Integration of Semantic Knowledge to Enable Re-Use of Robot Programs

Maj Stenmark

Master's Thesis in Computer Science

Lund University, Sweden 2011

Abstract

To be able to compete globally, European production needs to increase its cost efficiency and have manufacturing lines that can be easily adapted to different products. Also, tasks that are mainly manual at present, such as the assembly of consumer electronics, should be automated. One solution is to develop robot systems that are able to carry out more advanced actions, while autonomously improving the task and being able to work safely side by side with human colleagues. To enable flexibility, the robots should be easily instructed.

To simplify the robot programming, instructions should be high level, programs and tasks should be re-usable and shared in a common online database and tasks should be self-improving, optimized according to the platform.

The large European research project ROSETTA aims to develop architecture for knowledge integration, learning and safety. The knowledge is stored in a database called the knowledge integration framework and accessed by simulation and control tools, here called the engineering system and the ROSETTA controller respectively. This Master's thesis project was carried out within the ROSETTA work groups developing the knowledge integration framework and the engineering system.

The master's thesis investigates how robot skills, that is, an implementation of a robot routine or action, can be shared and re-used. The following topics are addressed: the types of knowledge that should be stored in framework to enable knowledge sharing and re-use of skills and how this information can be communicated between the knowledge integration framework and the engineering tool.

The types of knowledge that are considered are 1) state machines describing the skills as a sequence of states where 2) each state has a set of parameter values that should be annotated and 3) procedures written in programming code that are stored in a re-usable way.

The result is a description of the structure for storing the information and the outline of tools for adding and retrieving the information. The concepts are concretized by demonstration tools.

Sammanfattning

För att möta den ökande konkurrensen inom tillverkningsindustrin behöver europeisk produktion öka kostnadseffektiviteten, snabbheten och vara mer anpassad till en ökad grad av variation bland produkterna. Områden som utförs manuellt i dagsläget, som montering av konsumentelektronik, bör automatiseras.

Ett sätt att lösa problemet är att automatisera produktionen med flexibla och användarvänliga robotsystem som gör det möjligt att snabbt växla mellan tillverkningsmetod och som självständigt optimerar uppgiften för varje plattform. Robotarna ska tryggt kunna arbeta sida vid sida med människor och vara lätta att instruera.

För att förenkla programmeringen av robotar bör instruktionerna fokusera på uppgiften istället för detaljreglering. Uppgiftsbeskrivningar och data ska lagras i ett gemensamt kunskapsbibliotek.

ROSETTA är ett europeiskt forskningssamarbete som utvecklar en arkitektur för att integrera kunskap samt utvecklar lärande och säkerhetssystem. Examensarbetet utfördes inom arbetspaketen som arbetar med att integrera kunskap och utveckla ingenjörserktyg.

Examensarbetet undersöker hur implementationer av robotrutiner, så kallade *skills*, kan lagras och återanvändas i en kunspasdatabas. Närmare bestämt: vilken kunskap som ska integreras, hur denna ska lagras på ett återanvändbart sätt och hur kommunikationen mellan datasen och ingenjörsverktyget kan se ut.

Tre typer av kunskap berörs, 1) tillståndsmaskiner som beskriver en uppgift och där 2) varje tillstånd har en uppsättning parametrar som ska annoteras samt 3) hur robotrutiner skrivna i programkod ska sparas på ett återanvändbart sätt.

Resultatet är en beskrivning av strukturen för hur denna information ska sparas i databasen och designförslag på verktyg för att hämta och lägga till information. Koncepten konkretiseras med demonstrationsverktyg.

Acknowledgments

The research in the ROSETTA project receives funding from the European Community's Seventh Framework Programme[1].

I am grateful for all the help and support I have received from the members of the ROSETTA work packages 1 and 6 at Lund University and ABB Corporate Research, without whom the project would not have been possible.

To mention a few: Pierre Nugues, my supervisor at the department of computer science at Lund University and leader of Work package 1, who introduced me to the project in the first place and who guided me through the shadows of confusion to the twilight zone. Mikael Hedelind, my supervisor at ABB Corporate Research, who with angelic patience and humor endured my questions and lack of formality. Klas Nilsson, my examiner, let the judgment fall. Jacek Malec, who took over the torch as team leader after Pierre and always kindly reminded me to send him the reports I had neglected. Anders Björkelund, who, like an oracle, knew the solution to any programming problem.

Special thanks to Max, who gave brutally honest but constructive criticism on my report, I will pay you back big time, brother.

Finally, I want to thank my fellow thesis workers, both in Lund and at ABB, for the friendship and fun. I wish you all good luck.

[1]P7/2007-2013 – Challenge 2 – Cognitive Systems, Interaction, Robotics – under grant agreement No 230902 - ROSETTA.

Preface

At 19, I wanted the same thing from life as any other teenager, that is, to play with expensive toys while someone else pays the piper. The details were a bit fuzzy though and while sorting them out, I decided to chill out studying engineering physics, so that I, before going anywhere in the world, would know how it ticks. Following the path of curiosity, I spent a year taking computer science classes in sunny California that resulted in a specialization in software.

When the time came for the master's thesis, I run into the ROSETTA project, which fulfilled my requirements splendidly: it did fun stuff with expensive toys. That is, creating a collective consciousness for robots, with shared memory and reasoning, taking us one step closer a dystopian society where the machines have conquered the world, turning it into a robot anthill governed by a collective brain. Sweet.

This report is the product of that work. The target audience is other students that are familiar with the basics of computer science. My goal was to avoid technical detail and to explain everything in a clear and enjoyable manner.

Maj Stenmark, Lund, April 18 2011

Contents

Chapter 1

Introduction

A spectre is haunting Europe – the spectre of automation. European industry is threatened, production is moved to low-wage countries and unless something is done, European manufacturing will be diminished.

In addition to the global competition, future product lines will be more diverse, products will have a shorter lifespan and come in various versions. This demands more cost efficiency and higher levels of flexibility in the automated manufacturing processes.

This challenge is met by a new generation of industrial robots, that will assist human workers by automating parts of the production. The objective is to develop flexible robot systems that can be easily instructed to handle new tasks and autonomously improve its performance by learning. Also, the robots should be safe around human colleagues, to enable human-robot collaboration.

The goal is also to simplify and speed up the robot programming, by using high level instructions. Knowledge is to be stored in a common knowledge framework and accumulated when working robots upload learned configurations and human programmers add task descriptions to a common repository.

The large European research project ROSETTA was launched in 2009, aiming at developing a framework for knowledge integration and safe autonomous robot systems. The objectives of the thesis project is to 1) describe what information regarding implementations of robot routines to store in the framework, 2) define how to store the data in a generic and re-usable way, and 3) implement tools to demonstrate the concepts.

1.1 The ROSETTA Project

Can't we talk to the humans and work together, now?

– from "Robots" by Flight of the Conchords

ROSETTA is the not entirely obvious acronym for ***RO**bot control for **S**killed **E**xecu**T**ion of **T**asks in natural interaction with humans; based on **A**utonomy, cumulative knowledge and learning*. It is a 4-year large-scale European research project that started in March 2009. The project has a budget of 10 million Euro and is funded by the European Community's Seventh Framework Programme (ROSETTA Project information, 2011). The project is carried out in co-operation between the following companies and universities:

- ABB AB (Sweden) – Coordinator
- ABB AG (Germany)
- Dynamore GmbH (Germany)
- Fraunhofer IPA (Germany)
- K.U. Leuven (Belgium)
- Ludwig-Maximilians-Universität Munich (Germany)
- Lunds Universitet (Sweden)
- Politecnico di Milano (Italy)

that together form the project consortium (ROSETTA Consortium, 2011).

1.1.1 Background

The project is based on the belief that future factories will produce high volumes of goods, but in many variants with short lifespan. Robot systems can provide such an automated and flexible production line. However, to allow maximum flexibility, robots and humans should be able to work side-by-side in a safe environment, where the robots can perform standardized tasks while the human workers can do the more advanced (or nonautomated) labor, thus complementing each other (ROSETTA Project information, 2011).

1.1.2 Goals

The robots should be able to perform complex tasks with flexibility and robustness and it should be possible to change products fast, thus making the production efficient and cost effective.

The robots should appear human-like and interact with the human co-workers in a safe and intuitive manner, for example by using speech interaction, imitating human motion patterns and avoid situations that can injure humans beings. The latter should be done by sensor, control and decision making system to provide a safe physical environment for the human workers (ROSETTA Project presentation, 2010).

Programming the robot systems should be intuitive and related to the task and not require an experienced robot programmer. Once the robot is programmed, it should be able to improve the task autonomously. The parameters for the optimized task can then be shared with other robots via a central server (ROSETTA Project information, 2011). For the robot systems to be successful, the cost for the robots should match the price for labor in a low-wage country.

1.1.3 Expected Outcome

The new robot systems will make it possible to automate the manufacturing of new areas of production, especially where the products are frequently changed, for example consumer electronics. This should boost the competitiveness of the European industry by decreasing manufacturing cost and increasing production volume and quality.

Another outcome is the understanding of human-robot interaction in the industry, for example, a classification of injury risk makes it possible to develop new standards for robot safety in the industry (ROSETTA Project information, 2011).

1.1.4 Research Areas

The project is divided into four main research areas (ROSETTA Project presentation, 2010):

Intuitive robot instructions	How to instruct robots on task level rather than programming them; how to represent knowledge in a shared database and creating a simulation tool.
Learning	Autonomous systems that can optimize tasks and share information.
Robot control	Sensor integration and assembly operations.
Safety	Human-robot interaction, supervision of the work area and injury criterias.

Thus, this project does not develop the robots themselves, but use existing robot platforms for testing.

Each area develops different tools that are dependent on each other, see the overview in Figure 1.1. The knowledge integration framework, abbreviated KIF (see Section 1.2) is a server where the knowledge about devices, their capabilities and injury criteria are stored and reasoned upon. The ROSETTA controller is a platform-independent controller, that imports knowledge from the server and sends it to either the platform dependent robot controller or safety sensors (cameras etc). In turn, the robot controller communicates directly with the robot, the ROSETTA project uses the ABB concept robot FRIDA (FRIDA, 2011), seen in Figure 1.1. The engineering system is a tool for station configuration and simulation, hence it runs a virtual controller.

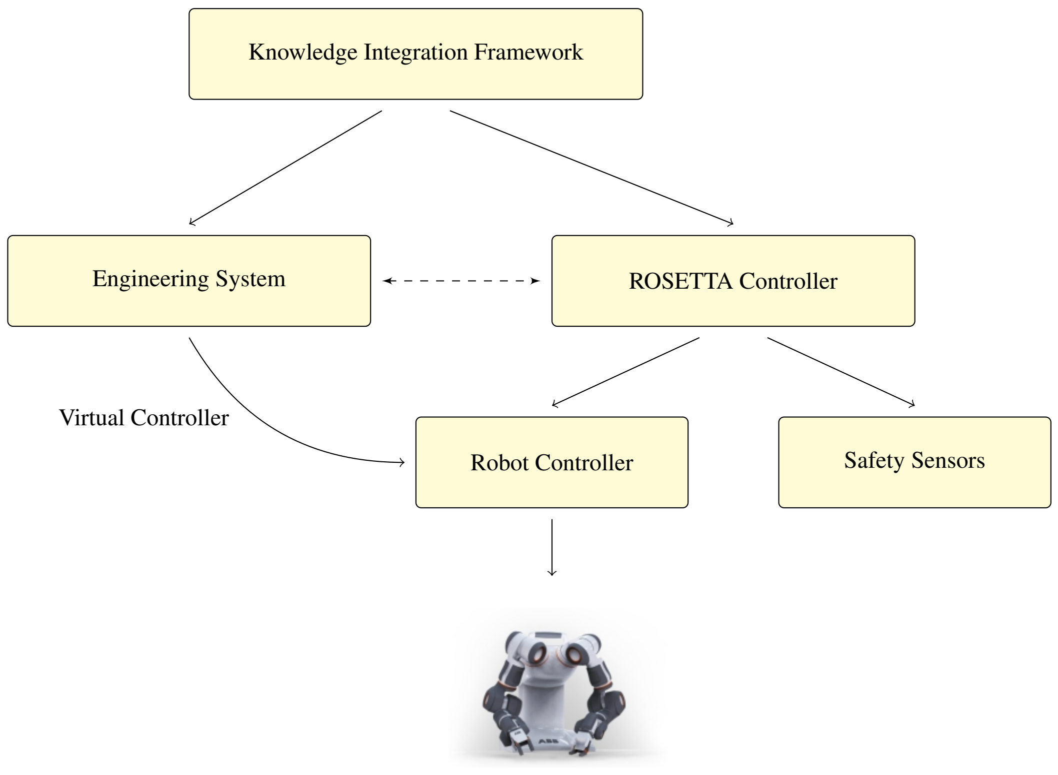


Figure 1.1: Overall architecture of the ROSETTA project. The robot photo is provided by ABB (FRIDA, 2011).

1.1.5 Related Work

There are several other ongoing large scale projects addressing the same issues of information sharing and re-use as the ROSETTA project. To mention two:

RoboEarth aims at letting robots perform complex tasks in a dynamic environment and share re-usable knowledge between robots. Their models are based on OWL (see Section 2.1.3) and the use of *action recipes* to describe tasks. The goal is to be a World Wide Web for robotics. (RoboEarth, 2011).

GeRT aims at making robots robust in novel environments by autonomously generalizing its manipulating skills to new objects. The projects focus is task planning for service robots (GeRT, 2011).

The outcome of these projects will be very interesting as will the exchange of ideas between them.

1.2 Knowledge Integration Framework

Those people who think they know everything are a great annoyance to those of us who do.

– Isaac Asimov

The KIF, is a database with information on *E.g.* CAD and device data, stored in a standardized machine readable format. In the present prototype of KIF, the data is stored in repositories as RDF triples, which means that knowledge is stored as a graph with objects as vertices and with edges representing the relationships between the objects, see Section 2.1. The information is extracted from the database using the query language SPARQL, see Section 2.2.

RDF is chosen because it is an open, standardized format that can describe objects platform independently, thus different robot models, even from different manufactures, share the same knowledge.

Robot and automation data can be expressed in the open, XML-based markup language AutomationML [1], which in turn can be transformed into RDF and stored in a repository, that can be queried.

Figure 1.2 displays the KIF architecture schematically, and is slightly adapted from (Persson, et al. 2010). The server, which is an Apache Tomcat servlet container [2], uses SPARQL and the RDF-triple store in the sesame framework [3] for querying and analyzing data. Via the server, the clients can send queries to the repositories and receive data as a table or graph (ROSETTA Project presentation, 2010). The architecture has three levels, separating the data storage from any logic on the server side and data presentation in for ecample a web browser on the client side, making it easy to change and update the different levels separately.

[1]http://www.automationml.org/

[2]http://tomcat.apache.org/

[3]http://www.openrdf.org/

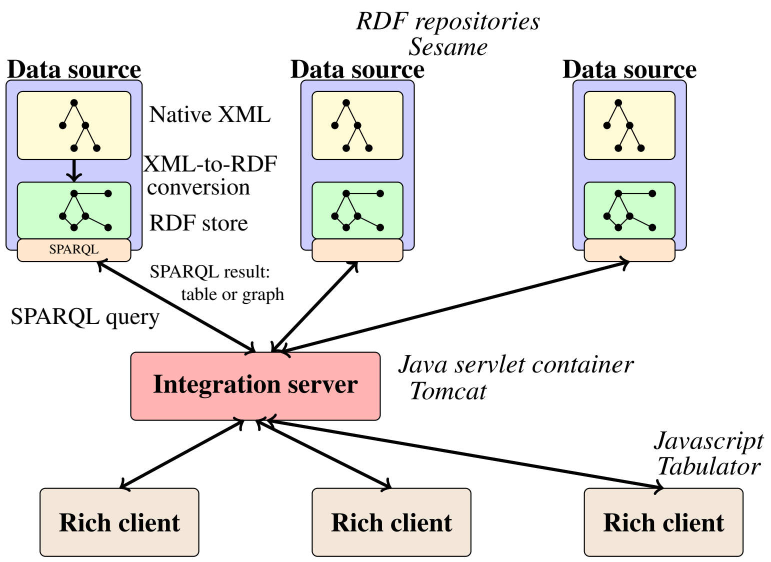


Figure 1.2: Complete architecture of the knowledge integration framework (KIF) including the visualization clients. Adapted from Persson et al (2010).

1.3 The Thesis Project

I can do whatever I want. They will tell me if what I am doing is stupid or a total waste of time. I may tell them that they are wrong, and we will come to an agreement.

– Bill Budge

One way of instructing the robots on a higher level is to re-use already written pieces of robot code, called *skills*. In a graphical user interface, the user can compose the task by choosing actions from a list of implemented skills. However, to enable this, the skills should be stored online in the KIF in a way that allows re-use.

There are two possible approaches for storing the programs. The skill can be described in an entirely general language, as a state machine or a list of actions from which robot code can be generated to any given robot language. A generic description requires a platform-dependent interpretation to be able to run it. Another option is to store pieces of program code as chunks or binary large objects, BLOBs, in the KIF database. These views are opposite in how much code and how much interpretation effort they require, see Figure 1.3.

So, in essence, it is only a matter of scale. A generic description of a skill would require a long list of actions (or subskills) that each, on some level, has an interpretation as robot code. The more specific approach has everything as one unchangeable piece of code. Hence, when populating the database with skills, it is all about finding what code pieces might be interesting to re-use.

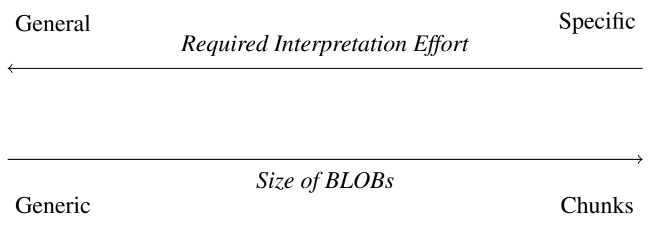


Figure 1.3: In a general representation, a very small piece of code, maybe only one line of code, will be generated from the skill, which increases the number of skills that must be used to program a task and vice versa.

1.3.1 Purpose

The overall purpose of this thesis project is to investigate how to integrate knowledge about skills in the knowledge integration framework. The project is divided into two parts: the first part focuses on populating the KIF and data annotation, while the second part on re-using robot programs.

The first part, performed at Lund University, investigates how to add skills described as finite state machines with associated parameters to the KIF.

The second part, carried out at ABB Corporate Research, aims at identifying how re-usable pieces of robot code can be defined so that they can stored in KIF. It includes both proposing a structure for the description and implementing an example where a skill is exported from the engineering system to KIF and later retrieved and used in another robot program.

The result should be a description on how skills can be defined in the KIF and how the knowledge can be transferred into the KIF. One important aspect of the result is to implement tools to demonstrate the concepts. That is, to create user-friendly interfaces in form of webservices or extensions of existing engineering tools.

The work is valuable because it enables the user to populate the KIF with knowledge and create new programs based on already existing skills and re-using knowledge from the database.

1.3.2 Delimitations

The thesis work is limited to investigating the storage of re-usable information about skills in KIF with a focus on annotation and re-use of code. The work also investigates the information exchange between KIF and the engineering system. The results can be used as guidelines and the demonstration tools works as a proof of concept.

1.3.3 Work method

The skill description is created in RDF and test and demonstration tools are made using Java servlets and engineering and simulation tool ABB RobotStudio.

The work was carried out together with team members of work package 1 working with KIF at the department of computer science at Lund University and members of work package 6 working with the engineering system at ABB Corporate Research in Västerås.

1.4 Structure of the Report

The report starts with a theoretical background in Chapter 2 where the concepts of RDF and SPARQL are explained. In Chapter 3, types of knowledge, re-usable information and requirements are discussed. The design and implementation is described in Chapter 4, while Chapter 6 summarizes and demonstrates the results of the thesis and the epilogue addresses conclusions and future work.

Chapter 2

Theoretical Background

Before continuing to how the skills can be stored in the database, it is important to understand what is hiding under the hood of the KIF. Thus this chapter is an introduction to RDF, inference, and SPARQL.

2.1 Resource Description Framework

The Internet is the world's largest library. It's just that all the books are on the floor.

– John Allen Paulos

The Resource Description Framework, RDF is a standard model for data interchange on the Web. The World Wide Web is a huge information store of human-readable web pages. The web pages can be extended with metadata that makes the semantic, or the meaning, of the information understood by machines. This web of data about the information on the World Wide Web is called the Semantic Web. To enable linking and reasoning upon the Semantic Web, RDF was created as a suite of specification by the World Wide Web Consortium, W3C (http://www.w3.org/).

Even though RDF can be used to link the entire World Wide Web [1], the principles can be applied to smaller data stores as well, thus enabling a little "reasoning" on the data set (see Section 2.1.3).

2.1.1 RDF-triples

In RDF, information is represented nodes and edges in a graph. The information is stored as *statements* with a *subject*, a *predicate* and an *object,* very much like natural languages. The subject and the object are nodes and the predicate is the directed edge connecting them, pointing from the subject to the object.

[1]The details are left as an exercise for the reader.

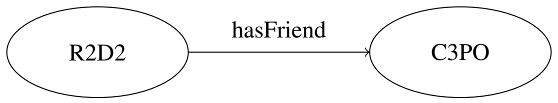


Figure 2.1: RDF representation of the statement: *R2D2 has (a) friend C-3PO in RDF.*

Example 1

To store the statement regarding the friendship between two robots: *R2D2 has a friend C-3PO,* the subject would be the robot *R2D2,* the object is the robot *C-3PO* and the predicate would denote the relationship between them. Figure 2.1displays the graph.

Since every statement contains three resources, a subject, a predicate and an object, another frequently used word for the statement is *triple*. The predicate is also called *property*, since it describes some property of the subject.

2.1.2 URI, Prefix and Context

When talking about RDF, there is a notion that *anyone can say anything about anything.* This is often referred to as an *open world* description, which allows anyone to extend the database with information. An open world is necessary, since RDF was created to describe the World Wide Web. However, the knowledge in the KIF must be consistent and hence rules governing the statements are added in a system called *ontology* (see Section 2.1.4).

Another step to structure the information is to give each resource a unique ID. Since RDF is modeling the web, it is natural to take Uniform Resource Identifiers, URI:s, as unique ID:s.

In example 1, the subject is given the URI:s *http://starwars.com#R2D2,* the predicate is *http://starwars.com#hasFriend* and the object *http://starwars.com#C3PO.*

Spelling out the entire URI:s might be tedious, hence it is possible to declare *prefixes*. In this case, the following prefix could be used:

```
prefix sw: http://starwars.com#
```

Instead of writing the full URI:s, the prefix can be used with a colon to write the statements as follows:

```
sw:R2D2 sw:hasFriend sw:C3PO
```

In the example, the object node can connect to other nodes but does not contain any data. Therefore, it is also possible to make the object node to a *terminal node* with a datatype and value. The datatypes can be strings, Boolean values or numbers. The values of the terminal nodes are called *literal values* or simply, *literals*. *E.g.* giving R2D2 a nickname would look like:

```
    sw:R2D2 sw:hasNickname "Artoo"
```

Statements can be "bundled" together in a *context*, which is also expressed as a URI.

2.1.3 Inference, RDFS and OWL

In RDF there can be object classes, just as in object oriented programming. A resource can be declared as an instance of a class by using the pre-defined property *http://www.w3.org/1999/02/22-rdf-syntax-ns#type,* which can be abbreviated *rdf:type*, using the prefix *rdf* for *http://www.w3.org/1999/02/22-rdf-syntax-ns#.*

Just as in object oriented programming, there is a class hierarchy. A class can be declared as a subclass of another class by using a *subClassOf* property in the RDF Schema language, RDFS (RDF Schema, 2011), usually used with the prefix *RDFS* for the URI *http://www.w3.org/2000/01/rdf-schema#.* Of course, an instance of a subclass, will also have the type of the superclass. Thus extra information is deduced. This is called *inference* and is illustrated in example 2.

Example 2

There can be a class *sw:Droid* which is a subclass of *sw:Machine.* Using the property *rdfs:subClassOf* the following can be stated:

```
sw:Droid rdfs:subClassOf sw:Machine
```

Adding an instance of the *Droid* class:

```
sw:R2D2 rdf:type sw:Droid
```

will also give the additional triple through inference:

```
sw:R2D2 rdf:type sw:Machine
```

An external tool called *inference engine* is used to add inferred triples to the repository. Another set of inference rules are given by the Web Ontology Language, OWL (Web Ontology Language, OWL, 2011). Two uses of OWL are given in example 3.

Example 3

Continuing with the robot example, making the naive assumption that friendship is mutual, the *hasFriend* property should be symmetric. Of course, this could be stated explicitly that *C3PO has (a) friend R2D2*, but this can be inferred instead. In OWL, the there is a class *SymmetricProperty* that can be used:

```
sw:hasFriend rdf:type owl:SymmetricProperty
```

This means that if the statement

```
sw:R2D2 sw:hasFriend sw:C3PO
```

is present, the other statement

```
sw:C3PO sw:hasFriend sw:R2D2
```

will be inferred. The new triple will be added by an *inference engine.* Just as classes can be subclasses, properties can be subproperties:

```
sw:hasFriend owl:subPropertyOf sw:isFamiliarWith
```

will now infer both triples

```
sw:R2D2 sw:isFamiliarWith sw:C3PO
sw:C3PO sw:isFamiliarWith sw:R2D2
```

OWL and RDFS both have a wide range of useful inferences, however, in this thesis project, it is mostly *rdf:type* and *rdfs:subClassOf* that are used.

2.1.4 Ontologies

The use of inference is not limited to OWL and RDFS, it is possible to define an own set of rules in a *ontology*. An ontology describes the semantic web, it can include inference rules, property cardinality (*E.g.* limiting the number of parents to two) and declare resources with different URI:s as the same or strictly different, for mentioning a few.

One example of an ontology, which is used in this thesis project, is the NASA unit ontology (Nasa unit ontology, 2011). It maps units to quantity kinds, gives each unit conversion ratios and so forth, for example meters and millimeters are both length units and the conversion ratios are 1 and 1 000 respectively.

The material and examples covered in this chapter should be sufficient to understand all the details of the thesis project, however, for deeper studies there is an excellent book: *Semantic Web for the Working Ontologist: Effective Modeling in RDFS and OWL*, by Dean Allemang and Jim Hendler (2008).

2.2 SPARQL Query Language

The answer to the ultimate question of life, the universe, and everything is... 42!

– *Deep Thought,* The Hitchhiker's Guide to the Galaxy, *Douglas Adams*

A query language is needed to extract information from the RDF repositories. The two main query languages for RDF: SPARQL and SeRQL (pronounced "sparkle" and "circle"). Both can be used in KIF, but SPARQL is the main language in this project.

A query contains the following elements:

- Prefixes that are used (optional).
- Keyword for how to present the query result (“SELECT”, “CONSTRUCT” etc).
- Output parameters that are written with “?” to signal that they are variables.
- If only a certain context should be searched, “FROM *context*” is added to the query string (optional).
- The premises for the search is added within a “WHERE{ ... }”.
- Finally filters can be added (optional).

There can be zero, one or multiple results depending on the number of matches for the query. Using an appropriate API, it is straightforward to write a client that queries a repository directly, however, the KIF server also provides a user interface for queries, where the query is written in a form and the result presented in a table. The three examples below illustrates how the different elements of the query works.

Example 4

This is what a simple query look like. It answers the question: *Who is familiar R2D2 with?*

```
PREFIX sw:<http://starwars.com#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?x FROM <http://simple.example.se>

 WHERE{sw:R2D2 sw:isFamiliarWith ?x

}
```

The single result would be presented as:

Variable	x
Value	sw:C3PO

The multiple search criterias are separated with a “.” and the variable *x* is bound to fulfill all statements. The FROM keyword is used to set the context, it is not necessary in this case and will be left out in the other examples.

The SELECT method just returns the variables in a list. CONSTRUCT, on the other hand, returns triples that form a graph.

Example 5

Assuming that the repository contains multiple (an possibly conflicting) statements about droids, humans and everything in between. The nodes that are extracted in a query can be used to create a new graph with the CONSTRUCT method:

```
PREFIX sw:<http://starwars.com#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
CONSTRUCT  {?x rdf:type sw:Cyborg}
WHERE{{?x rdf:type sw:Cyborg.}
UNION {
?x rdf:type sw:Machine. ?x rdf:type sw:Human}
}
```

This will return a set of triples of all statements where the subject is of type *Cyborg* AND where a resource is stated to be both a machine and a human although doublets will not be returned. This possibility means that the created graph not necessarily resembles the graph it queries.

A final example demonstrates how filters (SPARQL Query Language, 2011) can be used:

Example 6

To get a copy of the entire graph, the CONSTRUCT and WHERE can contain the same variables *?s*, *?p* and *?o*. It is also possible to filter the result to only return certain statements. Filters can match data types, strings, evaluate logical expressions using AND ("&&") or OR ("| |"). In the SPARQL query below, the only statements that are interesting are the ones where (at least) one of the resource URI:s contains the substring "R2". This is done by finding a match between the URI string and the sought string using regular expressions.

```
PREFIX sw:<http://starwars.com#>
CONSTRUCT {?s ?p ?o}
WHERE {{ ?s ?p ?o}
FILTER (regex(str(?s), "R2")
|| regex(str(?p), "R2"))
 || regex(str(?o), "R2")) }
```

These examples only highlight a few methods. The search criterias and filters can be very elaborate, making the query language quite powerful. A word of caution thought, when it comes to queries, if the answer seems strange, the query is probably badly formed, because after all: *You get what you ask for.*

Chapter 3

Skill Requirements

This chapter examines the definition of skills and the information that is relevant to store and re-use and required data for enabling re-use.

3.1 Skills

If the power to do hard work is not a skill, it's the best possible substitute for it.

– James A. Garfield, 20th president of USA

In plain English, a skill denotes a special ability obtained by training, particularly requiring the use of hands or body. Robot skills are similar, it is a task that the robot can perform with its extremities or a computation capability, such as the ability to visually identify objects.

Since the skills can be a set of robot instructions, a skill is whatever the instructor decides to call a skill. In this case, the focus lays on generic skills that can be used in more than one setting. Thus it is worthwhile to store and distribute the generic skills in a database.

3.1.1 Word Definitions

These definitions are not strict and some terms are used in multiple contexts, each time with slightly different meaning:

Task A task is a job that the robot performs, *E.g.* to assemble a cell phone. The assembly can in turn be divided into a series of activities: locate part, move and pick part, these activities are called skills. Thus a task is composed by a set of skills.

Skill an action or activity performed by the robot, such as a movement, locating a part or gripping it. Skills can have different levels, *E.g.* an *assembly skill* is composed by movements, location of parts, gripping and releasing, thus it is more complex than perhaps a single movement.

3.2 Types of Knowledge

The KIF should contain and link different types of knowledge. A skill can be described as a finite-state machine (see below) or as a piece of program code (see Section 3.2.3). These descriptions could be exchangeable and it should be possible to mix them when instructing the robot. The KIF should be able to store several types of knowledge. Some actions are more easily expressed in robot code directly; some will need a state machine.

3.2.1 Finite-State Machines

The skill can be described as a finite-state machine, each state representing a step in the execution. In the case of robot programs, each state is either a movement, called *Motion*, or information processing, simply denoted *Action*(Bruyninckx, 2010). These are connected with *Transitions*. Thus a finite-state machine can be described as: *Initial Step (Motion/Action)* $\rightarrow$ *Transition* $\rightarrow$ *Step (Motion/Action)* $\rightarrow$ ... possibly with parallel execution.

The task or action can be described as a state machine that is created in a tool called JGrafChart (JGrafChart, 2011), see Figure 3.1. Each state is drawn as a square and represents a motion or action as mentioned in Section 3.2.1. The states can have parameters with values, such as the speed for executing a task.

The statemachine is saved as an XML-file. The XML description can be interpreted and transformed into RDF triples and stored in the KIF repositories. See Section 5.1 for details.

3.2.2 LabComm Samples

The parameters in the state machine are sent to the robot controller using a transmission protocol called LabComm (LabComm, 2011). It is used for one-way communication in real-time systems where the overhead needs to be kept to a minimum. The parameters are written as *samples*, for example:

```
sample double f_switch;
```

In this report, the samples are carelessly called *parameters*, however, each sample use one of four channels and is as such either a parameter, log, input or output data. The file with the samples written in plain text can also be translated into RDF triples and stored in the database, see the implementation description in Section 5.1.

3.2.3 Storing Robot Programs in KIF

When it comes to re-using robot code, some basic requirements must be fulfilled. First, each program should have a description of what it does. In this case, it should be both

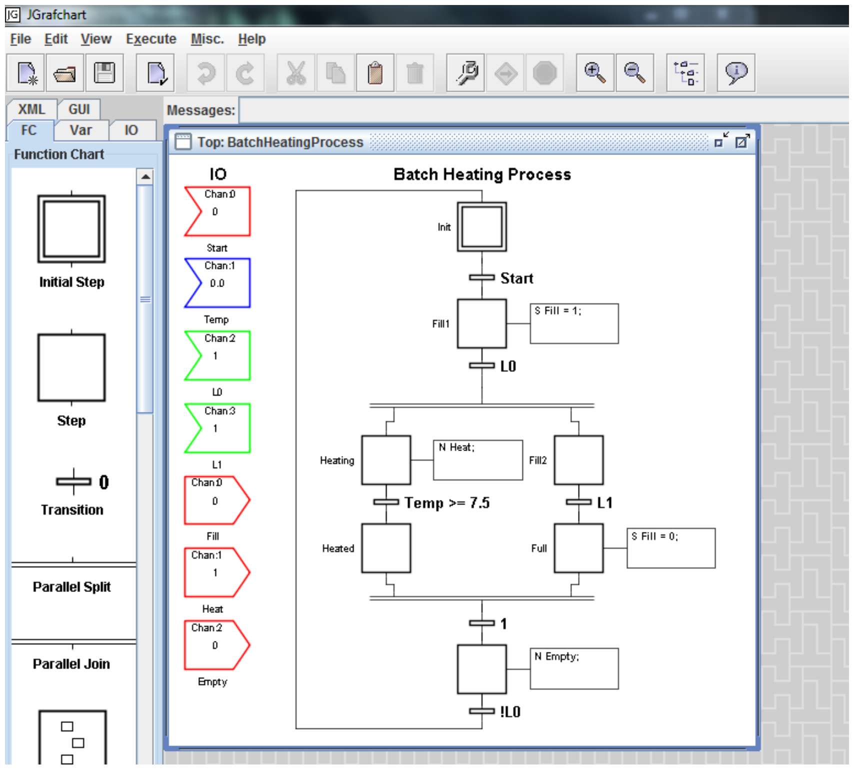


Figure 3.1: Example of a finite statemachine from JGrafChart (2011).

a machine-readable label and a description in natural language. The skill should also account for the tools or information it needs to operate, *E.g.* a grasping skill needs a gripper.

The program stored in KIF should contain enough information to generate working code. In this case, there will be pieces of robot code attached to it. However, transferring pieces of code requires some care to avoid name conflicts and missing variable declarations. Hence, everything that is not purely internal must be declared. In essence, that means that all global or external variables must be arguments to the procedure, and similarly, all output parameters should be listed.

For the purpose of demonstration and testing, the programs are written in RAPID, the language used for ABB industrial robots, and the engineering and simulation tool associated with it, RobotStudio. The robot programs are created in RobotStudio and stored in the KIF database and later retrieved and re-used in RobotStudio. Creating the procedure does require familiarity with the programming language (when introducing composite skills this will not be necessary). Importing the skill as a BLOB from KIF and running a simulation in the engineering tool should not require any changes in the code. However, running it on a robot will very likely require adaptions.

3.2.4 Example of a Robot Program

At this point, it is time to stop beating around the bush and finally look at an executable robot program. As a demonstration, it should be possible to export similar programs from the engineering tool to the database and later import them and use them in a different station. Below is an example of a program written in RAPID. It is a program that resets all signals (`closegripper` and `opengripper`) and moves the robot into a *home position*.

```
CONST jointtarget home_pos:=[[0,0,0,0,30,0],
    [9E9,9E9,9E9,9E9,9E9,9E9]];
PROC main()
    VAR string error_string;
    initialize;
ENDPROC
PROC initialize()
    Reset closegripper;
    Reset opengripper;
    MoveAbsJ home_pos,v1000,z100,MyTool,
       \WObj:=wo_station2;
ENDPROC
```

Constant global variables are declared outside the procedures with the keyword `CONST` and the data type (here *jointtarget*), name and assigned value (using ":="). Each procedure is written between PROC- and ENDPROC-flags. Local variables are declared in the beginning of a procedure with a flag, type and name. Procedures are called by name followed by arguments. *E.g.* *MoveAbsJ* is a built in procedure with the arguments:

home_pos as target position.

v1000 the velocity 1000 mm/s.

z100 that is the proximity radius to the target position, thus how close to the target is close enough.

MyTool is the tool that should be moved into the target position.

\WObj:=wo_station2 sets an optional variable *WObj* to *wo_station2* which is the name of a work station created in the simulation environment.

There will be a few more examples of robot code in Chapter 4 to concretize the use of skills.

Chapter 4

Design

Now, when previous chapters have given the background, defined the problem and how the system works, it is time to define how to store the information in the database.

First, we create a design of how to represent a basic skill containing a BLOB of code in RDF. The design involves a type classification, a description of how to describe the arguments generically, and how a skill is written in robot code to enable re-use.

The next step is the design of composite skills, that are a combination of other skills. This involves an argument mapping so that information can be shared between the inner skills.

4.1 Basic Skills

We are programmed just to do, anything you want us to.

– from The Robots *by Kraftwerk*

Here, we create the structure of storing a piece of program code. Initially, the skills are classified in different types according to their purpose. The execution can be described abstractly in a finite-state machine (Section 3.2.1) and the executable robot program, can be attached to the skill together with a generic set of arguments containing the input and output parameters (see Section 4.1.2).

4.1.1 Classification and Description of Skills

If the robot should move the arm into a certain position, it is desirable to receive a list of already defined moving skills for the robot from the database. The first step is therefore to classify skills according to their purpose, creating a class hierarchy.

A Simple Example:

As a first simple example, a robot picks up a cube at one station (Station 1) and leaves it at Station 2. The robot locates the position, approaches it, grasps the object at the correct position, and retracts. A similar procedure is used to place the object at the release position. Schematically, it looks like this:

```
Locate Pick Position
Approach Pick Position
Grasp
Retract
Locate Release Position
Approach Release Position
Release
Retract
```

From this example, a first set of skill types can be identified, locating skills, motions and tool actions.

Classification

To organize the skills in a sensible structure that is easily extended, the skills are ordered in a class hierarchy. Every skill class is a subclass of the superclass `Skill`, but there are for example several types of motion skills. Below are a few examples expressed as subclasses in RDF- triples (the namespace is left out):

```
:Motion rdfs:subClassOf :Skill
:Approach rdfs:subClassOf :Motion
:Retract rdfs:subClassOf :Motion
:Grasp rdfs:subClassOf :Skill
:Release rdfs:subClassOf :Skill
```

Assuming there is a skill instance called `myApproachingSkill`, the classification information can easily be stored as a triple in KIF:

```
:myApproachingSkill rdfs:type :Pick
```

and the RDFS-inference will generate the additional triple:

```
:myApproachingSkill rdfs:type :Skill
```

In addition to the class, each skill is given a label in natural language with a description of the skill. This is simply stored as a triple:

```
:myApproachingSkill rdfs:label "The first skill in KIF."
```

4.1.2 Input and Output Parameters

A simple example of RAPID code will serve as an illustration of how to store the arguments in the KIF.

A Simple Example: Approach

This is a small RAPID program that moves the robot tool (*`MyTool`*) into a target position (*`pick_pos`*) at a workobject (*`wo_station`*). The motion has initially high speed (*`high_speed`*), but at the distance *`distance`* of the target position it slows down to *`low_speed`*.

```
PROC my_approaching_skill(tooldata MyTool, wobjdata wo_station,
robtarget pick_pos,
speeddata high_speed, speeddata low_speed, num distance)

  MoveJ RelTool(pick_pos, 0,0,-distance),high_speed, z20,
      MyTool\WObj:=wo_station;
  MoveL pick_pos, low_speed, fine,
      MyTool\WObj:=wo_station;
ENDPROC
```

In this design, the signature, in the example given by *`my_approaching_skill(tooldata MyTool, wobjdata wo_station, robtarget pick_pos, speeddata high_speed, speeddata low_speed)`*, is not stored with the rest of the BLOB, to 1) avoid name conflicts with other existing procedures and 2) both the procedure call and the signature is generated from the set of generic arguments, as described in Section 5.2.

The BLOB is everything between the PROC and ENPROC- key.words Attaching the RAPID BLOB to `myApproachingSkill` is simply done:

```
:myApproachingSkill :hasRapidBlob "MoveJ RelTool ..."
```

Here the code is stored as a literal.

An argument is added like this:

```
:myApproachingSkill :hasArgument :high_speed
```

where each argument has a generic type, a name and tags whether they are references, optional or both (the Boolean literals are emphasized):

```
:high_:speed rdf:type :Speed
:high_:speed :hasName "high_speed"
:high_:speed :isReference true
:high_:speed :isOptional true
```

If the skill returns a value, it is a *function*. A locating skill can be a function returning position coordinates. Then the skill will keep track of the return type in a separate statement:

```
:myLocatingSkill :hasReturnType :Position
```

However, the locating skill could just as well be a procedure with a position reference, the only difference is that a function *must* have a return value, while a procedure reference could be entirely ignored in the code. In general, the skills are procedures.

There is nothing limiting what can be said about the arguments, *E.g.* giving them default values, here using the NASA unit ontology QUDT is done like this

```
:high_speed :hasDefaultValue :def_high_speed
:def_high_speed :hasUnit qudt:MeterPerSecond
:def_high_speed :hasValue "1.0"
```

Now, if a application needs these default values it can query KIF for them, while they are ignored by all other applications.

When creating new skills, the robot programs must have the same structure as given in the approach example. All variables from the station such as tools, objects and signals must be arguments to the procedure, to ensure that they will be given a value from the new station when re-used.

4.1.3 Relationships Between Skills

As mentioned previously, creating skills calls for some programming proficiency, but if it also required knowledge of RDF, KIF would be sparsely populated. Hence, all triples must be generated automatically and still have a unique ID, either as a part of the URI or stored in a separate node.

The skills in the KIF keep track of previous and updated versions using the transitive and inverse properties *previousVersion* and *updatedVersion* respectively. Time stamps and creator IDs etc can be added.

4.1.4 Summary of the RDF Representation

To minimize the headache of implementation, the skill representation in KIF is minimalistic. Every single node must have a raison d'être. The skill implementation has language specific BLOBs of code and a set of arguments. To minimize the number of nodes in the graph, these are adjacent to the skill node, see Figure 4.1.

Each argument keeps track of its own type (*Tool, Speed* etc), whether or not it is optional or a reference and a name (which can be language specific).

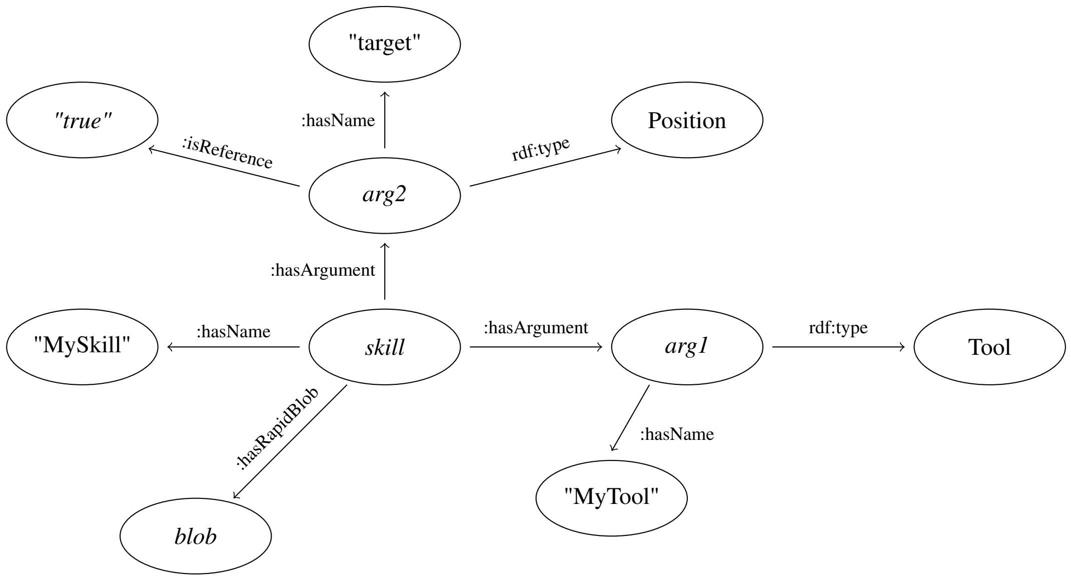


Figure 4.1: Skill representation in the KIF. The blobs are adjacent to the skill node as well as each argument. Each argument keeps track of its type and name.

4.2 Composite Skills

The longer you look at an object, the more abstract it becomes.

– Lucian Freud, artist.

Assume that the KIF is populated with some sensible skills, that is, skills for locating objects, motions and gripper movements. Certain combinations of skills are more common than others; when a part is picked at a location, four skills are used:

```
Pick

    Locate
    Approach
    Grasp
    Retract
```

`Pick` is composed by a set of other skills and is hence named a *composite skill*. A composite skill does not have own program code attached to it, it only refers to a list of regular skills. Therefore, the composite skills is an own type of skill. In KIF, the composite skill is represented as a linked list of nodes pointing to skill nodes, see Figure 4.2.

Anyone can say anything about anything; so nothing limits the number of nodes that can be added to each link. If one of the skills lacks a language implementation that

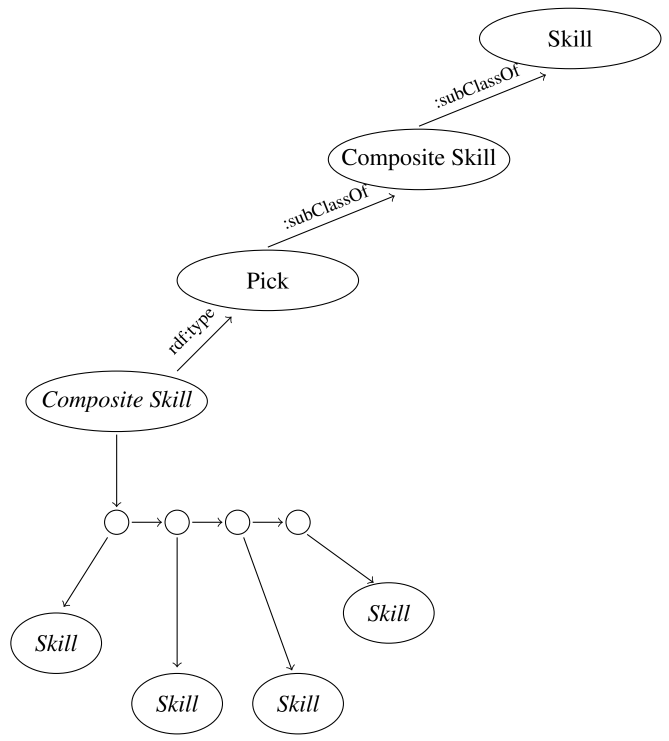


Figure 4.2: A schematic figure of composite skills. The nodes with italic names are instances, the others are classes. Composite skills refer to a list of skills but does not contain own program code.

exists in another skill, the other skill could be added to the list node. Only the suitable language implementations are selected.

However, just keeping the skills as a list does not account for the relationship between the skills. *E.g.* the target position returned from `Locate` and used by `Approach` is an entirely internal variable and, as such, the user does not need to be concerned with it. Similarly, if the same variable is used in several of the skills, *E.g.* it is the same tool moving, gripping and so on, it would be tedious to specify it as an input parameter for each routine every time the composite skill is used.

When importing `Pick`, the user does not (necessarily) care about composite skills, he or she just wants the robot to pick up some object. Of course it is the same tool that is moved into place and then gripping the object, why should the user need to specify it for both `Approach` and `Retract`? Even though the target position is an input variable in three skills, it does not have to be an argument to the `Pick` skill and is thus an internal argument.

So instead of just having a list of skills, the composite skill also contains an argument mapping of how the set of arguments are shared by the inner skills.

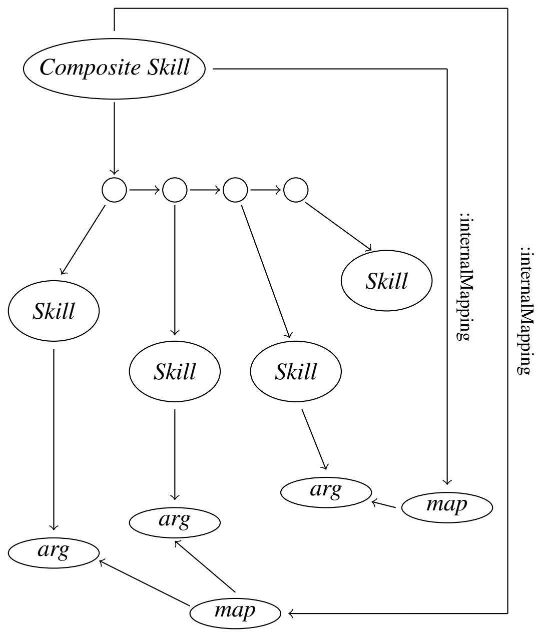


Figure 4.3: Arguments with the same input variable are connected by map nodes in the composite skill. The map nodes are pointing to arguments that are the same. Only the internalMapping property is explicitly written in the graph.

4.2.1 Argument Mapping

It is not entirely obvious how to represent an argument mapping in KIF in a good way. However, there are some conditions for the mapping:

1. The mapping is connected to the composite skill, not only the arguments.
2. The mapping must be able to handle a nested composite skill with its own mapping.
3. The underlying skills may not be changed by the composite skill.

If the above requirements are accounted for, the mapping and the skill list can be separated and certain argument mapping nodes added to the graph. One example of what the mapping can look like is illustrated in Figure 4.3**.** In this case, there are no composite skills in the list. As seen in the figure, this design involves map nodes that connects the arguments that are shared. When importing the composite skill, the map nodes have the same function as the argument nodes have in a basic skill. Each map node is assigned a value by the user, and that value is assigned to all the arguments in the subskills that are adjacent to the map node.

Now what happens if the composite skill points to another composite skills? The internal mapping of the inner composite skill is not changed, but the outer composite skill can map arguments from any level. So even if elements are mapped between the regular skills and the nested composite skill, the latter still keeps track of the internal mapping between its listed skills. See Figure 4.4.

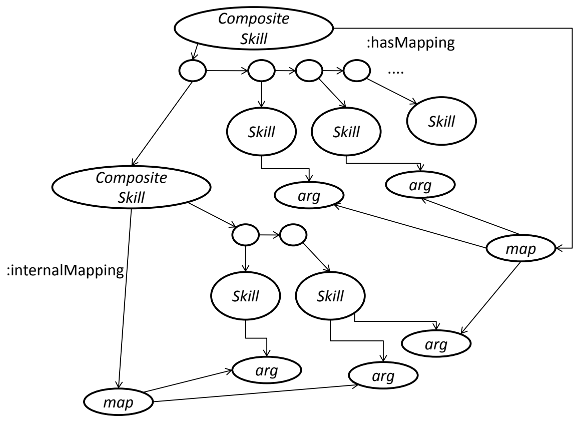


Figure 4.4: Mapping arguments with nested composite skills. Internal mappings are "local".

Referring to argument nodes that are under some other skill's control makes the mapping structure fragile in case of changes. Thus when importing a composite skill any existing mapping must pass the consistency check, that is, the arguments in a mapping must have the same type declaration and belong to the listed skills. If not, the mapping is omitted.

Chapter 5

Implementations

Two tools were created for the purpose of demonstration. The first is a Java servlet for uploading finite state machines and adding LabComm parameter information to the KIF. The second is an plug-in to the simulation tool RobotStudio where robot program code was exported to the KIF for later re-use.

5.1 State Machine and LabComm Servlet

The intent is there, but implementation is lacking.

– Kiran Karnik, former president of NASSCOM

The servlet can do the following:

- Upload a finite state machine describing a skill in XML-format, transform it to RDF, and add it to a KIF repository.
- Visualize the states in the finite-state machine.
- Assign LabComm parameters to the finite-state machine.
- Tagging the LabComm sample as either a *Parameter*, *Log*, *Input* or *Output*.
- Adding and editing sample comments.
- Adding and editing sample quantities and units.

Uploading the state machine as an XML-file and the LabComm samples to the server is straightforward. The user specifies the server, the repository, the context, and base (namespace) URI. Then the XML content is transformed into RDF using the Saxon library with XSLT stylesheets (Saxon, 2011), that are used to translate information between formats. Finally the RDF triples are added to a repository.

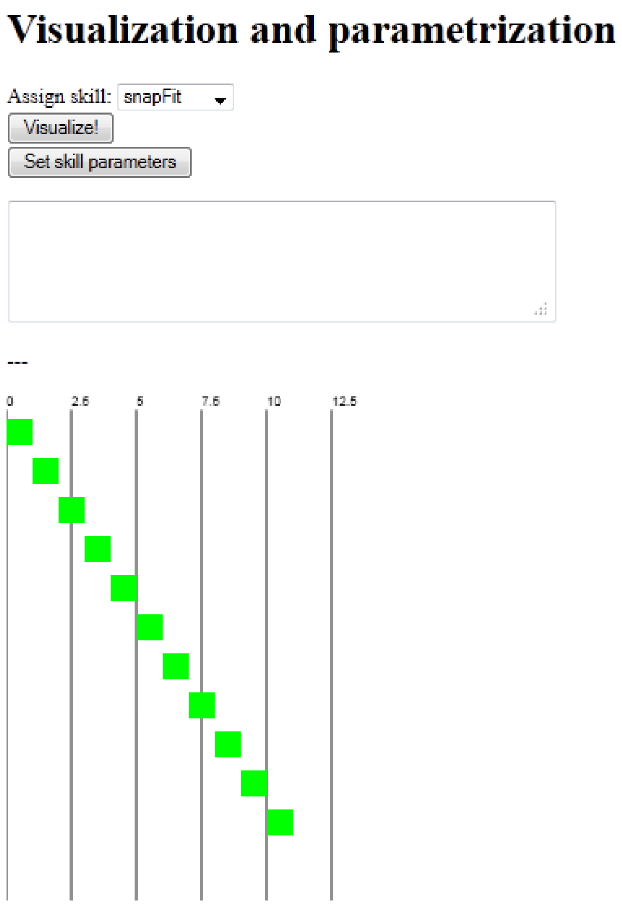
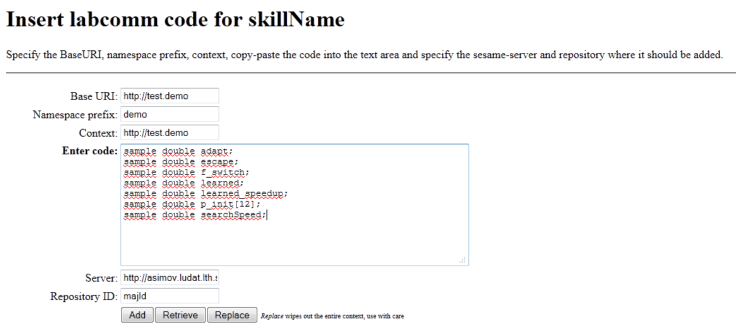


Figure 5.1: Screen shot of the demonstration. The drop down menu displays all skills in the KIF. The visualization button calls a JavaScript that draws a chart with the states. The button labelled "Set Skill Parameters" directs to the page shown in figure 5.2.



Insert labcomm code for skillName

Specify the BaseURI, namespace prefix, context, copy-paste the code into the text area and specify the sesame-server and repository where it should be added.

Base URI: http://test.demo

Namespace prefix: demo

Context: http://test.demo

Enter code:

```
sample double adapt;
sample double escape;
sample double f_switch;
sample double learned;
sample double learned_speedup;
sample double p_init[12];
sample double searchSpeed;
```

Server: http://asimov.ludat.lth.s

Repository ID: majld

Replace wipes out the entire context, use with care

Figure 5.2: Screen shot of the demonstration: The LabComm samples are uploaded to the server using a servlet.

When choosing to visualize the finite state machine, a JavaScript is called that retrieves necessary node information from the repository to create the chart pictured in Figure 5.1.

The same servlet is used to link LabComm parameters to the state machine, by uploading them to the server. See Figure 5.2. Again the user chooses the server, the repository, and the base and context URI:s and the LabComm code is added in the text area.

The sample are also transformed into RDF and then annotated with the same webinterface with units and comments. To describe the units, the NASA unit ontology QUDT is used (Nasa unit ontology, 2011). The unit ontology has, as mentioned in Section 2.1.4, mapped the quantity kinds to units and the ratio between them. See the screen shot in Figure 5.3. This information is stored as RDF triples as well and viewed again when the user wants to edit old information.

5.2 Robotstudio Plug-in

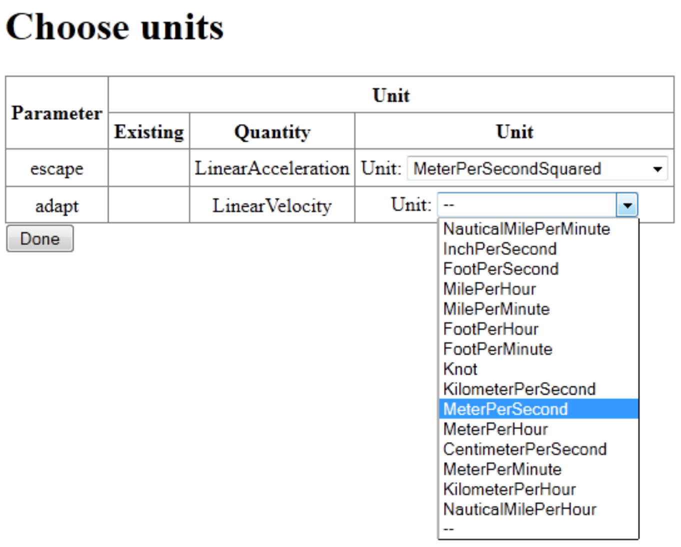
We allow no geniuses around our Studio.

– Walt Disney

The plug-in is written in C# and incorporated with the rest of the RobotStudio as a tab on the menu. It is used for exporting and importing BLOBs of RAPID code to the KIF and is has an updated online connection to KIF.

Creating a Skill in RobotStudio

The skills are written in RAPID directly in RobotStudio with all parameters as input arguments to a procedure as described in Section 4.1.2. See the right side of Figure 5.4. In the **Demo** ribbon there is a **Test Export** button that opens a window for exporting code. See the window to the left in Figure 5.4. The user names the skill, assigns it a type and posts the entire procedure in the text area.



Choose units

<table>
<tr><th rowspan="2">Parameter</th><th colspan="3">Unit</th></tr>
<tr><th>Existing</th><th>Quantity</th><th>Unit</th></tr>
<tr><td>escape</td><td></td><td>LinearAcceleration</td><td>Unit: MeterPerSecondSquared</td></tr>
<tr><td>adapt</td><td></td><td>LinearVelocity</td><td>Unit: --</td></tr>
</table>

Figure 5.3: Screen shot of the demomonstration: The LabComm samples are assigned quantity kind and units.

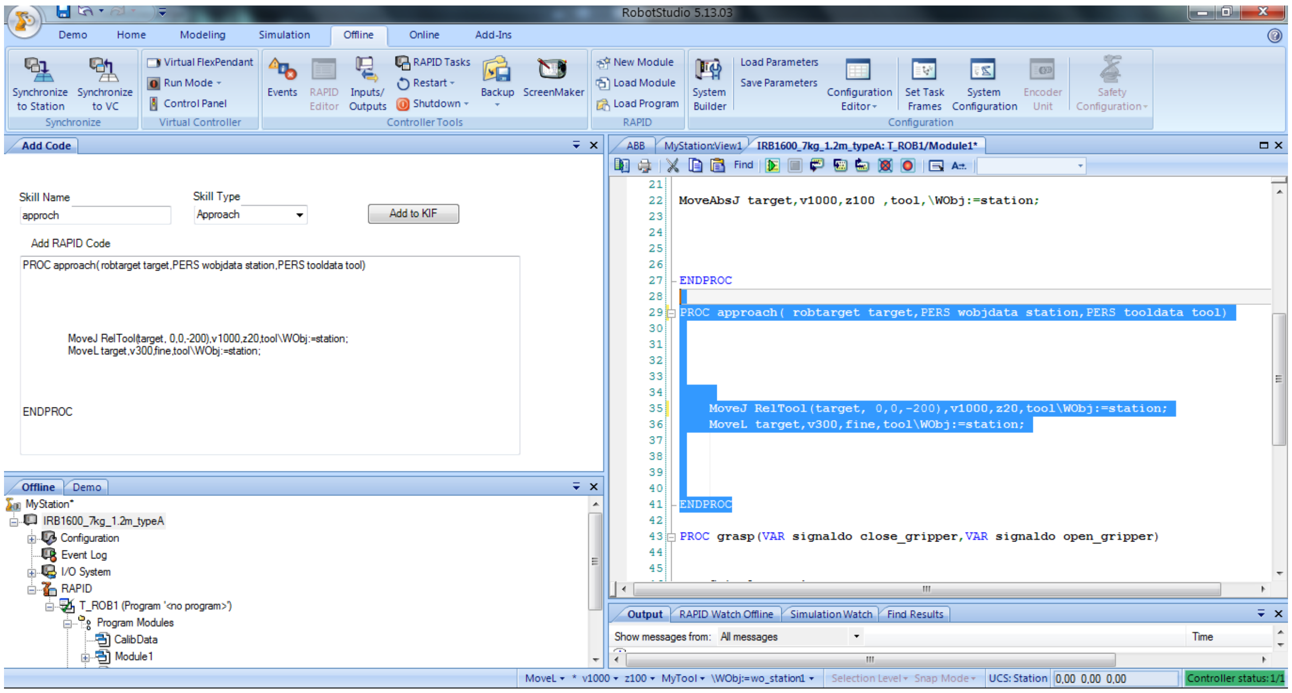
Generating RDF and Connecting to the KIF

Pressing **Add to KIF** will parse the code for the signature and the arguments and create a `Skill` object, with the the code between the PROC- and ENDPROC-keywords stored as a BLOB and the set of arguments as fields.

There is no official API for handling RDF in C#, so a custom Repository class was created. The `Skill` object is transformed into RDF triples in the local `Repository` class. Each argument type is mapped from robot code to the generic KIF type using a static `TypeMapping` class with dictionaries for how robot code is represented in RDF. The types are written as statements where the fields (name, type, etc) are objects to the skill or argument nodes. The repository is then written in an RDF format called N3, as the example shown in Figure 5.5.

The implementation has a separate class for handling the connection to KIF. This string of RDF triples are sent to KIF via a third party application called SemWeb[1] as a `Post` command to a import servlet in KIF. The same class also uses SemWeb to send SPARQL[2] queries when receiving information.

[1]razor.occams.info/code/semweb/
[2]http://www.w3.org/TR/rdf-sparql-query/


Skill Name: approch

Skill Type: Approach

Add RAPID Code

```
PROC approach(robtarget target,PERS wobjdata station,PERS tooldata tool)

        MoveJ RelTool(target, 0,0,-200),v1000,z20,tool\WObj:=station;
        MoveL target,v300,fine,tool\WObj:=station;

ENDPROC
```

```
MoveAbsJ target,v1000,z100 ,tool,\WObj:=station;




ENDPROC

PROC approach( robtarget target,PERS wobjdata station,PERS tooldata tool)




    MoveJ RelTool(target, 0,0,-200),v1000,z20,tool\WObj:=station;
    MoveL target,v300,fine,tool\WObj:=station;




ENDPROC

PROC grasp(VAR signaldo close_gripper,VAR signaldo open_gripper)
```

Figure 5.4: Screen shot of the demonstration. To the RAPID code from the program to the right is exported to the KIF in the Add Code- window to the left. A name and a skill type is selected before exporting it.

Receiving RDF Data in the KIF

A KIF servlet receives the RDF data as a `Post` command and, using the Openrdf Java API (API, 2011), adds it to a repository.

Importing Basic Skills to RobotStudio

The **Import Skill** dropdown menu in the demonstration window shows the types that have a RAPID implementation in the KIF. The information in the menu is updated, via a SPARQL query in the KIF connection class, when the button is pressed. All existing class types, each listing the names of implemented skills are displayed in the **Import Skill** menu.

Each type lists the existing skills as shown in Figure 5.6.

When a skill is selected each input argument is to be assigned. If the argument is a tool, all existing tools in the station are listed, similarly, if the argument is a robot target, already defined targets are presented. Now if the procedure has a reference variable, *E.g.* if it is a locate skill, there is an option to save the returned variable and re-use as input to another skill. See Figure 5.7 for how the arguments are selected.

Behind the scenes, data structures mirroring the RDF representation are created. An instance of a `Skill` implementation is created, that has fields containing the BLOB, the arguments and the skillname. Each argument has in turn a name, type, and indicators whether it is a reference or if it is optional. The skill implementations are kept in a

```
@prefix demo:  <http://rosetta.skills.demo#>.
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.

@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#>.


_:node0 demo:hasName "approach" .
_:node0 rdf:type demo:Approach .
_:node0 demo:hasArgument _:node1 .
_:node1 demo:hasName "target" .
_:node1 rdf:type demo:Position .
_:node0 demo:hasArgument _:node2 .
_:node2 demo:hasName "station" .
_:node2 rdf:type demo:WOBj .
_:node0 demo:hasArgument _:node3 .
_:node3 demo:hasName "tool" .
_:node3 rdf:type demo:Tool .
_:node0 demo:hasRapidBlob "
MoveJ RelTool(target, 0,0,-200),v1000,z20,tool\\WObj:=station;

MoveL target,v300,fine,tool\\WObj:=station;
" .
```

Figure 5.5: The RDF content that is sent to KIF describing the BLOB from Figure 5.4. At the top, the prefixes are declared. The blank nodes starting with underscore, are the skill node and the argument nodes.

`Program` class and the entire instance is sent to a routine that generates robot program code.

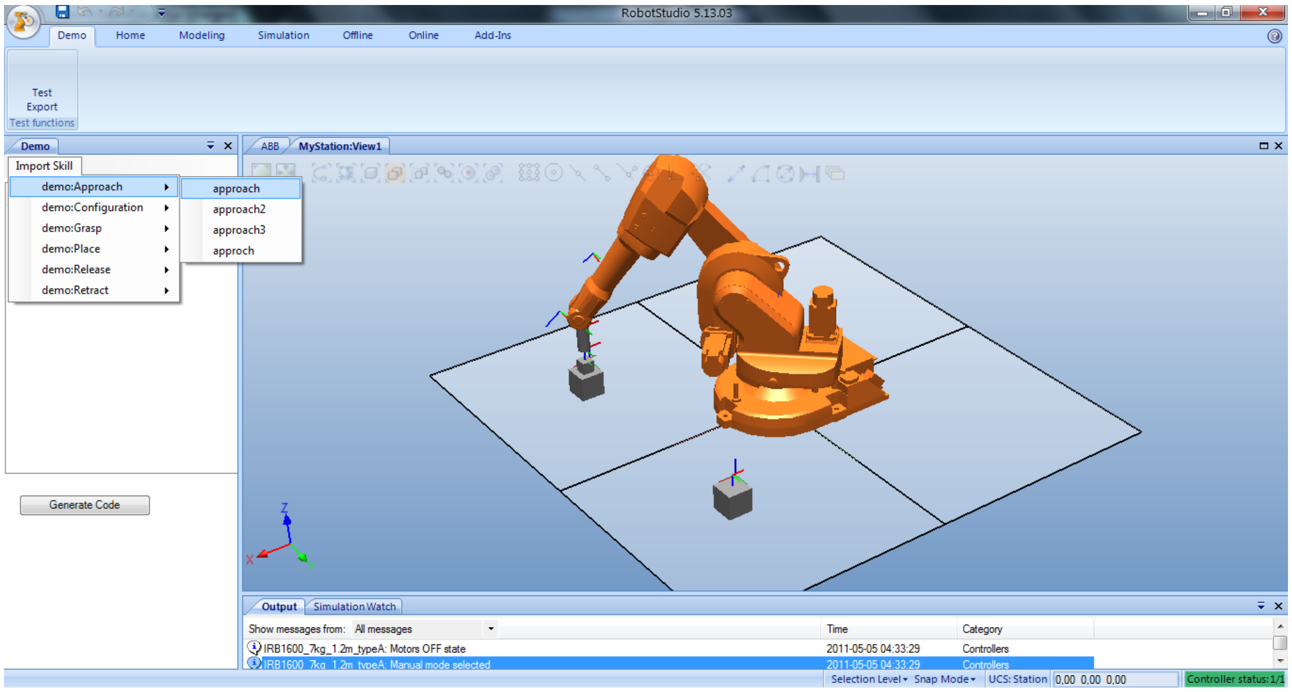
After selecting a set of skills and assigning the arguments values, RAPID code is generated by a buttom press. Each argument type is mapped back to RAPID and language specific keywords are added for each type, references, optional variables.

The code is written to a file with a main procedure that calls each skill with its specified name and arguments. The file is loaded into RobotStudio as the primary module (Module1), overwriting existing code in that module. Once the code has been added to the controller and found free of errors, simulation can be started. As long as the skills saved in the KIF are free of errors themselves and created according to the guidelines this should be done without any manual effort.

Limitations in the Demonstration

- When adding skills to the KIF administrative data such as unique IDs, time stamp or version information is omitted.
- It is not possible to update an existing skill or adding other language BLOBs, configurations, state machines, requirements or even comments.

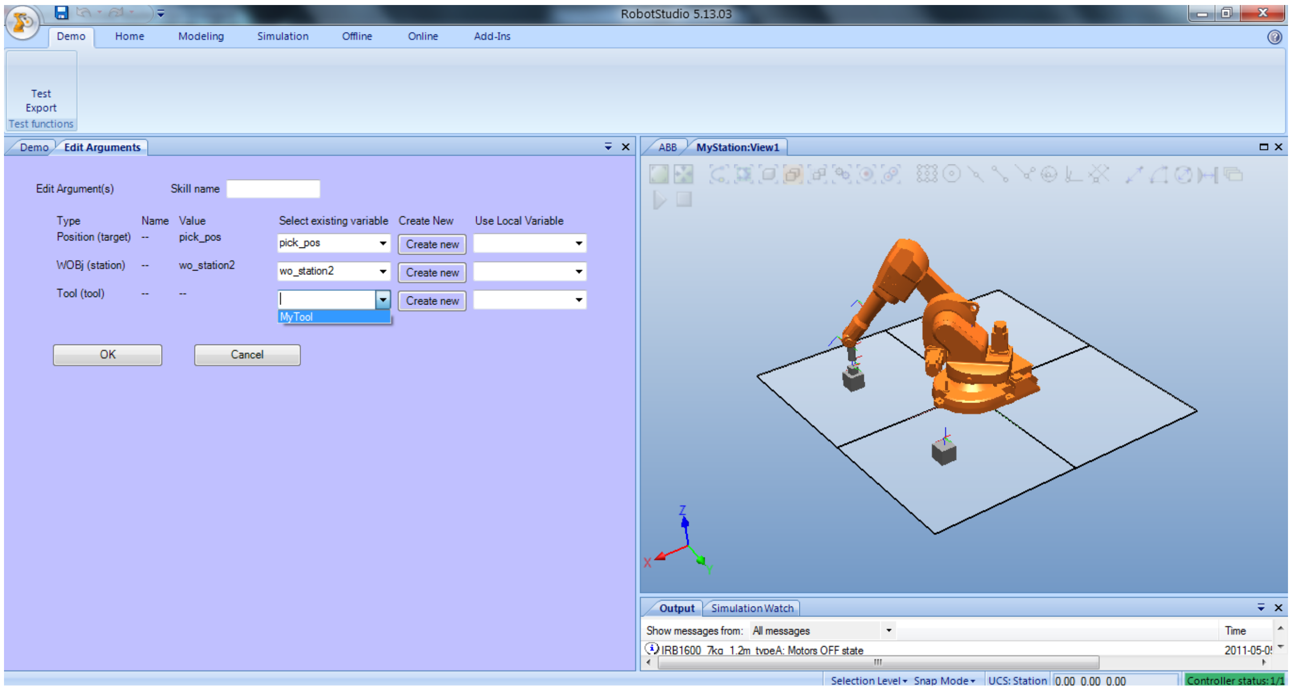
- It is not possible to create composite skills in the user interface. It would, for example, be desirable to be able to export a series of skills as a composite skill with the existing arguments as an argument mapping (described in Section 4.2.1).
- As a demonstration tool, this is not fool proof in any way, there are for example no checks that user input is valid.



Show messages from: All messages

	Time	Category
IRB1600_7kg_1.2m_typeA: Motors OFF state	2011-05-05 04:33:29	Controllers
IRB1600_7kg_1.2m_typeA: Manual mode selected	2011-05-05 04:33:29	Controllers

Figure 5.6: Screen shot of the demomonstration tool. The dropdown menu displays all classes in the KIF with associated skills (with a RAPID implementation).



Edit Argument(s) Skill name

Type	Name	Value	Select existing variable	Create New	Use Local Variable
Position (target)	--	pick_pos	pick_pos	Create new	
WOBj (station)	--	wo_station2	wo_station2	Create new	
Tool (tool)	--	--		Create new	

Show messages from: All messages

	Time
IRB1600_7kg_1.2m_typeA: Motors OFF state	2011-05-0

Figure 5.7: Screen shot of the demonstration tool. When importing a skill the arguments should be selected from existing variables in the station, such as the tools, or new temporary variables can be created.

Chapter 6

Evaluation

The results presented in the thesis are the following:

- Identifying what type of knowledge about skills to store in the framework.
- Designing an extendable structure for storing robot programs as basic skills with BLOBs of code in the KIF.
- Proposing a structure for composite skills and argument mappings.
- Demonstrating the concepts with implemented tools.
- Creating an online connection between the engineering tool and the KIF.
- Annotating LabComm parameters with quantities, units, and comments.
- Investigating how to to write re-usable robot programs.

6.1 Results

Success is a science; if you have the conditions, you get the result.

– Oscar Wilde

This report has presented three types of knowledge about skills that can be stored in the KIF:

- A state machine description that is transformed from XML to RDF.
- LabComm parameters, associated with a state machine, where the parameters are given quantities, units and descriptions.
- Procedures stored as BLOBs of robot code with associated arguments.

The RDF structure for storing the BLOBs of code as basic skills and composite skills is described in Chapter 4. To enable re-use of program code, the basic skill has a set of arguments and BLOBs of code. The composite skills contains a list of other skills and an argument mapping as described in Section 4.2.1.

Chapter 5 consists of a Java servlet and a plug-in to RobotStudio. The servlet is used for uploading finite state machines in XML-format to the KIF, visualizing the state machine and assigning it LabComm parameters. Using the same servlet, these parameters are given quantities, units and comments.

The plug-in to RobotStudio has an online connection to the KIF and can be used to export and import robot programs. The program is written with all station variables as arguments as described in Chapter 4 and, using the plug-in, exported to the KIF. Using the same plug-in, the procedures are imported, the arguments given values from the existing station and RAPID code is generated.

Figures 6.1 to 6.3 show tables of a set of standard basic, composite skill classes, and a list of generic argument types in the KIF . The hierarchy is intended to be easily extended by the users.

Skill Class	Description
Approach	Moves the tool to a position, first at a high speed and slowing down when closing the target. Typical input arguments: tool and the target position. Subclass of Motion.
Grasp	Closes the gripper. Depending on the implementation the arguments can be tool signals or the tool itself.
Locate	Locates an object and returns a position.
Motion	A motion of the tool, a superclass for *E.g.* approach and retract. Input can be tool, target location, speed etc.
Release	Opens the gripper. Similarly to grasp, the skill needs either tool signals or the tool itself.
Retract	Moves the tool away from a position. Input can be the tool to be moved, the position etc. Subclass of Motion.
Vision	A subclass of Locate. Requires *E.g.* a camera, has the target object as input and returns the position coordinates.

Figure 6.1: Plausible set of skill classes, all subtypes to the superclass Skill.

Composite Skill Class	Description
Assembly	A set of pick and place operations.
Pick	A list of basic skills, *E.g.* locate, approach, grasp and retract.
Place	A list of basic skills, *E.g.* locate, approach, release and retract.

Figure 6.2: Classes of basic and composite skills, all subclasses to the superclass composite skill.

Argument Type	KIF Representation
Tools	rosetta:Tool (or a subclass).
Targets	rosetta:Position or a subclass such as rosetta: Destination.
Signals	A digital signal is rosetta:DigitalIn, rosetta:DigitalOut or rosetta:DigitalInOut and equivalent for analog signals.
Work Objects	rosetta:WorkFrame
Numerical values	xsd:double
Boolean values	xsd:boolean

Figure 6.3: The KIF representation of typical arguments. Here the namespace rosetta stands for http://asimov.ludat.lth.se/rosetta.owl# and xsd for http://www.w3.org/2001/XMLSchema#.

Chapter 7

Epilogue

7.1 Discussion

Robots... I think that is a hot topic.

– Bill Budge

This work is important because it enables re-use of knowledge about implemented skills. The use of already defined skills from the database, simplifies the robot programming, and makes it possible for the robot to use stored parameters to speed its learning process.

The generic representation, with standard units for LabComm samples and platform independent arguments for programs, enables reasoning upon the information. With the unit ontology, LabComm samples can be compared in quantity and kind. The integration of the NASA unit ontology is important because it makes the variables self-descriptive, which for example gives built-in information on how to convert between units and unit systems[1]. The user-interface for adding human readable comments describing the parameters is needed until the world reaches the utopian stage of total automation.

Even though the BLOBs are platform dependent, the generic argument makes it possible to share the BLOB between versions using the same robot language. For example six- and seven axed robots, where position is represented as axis angles, can still use the same code when the position is a procedure argument.

Those familiar with the magic behind compilers should have noted the resemblance between the RDF representation for storing BLOBs and how programs are represented in an abstract syntax tree, as well as the type mapping and syntactic flags in the implementations. The RDF representation works as a little piece of an abstract syntax tree that is common for robot languages.

[1]Noteworthy is that NASA lost the multimillion dollar Mars climate orbiter due to metric/imperial confusion (http://mars.jpl.nasa.gov/msp98/orbiter/).

The standard skills will have multiple language implementation, so a user-defined sequence of actions can be implemented on several platforms, each using the suitable language implementation. The generic arguments can also be used when algorithmically generating skill sequences from information about the existing station.

Composite skills are transferable between platforms, as long as the listed skills have the suitable language implementations. The argument mapping in the composite skill makes it possible to load an entire list of skills without having to specify the arguments used in each skill separately, making re-use very user-friendly**.**

Of course, the creation, storage and the re-use of the program can be separated in both space and time.

The idea of connecting robots to a collective memory, where they share knowledge about tasks and the world, is a hot topic. Hence there are other research projects similar to ROSETTA. RoboEarth uses OWL to describe action recipes as well, which means that the research community pulls in the same direction. Other interesting ongoing projects are GeRT and Proteus (Proteus, 2011), a project for transferring knowledge within the French robot community . The sharing and re-use of knowledge and learned information between the projects is still in the design phase.

7.2 Conclusions

A conclusion is the place where you got tired of thinking.

– Martin Henry Fischer

The purpose of this Master's thesis was to investigate what knowledge about skills to integrate in the KIF and how to represent it. The aim was to address finite state machines with associated parameters and program procedures. I have 1) created a servlet that transforms files describing finite-state machines and LabComm samples to RDF triples and integrated a the NASA unit ontology for annotating the parameters and 2) designed an RDF structure for storing BLOBs of robot code with generic arguments and a tool for demonstrating the concept.

7.3 Future Work

None, because the robots stole the jobs.

– Anonymous

These intelligent robot systems will be a paradigm shift in European industry, however, there are a few steps left on the ladder. Future work within the ROSETTA project involves agreeing upon a definite structure for representing the the finite state machine and the skill ontology.

Another aspect is to connect the chain of knowledge flow from the Engineering Tool to the KIF, from KIF to the ROSETTA controller and from the controller to the robot platform. This is not merely a one way pipeline, the learned parameters and log data from the execution should be fed back from the robot to the ROSETTA controller where the relevant data is extracted, generalized and stored in the KIF for re-use. Future work includes writing a client for the ROSETTA controller and connect it to the KIF to retrieve information about skills. Then the program can be written in the engineering tool, uploaded to the KIF, downloaded to the ROSETTA controller and used in a robot, thus connecting the entire chain from modelling to execution.

Since the KIF is a library, it should be populated with "template" knowledge of tasks and skills. There should be user-friendly tools for integrating and extracting knowledge from the KIF as well as for intuitive robot programming. The focus in the ROSETTA project lays on generic knowledge representation and information exchange rather than on developing end-user applications.

Considering the amount of grants assigned to research projects like this, it is only a matter of time before a new industrial revolution takes place, in which the robot workers of the world unite their minds in a collective memory and brain.

Bibliography

Allemang, D. and Hendler, J. (2008). *Semantic Web for the Working Ontologist modelling in RDF, RDFS and OWL*, pages 79–103. Denise E. M. Penrose.

API, O. R. (2011). http://www.openrdf.org/doc/sesame/api/. Last access on April 14, 2011.

Bruyninckx, H. (2010). Definitions of task, skill, motion and action.

FRIDA (2011). Abb concept robot. http://www.abb.com/cawp/abbzh254/8657f5e05ede6ac5c1257861002c8ed2.aspx. Last access on April 24, 2011.

GeRT (2011). http://www.cs.bham.ac.uk/research/groupings/robotics_and_cognitive_architectures/projects/gert/. Last access on April 14, 2011.

Jacob Persson, Axel Gallois, e. a. (2010). A knowledge integration framework for robotics.

JGrafChart (2011). http://www3.control.lth.se/user/karlerik/Grafchart/JGrafchart.html. Last access on April 13, 2011.

LabComm (2011). http://torvalds.cs.lth.se/moin/LabComm. Last access on April 13, 2011.

Nasa unit ontology (2011). http://www.qudt.org/qudt/owl/1.0.0/qudt/. Last access on March 22, 2011.

Proteus (2011). http://www.bourges.univ-orleans.fr/CAR08/14_CAR08-Patin_PROTEUS.pdf. Last access on April 14, 2011.

RDF Schema (2011). http://www.w3.org/wiki/RDFS. Last access on March 9, 2011.

RoboEarth (2011). http://www.roboearth.org/. Last access on April 14, 2011.

ROSETTA Consortium (2011). http://fp7rosetta.org/?q=node/4. Accessed February 28, 2011.

ROSETTA Project information (2011). http://www.fp7rosetta.org/?q=node/2. Accessed February 28, 2011.

ROSETTA Project presentation (2010).

Saxon (2011). http://saxon.sourceforge.net/saxon7.7/using-xsl.htmll. Last access on April 14, 2011.

SPARQL Query Language (2011). http://www.w3.org/TR/rdf-sparql-query/. Last access on March 9, 2011.

Web Ontology Language, OWL (2011). http://www.w3.org/TR/owl-ref/. Last access on March 9, 2011.