Integrating Components into SORASCS

Bradley Schmerl, Michael W. Bigrigg,
David Garlan, and Kathleen M. Carley

August, 2010
CMU-ISR-10-122

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Center for the Computational Analysis of Social and Organizational Systems
CASOS technical report.

This work was supported in part by the Office of Naval Research - ONR -N000140811223. Additional support was provided by the center for Computational Analysis of Social and Organizational Systems (CASOS). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Office of Naval Research, or the U.S. government.

Keywords: SORASCS, Services, Integration, Thick Services, Thin Services

Abstract

This document describes how components may be integrated into the SORASCS framework as web services. Two modes of integration are described: 1) Integration of web services as thin services that can be deployed and run on a SORASCS server; and 2) Integration of thick tools as thick services that run on user machines but can participate in SORASCS workflows.

Table of Contents

1 Introduction 1

2 SORASCS Architecture in a Nutshell 1

2.1 SORASCS Services 3
2.2 Provenance 4
2.3 Data 4
2.4 Registry 5

3 Integrating Thin Services 5

3.1 SORASCS Service API 5
3.1.1 Synchronous Interaction 5
3.1.2 Asycnhronous Interaction 6
3.1.3 Operation Parameterization 7
3.1.4 Categorization 9
3.2 Tool Integration 11
3.2.1 Java Web Service Interface 11
3.2.2 Providing the Operation 12
3.2.3 Parameterization 15
3.2.4 Bundling and Deployment 15

4 Integrating Thick Services 16

4.1 Kinds of Thick Integration 17
4.2 User Installation: SORASCS Client-side Server 17
4.3 Developer Integration 18
4.4 SORASCS User Registration 19

5 References 21

6 Appendix 21

6.1 Parameter Schema Definition 21

1 Introduction

Over the past decade there have emerged a large variety of tools supporting analysis of heterogeneous information to understand complex real world relationships and trends, including natural language processing, network analysis, simulation, and what-if reasoning. Most analysis ensembles assume a relatively limited set of model types and input sources. They are wired together with special purpose, tool-specific and ad hoc integration code, making them brittle and requiring low-level system expertise to reconfigure in new ways or add new capabilities. The underlying problems stem from the lack of a coherent and flexible architectural framework that will allow tools and models to be seamlessly incorporated and composed by end users. Having such a framework would ideally allow information analysts to bring together tools from various vendors and research groups, dynamically compose their analyses in new ways, store and reuse workflows and settings, ingest new forms of information, and interactively fine tune analyses, simulations, and report generation through consistent and uniform forms of interaction.

Work on SORASCS is developing an architecture for the intelligence community that will enable these new capabilities. This architecture is based on service oriented architectures (SOA), which are typically used in the business community for integration and use of different business systems and processes developed by different companies at different times. The key aspects of SOA that enable this are:

- Modularity of components as services, providing simple operations that are loosely coupled. This allows applications to be easily composed out of existing services.
- Standards for communicating among distributed services over the web.
- Standards based support for discovery, orchestration, governance, security, etc.

The goal of SOA is to support the development of applications using web services, and therefore many of the features provided are general in nature, so that they can be used in many domains. SORASCS is layered upon these general features to provide features customized to the domain of intelligence analysis.

The purpose of this document is to serve as a guide to tool integrators to describe how to integrate tools into SORASCS. The following section provides a brief summary of the rationale behind SORASCS and some of the foundational services that are provided by SORASCS. Following that, instructions are given for tool integrators to integrate their components and algorithms as thin services. Integration of existing tools as thick services is then described. Finally, we describe how to build upon SORASCS to new tools in this domain.

2 SORASCS Architecture in a Nutshell

While SOAs provide many foundational technologies and approaches that can be used in the intelligence domain, there are a number of key challenges that must be addressed to apply SOA:

- SOA provides general features for constructing web-based applications. In the intelligence community there is a need to provide features customized to this domain that allow: a) easy integration of intelligence tools, and b) providing workflow use and creation interfaces that are amenable to users from the intelligence community. This bridge between current SOA technologies and technologies that can be used by the community is a key challenge for SORASCS.

- Tools in the intelligence community are data-intensive, meaning that they work on and share large sets of data. The distribution, security, and staging of this data are crucial to address in this domain.
- Service invocations are potentially long-lived. What are the best models of interaction with services in this domain?
- Intelligence analysts desire to use a mixture of existing, workstation-based tools with which they may be familiar, and services of others to get their job done. For example, they may be familiar with one tool, but require a piece of technology that is not provided by that tool's vendor. How can a mixture of these thick and thin services be supported, both in daily use, and in developing workflows.
- Workstation-based (thick) tools are typically UI intensive, allowing interactivity such as network manipulation, interaction with maps, etc., whereas SOA typically support call-return style operations not requiring user interaction. How can these thick tools best be incorporated into a SOA setting

Work on SORASCS is developing an architecture that will be useful with both web-based and non web-based models and analysis tools. The aim of SORASCS is to be:

- Light: SORASCS should not involve providers completely rewriting their tools to conform to SORASCS. SORASCS should automate mundane tasks such as security, data use, etc., so that users and integrators are not burdened with these issues.
- Extensible: SORASCS should easily accommodate new kinds of tools, models, and services that are currently being developed in this thriving community.
- Community-based: Provisioned services will not be provided by SORASCS, but by tool providers in the intelligence community. The architecture therefore needs to be community based and open source.

In a nutshell, SORASCS will serve as the glue for linking a set of heterogeneous data sources and tools needed for modeling and analysis to support the needs of the intelligence community.

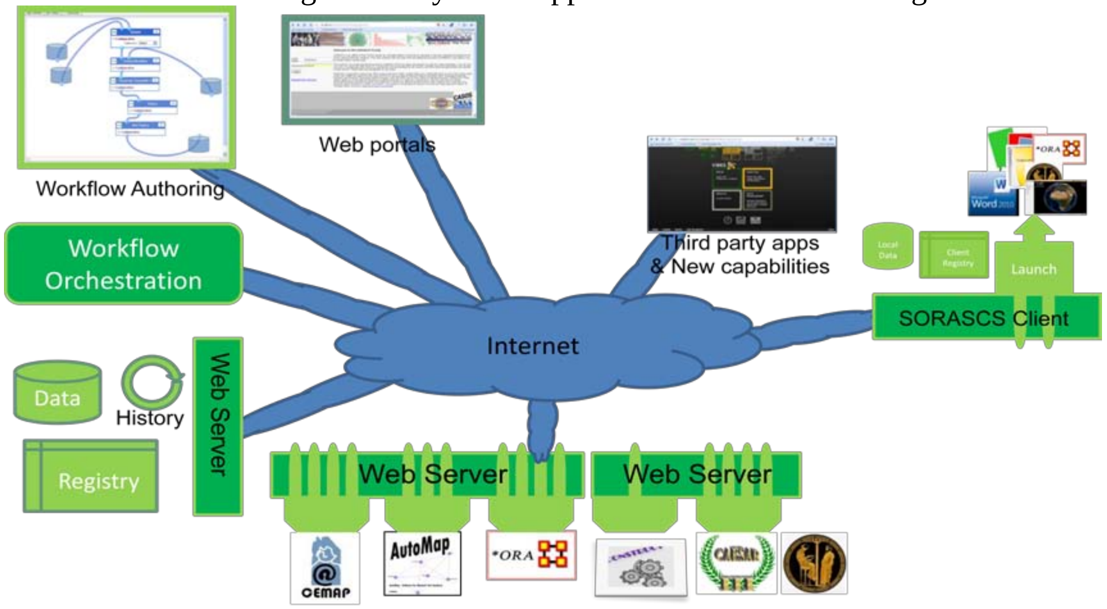


Figure 1. The SORASCS Architecture.

Figure 1 outlines the architecture for SORASCS; it consists of:

- Tools and components wrapped as web services and provided to the community via standard web service interfaces. Services are invoked by sending them messages (formatted as SOAP messages) through these interfaces. SORASCS defines a set of domain-specific APIs that SORASCS clients (web portals, workflow tools, and other custom tools) can use to interact with SORASCS services.
- A service registry, built on top of standard web service registry technologies, that provides additional information about services (currently parameterization, but other information such as quality of service, security related information, etc., is envisioned to go here).
- Data Services, that provide files and data required by services. References to data are passed to services and then resolved by SORASCS to reside on the local machine, for access by components and tools.
- Client-side services that provide SORASCS access to thick client tools that can be used on a user's local machine
- SORASCS level workflows, that are defined using a domain-specific workflow language that is designed to be accessible to data analysts. This workflow language allows analysts to capture and share common tasks with other analysts. The domain-specific language is translated into the technical language, BPEL, which is a standard SOA orchestration language and then executed on an open source execution platform. This translation allows a number of mundane steps to be automatically added to the analyst level workflow, such as

This document focuses on the first four aspects of SORASCS, which are of primary concern to integrators of tools into SORASCS.

2.1 SORASCS Services

A service in SORASCS provides a single HSCB operation, such as applying a delete list, generating a report, visualizing a network, etc. This is consistent with the standard definition of web services. However, in SORASCS, services are divided into two broad categories of services: Thin and Thick. A thin service is a service that can be run remotely, and does not require any user interaction other than when invoking the service. On the other hand, a thick service can be seen as the web service implementation of an existing tool, providing a user interface, user interactivity, etc. These latter services are not typically supported in by SOAs, and in fact run contrary to the accepted definition of a web service: they provide many operations, are often tightly coupled and monolithic, and their user interfaces are not amenable to distributed operation. However, in this domain, many users are familiar with these large tools, and in fact want to be able to automate them in some way with thin services into workflows to capture common tasks. SORASCS supports both thin and thick services.

Regardless of whether a service is thin or thick, it must provide some information to SORASCS in order to integrate with other SORASCS services. These interfaces are described in more detail in later sections, but they are broadly defined as:

- Ports for providing both synchronous and asynchronous communication protocols for service invocation. Synchronous operations are the typical way that clients interact with web services. However, in this domain, service operations may take exceptionally long on large data; SORASCS therefore provides an asynchronous interface that allows clients to submit operations to be done and then check later whether the operation has completed and get results. Synchronous operations should only be used in cases where operations are being performed on small amounts of data. In SORASCS, the normal mode of interaction will be through the asynchronous interfaces.

- Registry information. Standard SOA service descriptions, such as WSDLs, provide syntactic information about the API, but are impoverished with respect to the semantic meaning of an operation. Languages such as OWL are used for describing ontologies of services and provide more semantic meaning. SORASCS is exploring the use of OWL, but in the meantime SORASCS requires something in between that describes the optional parameters available for an operation. For example, a delete service, while requiring inputs and outputs and a list of words to delete, may also optionally allow clients to specify whether adjacent or rhetorical delete is to be used; services for visualizing networks may allow the layout algorithm to be specified. SORASCS provides a way of describing both the parameters that can be passed to a service and the results that can be obtained. This description is useful for a client to know what parameters the service understands, and what defaults are. Furthermore, such a description can be used by clients to construct user interfaces allowing users to specify the parameters. This port is primarily used by the registry when a service is registered to get additional information about the service.

2.2 Provenance

One of the goals of SORASCS is to provide to users some form of provenance, so that users can understand and inspect things that they and others have done before. Ideally, this consists of at least two aspects: 1) What services have been used to construct a result, their parameters and so on; and 2) any intermediate data that was produced. Currently, SORASCS provides information about the first aspect only through a service called the History Service.

Whenever a service is called in SORASCS, a record is kept by the History Service of the service called, the inputs that were passed, the outputs produced, who called the service, whether it was part of a workflow, and when the service was started and ended. SORASCS does this unobtrusively by intercepting all service calls and returns between the client and the service and keeping a record. The history service can be queried by the user to get this service call history. This service will not be described further in this document, as the developer and the client may interact with SORASCS unaware that the service calls are being stored.

2.3 Data

SORASCS services typically operate on large sets of data, such as text files, networks, etc. In a distributed web service environment, web services require access to this data. Because SOAs have restrictions on the size of messages that can be sent to services, sending data along with messages to the services is not possible. Many of the components that need to be integrated in SORASCS manipulate files, and expect them to be available on the machine on which the component is executing. We require clients to pass the files *by reference* to services and provide a Data Service that fetches the data as required. This data service is currently in rudimentary form. When integrating components into SORASCS, the Data service is invoked behind the scenes, to shield integrators as much as possible from a part of SORASCS that will rapidly change. When integrating a component in SORASCS, integrators are passed absolute filenames to files and directory on the local machine, that have been mapped by SORASCS from the parameters that clients pass to the service.

2.4 Registry

The SORASCS registry is where clients can go to discover services, determine their interfaces, parameters, and locations. Other meta data about a service will be stored in the registry as SORASCS develops. Once a component has been integrated into SORASCS by implementing the APIs, the integrator must register the new service with SORASCS. A governance process for this, which would involve authorization, testing, etc., could be implemented by SORASCS. In the meantime, service registration is currently done through the SORASCS portal.

3 Integrating Thin Services

Thin clients are services that provide a single operation and do not require a user interface (other than to start the operation) or user interaction. Such services can be run on a server. Thin clients are the building block for more powerful services, embodied in workflows. There are several advantages to having thin client services:

- Data-intensive operations can be run on powerful server clusters;
- There are opportunities for parallelism / data farming / cloud services
- Service providers can provide alternative versions of operations that can be used in workflows

In most cases, users will not interact directly with these thin clients. However, they are required for building up workflows, or for special purpose tools built on top of these services.
There are two aspects to a service, and this document describes both: a) the protocols/ports that each service must provide to so that it can interact with other parts of SORASCS; and b) the tool facing interface of the service, specifying how developers may integrate (wrap) a third-party tool or component into SORASCS. Developers need to understand both.

3.1 SORASCS Service API

SORASCS thin client services are responsible for providing the following kinds of interactions, or ports.

1. They must provide a synchronous interaction mode, in which the client blocks waiting for the operation to be performed.
2. They must provide an asynchronous interaction mode, in which the client submits a request to run the operation and then polls until the request has been finished.
3. They must provide a port that provides a description of how the operation is parameterized, which can be used by clients to build forms or interfaces to get parameters from the user.

These ports are client-facing ports, in that tools and clients of SORASCS services will assume that these ports exist and use them. They are operations that will appear in the web service description of the service. These ports are described in more detail below.

3.1.1 Synchronous Interaction

Synchronous interaction is the most common programming paradigm, and developers often strive to implement this interaction mode even when it is not provided. SORASCS provides this mode of interaction in the API, however it should be used judiciously; SORASCS services can

take a long time to complete on large data sets, and so depending on the client, waiting for completion may time out, or cause blocking of the client for large amounts of time.
The synchronous interaction mode is a single web service operation. Consider the synchronous operation on a thin client graph visualization service, IGraphVisualizer:

```
String visualizeGraph (String graphFile, Properties additionalProperties)
```

The way to read this operation is that the name for the operation is visualizeGraph. It has one *required* parameter, called graphFile[1], that is an argument to the port. The other argument to the port is a Properties argument, which is used to pass «key, value» pairs to the operation. All SORASCS operations have as their last argument this Properties object. The intent here is two-fold:

1. Optional parameters to the operation can be passed in. In the case of this operation it might include the algorithm to use for laying out the graph, which nodes in the graph to focus on, etc. The kinds of properties that an operation understands are described by an XML document that the service provides through the description port, described later in this section.
2. SORASCS passes some information to the service through this object, such as the identification of the caller, their credentials, etc., that are required by the SORASCS infrastructure.

If the operation fails for any reason, it throws a SORASCSException.

3.1.2 Asynchronous Interaction

Asynchronous operations allow clients to request the service to execute an operation, and then the clients are free to come back later to check on progress and get results. For large sets of data, operations can take hours or days, and typical synchronous web service calls will time out if a reply is not returned within a certain time period. SORASCS implements the asynchronous port as a pattern of methods for submitting the request, checking completion, and getting any results. Consider again the interface to a Graph Visualization service. It has three methods:

```
String submitVisualizeGraphRequest (String graphFile, Properties additionalProperties)
boolean isDone (String clientId)
String getVisualizeGraphResults (String clientId)
```

The first method is used for submitting the request to do an operation. The arguments to this method are the same as the arguments to the synchronous port. The name of the method is also based on the name of the method for the synchronous port, and has the pattern submit<operation>Request. This method always returns a String, which is a universally unique identifier that the client can use for interaction with the other methods on the asynchronous port.

The second method, isDone, is used for checking if a request has completed. The argument to this method is a clientId that a client received from the request method. It returns true if the operation for the client has completed.

The third method is optional, and is implemented in the case that an operation returns a result. Again, the pattern of the method name is based on the name of the synchronous port:

[1] How SORASCS deals with files in a distributed web services environment is discussed later in this document.

get<operation>Results. This method passes in the clientId that was returned to the client when it made a request.

It is possible for a service to provide numerous operations, although this is not common. For example, if the graph visualize service also provided an operation to return the number of nodes of a certain type in a graph, the service might provide the following methods:

```
int numberOfNodes (String graphFile, String nodeType, Properties additionalProperties)
String submitNumberOfNodesRequest (String graphFile, String nodeType, Properties additonalProperties)
int getNumberOfNodesResults(String clientId)
```

Notice that the method isDone is required only once by the web service. The universally unique id will also distinguish which operation is being checked for completion.

3.1.3 Operation Parameterization

Each service provides a port to get further description the operations that the service provides. This is different to the web services description, typically embodied in a WSDL, which merely provides the API to the web service. The description provides further information about the required parameters, and also what optional parameters the operation understands. An additional intent of this description is that clients can construct user interfaces, or forms, for users to parameterize the services. For example, the portal uses this information to dynamically construct HTML forms for calling a service; the workflow tool uses it for constructing a UI for parameterizing services.
The method that implements this port is:

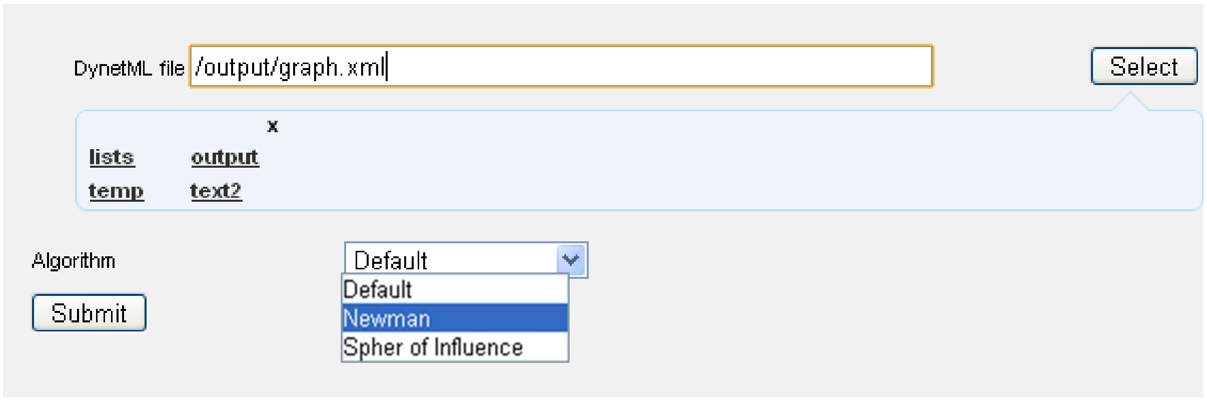
```
String getParametersForMethod (String operation)
```

Consider the operation visualizeGraph from above. The XML description for an instance of this operation may look like:

```
<parameters operation="visualizeGraph">
  <parameter name="graphFile" type="file" filetype="xml" required="true">
    <prompt text="DynetML file" />
  </parameter>
  <parameter name="algorithm" type="choice" mode="property" key="algorithm" default="none">
    <prompt text="Algorithm">
      <description>The algorithm to use in visualizing the graph.</description>
    </prompt>
    <options>
      <option name="None" value="none"/>
      <option name="Newman" value="newman"/>
      <option name="Sphere of Influence" value="soi"/>
    </options>
  </parameter>
</parameters>
```

The schema for this XML is provided in Appendix I. In this case, the first parameter graphFile, which is required, is intended to be a reference to an XML file, and is required. This is the description of the first parameter to the visualizeGraph methods. The second parameter element is an example of a parameter that can be passed in the Properties argument, as indicated by the mode="property" attribute. This parameter is a choice, the default being one, the options being described in the options section. The key attribute specifies the key that will be used. Given this specification, it is possible to pass a «key, value» pair of «"algorithm","newman"» or «"algorithm","none"». Other kinds of properties are described in the schema, and include integer, string, and boolean.

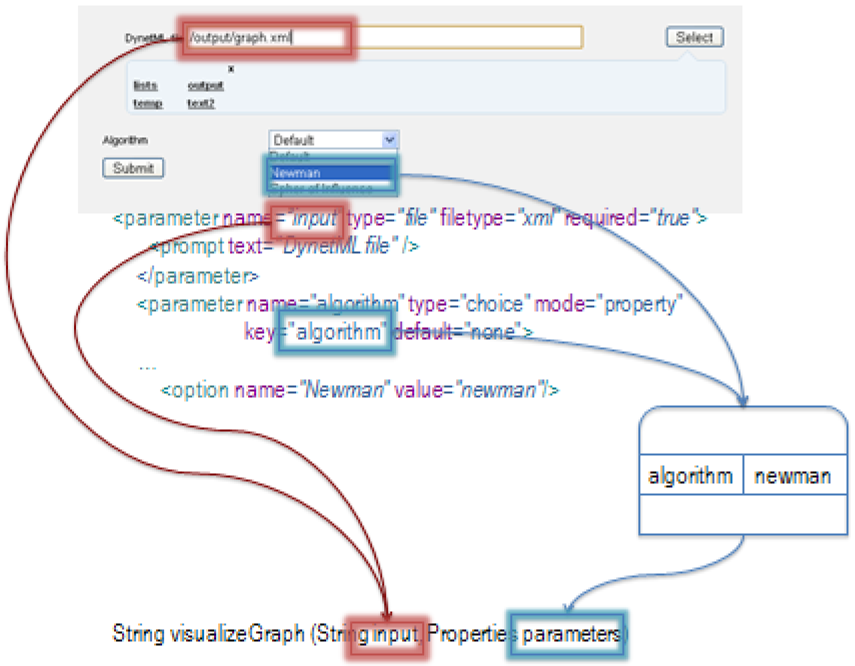
The figure below shows an example of how this XML is used to generate a form in the User Interface. The developer will likely not see this, but it is useful to understand the context of what the user interface may look like. Also note that in addition to generating an HTML form, the XML could be used to generate a Java interface, or translated to JSON for use in other UIs.


DynetML file /output/graph.xml

Algorithm: Default

Figure 2. A User Interface Generated from the parameter XML Specification.

When the user fills in the values in the UI, the values are eventually passed to the operation of a thin service as Java values. The figure below illustrates how this works. The value /output/graph.xml is passed as the input parameter to the operation. The value "Newman" selected from the drop down list is passed to the operation in the parameters parameter of the operation, with the key "algorithm" and the value "newman".



```
<parameter name="input" type="file" filetype="xml" required="true">
    <prompt text="DynetML file" />
  </parameter>
  <parameter name="algorithm" type="choice" mode="property"
               key="algorithm" default="none">
  ...
    <option name="Newman" value="newman"/>

String visualizeGraph (String input, Properties parameters)
```

Figure 3. Translating parameter specification to code objects.

3.1.4 Categorization

SORASCS currently provides the following categories of services, based on a range of common services within the CASOS toolset.

Category	Description
Extraction	boolean extract (String output, Properties…) Extracts information from various resources and result in text files in SORASCS (e.g., extracting from Mail, News, Web, ...). The output parameter specifies where in SORASCS the results should go. Information about the source of the extraction is passed in as Properties.
Conversion	boolean convert (String input, String output, Properties…) Converts between one format and another. The input parameter specifies the input to be converted, the output specifies where the output should go.
Transformation	boolean transform (String input, String output, Properties…) Performs some transformation of data in SORASCS. The original intent of this category was to transform text from one form to another (for example, by changing its case),

	but it is now used more generally than that (for example, to transform text into a network).
List-based Transformation	boolean transform(String input, String output, String list, Properties…) Performs some transformation of data in SORASCS that relies on a list. For example, deleting text from the input requires a delete list.
Reports	String report (String graphFile, Properties…) Generates a report based on the network passed in as the first parameter. All reports accept the following parameters: "outDir": The location in the clients server directory to put results. Omitting this will cause results to be placed in the user's top user directory. "format" The format the results will be in. Accepted values are: "html" HTML will be generated "text" Text will be generated "csv" Comma separated files will be generated “powerpoint” Powerpoint will be generated The default behavior is that HTML will be generated "mode" The return mode. Accepted values are: "link" A link to the index page for the report "zip" A zip file is returned containing all contents of the report. The default is "link"
Merge	boolean merge (String input, String output, Properties…) Merges data. The data is contained in the input parameter (which may refer either to a directory containing files to merge, or a comma separated list of files – as specified in the XML description of the operation). The output specifies where to put the merged data.
GraphVisualization	String visualizeGraph(String graphFile, Properties…) Visualizes graphs. The graphFile is a reference to the network to be visualized. The return string is a reference to the URL containing the visualized image.

All follow same interface scheme: required parameters are passed as service parameters, optional parameters passed as properties in last parameter; they have both synchronous/asynchronous interaction patterns.

3.2 Tool Integration

This section runs through an example of integrating a thin client into SORASCS. The example we will use is integrating an existing component that takes an internet chat log and produces a list of names of participants in the chat. Chat logs are text files that contain lines like:

```
[00:03:17] <kevglass> awesome, just generated some quests based on heroquest
[00:03:23] <kevglass> now playing levels I've not seen before :)
[00:04:33] <kappaOne> cool
```

The service will take these lines and produce a file containing just the names:

```
kevglass
kappaOne
```

How it does this is not important for integration. For integration, it is sufficient to know that this component has a main method that takes two arguments: 1. A file on the machine where the component is running that contains the chat log, and 2. The location and name of the resulting file that will contain the list of names.

It is possible to implement from scratch the service that implements the client side interface directly. However, doing this, the developer will need to write code for managing file distribution, any user validation, threading, etc. Instead, SORASCS provides a number of abstract classes that handle this additional plumbing as part of the SORASCS framework, and instead can write a method that simply calls the component to do the transformation.

To implement the service, the developer must write the following, not necessarily in the order described:

1. The XML describing the parameters for this operation;
2. A Java interface extending one of the Java interfaces for one of the predefined categories in SORASCS;
3. An implementation of one of the abstract SORASCS framework classes for one of the predefined categories;

We will cover each of these steps in detail. Let's decide that this component will be a transformation kind of service, because it is transforming one kind of file (chat logs) into another (a list of names).

3.2.1 Java Web Service Interface

The SORASCS framework uses Apache CXF as the means for providing web service interfaces of Java classes. Apache CXF provides a means of automatically transforming an annotated Java interface to a standard Web Services Description Language (WSDL) file that is the W3C standard for web services. Each service requires a corresponding interface. Because we are providing a transformation service, and this is provided by SORASCS, the definition of this interface is very straightforward:

```
package edu.cmu.casos.sorascs.example;

import javax.jws.WebService;
import edu.cmu.casos.sorascs.services.ISimpleTransformation;

@WebService()
public interface IRCNamer extends ISimpleTransformation {}
```

We will assume that all our classes will be in the edu.cmu.casos.sorascs.example package, though developers are free to place this anywhere. The service is called IRCNamer, and the interface is intended to be a web service. The class is therefore annotated with the javax.jws.WebService annotation, which will name the service IRCNamer by default. Because the service is a transformation service the interface extends edu.cmu.casos.sorascs.services.ISimpleTransformation, which is a SORASCS framework interface. That interface provides the methods and annotations to define the client facing interface for a transformation service.

3.2.2 Providing the Operation

Each supported category in SORASCS has a corresponding abstract Java class, that provides support for:

1. Localizing any file arguments. This interacts with the SORASCS Data Service to download any files required from a user's SORASCS workspace to the local file system and uploading the files back to the SORASCS workspace when done.
2. Handling the mapping of the developer provider operation to the client side interface. This allows the developer to implement the algorithm for their service only once.
3. Handling threading and client correlation of requests to responses for the asynchronous client side port.
4. Managing user validation.

The intent of the SORASCS framework is to have the developer implement the lease amount of code to implement the operation, and have the SORASCS framework handle as much of the plumbing as possible. To this end, the developer need only provide two things to extend the SORASCS framework:

1. The implementation of the operation; and
2. The WorkerThread that will be used for buffering asynchronous operations.

Implementations of this abstract class provide the mapping between the client-side interface and the actual implementation of the operation.

The actual operation is implemented in SORASCS as an object. This operation must extend IOperation, and encapsulates the parameters required by the operation and passed in from the service. SORASCS provides abstract classes for each of category of service that provides getters and setters for the parameters, and requires the developer to only implement the doOperation method in the subclass.

The code below provides the implementation for the IRCNamer:

```
package edu.cmu.casos.sorascs.example;

import edu.cmu.casos.sorascs.SORASCSException;
import edu.cmu.casos.sorascs.services.AbstractTransformationService;
import edu.cmu.casos.sorascs.services.TransformOperation;
import edu.cmu.casos.sorascs.util.WorkerThread;

public class IRCNamerImpl extends AbstractTransformationService implements IRCNamer {

  // The thread to handle asynchronous requests.
  // This needs to be static because CXF creates a new instance of
```

```
        // the class per service invocation, and we want only one thread
        // for all instances of the operation.
        static WorkerThread worker = new WorkerThread ("IRC Namer");

        @Override
        public WorkerThread getServiceThread() {
           return worker;
        }

        // Return the operation of implementing the guts of the code
        @Override
        protected TransformOperation buildOperation() {
           return new TransformOperation() {
              @Override
              public Boolean doOperation() throws SORASCSException {
                 try {
                    edu.cmu.casos.sorascs.example.thirdparty.IRCNamer.
                       main(new String [] {getInputParameter(), getOutputParameter()});
                 } catch (Exception e) {
                    throw new SORASCSException (e.getMessage (), e);
                 }
                 return true;
              }
           };
        }
     }
```

The AbstractTransformationService class requires the developer to implement two methods:

```
TransformOperation buildOperation ()
WorkerThread getWorkerThread ()
```

The method buildOperation returns an instance of the corresponding TransformOperation class (because we are implementing a Transformation service, it is required that the operation be a subclass of TransformOperation). TransformOperation provides the following methods that are useful for the developer (in addition to doOperation, which the developer must implement):

```
public Properties getProperties ()
public String getInputParameter ()
public String getOutputParameter ()
```

Each of the getter methods above is how the operation gains access to the web service parameters. Each SORASCS operation class provides getters for the parameters of the corresponding category's operation. In the implementation code above, we create an anonymous TransformOperation class that provides the doOperation method. This method access the input and output parameter through the appropriate getters and passes it through to the actual implementation of IRCNamer, in the third party class.

When a call is made to the service's operation, the SORASCS framework does the following:

1. Calls buildOperation to construct the operation object

2. Sets the properties object of the operation object to the properties passed in to the call
3. Sets the input and output parameters passed in as parameters to the call. The input and output files are transferred to the location of the service, if necessary.
4. Places the operation on a queue to be processed
5. When the operation is dequeued by the worker thread, then doOperation is called.

The figure below provides a summary of the classes and interfaces that are involved in integrating a thin client. Above the dashed line are the classes and interfaces defined by the SORASCS framework. The integrator must supply the classes and interfaces below the dashed line.

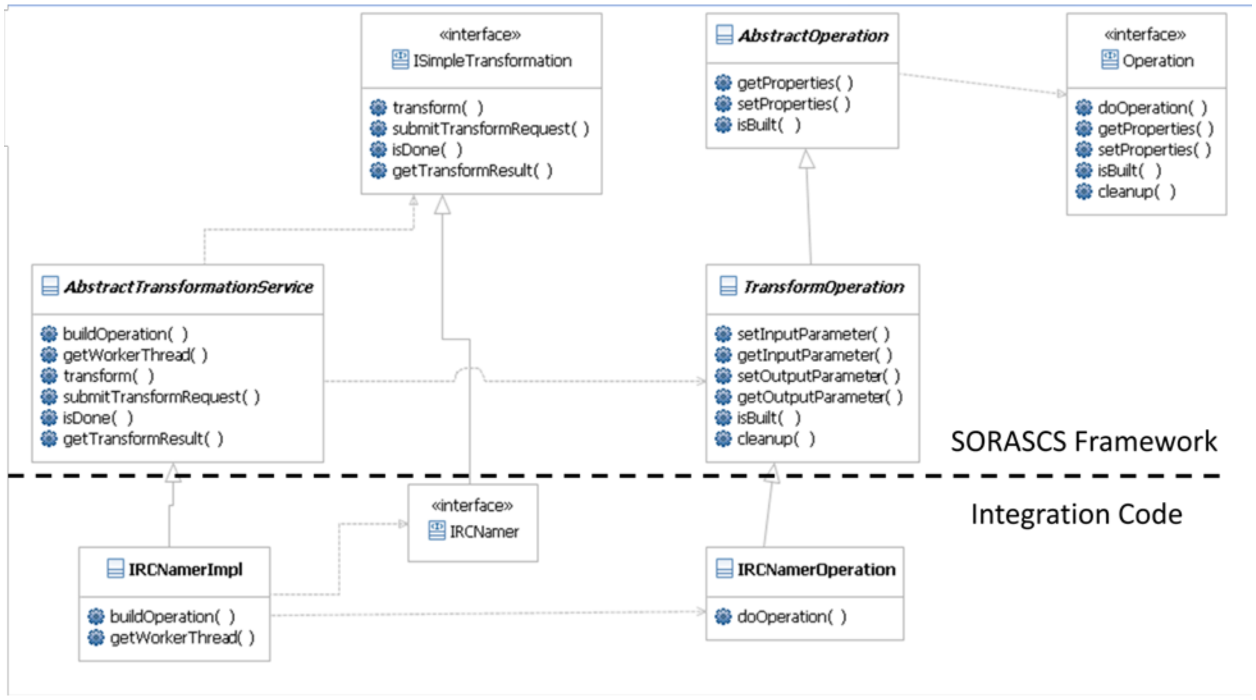


Figure 4. Class hierarchy for SORASCS Framework and Tool Integrator code.

The SORASCS-provided interfaces and classes are:

ISimpleTransformation: Defines the web service interface for a transform operation. This is the interface seen by clients to the service.

Abstract TransformationService: Defines an implementation of the ISimpleTransformation interface, implementing the bridging code between the web service interface and the integrator-supplied operation. It defines two abstract methods: buildOperation, which will be called to get the integrator's operation object, and getWorkerThread, which returns the thread that will be used for managing asynchronous operations.

Operation: defines a base level Operation interface that all operations will realize.

AbstractOperation: provides implementations of some Operation methods for setting and getting the properties of the operation.

TransformOperation: provides a means for setting and getting the input and output parameter for an operation. This operation also handles data transfer.

The table below provides a summary of the SORASCS framework classes that need to be implemented for each of the currently supported service categories

Category	Extend Web Service Interface	Extend Abstract Implementation Class	Extend Operation
Extraction	IExtractor*	AbstractExtractorService	ExtractionOperation
Conversion	IConverter	AbstractConversionService	ConversionOperation
Transformation	ISimpleTransformation	AbstractTransformationService	TransformOperation
List-based Transformation	IListBasedTransformation	AbstractListTransformationService	ListTransformationOperation
Reports	IReport	AbstractReportService	ReportOperation
Merge	IMerge	AbstractMergeService	MergeOperation
GraphVisualization	IGraphVisualization	AbstractGraphVisualizationService	GraphVisualizationOperation

*All classes and interfaces are in the package edu.cmu.casos.sorascs.services, in the sorascs.jar file.

3.2.3 Parameterization

The XML describing the parameters for this operation are fairly straightforward. The input and output will both be a text file. While it is certainly possible for the developer to define their own way for returning the XML, the SORASCS framework provides a default behavior that will be sufficient for most cases. The XML needs to be placed in a file that will be placed in a certain location, and the file needs to be named using the following convention:

<ServiceName>_<operation>Param.xml

Let us call our service IRCNamer. Because it is also a transformation service, the XML file name should be called: IRCNamer_transformParam.xml. Furthermore, the XML file should be placed in the directory edu/cmu/casos/sorascs/example directory in the bundle for the service. The XML file for the IRCNamer service will contain:

```
<parameters>
  <parameter name="inputFile" type="file" filetype="txt" required='true'>
    <prompt text="IRC Chat Log File"/>
  </parameter>
  <parameter name="outputFile" type="file" required='true'>
    <prompt text="Resulting file"/>
  </parameter>
</parameters>
```

3.2.4 Bundling and Deployment

After the service has been implemented, it must be bundled up into a WAR file and deployed on a web server. The WAR file (or Web ARchive file) is a bundle that contains the following things:

1. The classes associated with the service implementation
2. All the JAR libraries necessary for the service to run (this includes sorascs.jar and the Apache CXF jars)
3. A web.xml that describes how the package should be deployed
4. A beans.xml that points Apache CXF to the service implementation object

The picture below illustrates what the file system should look like for your example. The files that need to be changed are: src/, beans.xml, and build.xml. The other files will be the same as those that are included with the IRC example.

src/: The src/ directory contains the Java source files for the service, as well as the parameter XML file.

beans.xml: Will be changed to point to the class implementing the service. The lines below will need to be in the beans.xml. The implementor attribute should point to the service implementation (not the interface). The address is where CXF will publish the service, relative to the server. The pattern used is http://server/<*warname*>/*examples/IRCNamer*.

```
<jaxws:endpoint
  id="IRCNamer"
  implementor="edu.cmu.casos.sorascs.example.IRCNamerImpl"
  address="/examples/IRCNamer" />
```

build.xml: Will be changed to appropriately named the WAR file.

The build file defines a number of targets:
clean: remove any files that were generated as part of the build process
build: build the Java class files (i.e., compile the source)
deploy: builds the war file, and deploys it on the server by copying it to the server location
undeploy: removes the war file (and it’s expanded directory) from the server location

4 Integrating Thick Services

Unlike thin clients, Thick Clients in SORASCS are characterized as having a rich user interface and intensive user interaction. The kinds of thick clients can range from complete tools (such as ORA or PowerPoint) to component tools (such as using the ORA visualizer). Being that they are user interface intensive, it is not possible to run them on a server: user interfaces need to be brought up on the client.

Thick clients are integrated into SORASCS so that they can be

- Included as part of a SORASCS workflow
- Can be invoked as a web service by other tools

Furthermore, many users have been trained in using existing tools, and are accustomed to their user interfaces. In order to accomplish integration of thick clients in SORASCS, SORASCS must be able to communicate with client machines to launch and direct applications. Because this opens up the client to communication with the outside network, we have been careful to give the users full control over what is available and not available on their machines. We do this by:

- The user must have tools that are used as thick clients installed on their machine already. SORASCS will not install programs on client machines.
- The user has complete control over what programs can be directed by SORASCS. Programs must be explicitly registered by users with SORASCS
- SORASCS provides a dashboard showing what programs SORASCS has launched, and what files on the local machine they are working on

The SORASCS client, once installed on the client machine, acts as a web application server that provides client-based services to SORASCS. The SORASCS client uses the same technology as SORASCS (apache, CXF), rather than a hand-crafted application server, to enable the full range of security and protection available from a modern web server.

4.1 Kinds of Thick Integration

Thick clients can be installed with SORASCS in a number of ways, depending on the granularity of the tools and integration with SORASCS. Tools can be launched from SORASCS, and passed data to open on launch. Alternatively, if the tool can be directed, then thick clients can be registered as component-level services. For example, the ORA tool for managing meta networks can be launched to open meta networks, in which case the user is free to manipulate the networks in the manner they are used to, or a Graph Visualization service could be registered on the client that launches the ORA visualize on the client.

The SORASCS Data Service will transfer files between SORASCS and the client.

4.2 User Installation: SORASCS Client-side Server

SORASCS requires the installation of a Client-side server on a user’s machine. This client manages the calls from SORASCS to the Thick Clients installed on the user’s machine, and provides a dashboard for informing the user of SORASCS’ activity on the client machine. In the future, it is likely that the dashboard will be used for workflows requiring user interaction.

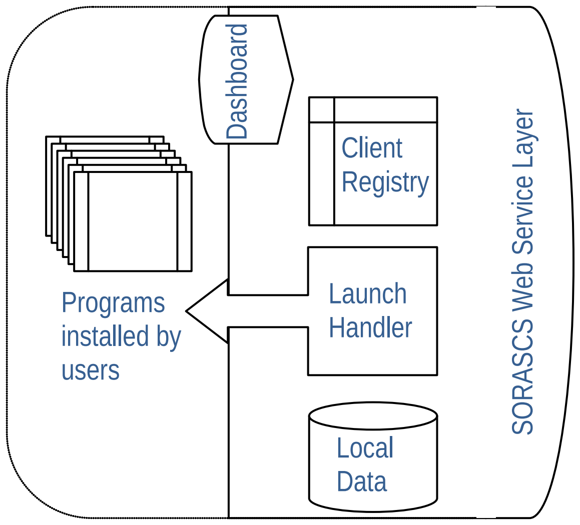


Figure 5. Components running on SORASCS Client machine.

The figure above shows a schematic of what runs on the client machine.

Client Registry: The client registry contains information about services that can be run on the client machine. Tool integrators are responsible for defining this information; users are responsible for registering the tools they want to enable SORASCS to execute on their machines.

Launch Handler: This component is responsible for starting processes on the client machines.

Local Data: Data that tools need from SORASCS are managed by the local data component. It is responsible for synchronizing data with SORASCS. When a request is made to execute a tool by SORASCS, the Launch Handler invokes this component to ensure that up-to-date files are located on the client machine. After the tool is completed, this component is responsible for synchronizing files back to SORASCS.

Dashboard: The dashboard is responsible for informing the client of any tools that SORASCS has launched and the files they are working on.

4.3 Developer Integration

This section of the document describes how a developer of a tool will integrate the tool as a thick client with SORASCS. SORASCS strives to keep this simple. To integrate a thick client as a program that SORASCS can launch, a developer must simply provide an XML file describing the parameters for integration with SORASCS and provide this file as part of the installation of their tool. For example, let's say that ORA will be installed in C:\Program Files\ORA\ on the client machine, and the executable program will be C:\Program Files\ORA\ORA.exe. The developer will provide a file sorascs_ora_thick_client_params.xml that will specify the parameters. The XML in this file will be the same format as the XML for the thin clients, except that parameters will have the mode="parameter" attribute set. Parameter mode parameters will be passed on the command line in the order that they are specified in XML. Imagine that ORA had the following command line options:

```
ORA.exe [/report=<report>|/visualize] file.xml
```

Where <report> could be "Intelligence" or "Content". The developer would specify the XML for parameterization as:

```
<parameters>
    <parameter name="report" type="choice" mode="parameter" key="param" default="">
      <prompt text="Tell ORA to:">
        <description>Optionally select an operation for ORA to do</description>
      </prompt>
      <options>
        <option name="Just open the file" value=""/>
        <option name="Generate Key Entity Report" value="/report=intelligence"/>
        <option name="Generate Content Analysis" value="/report=content"/>
        <option name="Display the network" value="/visualize"/>
      </options>
    </parameter>
    <parameter name="input" type="file" filetype="xml" mode="parameter" key="input">
      <prompt text="Dynetml File"/>
    </parameter>
  </parameters>
```

4.4 SORASCS User Registration

If a program provides the parameters in the XML as above, the user must still explicitly register the program on their machine as one that can be called by SORASCS. This is done by the user (or local administrator) through the SORASCS dashboard. The user must select the "Programs" button on the dashboard, which brings up a list of the currently registered Thick Services.

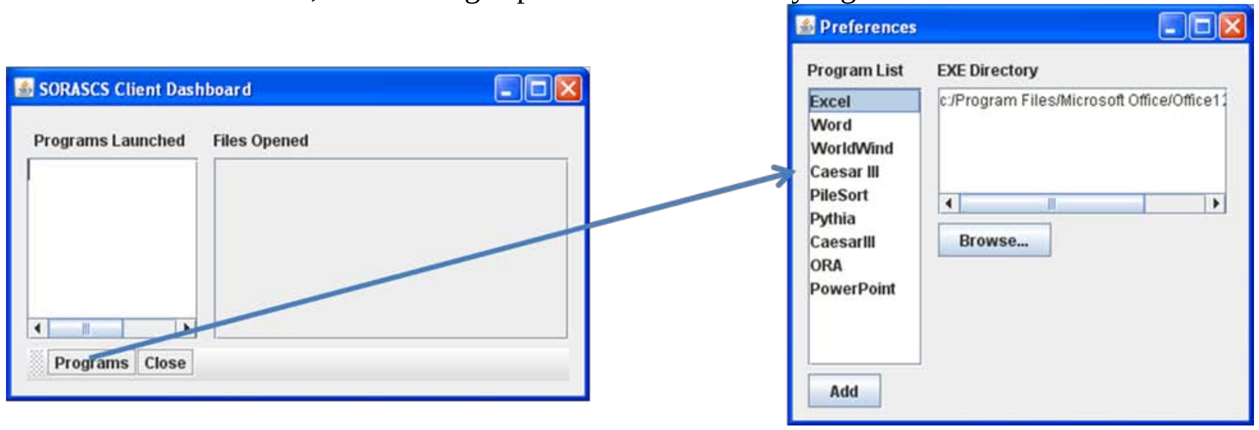


Figure 6. Registering a new Thick Service.

The user must then select the Add button, which allows them to add a program, first by giving it a name, and then by selecting the location of the executable for the program on their local machine.

5 References

[1] Tsvetovat, M., Reminga, J., and Carley, K.M. (2003). DyNetML: Interchange Format for Rich Social Network Data. NAACSOS Conference 2003, Day 2, Electronic Publication, Pittsburgh, PA.

[2] Carley, K.M., Reminga, J., Storrick, J., and DeReno, M. (2009). ORA User's Guide 2009. Carnegie Mellon University, School of Computer Science, Institute for Software Research, Technical Report CMU-ISR-09-115.

[3] Garlan D.K., Carley K.M., Schmerl B., Bigrigg M.W. and Celiku O. 2009. Using Service-Oriented Architectures for Socio-Cultural Analysis in Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE2009), 2009 July 1-3; Boston, MA.

[4] Carley, K.M., Columbus, D., DeReno, M., Bigrigg, M.W., Diesner, J., and Kunkel, F. (2009). AutoMap User's Guide 2009. Carnegie Mellon University, School of Computer Science, Institute for Software Research, Technical Report CMU-ISR-09-114.

[5] Whorf, B. "Language, Thought, and Action" in Twenty Questions: An Introduction to Philosophy. Bowie, Lee, Michaels, and Solomon (eds.), HBT, 1908.

[6] Berlin, B. and Kay, P. Basic Color Terms: Their Universality and Evolution, University of California Press, 1991.

6 Appendix

6.1 Parameter Schema Definition

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.example.org/SORASCSParameters"
  xmlns:tns="http://www.sorascs.casos.cs.cmu.edu/SORASCSParameters"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">

     <element name="service">
      <complexType>
        <sequence>
          <element name="mainOperation" type="string"/>
          <element name="parameters" type="tns:ParametersList"/>
          <element name="results" type="tns:ResultsList"/>
        </sequence>
         <attribute name="service" type="string"/>
      </complexType>
     </element>

  <complexType name="ParametersList">
    <sequence>
      <choice maxOccurs="unbounded" minOccurs="0">
        <element name="parameterGroup" type="tns:ParameterGroup"/>
        <element name="parameter" type="tns:Parameter"/>
        <element name="prompt" type="tns:Prompt"/>
        <element name="pagemap" type="tns:PageMap"/>
      </choice>
    </sequence>
  </complexType>

  <complexType name="PageMap">
    <sequence>
      <element name="pagejump" type="tns:PageJump"/>
    </sequence>
  </complexType>

  <complexType name="PageJump">
    <attribute name="page" type="integer" use="required"/>
    <attribute name="next" type="integer" use="required"/>
    <attribute name="param" type="string"/>
    <attribute name="value" type="string"/>

  </complexType>

  <complexType name="ResultsList">
    <sequence>
        <element name="result" type="tns:Result" maxOccurs="unbounded"
minOccurs="0"/>
    </sequence>
  </complexType>

  <complexType name="ParameterGroup">
    <sequence>
      <element name="text" type="xs:any" minOccurs="0" maxOccurs="1"/>
    </sequence>
    <attribute name="id" type="string" use="required"/>
    <attribute name="text" type="string" use="optional"/>
  </complexType>

  <simpleType name="paramtype">
    <restriction base="string">
      <enumeration value="integer"/>
      <enumeration value="choice"/>
      <enumeration value="directory"/>
      <enumeration value="file"/>
      <enumeration value="date"/>
      <enumeration value="multichoice"/>
      <enumeration value="radiochoice"/>
      <enumeration value="boolean"/>
      <enumeration value="real"/>
      <enumeration value="string"/>
      <enumeration value="upload"/>
      <enumeration value="multiselect"/>
    </restriction>
  </simpleType>

  <simpleType name="modetype">
    <restriction base="string">
      <enumeration value="property"/>
      <enumeration value="parameter"/>
    </restriction>
  </simpleType>
  <simpleType name="inouttype">
    <restriction base="string">
      <enumeration value="input"/>
      <enumeration value="output"/>
    </restriction>
  </simpleType>

  <complexType name="Parameter">
    <sequence>
      <element name="prompt" type="Prompt" minOccurs="0" maxOccurs="1"/>
      <element name="options" type="Options" minOccurs="0" maxOccurs="1"/>
      <element name="range" type="Range" minOccurs="0" maxOccurs="1"/>
      <element name="store" type="Store" minOccurs="0" maxOccurs="1"/>
    </sequence>
    <attribute name="name" type="string" use="required"/>
    <attribute name="type" type="tns:paramtype" use="required"/>
    <attribute name="mode" type="tns:modetype"/>
    <attribute name="key" type="string"/>
    <attribute name="required" type="boolean" default="false"/>
    <attribute name="default" type="string"/>
    <attribute name="group" type="string"/><!-- This should really be a
reference -->
    <attribute name="page" type="integer"/>
    <attribute name="filetype" type="string"/>
    <attribute name="direction" type="tns:inouttype"/>
  </complexType>

  <complexType name="Prompt">
    <sequence>
      <element name="text" type="any" minOccurs="0" maxOccurs="1"/>
      <element name="description" type="any" minOccurs="0" maxOccurs="1"/>
    </sequence>
    <attribute name="text" type="string"/>
    <attribute name="id" type="string"/>
    <attribute name="ref" type="string"/>
    <attribute name="description" type="string"/>
  </complexType>

  <complexType name="Options">
    <sequence>
      <element name="option" type="Option" minOccurs="0" maxOccurs="1"/>
    </sequence>
    <attribute name="source" type="string"/>
    <attribute name="default" type="string"/>
  </complexType>

  <complexType name="Option">
    <attribute name="name" type="string"/>
    <attribute name="value" type="string"/>
  </complexType>

  <complexType name="Store">
    <attribute name="id" type="string"/>
    <attribute name="class" type="string"/>
  </complexType>
  <complexType name="Result">
    <sequence>
      <element name="prompt" type="Prompt" minOccurs="1" maxOccurs="1"/>
    </sequence>
    <attribute name="name" type="string"/>
    <attribute name="key" type="string"/>
    <attribute name="type" type="string"/>

  </complexType>

</schema>
```