Master's Thesis

Distributed Knowledge Sources in SIARAS

Petter Lidén
petter.liden@tactel.se

Supervision:
Jacek Malec, jacek@cs.lth.se
Slawomir Nowaczyk, slawek@cs.lth.se

Examination:
Anders Nilsson, andersn@cs.lth.se

Abstract

SIARAS (Skill-based Inspection and Assembly for Reconfigurable Automation Systems) is an EU-funded project with main goal to build an automated system to support (semi-)automatic reconfiguration of manufacturing processes.

The main part of this system, known as the Skill Server, uses a knowledge source which has two main parts. The ontology file, representing the vocabulary needed to reason about devices and skills and their behaviour, i.e. the possible objects within the manufacturing process and how they relate to each other; and the device library which holds information about the devices and their properties.

With this information the skill server is able to reason about what can or can not be done in reconfiguring the system as result of some type of new requirement or in response to hardware/device changes.

The objective of this thesis is to analyze the knowledge source(s) and the flow of information to find a new architecture that facilitates both access to the information and the maintenance of it. The analysis should result in a prototype software, a knowledge server that maintains the information and provides a well-defined interface for accessing it. Specifically, the device libraries should be separate from the ontology (vocabulary) information.

Table of Contents

1 Introduction....5
  1.1 SIARAS....5
  1.2 Objective....5
  1.3 Structure....6
2 Background....7
  2.1 The Skill Server....7
    2.1.1 Ontology....7
      Devices....7
      Properties....7
      Skills....7
    2.1.2 Device Libraries....8
    2.1.3 Skill Server Architecture....9
  2.2 Ontology....10
    2.2.1 Introduction....10
    2.2.2 OWL....11
    2.2.3 Protégé....11
  2.3 Library Maintenance Tool....13
3 Design....14
  3.1 System Requirements....14
  3.2 Introducing the Knowledge Server....14
    3.2.1 The Ontology Module....15
    3.2.2 The Verification Module....15
    3.2.3 The Parsing Module....16
    3.2.4 Database....16
    3.2.5 Administration Interface....17
  3.3 Distributing the Device Libraries....17
  3.4 Database and Server Software....17
4 Implementation....19
  4.1 A Design for Promoting Extendability and Modularity....19
    4.1.1 The Model-View-Controller Design Pattern....19
    4.1.2 MVC in the Knowledge Server....20
  4.2 Separating Database, Server and Interface....21
    4.2.1 The Knowledge Database....21
      Tables....22
        Devices....22
        Properties....22
        Libraries....23
        Defaults....23
    4.2.2 The Knowledge Server....23
      Fetching....23
      Parsing....24
      Verifying....24
    4.2.3 The Administration Interface....24
      Connection....25
      User Interface....25
        Overview....25
        Device Database....26
        Device Libraries....26

Device Verification....................................27
Defaults Management....................................28
4.3 Tests....................................29
4.4 Code....................................29
5 Conclusions and Future Work....................................30
6 Bibliography....................................31

1 Introduction

1.1 SIARAS

SIARAS (Skill-based Inspection and Assembly for Reconfigurable Automation Systems) [2] is an EU-funded project with main goal to build an automated system to support (semi-)automatic reconfiguration of manufacturing processes.

The main part of this system, known as the *Skill Server (SkS)* [1], [3], uses a knowledge source which has two main parts. The *ontology* file, representing the vocabulary needed to reason about devices and skills and their behaviour, i.e. the possible objects within the manufacturing process and *how they relate to each other*; and the device library which holds information about the devices and their properties.

With this information the skill server is able to reason about what can or can not be done in reconfiguring the system as result of some type of new requirement or in response to hardware/device changes.

As an example take a car assembly line where one of the steps is the fitting of a windshield. Should it be decided that a new type of windshield is to be used, it needs to be investigated how this affects the manufacturing process. The new windshield may be heavier and a more powerful robot fitting it to the car is needed; or it may be that the windshield needs to be fitted in an entirely new way and there's a need to replace the robot(s) altogether.

Such reasoning is what the skill server is about; by using device data together with the ontology's information about how devices are interconnected and interact, and what skills they have, it can draw conclusions about what has to be altered in the manufacturing process, if anything.

1.2 Objective

The knowledge about devices, skills, their properties and relationships is currently technically stored in two separate structures (although logically it is one entity):

- An ontology file maintained through the *Protégé* tool.
- A text file, where the device library is described.

The objective of this thesis is to analyze the knowledge source(s) and the flow of information to find a new architecture that facilitates both access to the information and the maintenance of it.

The analysis should result in a prototype software, a *knowledge server* that maintains the information and provides a well-defined interface for accessing it. Specifically, the device libraries should be separate from the ontology (vocabulary) information.

While the ontology remains fairly constant, the actual devices provided by manufacturers may change over time and this information should therefore be handled and stored in a more dynamic way - a way that allows for easy updating as sources are modified.

Also, because device libraries are provided by different manufacturers the information may be in different formats. Moreover, it should be possible for manufacturers to supply and update information asynchronously with regard to interaction with the skill server. That is, the suppliers of device information should not need to alert the system that the information has changed.

The knowledge server should enable distributed knowledge sources. I.e., manufacturers should be able to publish new information for access by the knowledge server and ultimately the skill server independently of each other and without help from anyone maintaining the skill server system.

1.3 Structure

In section two the background of the project and its main concepts and components are detailed and discussed. Through this discussion we establish a set of requirements that will be used as a foundation for designing the actual system.

Section three concerns the design of the prototype software. We use the conclusions from section two to outline a design that will support the development of the system and that promotes key aspects of a sound software such as modularity and extensibility.

The fourth section describes implementation of the system and details the realization of the design choices made earlier. We look at actual technology choices and how the system will look and execute. Limitations to the design, possible extensions and improvements are also discussed here. The report is concluded with analysis and discussions of tests performed.

2 Background

2.1 The Skill Server

The Skill Server accesses the knowledge base and reasons about the objects and concepts stored there. The results are used to reason and make decisions about reconfiguration of a given automated manufacturing process. The knowledge base consists of two main parts; the ontology and the device libraries [4].

2.1.1 Ontology

The ontology is the vocabulary used to reason about and describe relationships between the objects and concepts that constitute the knowledge base. There are three main hierarchies in the ontology; devices, properties and skills. The ontology is described in the Web Ontology Language (OWL) [5],[6],[9].

Devices

Devices are objects that potentially have properties and skills; examples include drills, ethernet interfaces, sensors and so on. Much of the work of the knowledge server will be to verify the consistency between the devices in the device libraries provided by manufacturers, and the ontology. I.e. we need to make sure that all devices that go into the knowledge base are valid and otherwise consistent with what is defined in the ontology. Should this not be the case, the knowledge server needs to handle the problem in a graceful way and possibly alert the manufacturer that supplied the device that the information provided is insufficient or erroneous.

Properties

Properties are typed and valued entities that define the devices and skills which are part of the knowledge base. Typical examples are accuracy, number of connections, resolution and other properties describing the capabilities of the device or skill, but properties also include concepts such as cost.

Skills

Skills are descriptions of actions that can be performed by a device. A skill may or may not consist of several (simpler) skills; for example, the skill Drill is an aggregation of the skills *Approach*, *StartRotation*, *MoveLinear(down)*, *MoveLinear(up)* and *StopRotation*.

2.1.2 Device Libraries

Device libraries are collections of device descriptions. They are definitions of products supplied by manufacturers and together with the ontology they form the knowledge base used by the skill server.

Figure 2.1 shows an excerpt from an example device library. The first line defines a device *ABB_IRB-140* that is a of type *ArticulatedRobot*. The following lines each describes a property of the device.

If some property that the ontology specifies as required for the *ArticulatedRobot* type is missing here, the verification of the device will fail unless there is a suitable default value for this property in the current scope (*ArticulatedRobot*).

```
ABB_IRB-140 | ArticulatedRobot

ABB_IRB-140 | Class | ArticulatedRobot
ABB_IRB-140 | DegreesOfFreedom | 6
ABB_IRB-140 | EnclosureRatingIP | 67
ABB_IRB-140 | IntelligentCtrl | yes
ABB_IRB-140 | MaxAmbientTemperature | 45.0
ABB_IRB-140 | MaxVoltageSupply | 600.0
ABB_IRB-140 | MinAmbientTemperature | 5.0
ABB_IRB-140 | MinVoltageSupply | 200.0
ABB_IRB-140 | NumberOfJoints | 6
ABB_IRB-140 | Payload | 5.0
ABB_IRB-140 | PowerConsumption | 4.5
ABB_IRB-140 | Reachability | 810
ABB_IRB-140 | Repeatability | 0.03
ABB_IRB-140 | TypeOfActuation | Electric
ABB_IRB-140 | Weight | 460.0
```

Figure 2.1 Device library excerpt

2.1.3 Skill Server Architecture

Figure 2.2 depicts how the skill server fits in to the old architecture as a whole. The database device library in the bottom left corner, and the OWL ontology above it are the parts that (though in different form) will make up the knowledge server software.

How the skill server should access this new (sub-)system will be decided during the course of the design.

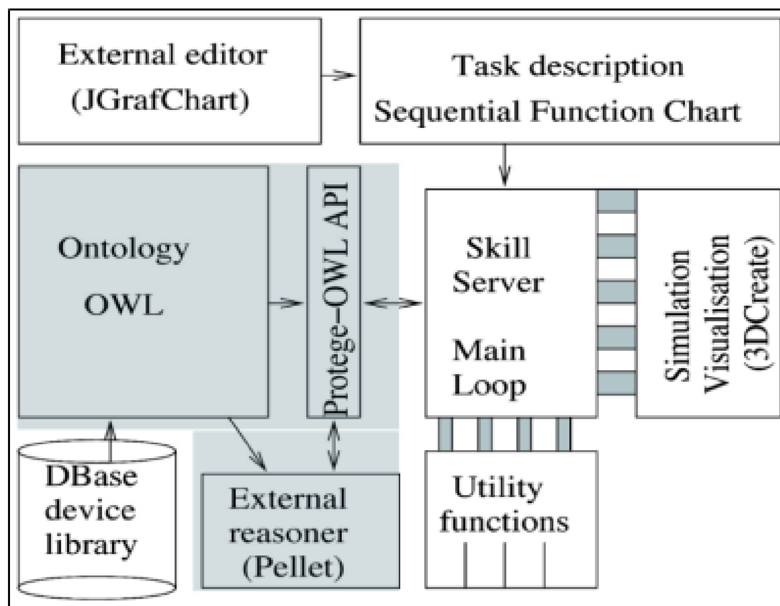

Figure 2.2 Current skill server system architecture

2.2 Ontology

2.2.1 Introduction

An *ontology* is used to represent objects and their relationships within a certain domain. It defines a *vocabulary* for reasoning about the objects in the domain, and formalizes how these concepts relate to each other.

In the ontology used by the skill server system we define concepts such as device types, property types, skills and their relationships. We also define restrictions on how devices may be configured. A certain *Drill* type for example, may require a property *Speed* for the skill server to be able to reason about it. Thus we require that such a property MUST be part of a *Drill* device description. Figure 2.3 shows an excerpt from the ontology:

```
<owl:Class rdf:about="#Displace">
  <rdfs:subClassOf rdf:resource="#Move"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#Orient"/>
  </owl:disjointWith>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="#Robot"/>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#isSkillOf"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <owl:disjointWith>
    <owl:Class rdf:about="#Convey"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#Pan"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#Position"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Feed"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Pass"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#Arrange"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#Turn"/>
  </owl:disjointWith>
</owl:Class>
```

Figure 2.3 Excerpt from the OWL ontology

2.2.2 OWL

OWL, the Web Ontology Language, is based on RDF (*Resource Description Frmework*) [7],[8] and standardizes the meaning of a set of XML elements/attributes that can be used for expressing the relationships of concepts within an ontology.

Consider the following example:

```
<owl:Class rdf:ID="Wine">
  <rdfs:subClassOf rdf:resource="&food;PotableLiquid" />
  ...
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasMaker" />
      <owl:allValuesFrom rdf:resource="#Winery" />
    </owl:Restriction>
  </rdfs:subClassOf>
  ...
</owl:Class>
```

Figure 2.4 OWL example

In figure 2.4 we see a definition of the type *Wine*, which states – among other things - that it is a subclass of *PotableLiquid*, i.e. that wine *is a* type of potable liquid.

However, the lines following this statement are more interesting; we have an OWL *Restriction* on one of the properties of *Wine*. The restriction is **on** the **property** *hasMaker*, and requires that all such properties be of the type *Winery*. In other words; if a wine has a maker, that maker must be a winery.

By applying restrictions like this on the contents of our ontology we define how the concepts in our domain relate to each other, and we get a well-defined vocabulary that can be used for reasoning.

2.2.3 Protégé

Protégé is an open-source tool developed at Stanford University for manipulating ontologies. Essentially it is an ontology editor written in Java. It is a fairly widespread tool with a registered user base of 100.000+ users. Figure 2.5 shows a screenshot of the application.

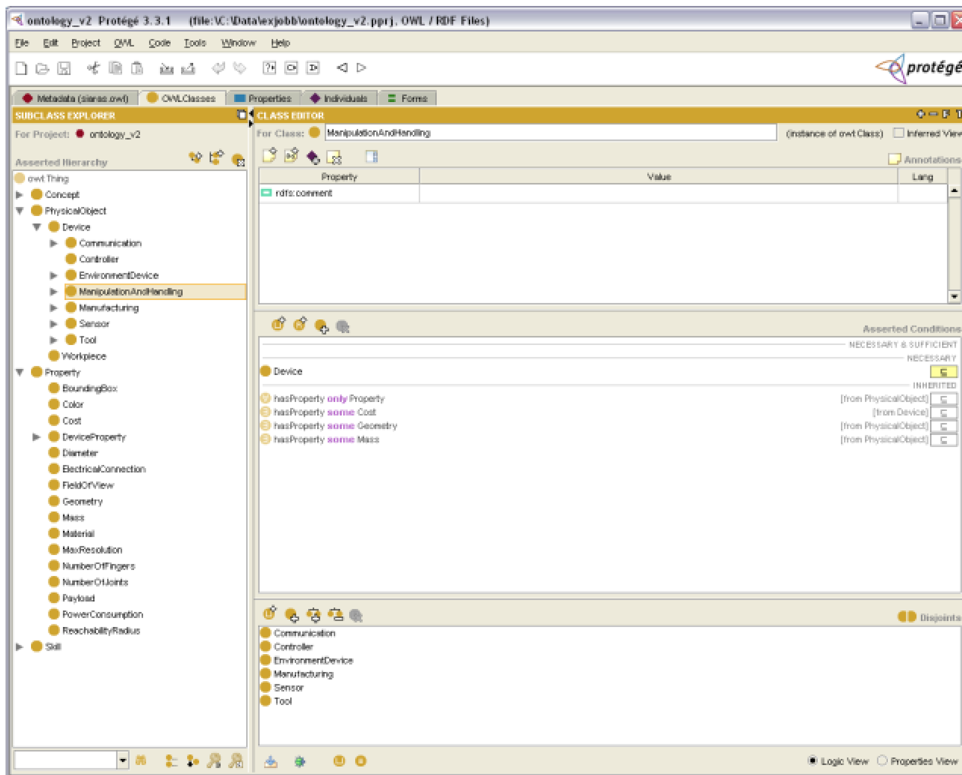

Figure 2.5 The Protégé tool

In SIARAS, Protégé is used for maintaining the ontology (OWL) file. Protégé has an API in Java that can be used for accessing and manipulating ontologies from any Java program. This API is powerful and will be used in the new architecture to extract necessary information to be used for building an independent (internal) representation of the knowledge when needed.

As the ontology is not expected to change very frequently it is only occasionally necessary to compile this information from the ontology. For the day-to-day running of the system, access to the ontology is not necessary, but the knowledge about the ontology can be refreshed on request if needed.

2.3 Library Maintenance Tool

The device library is currently maintained using a tool developed at Lund University, Dept. of Computer Science. It is written in Python and allows for inspecting current devices, modifying them, as well as adding new ones. A screenshot of the tool in use is shown in figure 2.6.

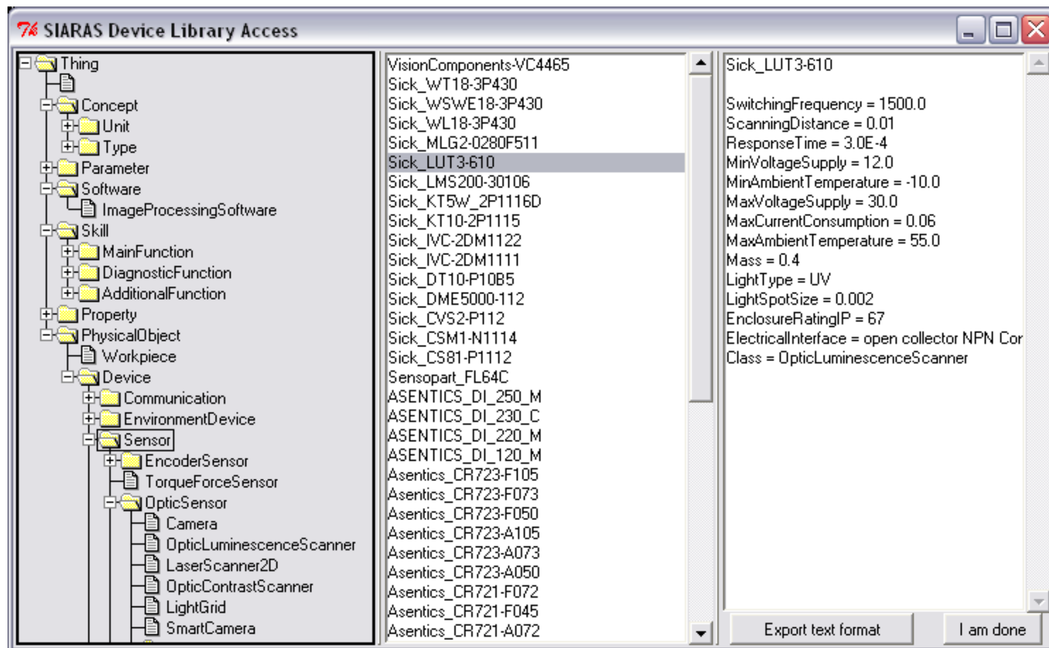

Figure 2.6 SIARAS Device Library Access

3 Design

3.1 System Requirements

Drawing from what we have discussed so far, a number of high-level requirements for the system can be identified. In particular, the system should do the following;

- Store the knowledge required by the skill server
- Enable distributed storage of the device libraries
- Verify that the devices provided by manufacturers or otherwise comply to the vocabulary specified by the ontology
- Handle any erroneous devices in a way that does not cause the system to fail
- Manage default values that may replace missing data in device libraries
- Provide an interface for access by the skill server
- Allow for device libraries to be stored in different formats
- Allow for access to the device libraries by different protocols
- Allow for propagating ontology changes (re-enforce consistency etc when the ontology is modified).
- Have a modular and easily extensible design, to allow for additional functionality required by new device library suppliers.

3.2 Introducing the Knowledge Server

The design of the knowledge server subsystem begins with identifying the major modules each fulfilling some of the requirements that were discussed in the previous section.

As we want to promote extensibility and have a modular design, it is important that the responsibilities of each module are properly defined. Lack of a thought-through design easily translates into code entanglement and nightmare refactoring work if updating or extending the system becomes necessary.

Furthermore, since this is a prototype software it will probably even be *necessary* to extend it in order to adapt it to actual systems. Therefore well-defined interfaces between modules are of great importance. Figure 3.1 shows the information flow between the modules.

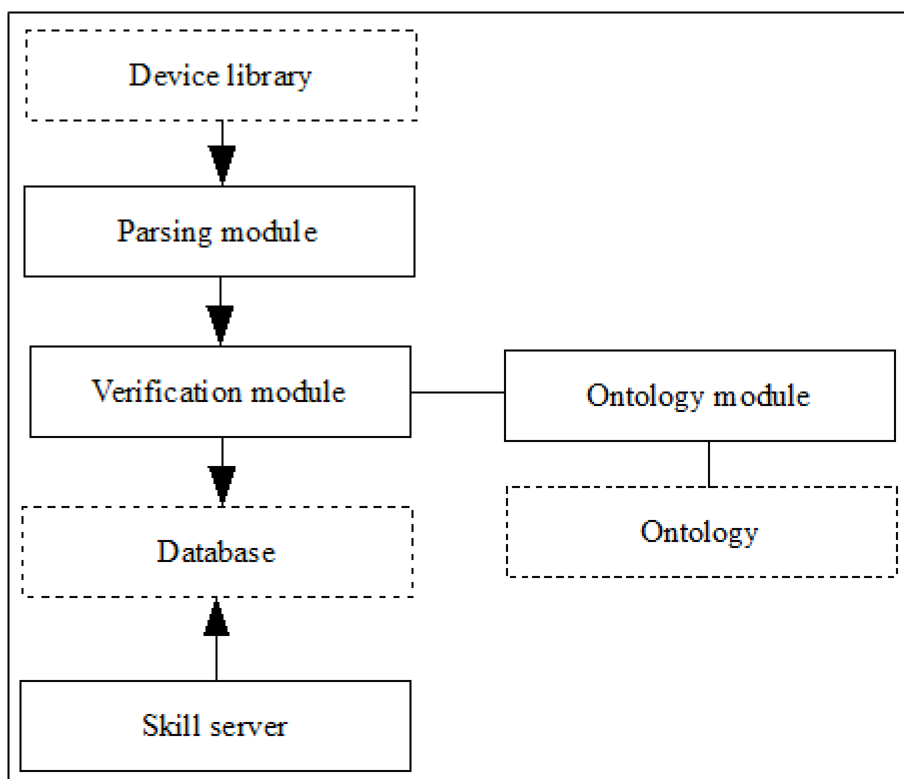

Figure 3.1 Knowledge server information flow

3.2.1 The Ontology Module

The knowledge server has access to the ontology/vocabulary and compiles data from it that allows for verification of the device libraries.

As the ontology is not expected to change very often this is solved by generating the required information from the ontology and saving it for use in the verification step until the ontology changes and the information has to be re-compiled.

So, the module's main responsibility is:

- Extract necessary data from the ontology and save it for future use by the verification module.

3.2.2 The Verification Module

Device manufacturers will supply new or edit existing libraries to reflect current and coming products. Device libraries could be stored locally or remotely on web servers, database servers or simply in plain text files.

When libraries are loaded by the knowledge server they have to be verified against the ontology to ensure they are consistent with the definitions of device types that are available.

For example, a certain device type may require that a number of properties are specified for it to be possible to use.

Using the information compiled by the ontology module the verification module of the software will perform these tests and, if needed, handle any errors that occur. Such errors may be device descriptions missing required information, and where possible the verification module should fill these in by using values from the defaults database. So for the verification module the responsibilities are:

- As device libraries are added or modified, verify the consistency of all device descriptions and if needed (if the descriptions does not comply with what is specified in the ontology), use the defaults database to fill in (if possible) missing required information or signal error, omitting the device.

3.2.3 The Parsing Module

As libraries can be stored in different formats we need a module that can interpret these and transform them into sets of data that are usable by the system.

To promote extensibility new parsers should be easy to add to the system, allowing for new formats to be handled without too much work. The parsing module needs to identify formats of device libraries and select the appropriate parser for doing the transformation into usable data to be verified by the verification module. I.e:

- Take device descriptions and parse the information, passing it to the verification module for consistency checks, ultimately adding it to the device database.

3.2.4 Database

While we need to store actual device data we also need to save information about locations of device libraries, what types of parsers need to be used to access them and so on.

Though there are a number of ways we could store the data, for this prototype system a centralized database will be used. As a possible extension, one could imagine a truly distributed database; i.e. not only aggregating information from distributed sources but in fact having the database itself being distributed. We get the following responsibilities for the database:

- Store device descriptions
- Store information about location and type of device libraries
- Store defaults data to be used for filling in (where possible) missing data in device descriptions.

3.2.5 Administration Interface

To be able to add new libraries, handle errors during parsing or verification, or just to inspect results output by the verification module, there should be an administration interface.

In particular this is necessary for entering rules regarding what should happen if the verification of new devices fails at some point. For example, in the case of a device missing one or several required properties, there should be an interface for entering default values for those properties.

If a certain device type is required to have a property ”color” and for a device being verified against the ontology, that property is missing, it should be possible to enter that any device this type should have the default color ”blue”.

It would be a fairly easy task to extend the set of possible rules used here; and since the interface will simply reflect a rules table, the editing of new rules will be similarly simple (editing the table).

3.3 Distributing the Device Libraries

One of the principal goals of the system is to allow for distributed device libraries. The idea is that each manufacturer or provider of devices should be able to ”publish” or submit new data to be used by the system, without the administrators of it having to do much extra work.

Even though one could impose format standards for this information, the providers may already have lots of data in their own format. Some device libraries may be presented in plain text in some custom format, some in XML, some as MySQL databases and so on. Also, accessing the published or submitted information may be done in different ways. Locally, from a web server or from a remote SQL database are some examples. The system thus should be able to handle both different formats and different means of access.

Making device library parsers that are pluggable or at least replaceable without much effort is therefore a good idea, as is the possibility to connect to information sources in a variety of ways.

3.4 Database and Server Software

The knowledge server consists of two main parts; the data storage – a database – and the actual server software that handles fetching, parsing, verification and error handling of the device libraries; as well as providing an interface for inspection of these activities and administration of error handling procedures.

The database stores verified devices and their properties, device library locations and access methods, and rules regarding how verification errors should be handled. Though not

implemented in the prototype software, it should also store settings controlling automated updating of the device libraries and basic settings such as credentials for accessing the database.

4 Implementation

Being cross-platform, Java is one possible choice for implementing the system as it can then be deployed in ”any” environment, and it is the one we choose here. Also, interfacing with databases through ODBC is very simple, as is writing a suitable user interface with Java Swing. The following subsections outline the realization of the design and discusses the implementation choices made.

4.1 A Design for Promoting Extendability and Modularity

Key to a system that should be easily extendable is that it is modular and that is uses a thought-through design that is appropriate for the domain in which it lives.
Many learn the hard way that an application that initially is pretty small and manageable even though it lacks a real design, can quickly become a nightmare to maintain as the code base grows.

4.1.1 The Model-View-Controller Design Pattern

Though being fairly old (first presented in 1979 [10]), the Model-View-Controller (MVC) [10] design pattern has been gaining a lot in popularity also in recent years. As ”small” Java applications in for example mobile phones are growing in complexity and size, the need for proper design has increased, driving interest in architectural patterns such as MVC.

The idea behind Model-View-Controller (figure 4.1) is that data and domain logic (*Model*) should be totally separated from the presentation layer (*View*). This is done by using a *Controller*. By keeping domain logic separate from the UI code each part can be easily replaced without affecting the other.

While the model is concerned with the data and domain logic and the view with presenting it, the controller handles user input, mouse events, and propagates the events, typically to the concerned models.

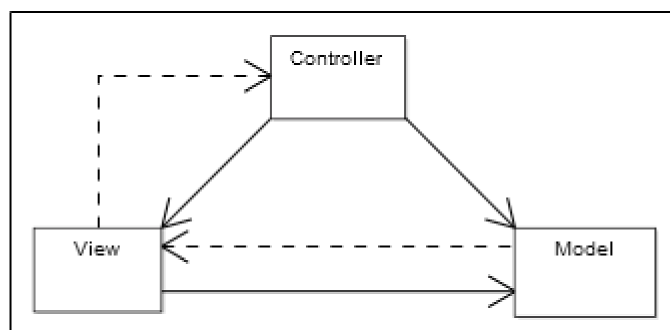

Figure 4.1 Model-View-Controller visualization (image from Wikipedia [10])

A typical chain of events in an MVC applications may be the following:

- The user enters a value and presses a key in the UI.
- The controller receives the event and notifies the models that something happened that may or may not be of importance to them.
- Each receiving model checks if the event was something that is should react to, and possibly updates its state to reflect the user action.
- The model notifies all of its *ModelListeners* that it has changed its state.
- The ModelListeners, typically views, re-render themselves based on the current state of the model.

The major point here is that the model has no idea of who listens to it, or what its data is being used for, it has no direct knowledge at all of the views or other listeners (other than that there exist listeners that it should notify).

It is easy to see that this pattern has a lot of potential when it comes to promoting a sound, modularized design with low entanglement. It also enables developers to work concurrently on each part as long as the interfaces in between the models, views and controllers are agreed upon.

Those familiar with Java Swing will recognize the ”Model-View” concept as being used for implementing the data flow to UI components such as JTable or JList. Plugging your own data model into a JTable is as easy as subclassing *AbstractTableModel* and overriding the methods used by the JTable to query for data.

4.1.2 MVC in the Knowledge Server

There is no predefined ”ultimate” way to apply the model-view-controller pattern to a system design. Obviously, a design should exist to simplify maintaining and understanding the code as well as promoting stability and robustness. Different design patterns are useful for different architectures.

In this design model-view-controller has been applied in those parts where it makes the most sense, such as in the user interface which consists of several ”views” and which benefits a lot from such a design.

By using model-view-controller here, adding a new view for monitoring or modifying new data is made very straightforward as there is no connection between the graphical representation and the data itself.

Extending the application can be done with little effort compared to a design where data and the visualization of it is mixed together in the classes. The data should never ”know how to render itself”, it should only notify any listeners that it has changed, and based on this a listener may choose to re-render the data.

Java Swing already has some of this type of architecture built in, in the JTable and JList classes. These classes are used for displaying most of the data in the application and so we subclass them and let our models implement the methods required by the ListModel interface. By using these classes and interfaces we get some of the model-view-controller pattern ”for free”.

The user interface connects to the actual server through a socket. When the database changes, a message is sent to the UI with the new device information. This represents an update to the model and triggers sending an event that informs listeners that the device set has changed.

4.2 Separating Database, Server and Interface

The system is divided into three main parts; a data storage, the server maintaining the data, and the interface through which the administrator can add and modify default property values, check verification results, reload device libraries etc.

4.2.1 The Knowledge Database

For storing the necessary data there are a number of viable alternatives, both regarding format and location of the stored data.

For this system, a ”traditional” SQL database is used; MySQL. It is free, well documented, easy to work with and interface with from Java. The benefits of using a standard database include not having to develop a custom format for storing the data, not having any concerns regarding scalability and performance, and – most important for this system – having a well-defined interface for accessing the data: SQL.

Using MySQL for storage means anyone developing the skill server need only be familiar with SQL to interface with the knowledge server.

The drawback of using this solution is obvious: we get a centralized database which needs to be updated regularly with data from device library ”sources”. One could certainly imagine having a database that is in itself distributed; essentially creating the knowledge server as a front-end (handling direct queries) for such a system. This however, is left as a possible extension or development of the system.

Tables

The different tables needed for storing the necessary data are pretty simple in their construction.

Devices

id	name	type	location
146	CLOOS_R350	ArticulatedRobot	c:database.txt
145	CLOOS_R320	ArticulatedRobot	c:database.txt
144	Bosch_GBM_10_RE	DrillMachine	c:database.txt
143	ABB_IRC5	Controller	c:database.txt
142	ABB_IRB-4400	ArticulatedRobot	c:database.txt
141	ABB_IRB-2400	ArticulatedRobot	c:database.txt
140	ABB_IRB-140	ArticulatedRobot	c:database.txt

Figure 4.2 Excerpt from devices table

The devices table (figure 4.2) consists of an id, the device name, the type and the location where it can be found.

Properties

id	owner	type	value
2095	CLOOS_R350	Color	0
2094	CLOOS_R350	Cost	0
2093	CLOOS_R350	Mass	205
2092	CLOOS_R350	TypeOfActuation	Electric
2091	CLOOS_R350	Repeatability	0.1
2090	CLOOS_R350	Reachability	2215
2089	CLOOS_R350	PowerConsumption	3
2087	CLOOS_R350	NumberOfJoints	6
2088	CLOOS_R350	Payload	15
2086	CLOOS_R350	MinVoltageSupply	350
2085	CLOOS_R350	MinAmbientTemperature	-15.0
2084	CLOOS_R350	MaxVoltageSupply	450
2083	CLOOS_R350	MaxAmbientTemperature	45.0
2082	CLOOS_R350	IntelligentCtrl	yes
2081	CLOOS_R350	EnclosureRatingIP	30
2080	CLOOS_R350	DegreesOfFreedom	6
2079	CLOOS_R320	Geometry	0
2078	CLOOS_R320	Color	0
2076	CLOOS_R320	Mass	205
2077	CLOOS_R320	Cost	0
2075	CLOOS_R320	TypeOfActuation	Electric

Figure 4.3 Excerpt from properties table

The properties table (figure 4.3) connects properties and their values to devices.

Libraries

location	parser
c:\database.txt	PlainTextFormat1
testlocation	MySQLFormat1

Figure 4.4 Excerpt from libraries table

Simplest of the tables is the library listing (figure 4.4), which keeps information on all current libraries. When the database is updated all of these locations are queried for their device libraries.

This table also holds information on which parser to use for interpreting the contents of the library.

Defaults

id	property	scope	value	newValue	reason
1	Cost	ArticulatedRobot	<missing>	1.0	required
3	Geometry	Controller	<missing>	0	Required
4	Mass	Controller	<missing>	10	Required

Figure 4.5 Excerpt from defaults table

When consistency checks of devices fail, the defaults data (figure 4.5) is queried for possible fallback values. A check is made if there is a default value for a certain property in the current scope; for example if there is a default *Mass* for *Controllers* available.

The filed value is intended to be used for differentiating between reasons for the failure. That is, one value could be used when the property is missing altogether, and another one for when the property is available but the value cannot be read.

4.2.2 The Knowledge Server

The heart of the system is the server itself. Although not actually serving any data it maintains and updates the device library information and verifies all the incoming data. Basically, it fetches data from the device library sources, verifies that it is consistent with the ontology and inserts it into the database. If errors occur it tries to resolve them or else discards the data. The process is logged and can be inspected by an administrator of the system.

Fetching

When the device database is to be refreshed, the database is queried for all currently registered libraries. Specifically, their locations and associated parsers are retrieved. The device libraries are fetched and the data is passed to the appropriate parser.

Parsing

Each parser implements the same interface, an interface that declares methods for extracting devices and properties from the libraries.

```
public interface DeviceLibraryParser {

  /*
   * Get device object populated with properties.
   */
  public Device[] getDevices(String location) throws DeviceLoadException;

}
```

Figure 4.6 Interface DeviceLibraryParser

This way, it is easy to write a new parser for each new device library format that is encountered. As long as the parser implements the *DeviceLibraryParser* interface (figure 4.6), it will be dynamically loaded when a library associated with it is retrieved.

As the libraries are loaded the devices are extracted and passed to the verification module for consistency checks to determine if they are to be inserted into the database or rejected.

The interface specification requires the implementing class to return an array of Device objects, each populated with Property objects and initialized with information about from which device library the device originates.

Verifying

Verifying a device means checking that it is a valid device type, and that its properties are those required by the ontology. Should one or several properties that are essential to the functionality of the device be missing, the device is discarded unless or until there are appropriate default values for those properties defined in the scope of that device.

If there are no fallback values, the device will not be entered into the database, but will be marked as failed. The administrator will see this in the interface and may enter such values and reload the library which should now be verified successfully.

4.2.3 The Administration Interface

For controlling the knowledge server, there's a user interface written in Java Swing. As previously discussed, one of the goals of the system is for it to be modular with low entanglement. This has been implemented particularly well in how the UI interfaces with the server.

Connection

To ensure very loose coupling between the interface, i.e. the graphical representation of the server state, and the server, they communicate through a socket connection. That is, the user interface is running as a separate process and connects to a port owned by the server. For one thing, this prevents any entanglement of data and its visual representation; also making it possible to replace the administration interface very easily. An additional advantage is that the interface can be running on a computer separate from the server software. The interface could even connect to the server over the Internet.

User Interface

The user interface is divided into five views, each presenting the user with controls for administrating or inspecting different features of the knowledge server.

Overview

The status overview screen (figure 4.7) displays information about connection status and some miscellaneous data such as the number of device libraries currently stored/monitored by the system.

There's also logging output from both the interface and the server displayed for monitoring the state of the system and facilitate debugging.

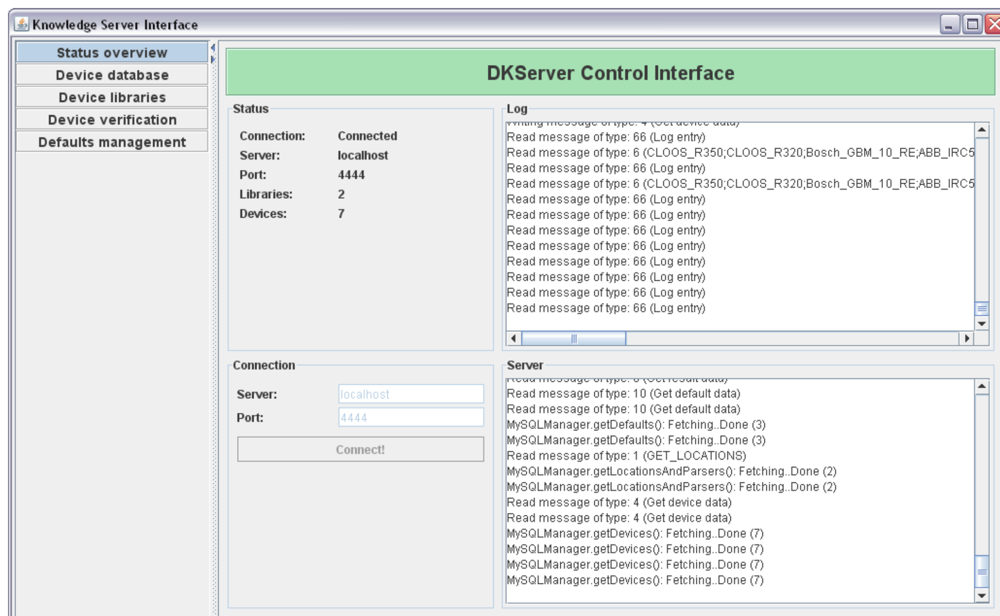

Figure 4.7 Screenshot of overview screen

Device Database

The device database screen (figure 4.8) shows all devices currently stored in the database in the left window. Selecting any one of them will cause its properties and their values to be listed in the right window.

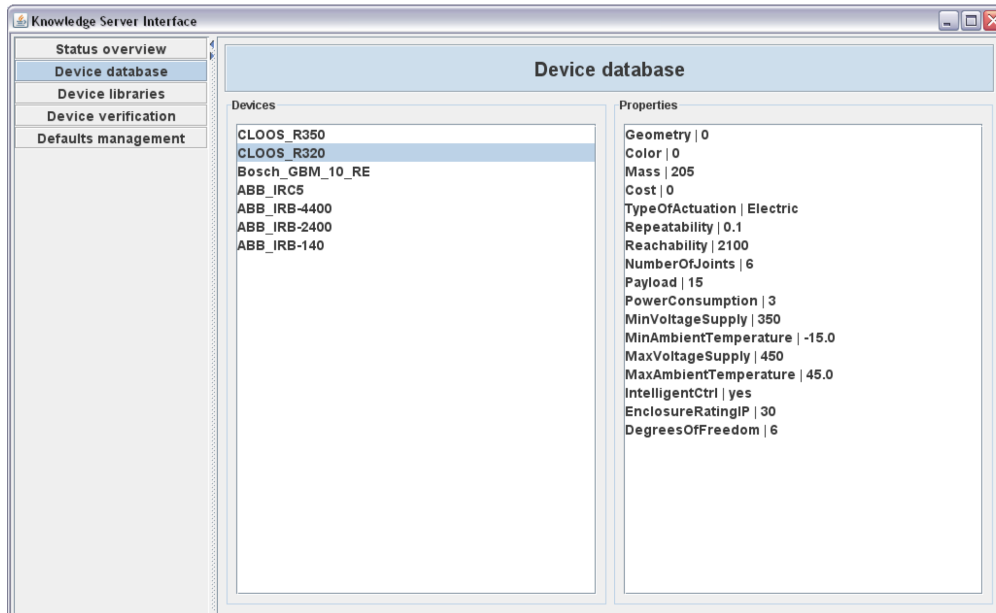

Figure 4.8 Screenshot of device database screen

Device Libraries

The device libraries view (figure 4.9) allows for adding new device library locations and associating them with appropriate parsers. Those parsers will then be used when fetching and updating the database with data from the libraries.

It is also possible to inspect which libraries' content is being monitored and stored in the database. There is also controls for reloading all the data in the database; i.e. reloading and verifying all data from the device libraries. Controls should ideally also be added for reloading single libraries.

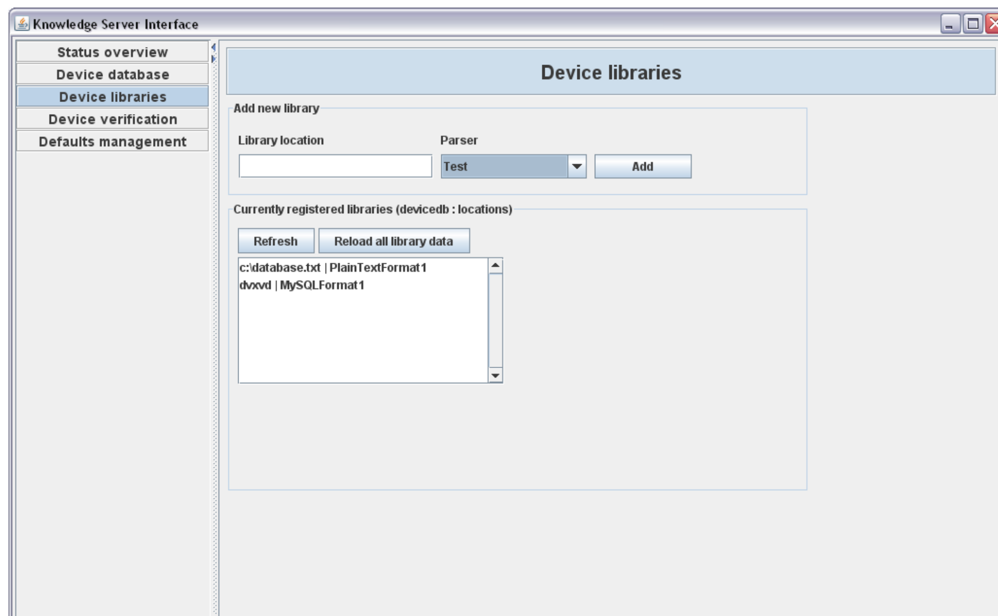

Figure 4.9 Screenshot of device libraries screen

Device Verification

When device libraries are reloaded into the database, the status of each device as it is processed in the verification step is saved and can be viewed in the device verification screen (figure 4.10).

Devices which are cleared for insertion into the database are marked green, while those which fail the consistency checks are marked red. By double-clicking a red device, a window is opened where default values for (for example) missing properties can be entered.

In other words, when a device fails verification the administrator of the system can enter replacement values for those properties and reload the libraries. Now, if all problems have been addressed, the device will be cleared and marked green.

Because the default values are only used as fallbacks, if the device is updated by the manufacturer or administrator of the library, those new and real values will automatically be used instead of the defaults as soon as the libraries are reloaded (see next section).

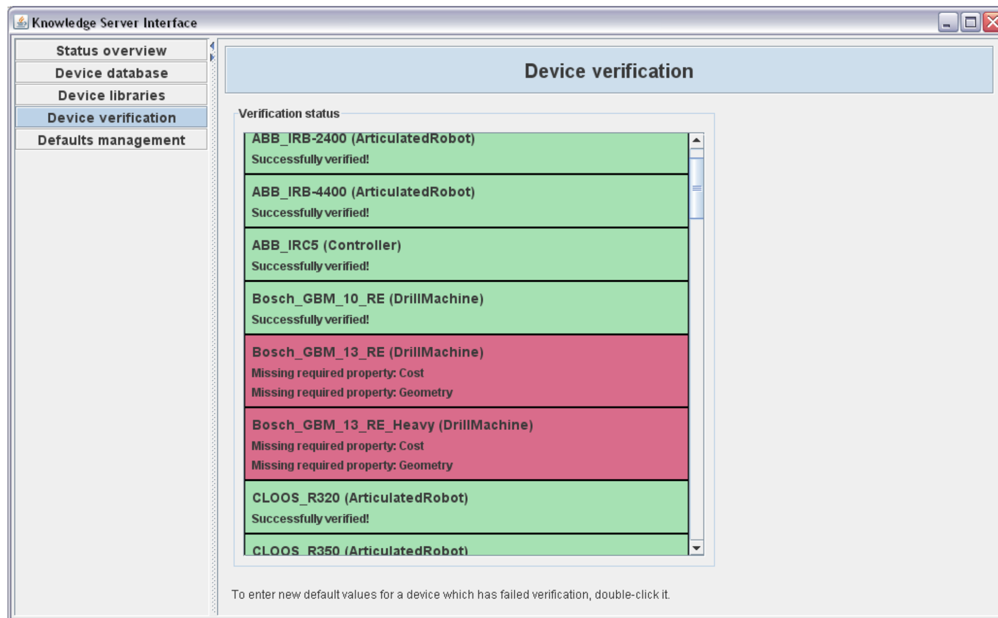

Figure 4.10 Screenshot of verification screen

Defaults Management

For handling cases where required properties are missing from objects being loaded from libraries, the software has a fallback mechanism based on default values that can be specified via the user interface of the server.

The default values are stored in the database in a separate table; and are maintained as shown in figure 4.11. The user can specify conditions for when the default property value should set in; for example for which device class and for which reason (e.g. the property is required).

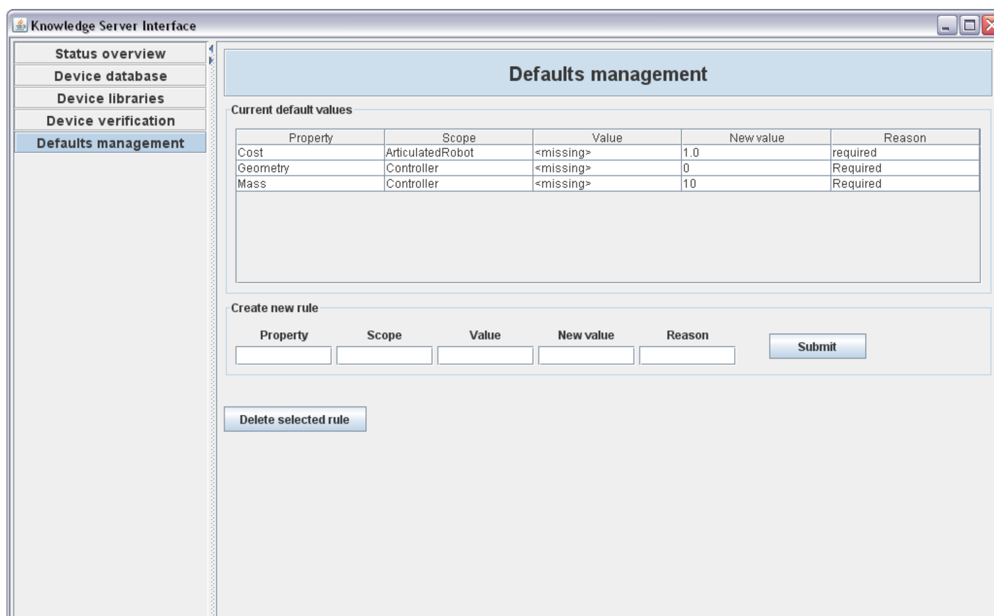

Figure 4.11 Screenshot of defaults management screen

4.3 Tests

For testing the application the most recent versions of the SIARAS textfile database of devices and ontology have been used. Many of the devices in the database do not fulfill the requirements as set by the ontology and consequently the fallback mechanism has been put to the test during testing of the system.

Tests have been limited to device libraries in plain text files, for which a parser has been written. Testing has shown that the verification and error handling steps of the server works as intended; with incomplete devices being rejected until addressed by appropriate default value rules.

4.4 Code

Source code is available at:

http://fileadmin.cs.lth.se/ai/xj/PetterLiden/KnowledgeServer.zip

5 Conclusions and Future Work

This project concerned separating devices (instances) from their descriptions and enabling having them distributed as several independent sources to simplify adding new ones for use by the Skill Server.

For realizing this a software, a *Knowledge Server* was created. Essentially it becomes the keeper of the information that is accessed by the SkS and aggregates device libraries from the different sources, enforcing the constraints set by the OWL ontology. The knowledge server serves the SkS by maintaining a coherent and valid database of the information that is required by the SkS.

By having multiple sources each separately editable from outside the system, and a well-defined (SQL) interface to the SkS, we get a system where information is both easy to update and access, in line with what was the objective of the thesis.

In this version of the system the information that is collected by the knowledge server is determined by a list of sources which can be modified by the administrator of the system. It would be easy to allow remote access to this list by the owners of device libraries, making the addition of new libraries involve only their creators and not those responsible for the knowledge server.

One could imagine new library sources being added to an inbox where the administrator approves them for inclusion into the system, or that they simply are included without further inspection. Any errors would trigger messages being sent to those responsible for the library, requiring them to modify the contents so that they are coherent with the vocabulary and can be accepted for use by the SkS.

This is only one extension of the current implementation that can be envisioned. This basic version of the system has been successful in modularizing the storage of device information; by writing a parser that implements a specified interface, any format (and location) can be used for storing the device libraries.

The system as a whole is easily extendable as adding support for new functionality only is a matter of specifying a new source where the corresponding device descriptions are stored and providing a parser that extracts information in the correct format.

During the project, thoughts on how the system could be extended and improved have emerged. For one thing, the database – which currently is centralized to the machine running the knowledge server – could be made truly distributed. Instead of collecting information from different sources and putting into one central place once verified, one would have a central data structure that, for example, only keeps track of where to look for certain types of devices and those locations are then searched on request (i.e. when the SkS requests that information) for complete descriptions.

There are other more practical improvements that can also be made. The protocol between user interface and server is almost entirely text-based and (as has been observed with a large database) becomes slow when lots of information has to be transferred between the two parts. For demonstrating the system design and its capability this is of secondary importance, but for a live system it's an issue that would need to be addressed.

6 Bibliography

[1] Angelsmark, O. Malec, J. Nilsson, K. Nowaczyk, S. (2006). *Skill-Based Inspection and Assembly for Reconfigurable Automation Systems – Framework for Task Representation*. Technical Report, Lund University, Dept of Computer Science.

[2] Angelsmark, O. Malec, J. Nilsson, K. Nowaczyk, S. Prosperi, L. (2006) *Knowledge Representation for Reconfigurable Automation Systems*. In Proc. 9th National Conference on Robotics, Piechowice, Poland.

[3] Malec, J. Angelsmark, O. Nowaczyk, S (2006). *Methodology for Skill Encoding*. Technical Report, Lund University, Dept of Computer Science.

[4] Malec, J. Nilsson, A. (2007). *Skill-Based Inspection and Assembly for Reconfigurable Automation Systems- Implementation of software for skill representation*. Technical Report, Lund University, Dept of Computer Science.

[5] Antoniou, G. Harmelen, F. *A Semantic Web Primer, 2nd Edition*. (2008). MIT Press.

[6] Costello, R. Jacobs, D. (2003). *A Quick Introduction to OWL – Web Ontology Language*. The MITRE Corporation.

[7] Champin, P-A (2001). *RDF Tutorial*. Université Claude Bernard Lyon 1, http://www710.univ-lyon1.fr/~champin//rdf-tutorial/rdf-tutorial.pdf. Verified May 10th 2009.

[8] Miller, E (1998). *An Introduction to the Resource Description Framework*. D-Lib Magazine, http://www.dlib.org/dlib/may98/miller/05miller.html. Verified May 10th 2009.

[9] W3C (MIT, ERCIM, Keio) (2004). *OWL Web Ontology Language Overview*, W3C, http://www.w3.org/TR/owl-features/. Verified May 10th 2009.

[10] Wikipedia (2009). *Model-View-Controller*. http://en.wikipedia.org/wiki/Model-view-controller. Verified May 10th 2009.