation functions using a model transformation engine.

Model transformations implemented by MigrationProcess tool:

**Table 2 Transformations occurring during the migration process**

<table>
<thead>
<tr>
<th>Source Model</th>
<th>Target Model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recovery model</td>
<td>Component model</td>
<td>Discover model components using <em>Component Discovery</em> tool</td>
</tr>
<tr>
<td>Component model</td>
<td>Architecture model</td>
<td>Extract the SoaML Architecture model form the Behaviour components, the Entity model form the entity components and the Presentation model form the Presentation components using the model transformation engine and the services provided by PersistentProfile tool and SoaML tool</td>
</tr>
<tr>
<td>Source Model</td>
<td>Target Model</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Architecture model</td>
<td>Implementation model</td>
<td>Generate a platform specific Implementation model from the Architecture model including service implementation code from activity diagrams.</td>
</tr>
<tr>
<td>Architecture model</td>
<td>Deployment model</td>
<td>Initialise the deployment model with deployable units extracted from the architecture components.</td>
</tr>
<tr>
<td>Architecture model Deployment model</td>
<td>Distributed Implementation model</td>
<td>Generate a platform specific Implementation model from the Architecture model taking into account the constraints related to deployment of an application on a cloud computing platform.</td>
</tr>
</tbody>
</table>

Developer services dedicated to the code generation from behaviour expressed as activity diagrams: One of the heaviest tasks made by this tool is the development of a set of services intended to facilitate the generation of the implementation model from the architecture model. Indeed, at the architecture layer, the behaviour of the service operations is modelled as activity diagrams that appear in the form of code contained into notes at the implementation layer.

This tool provides a complex set of services to ease the implementation of the model transformation function, including its dynamic transformation diagram (Activity, State, BPMN) to code (For more details please refer to chapter 3.1.2 Model Transformation methodology).

3.1.1.2 Model Transformation applied to Activity Diagrams

As part of the project REMICS, and to complete our range of products, SOFTEAM has sought to implement a generic framework for code generation from UML dynamic models (activity diagram, state machine…) and BPMN diagrams.

3.1.1.2.1 Why develop a specific generation framework for Activity Diagrams?

This tool has become necessary for the realization of transformation from the architecture model to the implementation model. Indeed, at the architecture layer, the behaviour of service is modelled as a UML activity diagram as those exist in the form of code at implementation layer.

![Figure 17: RemoveProvider behaviour at the Architecture and the Implementation layers](image-url)
The activity diagram representing the behaviour of an operation can be complex, with an overlap of conditional processing, looping or other data structure. To generate code from data structures induced by the Loop, Alternatives and other patterns of transitions, it is necessary to transform the activity model into structured model.

![Activity Diagram]

**Figure 18 : Structuring a dynamic diagram**

In addition, this issue for handling activity diagrams can be generalised to all types of dynamic diagrams. Once these structures have been discovered it becomes trivial to generate code from the model. For this purpose, development of a specific transformation tool is found necessary. The objectives of the project can be summarised in three points:

1. Define a reference architecture for the code generation and its reverse from diagram dynamics.
2. Define a set of generic services to facilitate the implementation of code generators.
3. Decouple aspects of "Model Treatment" and "Target Generation". Model Treatment aspect combines the functions of graph analysis and structure discovery, and Target Generation addresses issues of code generation to the desired programming language.

### 3.1.1.2.2 Architecture of DynamicCore framework

Architecture of the framework is organized around a component integrating a set of generic services dedicated to generating code from UML diagrams dynamics: The Dynamic CORE component. This component, which integrates with the code generators provided by Modelio, includes a generic pivot model for transformations. This generic model is presented as a union of concepts used by the UML dynamic diagrams.
Figure 19 DynamicCore Architecture

**Dynamic CORE:** Component incorporating a set of generic services dedicated to the code generation from the UML dynamic diagrams. This component includes a generic model used as pivot for transformations. The DynamicCore provides two extension points: an extension point that enables the registration of a service transformation from a dynamic diagram to pivot model, and an extension point which enable registration of service code generation from the pivot model.

**Generic Model:** The generic model is a union of concepts used by UML dynamic diagrams. It can be extended by specific components to solve specific business problems.

**CodeSpecific components:** A component offering specific code generation and reverse services from the pivot model. It treats specific target language generation such as Java, C++, C# or XML.

**Models Specific components:** A component offering model transformation services between specific UML models and the pivot model. This component provides a generic algorithm for extracting a structured model from a directed graph. This algorithm can be applied on any type of UML dynamic model and transform it into a structured pivot model.

The code generation process therefore involves the following steps:

1. Apply the structure discovering algorithm on the activity model
2. Instantiate a structured model based on *GenericModel* meta-model from the activity model.
3. Generate target implementation code by browsing the structured model

### 3.1.1.2.3 Structure discovering algorithm

The most interesting feature of this tool is the analysis algorithm of the graph which allows discovery of the existing controls structures. This algorithm is used by the ModelSpecific component and is based on an analysis of execution paths of the graph using matrices. In current tool, this algorithm allows us to create a structured pivot model from UML dynamic diagram. The algorithm works as follows.

**A.** Discover and store in a matrix all possible execution paths of the input graph
   
   **a.** Start from a unique element having no incoming transition
b. Browse outgoing transitions and add elements connected in the path

c. Whenever there are 2 outgoing transitions, duplicate the current execution path and go through the two alternatives.
d. If it falls on an item already in the current path, ignore this, it's a loop.

B. Discover data structures from the paths matrix

a. Browse the columns of paths matrix
b. Cut the matrix into three sub tables
   i. The prefix sub matrix: It gathers columns whose elements are identical to the beginning of the table
   ii. The variable sub matrix: It contains the columns whose elements are different
   iii. The suffix sub matrix: It contains the columns whose elements are identical to the end of the table

c. If there is at least one prefix or suffix sub matrix: The variable sub matrix represents an alternative to the graph
d. Otherwise distribute the matrix rows into subgroups until the discovery prefix or suffix (or until each group reduces to a single line). Each sub matrix then corresponds to an alternative.

C. Reapply the previous algorithm to the identified sub matrix

D. Discovering Loops

a. For each sub matrix, if there is a transition starting from a member of the row and ending towards a predecessor element of the same row, there is a loop between the two elements.

3.1.1.3 Getting Started

- Prerequisite

Using ProcessMigration tool requires the following dependencies:

a. Install the JDK 1.6 or higher
b. Download and install Modelio 2.2.00 open-source

The ProcessMigration module depends on the other the following modules:

<table>
<thead>
<tr>
<th>Module</th>
<th>Type</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaDesigner.mdac</td>
<td>Modelio Official Module</td>
<td></td>
</tr>
<tr>
<td>PersistentProfile.mdac</td>
<td>Modelio Official Module</td>
<td></td>
</tr>
<tr>
<td>SoaMLDesigner.mdac</td>
<td>Modelio Official Module</td>
<td></td>
</tr>
</tbody>
</table>

---

Creating a new Modelio Project

1. Click on “File\New project…”.
2. Select a project type. Project types preconfigure projects by automatically deploying certain modules. For example, a “Java” type project automatically deploys the Java Designer module.
3. Enter the name of the project.
4. You can choose to define a project location outside the Modelio workspace here.
5. Enter the author and the description of the project.
6. You can choose to use the SVN Teamwork Manager feature. This will deploy the Teamwork Manager module in your project.
7. You can choose to import an initialization model (available with certain project types).
8. Click on “Create” to create and open the project.

Figure 20 : Create Modelio Project
Installing modules

To deploy a module:

1. Click on “Extensions\Install a module”.
2. In the “Open” window, select the module you want to install.
3. Click on “Open” to run the installation procedure.

Create Migration Project command

The Migration Project offers support for the approach of migrating legacy applications to SOA application. To create a new Migration Project, use the "New Migration Project" command under the root package of Modelio project. A new migration project template will be created automatically.

Import XMI Model command

This function allows you to import a recovered application model into Modelio migration project using XMI exchange formats. The imported XMI model should correspond to an UML model of the recovered application and provided by the results of work package 3.

To import this model:

a. Select a XMI file in EMF UML3 format or older.

b. Import the application model

Extract Component Model command
This function controls the model transformation between the initially recovered Model and the Component Model. Applied to the initial model, it uses the services provided by the Component Discovery tools to produce a new Component Model.

For more details about the Component Discovery tool, refer to Section 3.4.2 of this document.

![Component Discovery Interface](image)

**Figure 23 Component Discovery Interface**

Once the application components are identified, you can assign a type through the property page of the MigrationProcess module.

![MigrationProcess property page](image)

**Figure 24 : MigrationProcess property page**

- Extract Architecture Model command

The Extract Architecture Model function triggers the transformation of the Component Model to the architecture model. Transformation algorithms applied are different depending on the type of component (Behaviour component, Presentation component and Entity component). Therefore, it is necessary that the components are correctly typed before triggering the transformation.
**Figure 25 : Component Model to Architecture transformation**

- Extract Implementation Model command

This command allows you to create an Implementation Model from the Architecture Model. In contrast to the architecture model, which does not depend on an execution platform, the implementation model is linked to a set of execution frameworks.

You must initially choose the target framework among J2EE, .NET or others more specialised frameworks. In a second step you can choose for each of the technologies involved in this framework an implementation among the list of available implementations.

Example: Choose the J2EE Framework and using JPA to manage object relational mapping.

**Figure 26 Framework selection**

- Extract Deployment Model
The **Extract Architecture Model** function triggers the initialisation of the Deployment Model based on the Architecture Model.

**Figure 27**: Architecture model to Deployment transformation

### 3.1.2 Component Recovery module

#### 3.1.2.1 Tool Approach

Service cloud platforms impose a new development paradigm through highly decoupled re-configurable SOA components while the architecture recovery activities usually result in monolithic and closely coupled architectures. Therefore a tool providing components discovery services was required.

The Component Recovery tool provides methods for components identification, discovery of their services, interfaces and dependencies. This tool is the first in the process of legacy application architecture decomposition into components compatible with cloud computing platforms, and achieves the breakdown of the application architecture into high level components providing business services.

The Component Recovery tool operates on the tree following principles: the dependence discovery between the model elements of the application architecture, the representation of dependencies in the form of dependency diagrams or a dependency matrix dedicated to human analysis, and the exploitation of this matrix to suggest reorganization application architecture into components.

A key point is that all the information required to achieve a division into business components cannot be discovered by analysing the input model. Some business constraints are not expressed in the original model. This is why this tool allows the user to define these constraints between the model elements, either in defining the dependencies of the business processes steps or by acting directly on the composition of a part of models components.

- **Tool Input**: The Component Recovery tool takes as input a UML model representing the architecture of the legacy application provided by recover phases of the REMICS migration process.

- **Search for Dependence**: The methodology for automatic discovery of dependencies between services based on dependency analysis of the UML input model.

The tools aims to integrate concepts of consolidation of services based on business functionality. However, these concepts are not directly extractable from the code, and must involve a human operator. The process therefore integrates into a single methodology the dependencies extracted from the UML model and the business dependencies outcome of the manual intervention.
Dependency Analysis: The analysis of these dependencies and the division into components is performed using the Design Structure Matrix, a mathematical tool that allows the relationships between elements of a system to be displayed in a compact and analytically advantageous format.

Dependency analysis is performed by applying a processing algorithm to the Architecture Design Structure Matrix thus helping to identify groups of model elements strongly coupled. At the end of this analysis, the tools suggest a breakdown of the original application into components.

3.1.2.2 Component Discovery methodology

3.1.2.2.1 Dependency Analysis for Architecture Decomposition

The extraction and exploitation of dependencies has been a subject of research since Parnas first formulated the notion of inter-module dependency in his early papers. The extraction of a dependency model from a UML model is based on two prerequisites: the definition of the units of the analysis, and the definition of business rules that characterise a dependency between two of these.

The first step is to define the units of analysis on which we will apply the dependency analysis. A UML model is a strongly connected model. In this context, it is necessary to work and apply the dependency extraction algorithm to a subset of the model. The choice of the elements to include in this subset is essential because these elements will become the atomic elements from which we will apply our component discovery algorithm.

The second step is to define a set of business rules that allow us to identify the dependency between two of these atomic elements. The definition of these business rules allows the user to distinguish those dependencies that are problematic because they violate architectural assumptions from those that are expected and reasonable. Some of these business rules are generic and can be applied to any UML model, while others are specific to the format of the analysed model. Here are some examples of generic business rules used for most of the analyses:

- When two elements that belong to two different components are linked, a link, which is called an abstract link, is established between these two components. An abstract link is a link between two different components containing any two linked elements, each belonging to one of the two components.
- If an element A is a redefinition of an element B, then element A depends on B.

Once these two prerequisites have been satisfied, we apply the following algorithm on the selected model element subset. Defined business rules are used in steps (b) and (c) to determine the dependency relationship between elements:

A. Retrieve the root element from which the analysis is initially launched. Let (X) represent the current level of analysis and it is initially equal to 0 (representing the root element).

B. For each element level (X):
   a. Get the type of the element being analysed.
   b. Add in (X + 1) the direct dependencies of the current element.
   c. If it is a composed element (an element with sub elements):
      - Analyse the dependencies in the component or sub components of the element.
      - Add abstract dependencies at (X + 1) level.
   d. Increment (X).

C. If (X) is not equal to the level entered, or if there are no more dependencies to visit, or if the root element is reached, go to step B, or else stop.

We are unable to extract all the information from an input data model that would be needed to fully realise the automatic discovery of components. Some information, such as the business role of the
architectural elements, is, in any case, not present in the input model. For this reason, the intervention of a user, who knows the business side of the application being upgraded, is required. To support this approach, the tool offers several documentation generators from the UML model. Generated documents are used to guide the user during the component discovery process, and to highlight the essential information extracted from the UML model.

3.1.2.2.2 Applying Design Structure Matrix to System Decomposition problems

3.1.2.2.2.1 Design Structure Matrix

A Design Structure Matrix is a mathematical tool that allows the relationships between elements of a system to be displayed in a compact and analytically advantageous format. A Design Structure Matrix is presented as a square matrix with identical row and column labels. The relationships between the system components are represented by a value at the intersection of the rows and columns of the matrix. Once initialised from data collected during the phase of dependency analysis between system components, we will be able to apply several algorithms to the Design Structure Matrix in order to highlight the coupling relationships between elements of the system. Using this mechanism, we will be able to propose to the developer a decomposition of legacy application architecture into consistent subsystems.

The design structure matrix (DSM), also referred to as dependency structure method and dependency structure matrix, is an accepted method that enhances and analyses the design of products and systems. The use of the design structure matrix in system modelling can be traced back to Warfield in the 1970s and Steward (1981). Interest in this mathematical tool grew in the 1990s. Eppinger (1994) used a matrix representation to capture both the sequence of, and the technical relationships among, design tasks, which was then analysed in order to find alternative sequences or definitions of the tasks. Eppinger (2001) also used DSM in the context of project management. The DSM lists all constituent subsystems or activities and the corresponding information exchange and dependency patterns.

![Design Structure Matrix](image)

**Figure 28**: Design Structure Matrix applied to a flow chart

There are several types of Design Structure Matrices developed to handle problems of different kinds, such as product developers, project planners, project managers, system engineers, and organizational designers:

**Architecture DSM (Component-Based)**: Used for modelling system architectures based on components or subsystems and their relationships.

**Team-Based or Organization DSM (Team-Based)**: Used for modelling organisation structures based on people or groups and their interactions.
Schedule DSM (Activity-Based): Used for modelling processes and activity networks based on activities and their information flows and other dependencies.

Throughout the REMICS project, we focus specifically on the Architecture Design Structure Matrix, which is very appropriate for modelling systems based on components. After being initialised with information regarding the dependencies between the elements of the system, the DSM becomes a good basis for the execution of a set of matrix operations that allow relationships between elements of the system to be identified.

In part of the REMICS project, we will use “Partitioning”, “Banding” and “Clustering” type processing algorithms. The use of these algorithms is described in Chapter 2.2.3: “Processing Algorithms of Architecture Design Structure Matrix”.

3.1.2.2.2 The Architecture DSM model

An Architecture Design Structure Matrix is a type of matrix that can be used to represent interactions among elements in system architecture. It can document system architecture in terms of the relationships between its constituent elements. The final goal of this approach is to highlight the system decomposition into subsystems.

The construction, initialisation and exploitation of an Architecture DSM in the component discovery process involve three steps:

A. Decompose the system into elements: With regards to the REMICS project, we work from a UML model that is already structured into services. These services are the elements we want to reorganise into components. We use them as the rows and columns of the matrix elements.

B. Collect and document the interactions between the elements: In the context of the REMICS project, collected interactions come from two sources: the UML model and business data provided by the user. We use the relationships resulting from the dependency extraction phase from a UML model (see 2.1.2 Dependency Extraction based on UML Metamodel).

C. Analyse the reintegration of the elements: We use the clustering, banding and partitioning algorithms on the matrix in order to bring up parallel, sequential or interdependent groups of services.

![Figure 29: Analyze application architecture using DSM](image)

For the REMICS migration process, we seek to manipulate two different types of dependencies, those resulting from the exploitation of the UML model and those modelled by business users. These two types of interaction data are not as important when it comes to implementing the processing algorithms of the matrix.

We must be able to differentiate between the two types of relationships between elements stocked in the matrix. Pimmler and Eppiger suggest two ways of doing this. Firstly, we can use a single
A three-dimensional matrix which can represent multiple types of interaction data if each off-diagonal cell contains a vector.

A second option is to use a quantification scheme in association with the matrix. Off-diagonal square marks in the matrix are replaced by a number, an integer that can associate a weight to the dependency between elements.

3.1.2.2.3 Processing algorithms of Architecture Design Structure Matrix

To realise the dependency analysis, we apply a processing algorithm to the Architecture Design Structure Matrix. Once created and initialized, the design structure matrices are found to be an excellent support for the “Partitioning”, “Banding” and “Clustering” algorithms. Each of these algorithms allows us to highlight different aspects of a relationship between elements of the system.

Clustering Decomposition: The clustering algorithm can provide a new organisation into system decomposition. The main objective is to maximise interaction between elements within clusters and minimize interaction between clusters. This algorithm also allows the size of clusters to be minimised.

Partitioning Decomposition: The partitioning algorithm is the process that brings together components in order to have the least data flow going back. It provides an automatic mechanism for architectural discovery in a large model. Partitioning eliminates cycles by forming subsystems. The groupings and orderings recommended by these algorithms can be applied in a straightforward manner, to reorganise the code base so that its inherent structure matches the desired structure.

Banding Decomposition: The banding algorithms (band decomposition) offer an organisation of components that provides the best execution path. The components belonging to a group are independent and can run in parallel. Components belonging to different bands run sequentially and cannot run simultaneously.

3.1.2.3 Architecture

The Component Model Recovery Tool is presented as an extension of the Modelio modelling platform. It is based on Eclipse plug-in technology (deployed in an RCP) and module technology specific to Modelio.

The architecture of the tool is organised around four main components: the Component Recovery Editor component, the Dependency Analyser component, the Component Discovery Engine component and the DSM Management component.
Figure 30 The component model recovery plug-in

Component Recovery Editor: The Component Recovery Editor manages interactions between the user and the component discovery process. As we are unable to extract all information from input data model that would be needed to realise a fully automatic discovery of components, interactions between users and the processes are essential and common.

Dependency Analyser: The Dependency Analyser component manages the analysis operation of the UML model to extract the dependencies between model elements. It is used to manage a dynamic set of business rules used to define the notion of dependency between analysed model elements. It is possible to enable or disable the business rules during dependency analysis, in order to adapt the process to the architecture of the analysed application. This component provides a Design Structure Matrix at the end of analysis operations.

DSM Management: The DSM Management component manages the persistence and execution of elementary operations on the Design Structure Matrix.

Component Discovery Engine: The Component Discovery Engine manages the application of algorithms applied to the design structure matrix to provide a breakdown of architecture application into components. To accomplish this task, it manages a set of algorithms for matrix analysis. The user chooses to use one or other of these algorithms based on the expected type breakdown

3.1.2.4 Getting Started

- Prerequisite

Using ComponentRecovery tool requires the following dependencies:

- Install the JDK 1.6 or higher
- Download and install Modelio 2.2.00 open-source
Table 4 Tool Dependencies

<table>
<thead>
<tr>
<th>Module</th>
<th>Type</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>SoaMLDesigner</td>
<td>Modelio Official Module</td>
<td></td>
</tr>
</tbody>
</table>

- **Installation guide**
  
  See section [ ] of this document.

- **Refactor Model**
  
  ![Refactor Model](image)

  The Refactor model command is used to initialise the components analysis process. It creates a copy of the analysed model and prepares it for process analysis.

- **Manage Business Dependency command**
  
  ![Manage Business Dependency](image)

  The Manage Business Dependency command opens an interface for editing business dependencies behind elements of input model. These dependencies are taken into account by discovering components algorithms.

  Through a diagram it is possible to model the relationships between model elements:

  1. **Required Link**: A required link is used to demonstrate a relationship of dependence between two elements
  2. **Exclude Link**: An exclude link is used to demonstrate a relationship of exclusions between two elements.

  ![Figure 31: Business Dependency management diagram](image)

- **Recover Component command**
  
  ![Recover Component model](image)
The Manage Business Dependency command opens the Component Discovery Interface

![Component Discovery Interface]

Figure 32: Component Discovery view

a) **Create business components**: The user is able to create and populate their own model of a component to meet business requirements

b) **Select the elements** to analyse

c) Select the rule applied in dependency analysis: during the process of dependency discovery, the user is able to choose what characterises a dependency between two elements via this interface. It can enable or disable parts of the rules applied in the process.
d) Begin the process of dependency analysis

e) Visualize and update dependency through a matrix display

f) Visualize dependencies as Dependency Diagram: Create a dependency diagram from the analysis
g) Select the dependency analyses algorithm to apply: Selection of the discovery component algorithm from Clustering Decomposition, Partitioning Decomposition and Banding Decomposition algorithm.

h) Apply dependency analysis algorithm

i) View the breakdown proposal

j) Accept and apply the breakdown into components: Apply the suggested decomposition creating the selected components and moving the model elements under to these components.

3.1.3 Pattern Composition modules

3.1.3.1 Tool Approach

At the architecture layer, the legacy application must be iteratively refactored and transformed to the targeted Service Clouds architecture. This process involves application of SOA and Cloud Computing design patterns to the source architecture. To solve this problem, we have developed the Pattern Composition tool.

A pattern represents a piece of a UML model which can be reproduced as many times as necessary in different environments, in the same way as a rubber stamp reproduces the same image again and again. A pattern can also have parameters to adapt it to different contexts when it is applied, just like the date can be changed every day on a rubber stamp. It is the standardised answer to a known problem which tends to appear often.

The implementation consists of two modules: The PatternCompositionDesigner tool which allows the creation of new patterns, and the PatternComposition tool which manages a pattern catalogue, allowing the application of these patterns to existing models and provides services for pattern composition. The PatternCompositionDesigner is an extension of a tool provided by Modelio which allows the creation of new patterns and adding support for extension points.

There are two types of extension points: required and provided. Required extension points allow the integration of the pattern to existing models, while provided extension points are used in the pattern composition mechanism.

![Pattern example](image)

**Figure 35: Pattern example**
Each pattern is identified by its name, its version and its dependencies, i.e. the list of external modules required in the current project to apply a pattern. These dependencies appear if the pattern has, in its definition, elements belonging to an external module. A pattern is associated with a category and has a description.

These two tools are organised around the following services:

- **Pattern Creation**: Creates a new pattern adapted to specific requirements.
- **Definition of Parameters**: Adds parameters on models, code extensions or documentation.
- **Definition of Extension point**: Allows the integration of an existing pattern with the model and the composition of this pattern with other patterns.
- **Pattern Composition**: Creates complex patterns by simple aggregation of patterns using extensions points. The composition pattern is graphically realised by means of an assembly diagram.
- **Provide a pattern catalogue**: Provides a catalogue of patterns dedicated to SOA Application Architectures and cloud computing platform manipulation.

### 3.1.3.2 Pattern Concepts

The storage format of a pattern is composed of two elements: a manifest and a file containing pattern implementation. The manifest is an XML file containing all the meta-data associated with the pattern, such as the version, name, description and external dependencies. It also contains information about the parameters and extension points of the pattern. This file is used at instantiation to resolve the relationship between parameters and pattern implementation.

The implementation of the pattern is stored in a special way: For reasons of pattern instantiation engine performance, we decided not to store patterns in any description format and rather use the Modelio Java API for manipulating a UML model. The template is stored in the form of Java code needed to instantiate these patterns. This code is then supplied as a compiled jar.

An example of a simple pattern: a class whose name is configurable associated with an attribute typed by an external type.

```java
public void createModel(HashMap<String, Object> templateParameters) {
    Class element_1 = model.createClass(); // Class creation
    element_1.setVisibility(ObVisibilityModeEnum.PUBLIC);
    element_1.setAbstract(false);
    element_1.setName("" + (String) parameters.get("$(name)")); // Replace name by parameter
    element_2 = model.createAttribute(); // Attribute creation
    element_2.setName("Attribute" + "");
    element_2.setValue("");
    element_1.setOwner((IPackage) parameters.get("Test")); // Set root of pattern as parameter
    element_2.setType((IClass) parameters.get("Type")); // Replace attribute type by parameter
    element_2.setOwner(element_1);
}
```
When packaging the template, we generate the corresponding Java code and compile it. At instantiation, we just have to load the jar and invoke the `CreateModel` function by introspection with the required parameters. The lists of required parameters were extracted from the metadata file.

### 3.1.3.3 Getting Started

#### Prerequisite

Using the `PatternCompositionDesigner` and `PatternComposition` tools requires the following dependencies:

1. Install the JDK 1.6 or higher
2. Download and install Modelio 2.2.00 open-source

#### Installation guide

See section of this document.

#### Create a new pattern

Creation of new patterns requires the `PatternCompositionDesigner` tool. To create a new pattern from scratch, carry out the operations described below:

**a)** Run the "Create Pattern Designer element / Pattern" command.
**b)** Under the Pattern Designer element, model your pattern by building a standard UML model (model elements, diagrams, …). This model will become the content of the new pattern that you are creating.

#### Define parameters for patterns

Once a pattern has been created, it can be immediately used but it will then produce exactly the same model as the one used to construct the pattern itself. Although this result is already interesting, it is more than likely that you will want more from the application of the pattern. For example, by applying our "Observer" pattern, you will want to be able to replace "Observer. View" by another name, in order to specialise it.

There exist two types of parameters:

1. **String type parameters**: All strings defined in a UML model can be transformed into pattern parameters. For example, the name of a model element, the content of a code note, or an element property can all become String type parameters. To define a parameter, simply replace the string by `$(ParameterName)`. For the parameter name, you can use any alphanumerical combination.

2. **Model Element type parameters**: Model elements can also become pattern parameters. These parameters will have to be instantiated with a model element of the same type when the pattern is applied. To define a model element type parameter, select the element that you want to transform into a pattern parameter and check the "Parameter" tickbox in the "Pattern" tab.

#### Define extension points for patterns

An extension point allows the integration of existing patterns with the model, and the composition of this pattern with other patterns. There are two types of extension point: required extension points, which allow the integration of the pattern to existing models, and provided extension points, which are used in the pattern composition mechanism.
**Required extension point:** To define a model element as a required extension, define this element as a parameter and add the “ExtensionPoint” stereotype.

**Provide extension point:** To define a model element as a provided extension, add the “ExtensionPoint” stereotype to a given element.

- **Package and deploy the pattern**

To configure the properties of a pattern, select the pattern in question and run the "Pattern Designer / Edit the pattern". The following window then then opens.

![Figure 36: Packaging a new pattern](image)

Edit parameters and package the template:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Information &gt; Name</strong></td>
<td>Name of the pattern.</td>
</tr>
<tr>
<td><strong>Information &gt; Version</strong></td>
<td>Version of the pattern.</td>
</tr>
<tr>
<td><strong>Information &gt; Dependencies</strong></td>
<td>List of the pattern’s external dependencies.</td>
</tr>
<tr>
<td><strong>Information &gt; File</strong></td>
<td>Path where the pattern’s &quot;umlt&quot; file is generated.</td>
</tr>
<tr>
<td><strong>Properties &gt; Category</strong></td>
<td>Category to which the pattern belongs.</td>
</tr>
<tr>
<td><strong>Properties &gt; Description</strong></td>
<td>Description of the pattern.</td>
</tr>
<tr>
<td><strong>Properties &gt; Image</strong></td>
<td>Image to illustrate the pattern.</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>Definition of the labels of pattern parameters. See the &quot;Configuring pattern parameters&quot; chapter for more information.</td>
</tr>
</tbody>
</table>

- **Pattern Catalogue**
The pattern catalogue is a set of patterns dedicated to SOA application architecture and PIM4Cloud modelling. The catalogue is the first mechanism you use to apply a pattern.

To open the catalogue, select Repository view in the View menu of Modelio.

![Pattern Repository](image)

**Figure 37 : Pattern Repository**

a) List of patterns organised by category.

b) Description associated with the pattern. If there are unresolved dependencies, they will be listed here.

c) Parameters of the pattern. These must be filled to be able to instantiate the pattern.

d) Triggers the application of the selected pattern.

To deploy a new pattern in the pattern catalogue, you must put it in the following directory of your project: `/mda/PatternComposition/res/patterns/`.

- **Pattern Composition**

The second way to apply a pattern involves the mechanism of pattern composition.

To start pattern composition, you must create a new pattern composition diagram. For this, you can use the command “Pattern Composition diagram” of Pattern Composition tool.

This diagram allows you to create and assemble patterns.
Modelling Pattern Composition:
To assemble two patterns, you must first connect the extension points required and provided by these two patterns.

Figure 38 : Pattern composition diagram

a) Command to create available patterns for the composition.
b) Command to create dependencies between patterns.
c) A pattern created in the diagram.
d) A required extension point. The element type expected (here a package) is filled at the extension point.
e) A provided extension point. The type of element provided (here a package) is filled at the extension point.
Applying Pattern Composition:

Once all “Required Extension Points” have been resolved, you can apply this composition pattern and transform it into complex model. For this result, you can use the “Apply Pattern” command on owner Package of your Pattern Composition diagram.

3.1.4 Web Designer module

3.1.4.1 Tool Approach

The REMICS Migrate Toolkit provides a generic method for building service cloud applications starting from business models. This implies that we have to provide modelling tools and code generation for leading technologies used by the target applications. As no tools for web interface modelling and code generation existed in the commercial version of Modelio, we started with the implementation of the Web Designer tool. The Web Designer tool provides modelling services for web interfaces.

At the architecture layer, the tool is used to represent the front-end and backend of web interfaces, modelling the structure of web pages, the data model, and the behaviour of the controller used by the page as an activity diagram. As part of the REMICS process, the data model and the controller are created by model transformation from the component model.

The architecture model is organised around the notions of page and transitions between pages. Each of these pages includes a view, a model, a controller and a utility component.

The View part of the presentation model is used to model the graphical structure of the page. This structure is represented by typed elements, each representing a component of the web page.

![View part of the Presentation Architecture model](image)

The Model part is used to model the data model associated with the view. The model of the page is represented by a set of typed attributes.

![Model part of the Presentation Architecture model](image)

The Controller part of the presentation model is used to model the behaviour functions called by the view part. These services are represented as operations whose behaviour is in the form of
activity diagrams. Through these diagrams’ behaviour, it is possible to call business services modelled with SoaML.

![Controller Diagram](image)

**Figure 42 : Controller part of the Presentation Architecture model**

The Utility part of the presentation model has a structure similar to that of the controller. It is used to model the operations related to the internal mechanical view.

This tool also supports model transformation functions from the architecture layer to the specific implementation layer. Currently, we support model transformation for the JSF (JAVA–J2EE) framework. Others will be added later according to needs.

At the implementation layer, this tool provides code generation services for web pages. Concerning the Controllers, Models and Utility, it has been transformed during the transition from architecture to the implementation layer in code models, depending on the target implementation framework.

### 3.1.4.2 Getting Started

**Prerequisite**

Using the Web Designer tool requires the following dependencies:

1. Install the JDK 1.6 or higher
2. Download and install Modelio 2.2.00 open-source

**Table 6 WebDesigner Module Dependencies**

<table>
<thead>
<tr>
<th>Module</th>
<th>Type</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaDesigner</td>
<td>Modelio Official Module</td>
<td></td>
</tr>
</tbody>
</table>

**Installation guide**

See section □ of this document.

**Modelling Web Page**
The **Create Page** command creates a new model of a page. This function is available at the architecture layer on the presentation model. The View, Model, Controller and Utility sub models are created under the new page.

- **Modelling Web Page Model**

The **Create Model Property** command is used to create a new property on the data model of the page. This property can be typed by basic types or complex types defined as messages on the architecture behaviour model.

- **Modelling Web Page Controller**

The **Create Controller Operation** command is used to create a new behaviour operation under the Controller. The behaviour of this operation can then be modelled through an activity diagram.

- **Modelling Transition behind Pages**

The Web Designer tools allow the transitions mechanisms between web pages to be modelled through diagrams. The **Create Transition Diagram** command allows the creation of a specialised diagram dedicated to the modelling of these transitions.

- **Generate Implementation model**

For each technology supported by the modelling tools, a command allows the generation of the implementation model from the architecture model. The **Generate JSP Implementation** command is used to product a Java – JSP implementation model from the current presentation model.

- **Generate Web Page**

At the implementation layer, web pages are represented in a specific format. The **Generate Web Page** command is used to generate executable code from this format.

![Figure 43: Web Pages at the implementation layer](image-url)
3.1.5 PIM4Cloud Designer module

3.1.5.1 Approach Overview

We have the task of developing the PIM4Cloud profile, an extension for the OMG’s SoaML profile, in order to address the peculiarities of SaaS application modelling at the platform-independent level. The PIM4Cloud Designer module is the result of this profile implementation under Modelio.

The service oriented architecture modelling language (SoaML) is a specification from the Object Management Group that defines a modelling language for the design of Service-Oriented Architectures. This profile extends UML to support a range of modelling requirements for SOA, including the specification of systems of services, individual service interfaces and service implementations. The PIM4Cloud profile is an extension of the SoaML profile, for modelling specific architectural concepts of applications deployed on the Cloud Computing platform. The set of concepts represented by the PIM4Cloud profile is based on architectural elements previously modelled using the SoaML profile.

The PIM4Cloud profile addresses three issues related to the deployment of an application on the Cloud: modelling the technical architecture of the application, modelling the deployment of this technical architecture on public and private cloud computing platforms, and modelling the physical infrastructure used in conjunction with the cloud computing platforms.

Like the SoaML profile, the PIM4Cloud profile is part of the model-driven development process. It is presented as a PIM (Platform Independent Model) level profile, meaning that models of architecture and deployment remain independent of the technology and service providers used at the time of implementations. The models produced can be specialised later to a PSM (Platform Specific Model) and so be adapted to a specific technical context.

This tool provides a high-level user interface for PIM4Cloud modelling with contextual element creation and graphic diagram modelling to allow easy diagram creation, in order to graphically model the deployment architecture and the deployment project template.

3.1.5.2 Architecture

The PIM4Cloud profile is structured around four packages addressing different aspects of cloud application architecture modelling: the “Cloud Deployment” package, the “Technical Cloud Architecture” package, the “Infrastructure” package and the “Resource” package.
Cloud Deployment package: The Cloud Deployment package covers the modelling of the deployment of cloud applications on public and private cloud computing platforms. It is used to model all of the resources provided by these platforms, such as computational resources and storage resources. With a precise mapping of resources available, it becomes possible to model the deployment of the technical architecture of the application defined by the Technical Cloud Architecture package.

Technical Cloud Architecture package: The Technical Cloud Architecture package covers the modelling of the technical architecture of an application, by focusing on specific aspects of the Cloud. This package allows the modelling of virtual disk images, the unit used to deploy an application on a cloud computing platform at IaaS level. It makes the connection between the application modelled using the SoaML profile and cloud computing platforms by modelling the deployment of SoaML service components on these virtual disk images.

Infrastructure package: The Infrastructure package allows the modelling of the physical infrastructure involved in the process of application deployment, such as servers and networks. The Cloud Technical Architecture package uses these elements to model the technical architecture of the application, including all components of this architecture that are not deployed on the cloud computing platforms.

Cloud Deployment package: The Cloud Deployment package is used when one has to model the infrastructure of private cloud computing platforms.

Resource package: The Resource package covers the modelling of the representation of the "Property / Type" type, designed to allow the modelling of element-specific techniques that cannot be defined otherwise.
3.1.5.3 Getting Started

3.1.5.3.1 Prerequisite
Using the PIM4Cloud tool requires the following dependencies:
1. Install the JDK 1.6 or higher
2. Download and install Modelio 2.2.00 open-source

3.1.5.3.2 Installation guide
See section of this document.

3.1.5.3.3 Create Cloud Deployment Project command

The Create Cloud Project command is used to initialise creation of a new cloud modelling project.

![Figure 45: Cloud Modelling project](image)

3.1.5.3.4 Modelling Cloud Deployment Project
This is used to model the deployment of an application using the PIM4Cloud profile through the use of several diagrams. Each of these diagrams is dedicated to the modelling of a specific part of the deployment.

3.2 CloudScript
The Cloud-computing paradigm emphasizes the need for automated deployment mechanisms, abstracted from the underlying technical layer. As cloud-computing considers that the number of resources available in the cloud is not limited, it triggers new challenges from a deployment point of view. Even if several approaches consider the deployment target as “open” (i.e., new host machines can be added in the environment), the “virtually unlimited” dimension provided by the cloud-approach is not taken into account. Our contribution with this work is to propose a component-based approach to model software deployment in the clouds. This approach is provided as a Domain-Specific Language (DSL), which is given to the software designer. The language is based on a reduced component meat-model, and supports the modelling of the deployment relationship between components.

3.2.1 Motivating example: SensApp
This document uses the SensApp system as running example. SensApp is an application used to efficiently collect data from sensors. SINTEF uses SensApp daily to monitor its premises in Oslo, collecting data from several sensors (e.g., light, temperature, humidity). SensApp is also used to collect sensor data from different vehicles (e.g., bike, plane, car). Then, this data set can be (i) exploited by a data mining expert and/or (ii) consumed by third party applications offering
visualisations mechanisms. The development of SensApp is funded by two projects: ENVISION (EU FP7 – Environmental monitoring) and MODERATES (SINTEF strategic fund).

Figure 46 : SensApp overview

The application is small enough to be easily understood, and large enough to trigger real life situations with respect to cloud deployment. It is based on interconnected REST services hosted by a standard servlet container (e.g., Jetty). These services are implemented using the Spray framework (Scala language). The data backend is delegated to a MongoDB database for efficiency purposes.

SensApp can be deployed according to two different ways:

1. Monolith: this is the simplest deployment, where all the services and database are deployed on the very same server.
2. Distributed: this configuration allows the distribution of services on multiple servers.

In this deliverable, we focus on the Monolith flavour for the sake of simplicity. We follow a top-down approach. The deployment of SensApp is presented from the PaaS level to the IaaS level, digging into the details step by steps.

3.2.2 Deploying at the Software as a Service level

A graphical representation of what we want to model at this step (using “conventional” notations) is depicted in Fig 2.
SensApp is available as a WAR file, on its GitHub page. As a consequence, we model the SensApp system as an element that refines the “WarFileComponent” concept provided by CloudScript. This component has to provide an URL where the WAR file can be retrieved.

```scala
class SensApp extends WarFileComponent {
  val file = new java.net.URL("http://github.com/downloads/SINTEF-9012/sensapp/sensapp.war")
}
```

As the SensApp component mixes the WarFileComponent trait, the system enriches it automatically. Thus, it expects:

1. A SSH Service to support its deployment (attribute "ssh")
2. A String property describing where to deploy the WAR (attribute “hostPath”)

For now, we assume a platform component named "MonolithicHost" to be used as an host for this SensApp system, providing the expected elements. This component is (for now) considered as a black box. As a consequence, from a DSL point of view, modelling the deployment means to create a composite component that instantiates both SensApp and its host, and to bind the associated elements.

```scala
class SensAppSystem extends CompositeComponent {
  // internal components
  val host = instantiates[MonolithicHost]; host hasForUUID "sensapp-host"
  val system = instantiates[SensApp]; system hasForUUID "sensapp-system"
  // Deployment binding
  this deploys system.ssh on host.ssh
  // Property binding
  this sets system.hostPath using host.deploymentPath
}
```

From a technical point of view, implementing such a deployment means to:

1. Initialise the SensApp component:
2. Download the WAR file located at the “file” URL
3. Fill the “hostPath” property with the value of “deploymentPath”.
4. Copy the downloaded file to the deployment path

One can use Java to access to the content of a CloudScript component. For example:

```java
public static void main( String[] args) {
  SensAppSystem application = new SensAppSystem();
  for(Component c: application.containeds()) {
    System.out.println(c);
    System.out.println("  expects: "+ c.expected());
  } System.out.println(" offers: "+ s);
}
```

It produces the following output:
sensapp-host#net.modelbased.cloudscript.samples.sensapp.platform.MonolithicHost

expects: null

offers: Service[net.modelbased.cloudscript.library.Ssh]

sensapp-system#net.modelbased.cloudscript.samples.sensapp.SensApp

expects: Service[net.modelbased.cloudscript.library.Ssh]

Black box components are identified easily: they do not expect anything.

### 3.2.3 Deploying at the Platform as a Service level

We now focus on the modelling of the MonolithicHost component. As described in the Running Example section, SensApp requires the following components to be properly deployed:

1. A MongoDB database;
2. A Java Virtual Machine;
3. A Jetty server to host the WAR file.

These software components have to be deployed on a virtual machine hosted in the cloud, at the Infrastructure as a Service level. Fig 3 graphically describes this assembly.

#### 3.2.3.1 Using APT for standard components

![Figure 48: Platform as a Service layer](image)

The APT system is the classical way of installing software on debian-based linux distribution. Users can basically use "packages" (referred by name) to describe which pieces of software need to be installed. The dependency resolution algorithm provided by APT automatically handles the dependencies between applications. From a component point of view, one can model an APT package as a component that expects such mechanisms on its host. The CloudScript library
provides an “AptExpectation” concept that models this need. This concept extends the 
“SshOffering” one with APT specific extra-information.

More specifically, mixing the “AptExpectation” traits enforces the user to provide the list of apt 
packages that need to be installed.

```java
class MongoDataBase extends AptExpectation {
    val packages = List("mongodb")
}
class JavaVM extends AptExpectation {
    val packages = List("openjdk-7-jre")
}
```

Thus, to deploy a component that extends the “AptExpectation” concept, one must:

1. Connect to the host through the expected SSH service;
2. Execute on the remote machine:
3. `sudo apt-get install <PACKAGES LIST>`
4. It is more safe (at least once on each remote host) to
5. “sudo apt-get update”
6. “sudo apt-get dist-upgrade”

The “toCommand” method provided by the AptExpectation trait builds the associated command. For 
example, in Java:

```java
public static void main(String[] args) {
    MongoDataBase db = new MongoDataBase();
    System.out.println(db.toCommand());
}
```

```bash
sudo apt-get install mongodb
```

### 3.2.3.2 Using remote URL for file-based deployment

SensApp relies on asynchronous request handling (servlet 3.0 specification). This mechanism is not 
available in Jetty version earlier than 8. As this version is not provided through APT, we must use 
the “good-old” method for software deployment: do it manually.

The CloudScript library defines the “FileDownload” concept for this purpose. This trait enforces a 
SSH expectation, and requires the user to provide the following information:

1. A “sourceFile” to download;
2. A “targetPath” exposing where the installed software will be available;
3. A list of command to execute to implement the deployment

A piece of software installed manually might not start automatically. Thus, the CloudScript library 
provides the “StartupCommand” to describe the shell commands needed to start it.

As a consequence, one can model the Jetty8 server as the following:

```java
class JettyServer extends FileDownload with StartupCommand {
    val sourceFile = new java.net.URL("http://download.eclipse.org/jetty/stable-8/dist/jetty-distribution-8.1.5.v20120716.tar.gz")
    val targetPath = hasForProperty[String]; targetPath.data = "/opt/jetty8/webapps"
    val file_deploy_commands = List("tar zxvf /tmp/jetty-distribution-8.1.5.v20120716.tar.gz",
        "sudo mv /tmp/jetty-distribution-8.1.5.v20120716 /opt/jetty-distribution-8.1.5.v20120716",
        "sudo ln -s /opt/jetty-distribution-8.1.5.v20120716 /opt/jetty8")
    val startup_commands = List("cd /opt/jetty8/bin; ".jetty.sh start")
}
```

One can use Java to access to these information:

```java
public static void main(String[] args) {
    JettyServer srv = new JettyServer();
}
```
System.out.println("Deployment Script: \n" + srv.deployAsBash());
System.out.println("Startup Script: \n" + srv.startupAsBash());
}

Deployment Script:

#!/bin/bash
tar zxvf /tmp/jetty-distribution-8.1.5.v20120716.tar.gz

sudo mv /tmp/jetty-distribution-8.1.5.v20120716 /opt/jetty-distribution-8.1.5.v20120716

sudo ln -s /opt/jetty-distribution-8.1.5.v20120716 /opt/jetty8

Startup Script:

#!/bin/bash
cd /opt/jetty8/bin
./jetty.sh start

Deploying such a component means to:

- Connect to the remote host through SSH;
- Download the source file into /tmp;
- Execute the deployment script;
- Execute the start-up script.

These components need to be deployed on a Linux-based virtual machine. We assume an EC2VirtualMachine provided by the IaaS layer, that expects information such as cloud region or Amazon machine image (AMI) to be used on the virtual machine. Our SensApp instance will be hosted in Ireland, and use an UbuntuLTS (12.04) operating system. It also has to accept connection on the SSH port, as well as incoming request on the 8080 port (aka HTTP alternative, used by Jetty to host the WAR applications).

As the system is Linux-based, we can mix the “SshOffering” trait to model that it supports SSH connection. The Ubuntu LTS AMI provided in the Amazon Cloud suffers from a bad declaration of locale, preventing the operating system to properly start MongoDB. Thus, we can use the “InitCommands” and the “BashSetup” trait to model the needed commands to fix it:

1. BashSetup defines the “bashrc” list of commands, to be added into the ~/.bashrc file of the user
2. InitCommand defines the “init_setup” list of command to be executed only once to initialize the virtual machine.

As the CloudScript DSL is hosted in an object-oriented language, one can use inheritance to create hierarchy of components. In the following code, the “VirtualMachine” class is abstract, and factors the commonality of the virtual machines to be used to host SensApp instances. Then, we define two concrete components to be used: one relying on a “micro” virtual machine, and one relying on a “small” virtual machine.

```java
protected abstract class VirtualMachine
    extends amazon.EC2VirtualMachine with SshOffering with InitCommands with BashSetup {
        val ami = amazon.AMIs.UbuntuLTS
        val region = amazon.Regions.Ireland
        val allowed = List(amazon.Rules.SSH, amazon.Rules.HTTP_ALT)
        val bashrc = List("export LANGUAGE=en_US.UTF-8", "export LANG=en_US.UTF-8", "export LC_ALL=en_US.UTF-8")
```
val init_setup = List("sudo locale-gen en_US.UTF-8")

class MicroVM extends VirtualMachine {
  val name = "sensapp_host_micro"; val instType = amazon.InstanceTypes.Micro
}

class SmallVM extends VirtualMachine {
  val name = "sensapp_host_small"; val instType = amazon.InstanceTypes.Small
}

Based on these components, one can model the “MonolithicHost” previously used as the following:

class MonolithicHost extends CompositeComponent {
  // Internal components
  private[this] val vm = instantiates[SmallVM]
  private[this] val db = instantiates[MongoDataBase]
  private[this] val jvm = instantiates[JavaVM]
  private[this] val server = instantiates[JettyServer]
  // Properties
  val deploymentPath = externalize(server.targetPath)
  // Deployment binding
  this deploys db.ssh on vm.ssh
  this deploys jvm.ssh on vm.ssh
  this deploys server.ssh on vm.ssh
  // port promotion
  val ssh = promotes(vm.ssh)
}

3.2.4 Infrastructure as a Service

We focus here on the modelling of the Amazon EC2 infrastructure.

abstract class EC2VirtualMachine extends ScalarComponent {
  val name: String
  val ami: AMI
  val region: Region
  val instType: InstanceType
  val allowed: List[Rule]
}

The key point of the DSL is to support the definition of syntactic sugar that helps the final user. For example, this is how the regions are modelled:

case class Region(name: String, endpoint: String);
object Regions {
  val Ireland = Region("eu-west-1","ec2.eu-west-1.amazonaws.com")
  val SaoPolo = Region("sa-east-1","ec2.sa-east-1.amazonaws.com")
  val Virginia = Region("us-east-1","ec2.us-east-1.amazonaws.com")
  val Tokyo = Region("ap-northeast-1","ec2.ap-northeast-1.amazonaws.com")
  val Oregon = Region("us-west-2","ec2.us-west-2.amazonaws.com")
  val California = Region("us-west-1","ec2.us-west-1.amazonaws.com")
  val Singapore = Region("ap-southeast-1","ec2.ap-southeast-1.amazonaws.com")
}

The complete source code is available on Github: https://github.com/SINTEF-9012/cloudscript/blob/master/net/modelbased.cloudscript.samples/src/main/scala/net/modelbased/cloudscript/samples/sensapp/infrastructure/Infrastructure.scala
To start an instance, one has to use Amazon credentials (e.g., server key). This point needs more practice to identify clearly what must be modelled in the DSL, and what should be part of the deployment engine. This is still an ongoing work.

3.2.5 Off-the-shelf Components (CloudScript StdLib)

The standard library provided in CloudScript aims to factorize commonly used components. The point is to provide re-usable components, immediately available for the language user. All the components are located in the "net.modelbased.cloudscript.library" package.

3.2.5.1 Deployment Service

We identified the following services as commonly used to implement deployment in the clouds:

- SSH[1]: This protocol allows one to connect (securely) to a remote machine. This protocol is the most common one in the GNU/Linux ecosystem, and is offered by default on several distributions used in cloud setups.
- APT[2]: This software supports the installation of pre-defined software packages and their dependencies in the Debian GNU/Linux ecosystem (including Ubuntu, largely used in cloud setups).
- WAR[3]: WAR stands for "Web application ARchive". This file format is common in the Java ecosystem to package ready-to-deploy web application (e.g., Java server pages, servlets). Common java-based application servers (e.g., Tomcat, Jonas, Jetty) support the hot-deployment of such archive (through a dedicated protocol, or simply with file-system observation).
- FTP[4]: The FTP protocol is common to transfer files, and thus provided as part of the standard library. However, as it is "unsecured" by nature, users usually prefer to transfer files using SSH remote file transfer mechanisms (also known as SCP).

3.2.5.2 Off-the-shelf components fragment

We identified several mechanisms commonly used while modelling a cloud application. They are modelled using CloudScript, and offered in the standard library, so one can reuse it in his/her very own application. From a technical point of view, these mechanisms are implemented as "traits", to be mixed with users' components to enrich their descriptions. These components demonstrate how one can provides elementary bricks on top of a given protocol, using the CloudScript DSL coupled to standard Scala inheritance and mixins capabilities. Here, we built all the following components on top of the SSH service.

- **SshOffering**: This trait adds an offered deployment service named "ssh", exposing the SSH protocol.
- **SshExpectation**: This traits fills the "expected" deployment service of the component with an SSH service named “ssh”.

**APTExpectation (implies SshExpectation)**: A component that mixes this trait defines a list of APT package names. These package are basically the necessary pieces of software to be used to properly install this component. It also automatically loads the SshExpectation trait, enforcing a SSH connection to deploy it.

**FileDownload (implies SshExpectation)**: A component that mixes this trait relies on a remote file that contains the piece of software to be installed. Thus, user must define the URL to download this source file, as well as the shell commands to be used on the remote Linux machine (as it expects a SSH connection to connect to the host) to properly implement the deployment /typically unzipping
and moving to a well-known directory on the file system). It also exposes a String property named “targetPath” that must be filled by the user to publicely expose where the downloaded piece of software was deployed (avoiding collisions in an assembly).

**WarFileBasedDeployment (implies SshOffering):** This trait can be used to identify that a specific component can host WAR-based application, using file system hot deployment mechanisms. It exposes a String property named “deployPath”. The user must fill it with the file path to be used to drop new web applications (e.g., the “webapps” directory of a Jetty server).

**WarFileComponent (implies SshExpectation):** A component that mixes this trait exposes itself as a WAR file, captured in its “file” attribute. It expects a SSH port for the remote copy, as well as a property named “hostPath” that contains the file path to be used as deployment target. Typically, this property is filled by the “targetPath” property offered by a host component when it mixes the WarFileBasedDeployment trait.

**StartupCommand (implies SshOffering):** Some software needs to be automatically started when their host starts (typically a web server). This trait asks the user to provide a list of “startup_commands” to be used when the host start to properly start it. On a Linux based machine, these commands should be added in the runtime startup scripts (e.g., “/etc/rc.d” common mechanism). The trait provides a “startupAsBash” method to automatically generate a shell script that starts the software.

**InitCommands (implies SshOffering):** Some components need to execute commands only once, right after their installation (typically, creating a user in the database, fixing the environment on a virtual machine). These commands are captured in the “init_setup” list enforced by this trait. The “initAsBash” method automatically generates a shell script based on the provided commands.

**BashSetup (implies SshOffering):** Linux-based virtual machines that run the Bash shell might require some configuration to be enforced for each shell. This is typically done through a “~/.bashrc” file that contains several commands to be executed each time the user logs in. This trait requires the definition of a “bashrc” list of commands to be concatenated at the end of the bashrc file of the current user.
Here is an example in Scala to show how one can explore a component assembly.

```scala
def descrComponent(c: Component) {
  c.offereds foreach { s => println("offers "+s) }

  println("expects "+
    (if (c.expected == null) "nothing" else c.expected))

  c match {
    case _ =>
      case c: CompositeComponent => {
        println("contains "+c.containeds.map {_.toString}.mkString("","",""))

        println("connects:")
        println("from:"+c.from.offeredBy + "+"+c.from + ""]")
        println("to:"+c.to.offeredBy + "+"+c.to + "]")

        descrComponent(c)
      }

    case _ =>
  }
}
```
3.3 Desktop To Cloud Migration Tool (D2CM)

D2CM is a standalone tool that provides all necessary means from migrating a scientific application to cloud to defining which cloud resources are allocated, how the application is deployed on these resources, and to specifying the life-cycle of the deployed application. Following sections describe these means in detail.

3.3.1 Tool approach

To perform scientific experiments in the cloud, scientists would need significant knowledge of computer science, cloud computing and migration of applications to the cloud, which is difficult for non-computer scientists. For example, scientists who are running their experiments currently on grids and clusters are unaware of how the environment is configured, how the queues work on clusters, and how his job gets executed. All they are interested in is that they submit a job to some queue and after sometime they can collect the results.

Having this assumption in mind we have designed a tool that greatly simplifies migrating their applications to the cloud. The idea is to migrate the complete desktop environment, with which the scientist is working daily, directly to the cloud and freeing the user from re-creation of a software environment on the vendor-supplied (maybe supported) OS and cloud instances. The D2CM tool seeks to mitigate potential problems of this often laborious task by taking the entire environment as-is, provided it is within a virtual environment. The D2CM tool is developed in Python and provides two main facilities:

- Programmatical transformation of user-supplied virtual machine images to the target cloud image format. Currently it has support for the Amazon Machine Image format and Eucalyptus compatible XEN image.
- Defining the deployment configuration and life-cycle management of distributed applications, hosted on Amazon's Elastic Cloud Computing (EC2) platform (or compatible infrastructure, such as Eucalyptus).

Therefore, the requirement of the desktop-to-cloud process is that there is at least one existing virtual machine image that is able to run non-interactive experiments. At the moment, the tool is targeted at computationally intense scientific use cases and is not optimised for management of large datasets. The virtual machine images must have an SSH server installed in them and configured to allow root login using a private SSH key. The tool also has support for extracting the results and monitoring the health of the cluster while the application is running on the cloud.

3.3.2 Migration process

This subsection outlines what steps are needed to migrate an existing virtual machine image to cloud using the D2CM tool. D2CM currently supports only VirtualBox images. The tool expects a VirtualBox Disk Image (VDI) file of a stopped virtual machine where all the software components have been set up by running VirtualBox, installing Ubuntu and all the software components using the usual approach, whether this is the usual virtual machine the scientist is working or a newly created one. Once the image file has been created the user can proceed to migrating it using D2CM tool.

3.3.2.1 Specifying credentials

As the first step, we supplied the D2CM tool with cloud credentials for both Amazon EC2 and our local private cloud SciCloud. The required credential types are same for both EC2 and Eucalyptus.
based cloud as they use the same command line interface API for accessing the cloud services. (See Figure 50) The required credentials are:

- **Name** – Arbitrary name for the credentials, only used by the D2CM tool.
- **User ID** – Cloud user ID.
- **Access Key** – Textual key string provided by cloud.
- **Secret Key** – Textual secret key string provided by cloud.
- **User certificate** – Location of the user certificate file in the local file system.
- **Private Key** – Location of the private key file in the local system.

![Figure 50: Adding cloud access credentials](image)

### 3.3.2.2 Registering local Image

The user then registers a desktop image with the tool choosing a name, an image file located in the local machine and the default cloud the image should be associated with. The association can be changed later and a single image can be migrated to a more than one cloud. (See Figure 51)
3.3.2.3 Initiating Migration

After the image has been added to the system, user can select to initiate the migration process, by choosing the target cloud, its availability region, existing cloud kernel and ramdisk and a bucket name in the cloud file system in which the virtual machine image will be stored. (See Figure 52 to Figure 54)
Figure 53: Choosing a kernel

Figure 54: Choosing destination bucket name
3.3.2.4 Image Transformation

The tool then transforms the provided VirtualBox image to a compatible cloud virtual image, making any changes to a separate image file, to preserve the integrity of the source image. (See Figure 55)

Both EC2's and Eucalyptus' underlying machine virtualization is XEN (Note: Eucalyptus also supports KVM), a widely used open-source technology which supports full and para-virtualization. While Amazon provides a range of already configured images, it also supports user supplied images, provided they function with a kernel using a Xen compatible kernel. The kernel must have pv-ops with XSAVE hypercall disabled[2]. An image may either be configured to boot with Amazon supplied kernels, known as AKIs, or with a kernel packaged within the image itself. The latter option is used by D2CM, which uses Xen's PVGRUB boot process. In the kernel update section of the transformation process, the D2CM tool installs a new kernel, either 32 or 64 bit depending on the detected image type, and modifies GRUB configuration to boot with the new kernel by default. In the current implementation, the tool only supports transformation of a single disk partition. If an image has more than one partition, only the primary one can be migrated to EC2.

![Figure 55: Image transformation progress](image)

D2CM uses the Libguestfs library to inspect and manipulate virtual machine images. Using libguestfs is highly preferred to mounting the images within the user's file system as directly mounting the image requires the host OS to support the image's file system(s) and requires root privileges. Libguestfs' API includes functions to extract partitions from images and inserts SSH key files, which are required during the image migration process. Libvirt[4] is an abstraction library for interacting with desktop hypervisors. The D2CM tool uses libvirt to access the currently supported hypervisor, VirtualBox, which is an open source hypervisor that runs on Windows, Linux, Macintosh, and Solaris hosts and supports a large number of guest operating systems. In the future, adding support for other popular hypervisors such as KVM, Xen, and VMWare should be greatly eased by the use of libvirt. The currently supported IaaS platforms, EC2 and Eucalyptus, provide the same Amazon Web Services (AWS) API for managing machine instances. The web service is wrapped by
several libraries which provide programming language specific bindings. As we developed the D2CM tool using Python, we used the Boto library[5] which is the Python binding for AWS.

In addition to working with a compliant kernel, an image must also be able to mount devices in the manner supplied by the EC2 virtual machine. EC2 instances have a default device mapping that usually does not match the fstab entries within the user-supplied image. EC2 instances come in a variety of types that may have more or less storage devices than the image is configured for. Some instances do not provide swap devices. Most instances provide additional storage devices that the user may want to utilize. To conform to this new virtual hardware environment, D2CM changes the fstab appropriately.

3.3.2.5 Image Upload and registration

Upon the completion of the transformation of the image files, the tool uploads the images to Amazon's Simple Storage Service (S3) or to Walrus in case of Eucalyptus. Once cloud image has been successfully uploaded, the tool registers the image with clouds services, defining the default settings such as suitable system architecture, kernels, default instance type, etc. After the registration is completed, the images are ready to run on the specified cloud.

3.3.2.6 Summary

This process must be repeated for each Desktop image required for the distributed application. The duration of the entire transformation and upload procedure varies greatly depending on the image size and user's Internet connection speed. In our case, the size of the images was around 2GB.

3.3.2.7 Defining deployment template

Next step after migrating virtual machines is defining a deployment template that describes how the virtual machines are divided into roles and how these roles are executed in the cloud. A deployment consists of one or more roles. A role is defined by a deployment unique name, association to a cloud image, instance count and type, and life-cycle scripting. There may be a many-to-one relationship between roles and a unique cloud image. The instance count defines the number of virtual machines of specified type that will be allocated upon deployment start-up. Roles include optional life-cycle scripts that the deployment manager invokes at the appropriate stage.

When a user wants to run a scientific computing experiment, he has to define the following deployment configuration in the D2CM:

- Deployment name and the size of the experiment to be conducted. Size is used for comparing the computational complexity of separate deployments.
- Deployment roles for the virtual machines that will be deployed. For each role:
  - The name of the migrated virtual machine image that this role uses
  - The cloud instance type, defining how much computer resources role instances get.
  - Number of instances.
  - Input files that should be uploaded to the role instances.
  - Start-up command(s) executed then the virtual machine boots,
  - Initiation commands that specify what command starts the execution of the experiment inside the virtual machines.
  - End condition for checking when the experiment has been finished.
  - Download actions that specify which files are downloaded once the experiment is done.
  - What cloud the deployment will run on.
### 3.3.3 Deployment process

Once the deployment template has been defined, the user can create a deployment instance from the model and initiate the deployment execution by pressing the button "Deploy" as illustrate on Figure 56. User has also an option to delete an existing deployment or to create a clone of it. Cloning a deployment creates a new deployment with the same configuration, which the user can then modify or deploy, allowing the user to speed up updating deployments and still keeping all the data about previously run ones. This is especially useful for scientific experiments for which it is common to keep changing the parameters of experiments before repeating them.

![Diagram showing the deployment execution view](image)

**Figure 56 : Deployment execution view**
Figure 57: Deployment template for the experiment for running GPAW experiments.

Once user initiates the deployment, the tool prompts the user with a calculated hourly price for the chosen configuration and asks if the user is sure to continue. (see Figure 58)
Figure 58. Executing a deployment and confirming the price

When user presses “Yes” button, the tool continues into deployment lifecycle process and starts displaying the state of the deployment in “Log /Events” tab. (see ¡Error! No se encuentra el origen de la referencia.)

Figure 59. Deployment progress view
The life-cycle of the deployment proceeds through the following phases in the D2CM tool:

**Reservation** – Upon initiation, the tool iterates through deployment roles and reserves instances matching user specifications. If the request cannot be completely fulfilled by EC2, any successful allocation is released and the process ends. The moment instances are reserved, AWS begins charging the user's account. Prior to actual reservation, the tool prompts the user for confirmation, displaying the per hour rate in USD as determined by the deployment's instance type and count.

**Contextualization** – The tool enables all deployed cloud instances to discover each other’s IP’s and to be able to connect to each other. When all instances within a deployment are assigned IP addresses, the tool installs a text file containing all addresses on each host in the deployment. This is often required for parallel processing or any other kind of co-operative processing.

**Upload** – Tool executes the upload actions specified in the deployment description, allowing the user to install files on the instances which were not bundled with the image. These files may either correct minor image defects or supply experiment specific input.

**Start-up** – Tool executed start-actions specified in the deployment description. It executes all given commands for each instance in each role in serial manner. This functionality allows users to make modifications to deployment hosts that are guaranteed to finish before starting any main scripts, which are run using the asynchronous actions.

**Execution** - Once all the cloud instances have been stared up, the tool executes user defined asynchronous-actions defined in the deployment template that initiates the migrated system main process. In our case, these are the scientific experiments. The action is asynchronous in that it logs on to the remote instance, executes the user's script in a daemonising (with nohup) wrapper, and then immediately ends the remote connection.

**Completion monitoring** – The tool periodically checks the ending condition of the deployment to find out when the deployment should be terminated. The end condition can be an entry in a log file or existence of a specific file in the machine. The only option provided currently, is the ability to monitor for the presence of a file on an instance. The tool continually polls the instances of roles with the specified check. If no check is provided for a given role, that role is automatically assumed to be in the finished state. Only when all roles are in the finished state, the program proceeds to finalisation.

**Finalisation** – The tool executes user-defined download-actions to store the results of the deployment outside the cloud instances as any data on the instances is lost when the virtual machines are terminated.

**Termination** - Upon the completion of finalisation the deployment is shut down and all cloud instances associated with the deployment are terminated.

Once the tool has terminated the deployment, it prints out a path to a local deployment folder, from where all the downloaded results can be accessed by the user as seen from Figure 60.
3.3.4 Performance monitoring

D2CM tool also enables monitoring the condition of deployed virtual machines to see that they are working properly and to be able to estimate whether the deployment configuration, number of instances per each role and the instance capabilities were chosen correctly.

CollectD is a UNIX daemon that collects transfers and stores computer performance data in a Round-Robin Database (RRD). RRD is a fixed size circular data structure for time-series data. When the data structure gets full, the old data is overwritten in a circular manner and the entry-point of the structure is updated. This provides a convenient way to store and transfer computer performance parameters as the size of the database stays constant. For displaying the data in a graphical manner, we use the rrdTool software, which generates graphs from rrd files. CollectD must be installed on the target virtual machine, which the D2CM takes care of itself after the virtual machines boot.

The data that the D2CM tool collects and displays is CPU (Figure 61) and Memory (Figure 62) usage and machine load (Figure 63). Users can use this information to estimate how efficiently was the deployment configure chosen. For example, from Figure 61, it is clear that most of the memory is never used in full capacity and thus it would have more efficient to choose cloud instances with less memory instead. This shows that D2CM tool does not only help in migrating and deploying applications to the cloud but also helps in optimising the experiments, clusters and thus the costs for conducting scientific computing experiments on the cloud.
Figure 61: CPU usage

Figure 62: Memory usage
3.3.5 Getting started

The D2CM is a standalone tool and provides all necessary means from migrating to deploying an application to cloud, provided that the application and all its prerequisite software is installed in a VirtualBox virtual machine. The idea is to migrate the complete virtual machine directly to the cloud and free the user from re-creating the software environment on the cloud environment.

The tool takes the user provided virtual machine and transforms it to the target cloud image format. Currently, it has support for the Amazon Machine Image format and Eucalyptus compatible XEN image. Once the image is migrated, the user can define the deployment configuration and life-cycle of the application and execute it on Amazon's Elastic Cloud Computing (EC2) platform (or compatible infrastructure, such as Eucalyptus).

The tool can currently only be run on a Linux machine and we have tested it on multiple Ubuntu versions (from 9.04 to 12.04). Transforming a Linux virtual image to a desired cloud virtual machine format requires multiple libraries for modifying the internals of the image. Additional libraries are needed to communicate with the target cloud services for uploading and executing the migrated images. This section provides a guide on how to install the perquisite software and the tool itself on an Ubuntu (from version 9.04 to 12.04) operating system.

3.3.5.1 Installing prerequisite software

Required Debian/Ubuntu packages, which are installable from Aptitude are:

- python-setuptools (>=0.6.15)
- python-boto (>=1.9b)
- python-wxgtk2.8 (>= 2.8.10.1)
- python-sqlite (>= 1.0.1)
- ec2-ami-tools (>= 1.3.49953)
- euca2ools (>= 1.3.1)
- virtualbox-ose (>=4.0.4-dfsg)
- python-sqlalchemy
- python-rrdtool
- python-numpy
- python-scipy
- python-matplotlib
- libguestfs0 (>= 1.11.8) (http://libguestfs.org/download/binaries/ if not available from aptitude)
- python-guestfs (>=1.11.8) (http://libguestfs.org/download/binaries/ if not available from aptitude)
- libxml2-dev
- libgnutls-dev
- libdevmapper-dev
- python-dev
- libnl-dev

Libraries, which are not available from Aptitude are:

mockito-python. (http://code.google.com/p/mockito-python/) Can be installed using the following command:

easy_install mockito

libvirt. Since libvirt in Ubuntu repository does not support VirtualBox drivers, libvirt should be downloaded from ftp://libvirt.org/libvirt/ and compiled using the following commands:

- mkdir libvirt
- cd libvirt
- apt-get source -d libvirt
- sudo apt-get build-dep libvirt
- dpkg-source -x libvirt*dsc
- cd libvirt-0.8.8/debian
- gedit rules
- Change "--without-vbox" to "--with-vbox"
- gedit changelog
- Change "0.8.8-1ubuntu6.5" to "0.8.8-1ubuntu6.5.local"
- cd ..
- dpkg-buildpackage -us -uc -b -rfakeroot
- cd ..
- sudo dpkg -i *.deb

3.3.5.2 Compiling and starting the D2CM tool

In the project root directory run the following commands to compile the D2CM tool:

    make builddeb

This will create ../d2c_<version>_all.deb. To install the tool in the system, run:

    dpkg -i ../d2c_<version>_all.deb

Note that installing with dpkg explicitly will not resolve dependencies. If there are any broken dependencies, they have to be resolved using aptitude.

The tool can then be started with a command line command: d2c_gui
4 Annexes

4.1 Tools Reference Card

<table>
<thead>
<tr>
<th>Name</th>
<th>Migration Process tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal overview</td>
<td>The MigrationProcess module is the orchestrator of migration process. It offers a support for the approach of migrating legacy applications to SOA application deployed on cloud computing platforms. This module provides: Profile dedicated to the structuring migration project Model transformations functionality between different phases of the migration process. Service of code generation from behaviours expressed as activity diagrams. Integration of model analysis tools, transformations tools and code generation tools involved in migration process.</td>
</tr>
<tr>
<td>Type</td>
<td>Modelio module</td>
</tr>
<tr>
<td>Last Version</td>
<td>1.0.00</td>
</tr>
<tr>
<td>Archive</td>
<td>MigrationProcess.mdac</td>
</tr>
<tr>
<td>Link</td>
<td><a href="https://project.sintef.no/eRoomReq/Files/informatics/ReMiCS/0_97aa5/MigrationProcess-1.0.00.jmdac">https://project.sintef.no/eRoomReq/Files/informatics/ReMiCS/0_97aa5/MigrationProcess-1.0.00.jmdac</a></td>
</tr>
<tr>
<td>Dependencies</td>
<td>ComponentRecovery, PersistentProfile, JPAGenerator, WebDesigner, SoaMLDesigner</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Component Recovery tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal overview</td>
<td>The Component Recovery tool covers methods for components identification, discovery of their services, interfaces and dependencies. This module provides: Tools for analysing dependency between elements of the application architecture. Representation of dependencies in the form of dependency diagrams or matrices dedicated to human analysis. Service achieving breakdown of architecture into coarse-grain components providing business services</td>
</tr>
<tr>
<td>Dependencies</td>
<td>ComponentRecovery, PersistentProfile, JPAGenerator, WebDesigner, SoaMLDesigner</td>
</tr>
<tr>
<td>Type</td>
<td>Modelio module</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Last Version</td>
<td>1.0.00</td>
</tr>
<tr>
<td>Archive</td>
<td>Component Recovery.mdac</td>
</tr>
<tr>
<td>Link</td>
<td><a href="https://project.sintef.no/eRoomReq/Files/informatics/ReMiCS/0_97aa3/ComponentRecovery_1.1.03.jmdac">https://project.sintef.no/eRoomReq/Files/informatics/ReMiCS/0_97aa3/ComponentRecovery_1.1.03.jmdac</a></td>
</tr>
<tr>
<td>Dependencies</td>
<td>SoaMLDesigner</td>
</tr>
</tbody>
</table>

**Name:** PatternComposition tool and PatternCompositionDesigner

**Goal overview:**
The Pattern Composition tool is a generic solution to involve application of SOA and Cloud Computing design patterns to source architecture. It allows implementing the iterative refactoring of legacy architecture to the targeted Service Clouds architecture expressed with PIM4Cloud SoaML metamodel.

The PatternCompositionDesigner is an extension of PatternDesigner module that allows to defining and packaging new composed patterns.

This module provides:
- Option to define a new pattern from a model part
- Application of existing pattern to legacy architecture model.
- Creating a complex pattern by composing simple patterns

<table>
<thead>
<tr>
<th>Type</th>
<th>Modelio module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last Version</td>
<td>1.0.00</td>
</tr>
<tr>
<td>Archive</td>
<td>PatternComposition.mdac, PatternCompositionDesigner.mdac</td>
</tr>
<tr>
<td>Link</td>
<td><a href="https://project.sintef.no/eRoomReq/Files/informatics/ReMiCS/0_97aab/PatternComposition_1.0.00.jmdac">https://project.sintef.no/eRoomReq/Files/informatics/ReMiCS/0_97aab/PatternComposition_1.0.00.jmdac</a></td>
</tr>
<tr>
<td>Dependencies</td>
<td>No dependency but some others module are required to apply specific patterns</td>
</tr>
</tbody>
</table>

**Name:** WebDesigner tool

**Goal overview:**
The Web Designer module provides modelling services for web interface.
- Modelling front-end and backend of web interface
- Modelling backend behaviour as Activity Diagram
- Support dedicated to the integration of code generation tools from the web
### Implementation of a JSF (JAVA –J2EE) code generation tool

**Type :** Modelio module  
**Last Version :** 1.0.00  
**Archive :** WebDesigner.mdac  
**Link :** [https://project.sintef.no/eRoomReq/Files/informatics/Remics/0_97aa6/WebDesigner-1.1.00.jmdac](https://project.sintef.no/eRoomReq/Files/informatics/Remics/0_97aa6/WebDesigner-1.1.00.jmdac)  
**Dependencies :** JavaDesigner

### PIM4Cloud

**Name :** PIM4Cloud  
**Goal overview :** The PIM4Cloud module provides implementation of PIM4Cloud profile, an extension for OMG SoaML in order to address peculiarities of SaaS applications modelling on the platform independent level.  
This module provides:  
- Modelling deployment of application to cloud computing platform.  
- Integration of architectures components into deployment model.  
**Type :** Modelio module  
**Last Version :** 1.0.00  
**Archive :** PIM4Cloud.mdac  
**Link :** [https://project.sintef.no/eRoomReq/Files/informatics/Remics/0_97aa4/PIM4Cloud_1.1.01.jmdac](https://project.sintef.no/eRoomReq/Files/informatics/Remics/0_97aa4/PIM4Cloud_1.1.01.jmdac)  
**Dependencies :** NA

### CloudScript

**Name :** CloudScript  
**Goal overview :** The CloudScript Domain-Specific language supports the modelling of the cloud applications to be deployed using a component-based approach. It allows us to model using high level concepts the different software components to be deployed (e.g., a web server, a virtual machine), as well as their installation dependencies (e.g., this web server will be deployed on this virtual machine).  
**Type :** Scala internal DSL  
**Last Version :** 0.0.1  
**Archive :** cloudscript-0.0.1.zip  
**Link :** [https://github.com/SINTEF-9012/cloudscript](https://github.com/SINTEF-9012/cloudscript)
Dependencies : Handled by maven (basically scala library)

<table>
<thead>
<tr>
<th>Name</th>
<th>Desktop To Cloud Migration (D2CM) tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal overview</td>
<td>Support the migration of scientific applications to the cloud and provide a proof of concept for complete migration and deployment of applications to both private and public clouds.</td>
</tr>
<tr>
<td>Type</td>
<td>A standalone tool</td>
</tr>
<tr>
<td>Last Version</td>
<td></td>
</tr>
<tr>
<td>Archive</td>
<td>D2CM.zip</td>
</tr>
<tr>
<td>Link</td>
<td></td>
</tr>
<tr>
<td>Dependencies</td>
<td>VirtualBox</td>
</tr>
</tbody>
</table>

4.2 List of abbreviations


4.3 List of references

2. [https://www.virtualbox.org/](https://www.virtualbox.org/)
5. [https://github.com/boto/boto](https://github.com/boto/boto)