MASTER THESIS

Modular and Adaptive Assistance System for Manual Assembly – Engineering a Semantically Described CPS Module

MASTER THESIS

Modular and Adaptive Assistance System for Manual Assembly – Engineering a Semantically Described CPS Module

Bearbeiter: Amita Singh
Betreuer: Dipl.-Ing. Patrick Bertram
Prüfer: Prof. Dr.-Ing. Martin Ruskowski

Erklärung

Hiermit erkläre ich, daß ich die vorliegende Studien-/Diplomarbeit selbständig und ohne unerlaubte, fremde Hilfe angefertigt habe. Textabschnitte oder Bilder, denen fremde Quellen zugrunde liegen, enthalten Hinweise und sind im Literaturverzeichnis kenntlich gemacht.

Kaiserslautern, den 30. August 2017

Contents

Abstract

Assistance systems are used in production facilities to help workers during assembly. An assistance system consists of a central system and one or more CPS modules. CPS modules collect data from environment through various sensors and provide information to the central system. However, to fully extract the benefits of an assistance system, it is important that CPS modules can exchange data between them. This work provides a framework to design CPS modules and describe it semantically for interoperability across modules. The framework is then used to show how to design a weighing module for an assistance system, how to semantically describe it, and how to allow interoperability with other modules.

1. Introduction

An ever growing catalogue of products by companies, which may be relatively minor customization for different consumer segments, has trickled down to the number of variants being produced by the production facility. On the one hand, increasing global competition has led to low costs and shorter life-cycles and, on the other hand, changing demographics in many parts of the world has resulted in increasing labour costs which makes it difficult for manufacturers to compete in the market.

Also, the fluctuating demand makes it extremely difficult for the automotive sector to have a static set up as it is incapable of accommodating variations in production line-up. Automation is one plausible solution to reduce the labour costs in production. However, existence of a high number of variants makes it difficult for engineers to automate the process since setup of the facilities would require frequent change. This gave rise to the demand of reactive and proactive system that can respond to dynamic changes in the markets and has led to a number of changes in production facilities all across the world.

The emergence of multi-agent cyber-physical systems is the result of this demand. These agents help the production system be modular, decentralized, flexible and changeable in order to adapt to changes in product demand.

Since these agents cannot manage and resolve unforeseen situations in a production facility, they can be used to help workers in their task. An assistance system is being developed at *SmartFactory*^{KL} to help workers in deciding which step should be taken next in the assembly process. An assistance system is a multi-modal CPS which collects data with the help of sensors in different forms and decides the next step based on this information.

The aim of my thesis is to study the process of attaching a weighing module to the assistance system and develop a framework that helps interoperability of data among CPS modules. This can be seen as providing the guiding principles for allowing data collected by one sensor to be used by other modules.

One way of facilitating exchange of data is through ontologies. A framework for designing ontologies of the attached modules and exchanging data is described next. This framework starts with defining the parameters to gauge the effectiveness of adding a module, followed by defining the scope of modules and their intelligence. In the next step, an information model is developed and an ontology is created. In this last step, communication between the central system and modules is described.

The thesis is divided into 5 main chapters. Chapter 2 describes and motivates the research problem, i.e. interoperability between CPS modules. Chapter 3 describes the key ideas behind smart factories and assistance systems and discusses the state-of-the-art. Chapter 4 describes the methodology adopted to solve it and, the detailed solution is laid out as a framework in Chapter 5. Chapter 6 deals with the design choices which need to be made during implementation of such a system. The thesis concludes with the limitations of the suggested solutions and with discussion of the possible future work in this area.

2. Research Topic

perez2015cpps in their paper talk about adaptive production systems where workers can perform work aided by machines. These adaptive systems treat automation as an enhancement of worker's physical and cognitive capabilities. Manual assembly stations being developed by smart factories are one such example of adaptive automation.

The assembly stations are equipped with assistance systems in order to ease the assembly process for workers. These assistance systems help workers by showing the next step to be taken in the process. It consists of a central adaptive system and CPS modules thus making it both adaptive and modular. These modules measure different aspects and parameters of environment, for example, a hand tracking module or an eye tracking module can track the position of worker, a weighing module can give information about number of parts, a RFID tag reader can track the status of products, etc. The central system can be made more *aware* of the worker's environment by addition of such CPS modules. This will allow it to help workers more effectively.

The modules being developed follow the principle of plug-and-produce which requires them to be smart and adaptive themselves as well. Plug-and-produce is one of the key paradigms of Industry 4.0 as it facilitates the addition or removal of CPS modules as required for the automation of a process. **quint2016system** talk about such an assistance system developed by *SmartFactory*^{KL} and describe its system architecture. However, to fully exploit the benefits of such an assistance system, it is important that one CPS module can access data provided by other modules.

This is the gap I intend to fill with my thesis by developing a semantic model for different modules attached to an assistance system which may have heterogeneous data. For modules to be able to understand and exchange data, it is necessary that they have the same vocabulary. For this purpose, developing a semantic description of the modules is necessary. Once the semantic description of these modules is set, they can be developed in-house or independently by third-parties. To the best of my knowledge, this area has

not been explored by researchers, specially in the context of cyber physical production systems.

In this thesis, a concept for designing modules which can facilitate interoperability of data will be proposed. Further, a framework will be developed to help engineers semantically describe a CPS module and provide guidance to engineers on how to add modules to an assistance system. Some practical suggestions regarding design choices during implementation (i.e. both during semantic description and development of the prototype) will be given while using a weighing module as a use case. A part of the implementation of a weighing module will be covered. Other avenues that this thesis opens will also be discussed briefly as future work.

3. State-of-the-art

The vision of CIM (Computers in Manufacturing) of creating completely automated factories could not be realized due to the complexity involved in production processes [**zuehlke2008smartfactory**] The effort to implement CIM made it clear to engineers that completely automated factory is not a plausible solution as per the state-of-the-art. Humans are an indispensable part of production systems but automation at different stages of product is a necessity and a practical approach to the problems of increasing product variants, reducing product life-cycle and rising labour costs [**romero2015towards**].

Production facilities are focusing on cyber-physical systems (CPSs) that can interact with human through many modalities. CPSs are a combination of interacting embedded computers and physical components. Both computation and physical processes work in parallel to bring about the desired output. Computers usually monitor the physical processes via sensors in real-time and provide feedback to actuators [**lee2008cyber**; **jazdi2014cyber**]. A CPS consists of one or more micro-controllers to control sensors and actuators which are necessary to collect data and interact from its environment. These systems also need communication interface to exchange data with other smart devices and a cloud. According to **jazdi2014cyber** data exchange is the most important feature of cyber physical systems. CPSs connected over internet are also known as Internet-of-Things.

lee2015cyber in their paper proposed a 5C level architecture which defines functionalities of CPS very adequately. The levels of **5C** architecture are: (i) **smart connection** which can be a sensor network, (ii) **data-to-information conversion level** in which meaningful information is inferred from data collected by various sensors, (iii) **cyber level** which connects all machines to each other and transfers information to other machines, (iv) **cognition** which deals with proper presentation of the acquired knowledge and, (v) **configuration level** which takes feedback from physical and cyber levels and acts as a supervisory control to make machines self-configurable and adaptive.

With the increasing sophistication of actuators and sensors available in the market, availability of data has increased many folds. The CPSs used to create flexible and re-configurable production systems called Cyber-Physical Production Systems (CPPSs). CPPSs are build on the principle of modularity and decentralized control. Thus, these modules are loosely coupled with each other.

3.1 Industry 4.0

Industrial revolutions occur with new inventions that lead to the change of infrastructure and economy. The first industrial revolution is marked by the invention of coal and steam engine which led the transition from hand tools to machine tools. Second Industrial Revolution is characterized by centralized electricity, improved communication media, rail-road networks, improved water supply and use of oil and fossil fuel. Production line for mass production were introduced in this era. This industrial revolution was built on fossil fuels, but the need for new energy resources was eminent due to depleting resources and dangerous levels of CO2 emissions from factories. Advent of internet and renewable energies facilitated a new infrastructure ushering the Third Industrial Revolution [**rifkin2011third**]. Computers were introduced on factory floors that propelled automation in manufacturing sector. The Fourth Industrial Revolution focuses on modular, decentralized cyber-physical systems which can interact with each other and humans in real time, thus resulting in flexible production systems.

The term Industry 4.0 refers to the fourth revolution in manufacturing industry. The concept of Industry 4.0 was motivated by smart, modular and adaptive production systems with decentralized control [**hermann2016design**]. Its vision is to bring automation in the field of production and help combat the problems of increasing catalogue and labour costs. The information and communication technologies have trickled their way down to the production systems, paving the way for monolithic production systems to become modular and have decentralized control architectures. It is one of the most significant directions in computer science, information & communication technologies and manufacturing technologies. Industry 4.0 is characterized by a paradigm shift from centrally controlled to decentralized processes.

The core idea of Industry 4.0 is to integrate information technologies to operate business and engineering processes in a flexible and efficient with constantly high quality and low cost. The main feature of Industry 4.0 is horizontal, vertical and end-to-end integration. Horizontal integration facilitates inter-corporation collaborations of value

networks, vertical integration enables information flow through hierarchical subsystems in a production system resulting in more flexible and re-configurable production systems and end-to-end engineering integration enables engineers to foresee the effect of product design in production and services through software tools [**wang2016implementing**].

3.2 Smart Factory

Smart factories aim at the development, application and distribution of innovative, industrial plant technologies and create the foundation for their widespread use in research and practice [**zuehlke2010smartfactory**]. Research therein generally focuses on the use of innovative information and communication technologies in automated systems and deals with design challenges of such systems.

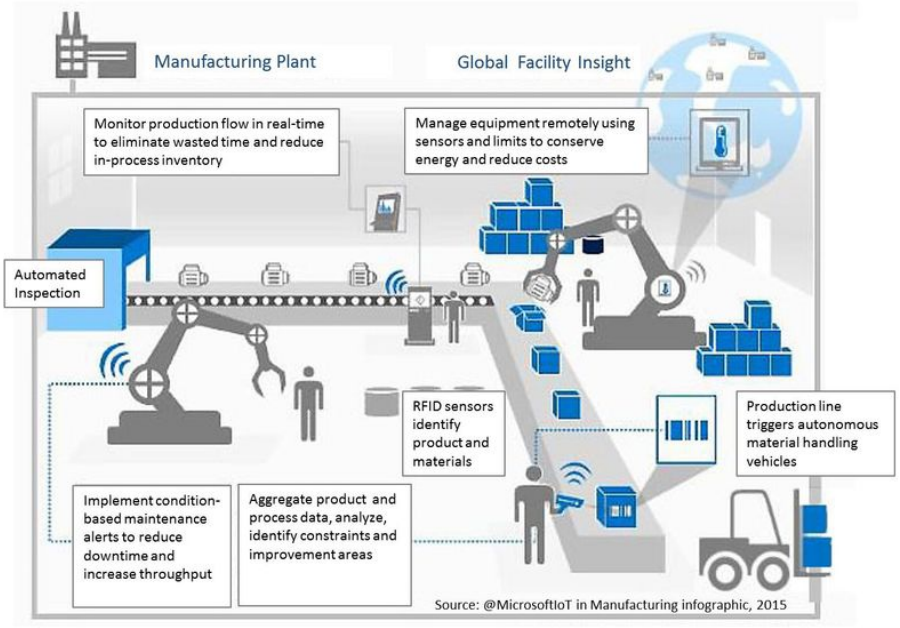

Figure 3.1: Shows an example of product and data flow in a Smart Factory. Product carries RFID tag which contains product information. Inspection and inventory triggers are automated. Source: MicrosoftIoT in Manufacturing Infographics, 2015.

They use smart, decentralized, modular and adaptive CPS. These CPSs reduce complexity in the production system by strict modularization on all levels of automation, decentralized

control architectures and loose coupling between modules. These modules are thus robust, adaptive and can self organize by negotiating among themselves. Smart factories, thus, can be seen as a hybrid production facility.

As shown in Figure 3.1, smart factories differ from traditional factories in their reconfigurable layout in contrast to the fixed line where each machine is tailored for a specific job. In a smart factory, machines are smart CPS modules which can be reconfigured as per requirements. The machines and information systems in smart factories are extensively connected and have decentralized control architecture which help CPS modules to negotiate with each other to organize to cope with system dynamics. Products in smart factories follow dynamic routing as opposed to fixed routing in case of traditional production lines. On the contrary, machines in traditional factories have a specified role and they perform functions assigned to them. Smart factories also make it possible for smart devices to collect data. This data can be processed for insights and training purposes, whereas in traditional factories, the data, if collected, is not well connected to cloud and thus is not efficiently used [**wang2016implementing**].

3.3 Human-in-the-loop

Computers in Manufacturing (CIM) established that it is important that the CPS systems developed should help humans instead of trying to replace them because with the current state of the technology it is difficult to replicate human cognitive skills. This has lead to human-centric workplaces. This approach to production was coined human-in-the-loop.

In complex production scenarios, the symbiotic man-machine systems are the optimal solution. This change in the nature of human-machine led to the paradigm shift from independently automated and manual processes towards a human-automation symbiosis called human cyber-physical systems. These systems are characterized by collaborative effort of workers and machines and aim at assisting workers being more efficient and effective [**romero2015towards**]. These systems are based on a trusting and interaction-based relationship, which has human supervisory control and human situation awareness, leading to adaptive automation to improve knowledge of worker and help the process. Human-system interaction is an indispensable part of the production systems and acts as an enabler of the intelligent decision making process [**gaham2015human**].

Since operators are to be kept in the production system, as discussed, it is important that they are equipped with smart devices which can help them in their work. This evolution is called Operator 4.0 and is characterized by smart and skilled operators who can perform

work aided by machines. The earlier generations are Operator 3.0 where humans work in collaboration with robots and machines, Operator 2.0 deals with humans assisted by computer tools like CAx tools and NC operating systems, Operator 1.0 represented manual and dexterous work aided by only mechanical tools [**romero2016towards**].

3.4 Manual assembly stations

Earlier, automation was considered a plausible solution to the problem of increasing variants and labour costs. However, it was soon clear that fully automated production facilities were not the solution in the current scenario because such facilities would be inflexible and, thereby, expensive. Neither traditional nor fully automated systems can respond effectively and efficiently to dynamic changes in the system [**leitao2009agent**]. Hence, workers should be assisted, as needed, in their work thus including automation with human aptitude as a trouble shooter. Manual assembly stations with assistance systems are developed based on this concept.

Traditionally, a manual assembly line consists of a sequence of stations. These stations are modular units, one in the chain of many automated/semi-automated stations. Products are assembled by workers at each station. Usually, production facilities are one piece flow. Depending upon the assembly plan of a product, one or more processes can be performed on a station.

Figure 3.2 shows a schematic description of such an assembly station. These stations are equipped with different visualization techniques and sensor technologies. Visualization techniques, like projectors and smart glasses as shown in Figure 3.2, help workers during assembly process by displaying instructions. Interactive screens can also be installed at assembly stations using which workers can interact with stationery computers when required. Assembly stations have areas dedicated for storing tools and parts used during assembly. Sensors can be employed to track usage of tools and control inventory of parts. RFID readers are installed on products and to know the current status of products in addition to the tools and parts as in the traditional workstations.

3.5 Assistance System

Assistance systems can play an important role in supporting humans during complex tasks [**gorecky2011cognito**]. Production facilities are focusing on CPS that can interact

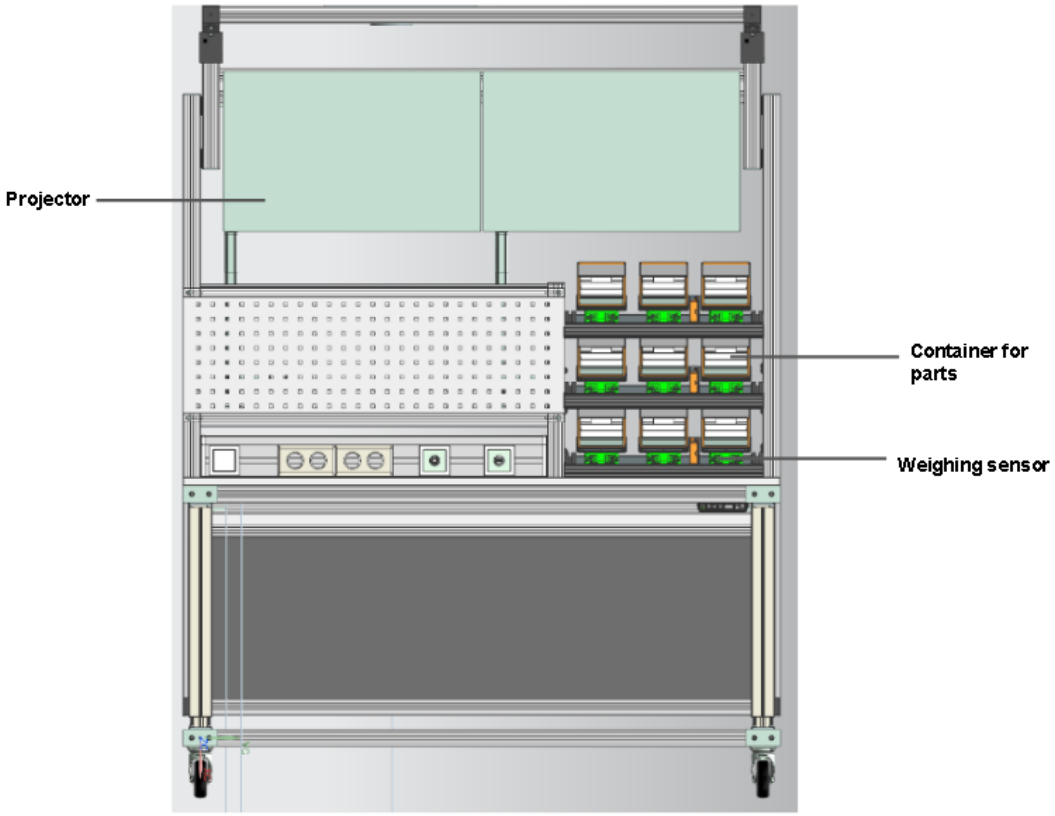

Figure 3.2: Shows schematic description of a manual assembly station. Assembly station may employ different visualization and sensor technologies, for example projectors and weighing sensors, to track the assembly process and help workers by displaying instructions. Source CAD model developed by SmartFactory^{KL}.

with human through many modalities. **pirvu2016engineering** talk about the engineering insights of such human centered yet highly automated cyber-physical system: keeping adaptive control in mind, a cognitive assistance and training in manual industrial assembly. The aim of such a system is to design a mobile, personal assembly work station which assists the worker in task solving in real time while understanding and inducing workflows. Standardized abstractions and architectures help the engineers in the design phase by reducing the complexity involved in building such systems [**kolberg2016cyprof**].

Assistance system derives its principles from the Operator 4.0 principle where workers are provided machines to aid their work. It helps workers by reading the product status available with products in machine readable format, collecting other information about

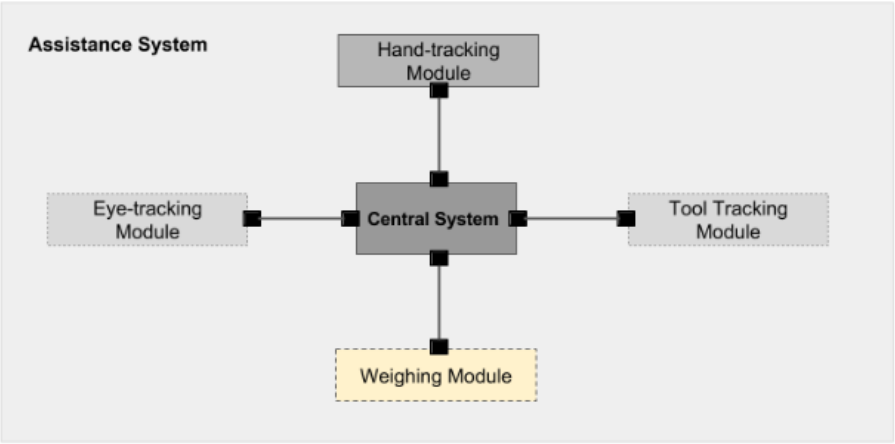

Figure 3.3: Shows schematic description of an assistance system. Hand-tracking module has been developed by SmartFactory^{KL}. Weighing module (highlighted in yellow) will be developed as a use case of this thesis. Eye-tracking and tool tracking modules are stated as examples which can be further developed.

the environment and helping the worker to decide the next step to be taken based on the information it receives. Hence, this system can be seen as a context aware human-centric cyber-physical system. As shown in Figure 3.3, this system consists of a central system and one or more CPS modules.

These CPS modules are built on the principle of plug-and-produce. The analogy is drawn from plug-and-play concept in computer science [**arai2001holonic**]. Plug-and-produce means a smart device can be easily added or removed, replaced without disrupting functioning of the system. The system should continue working while a CPS module is being added or removed. Additionally, the system should be able to recognize the newly added CPS. This process is different from the traditional processes in which systems need to be reprogrammed and machines are stopped for reconfiguration. Time taken in the complete process is counted as downtime. Similarly, in case of plug-and-produce systems maintenance can be done by removing only the required CPS module while the complete system continues working.

For this purpose, each CPS module should have its own environmental information and it should provide this information to the system to which it is being attached [**arai2001holonic**]. This gives central system the leeway to reconfigure and requires CPS modules to be smart and adaptive which demands CPS modules to have certain level of intelligence.

An assistance system with a hand-tracking module is described and implemented on an

assembly station by **quint2016system**. The paper describes an information model that illustrates the employed terminology and the system architecture of assistance in manual tasks. As shown in Figure 3.4, system architecture has four main components: *Messaging Server*, *Workflow Model*, *Detection* and *Views*. A *Messaging Server* connects all components and allows them to exchange messages. A *Workflow Model* contains states and transitions for changes between states for a particular assembly task. A *Trigger* monitors different data sources (cameras, buttons) for events and *Views* display instructions based on the current state of the assembly task. The system employs stationery computers, tablets and smart glasses to display information to the worker as is shown in Figure 3.4. If *Workflow Model* receives a relevant *Trigger*, it changes the internal state machine to a valid next state and the new state is broadcast to all *Views*. *Views* show instructions based on the current state of the assembly.

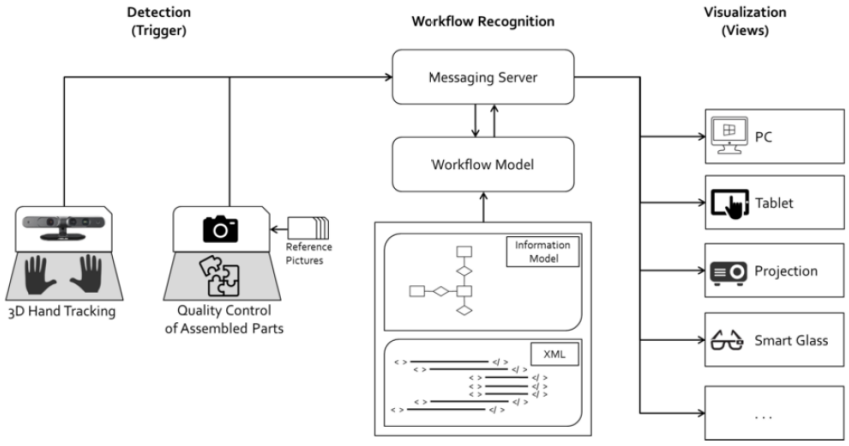

Figure 3.4: Shows schematic description of system architecture of assistance system. It contains four main components at the highest level: `Trigger`, `Messaging Server`, `Workflow Model` *and* `Views`. *Source [***quint2016system***].*

Further, various CPS modules can be attached to the assistance system like weighing module, eye-tracking module or tool usage module. In order to fully exploit the benefits of the CPS modules data exchange between these modules is necessary. **quint2016system** point out that standardized semantic self-description is required to facilitate interoperability between modules.

Recently, there have been some efforts towards discussing the need of bringing more semantics and data-driven approaches to Industry 4.0. **cheng2015semantic** identify varying degree of semantic approach and further provide guidelines to engineers to select

appropriate semantic degree for different Industry 4.0 projects. **wahlster2014semantic** talk about the importance of semantic technologies in mass production of smart products, smart data and smart services. Semantic service matchmaking in cyber-physical production systems is presented as a key enabler of the disruptive change in the production logic for Industry 4.0. **obitko2015big** introduce the application of semantic web technologies in handling large volumes of heterogeneous data from distributed sources. **grangel2016towards** describe an approach to semantically represent information about smart devices. The approach is based on structuring the information using an extensible and light-weight vocabulary aiming to capture all relevant information.

The aim of this work is to ensure that CPS modules attached to assistance system can share data and interoperate between themselves. Ontology is an efficient way of interoperability between heterogeneous information systems [**cullot2007db2owl**]. Further, **chen2003ontology**, **grangel2016towards**, and **semy2004toward** talk about ontologies being the key requirements for building pervasive context-aware systems in which independently developed sensors, devices and agents are expected to share contextual knowledge and to provide relevant services and information to users based on their situation.

3.6 Ontologies

An ontology is an explicit formal naming, definition of entities, relationship between entities, and constraints within a domain in order to have common understanding and information for participating people and machines [**cullot2007db2owl**]. It helps in explicitly stating the assumptions and analyzing the "domain knowledge": this helps people from different disciplines in understanding the relationships and constraints of a given domain.

The idea behind using ontologies in the thesis is to be able to formally and explicitly define the structure and data of a CPS module. Further, it can act as a basis for providing and fetching data from other modules attached to the central system. As the assistance system consists of a central system and CPS modules, it is necessary to understand how the ontologies for different CPS modules would be created and maintained. **hoehndorf2010upper** talk about upper ontologies and its usefulness in facilitating domain-specific ontologies. Upper ontologies can be created to provide a framework for domain specific ontologies of CPS modules. Upper ontologies will have common knowledge base and logic which module ontologies use.

3.6.1 Upper Ontologies

Upper ontologies are high-level, domain-independent ontologies, providing a framework by which disparate systems may utilize a common knowledge base and from which more domain-specific ontologies may be derived [**semy2004toward**]. Thus, upper ontologies facilitate interoperability between domain-specific ontologies by the virtue of shared common terms and definitions [**hoehndorf2010upper**]. The concepts expressed in upper ontologies are intended to be basic and universal concepts to ensure generality and expressiveness for a wide area of domains. An upper ontology is limited to concepts that are meta, generic, abstract and philosophical. Standard upper ontologies are also sometimes referred to as foundational ontologies or universal ontologies. They contain definitions and axioms for common terms that are applicable across multiple domains and thus provide semantic integration of domain ontologies. Since they provide well defined primitives, they help in resolving conflicts that may arise while extending the categories and provide common foundation for both existing and new ontologies.

On the other hand, domain ontologies have specific concepts particular to a domain and represent these concepts and their relationships from a domain-perspective. Multiple domains can have the same concept but their representation may vary due to different domain contexts. Domain ontologies inherit the semantic richness and logic by importing upper ontologies.

beisswanger2008biotop describe use of upper ontology for sharing vocabularies needed for consistently expressing meta-data in terms of semantic annotations and providing principled forms of conceptual inter-linkage between data. Another important feature of upper ontology is the structure that they impose on the ensuing ontologies: they promote modularity, extensibility, and flexibility. According to **semy2004toward**, upper ontologies can be built using two approaches: top-down and bottom-up. They discuss benefits and limitations of both approaches. In a top-down approach domain ontology uses the upper ontology as the theoretical framework and the foundation for deriving concepts [**semy2004toward**]. In a bottom-up approach, new or existing domain ontologies are mapped to an upper ontology. This approach also benefits from the semantic knowledge of upper ontology but the mapping can be more challenging as inconsistencies may exist between the two ontologies. For example, two teams may have different vocabulary for a similar semantic variable. In this case, mapping the two ontologies to an upper ontology would have inconsistencies. These inconsistencies are resolved as and when needed. However, usually a combination of both approaches is used to design upper ontologies.

The solution proposed to the problem of interoperability across modules relies heavily on the idea of upper ontologies. Upper ontology starts with defining a set of high level entities and then successively adding new content under these entities [**niles2001origins**]. The solution incorporates both the top-down and bottom-up approaches. The inevitable incompatibilities while making new ontologies are resolved as they are encountered. Depending on the need entities are added to the high level ontology.

In an ideal situation, eventually a stage should be reached where the demands of both high-level and low-level ontologies are satisfied. This resulting ontology may not contain all the possible high-level definitions and axioms but it should be comprehensive enough to attach to other domain-specific ontologies [**niles2001origins**].

Ontologies were developed for the use on Semantic web. Semantic web is an extension of web that help in semantically structuring data. The aim of Semantic web is to help software agents interact and share information over the internet. This is done by encoding the data in a machine interpretable language using constraints defined in the domain ontology. This lets software agents locate resources to extract and use information on the web. This differentiates ontologies from other traditional languages, like UML and SysML, used to describe software structure.

3.6.2 Mid-level Ontologies

A mid-level ontology act as a bridge between basic vocabulary described in the upper ontology and domain-specific low-level ontology. Generally, mid-level and upper ontologies are intended to provide mechanism to map concepts across domains. Mid-level ontologies may provide more concrete representations of abstract concepts found in upper ontologies. This category of ontologies may also encompass terms and definitions used across many domains but which do not qualify as concepts of a particular domain-specific system. They are also known as utility ontologies.

Figure 3.5 shows an example of upper, mid-level and domain ontologies. Most general vocabulary and concepts regarding `Process` and `Location` are defined in the upper ontology. Mid-level ontology is used to describe location in detail and defines variables pertaining to `Geographic Area of Interest`. The domain ontology extends from mid-level ontology and further defines `Airspace` and `Target Area of Interest` separately.

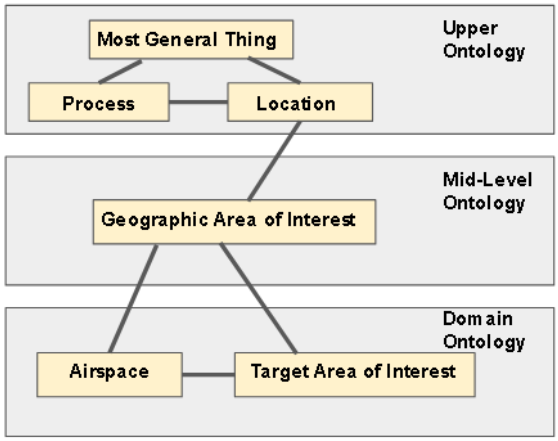

Figure 3.5: Shows a graphical representation of an example of `Upper`, `Mid-level` *and* `Domain ontology` *levels. Upper ontologies provide framework for disparate systems, domain ontologies represents concepts particular to a domain and mid-level ontologies act a bridge between upper and domain ontologies. Source: [***semy2004toward***]*

3.6.3 Representation of Ontologies

Ontologies are expressed as an abstract language: Web Ontology Language (OWL). However, to understand the reasons behind its structure, it is important to look at their representation and development. OWL has been developed on the basis of several layers of underlying infrastructural layers shown in Figure 3.6. The syntactical layer is a serialization layer and can be defined by XML or other markup languages. World Wide Web Consortium (W3C) recommends XML/XML Schema, JSON, N-Quad and Turtle for this purpose. Resource Description Framework (RDF) describes how to express relational data in triples, RDF Schema (RDFS) adds more structure to RDF to make it more human/real-world-modeling friendly and, finally, Web Ontology Language (OWL) adds vocabulary to allow reasoning with and exchanging of knowledge within a domain. RDF, RDFS and OWL are all W3C recommendations for knowledge representation in ontology building. This section will only touch upon the basics necessary to understand the concepts required for the solution proposed later in the thesis. Readers are encouraged to know more about the representation of ontologies from the W3C recommendations [**w3c2009owl**; **w3c2014rdf**; **w3c2009xml**].

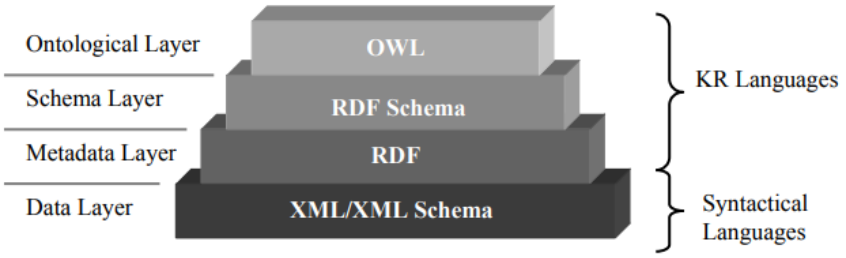

Figure 3.6: Shows four level modeling framework of ontologies. `XML/XML Schema` *provides syntax of encoding text whereas* `RDF`, `RDFS` *and* `OWL` *are used for knowledge representation. Source:* **atkinson2005detailed**.

XML / XML Schema

eXtensible Markup Language is a format to encode any structured data in a way that it is readable by both humans and machines. XML Schema defines how to formally describe an element in XML. XML/XML Schema provides syntactical constraints for knowledge representation languages.

An example XML snippet may looks like Listing 3.1. A `tagName` can be interpreted as class and an attribute as the *property* of the class. The `data` is used to give more detailed structural *content* of the class and its relationship with other classes (i.e. nested tags). The language can be eXtended by defining new tags (i.e. classes), properties (i.e. attributes) under new name-spaces (i.e. `ns`).

```
<ns:tagName  ns:tagAttribute1="attr1-value"
             ns:tagAttribute2="attr2-value">
  data
</ns:tagName>
```

Listing 3.1: An example XML snippet. The `ns` *stands for* namespace, *which is how the language can be extended. Also, the* `data` *can again be one or more tags, thereby allowing for expression of nested structures.*

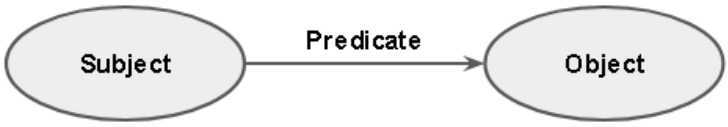

Figure 3.7: Shows schema of a triple. A triple consists of a `subject`*, a* `predicate` *and an* `object`*. Each such statement in RDF also has a unique IRI (see Section 3.6.8) associated with it, which further has three statements associated with it, identifying the subject, predicate and object of the statement (reification).*

RDF

The Resource Description Framework (RDF) is a general-purpose language for representing information in the Web.

RDF is a data model which serves to link all RDF-based languages and specifications. The abstract syntax is very simple: RDF graphs are sets of `subject-predicate-object` triples, where the elements may be IRIs (see Section 3.6.8), blank nodes, or typed literals. Example of a triple is shown in the Figure 3.7. It is easy to see how such triples can be used to describe the resources and their inter-relationships. A RDF database is an organized collection of RDF graphs related to a certain domain. Such a database forms the raw fodder for ontologies and allows exchange of data while still preserving the semantic meaning associated with them. RDF specification defines a vocabulary for this purpose under its own namespace `rdf`.

For example, a person named John is a teacher, likes music, plays football and has a bicycle. This information will be seen as Figure 3.8 for an OWL ontology and stored in the form of following triples in database.

- {John, isA, Teacher}
- {John, likes, Music}
- {John, plays, Football}

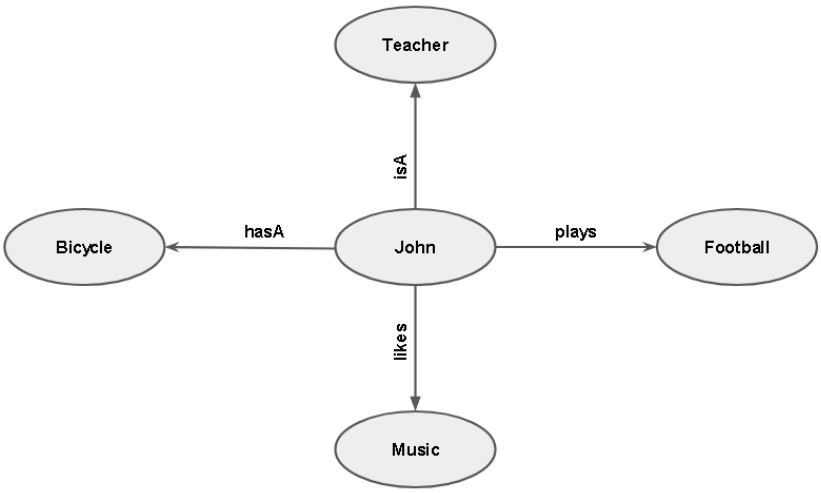

Figure 3.8: Example description of triples in database.

- {John, hasA, Bicycle}

RDF is an abstract data description framework and lacks a serialization format. W3C recognizes several valid serialization formats: XML (also called RDF/XML), Turtle, N-Quads and N-Triples. As XML was the first serialization format, and for simplicity of exposition, this thesis will only use the XML formatting. Back to the example, when serialized to XML/RDF format, it may look like the following snippet shown in Listing 3.2. The new vocabulary introduced by RDF are the terms with the namespace `rdf`, e.g. `rdf:Description`, `rdf:about`, `rdf:resource`, etc.

```
<rdf:RDF xmlns:props="http://URI/for/our/properties">
    <rdf:Description rdf:about="http://URI/for/person/John">
        <rdf:type rdf:resource="http://URI/for/class/Person" />
    </rdf:Description>

    <rdf:Description rdf:about="http://URI/for/person/John">
        <prop:isA>Teacher</prop:isA>
    </rdf:Description>

    <rdf:Description rdf:about="http://URI/for/person/John">
        <prop:likes>Music</prop:likes>
    </rdf:Description>
```

```
    ...

</rdf:RDF>
```

Listing 3.2: An Example snippet of XML for RDF Graph shown in Figure 3.8

RDF also provides is a mechanism for querying database using SPARQL. RDF allows *reification* of each triple in database to create (at least) three other statements: one identifying the the subject, one the predicate and the object of the statement. All statements are also treated as resources and URIs are assigned to them for this purpose. For example, say `John-plays-Football` is defined as a triple in RDF format. RDF treats each of the following entities {*John, plays, Football*} as subject of three statements and assign URIs (see Section 3.6.8) to them: `JohnURI-rdf:subject-statementURI`, `playsURI-rdf:predicate-statementURI` and `FootballURI-rdf:object-statementURI`. This makes the information more explicit, searchable, and gives OWL (discussed later) the ability to make inferences.

RDFS

RDF Schema provides additional data modeling vocabulary for RDF data. In particular, it allows:

- limiting the range and domain of predicates (i.e. attributes) via `rdfs:range` and `rdfs:domain`, respectively, and,
- Object Oriented Programming like hierarchical modelling of Classes and Properties via `rdfs:subClassOf` and `rdfs:subPropertyOf`, respectively.

RDFS also allows adding *labels* and *annotations* to RDF statements (via `rdfs:label` and *rdfs:annotations*). These can be used to make the statements more human-friendly and to add more meta-data to the statements for certain reasoners.

OWL

The Web Ontology Language (OWL) is a semantic markup language for publishing and sharing ontologies on the World Wide Web. It is a knowledge representation language, designed to formulate, exchange and reason with knowledge in the domain of interest [**w3c2009owl**]. OWL is designed for use by applications and developers alike. For-

mally, it is a vocabulary extension of RDF (the Resource Description Framework) and RDFS. OWL facilitates greater machine interpretability of knowledge than that supported by XML, RDF, and RDF Schema (RDFS).

Additionally, it also has formal semantics associated with its vocabulary which allows use of Description Logic [**baader2003description**] for making inferences. For example, OWL allows marking a Property as an inverse of another property by using the predicate `owl:inverseOf`. This allows writing a statement like `hasASonURI-owl:inverseOf-hasAFatherURI`. Now if an Ontology has a statement `John - hasASon - Sam` then by the virtue of inverse property, an OWL reasoner can infer that `Sam - hasAFather - John`.

```
<owl:ObjectProperty rdf:about="hasASonURI">
    <owl:inverseOf rdf:resource="hasAFatherURI"/>
    <rdfs:domain rdf:resource="PersonClassURI"/>
    <rdfs:range rdf:resource="PersonClassURI"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="hasAFatherURI">
    <rdfs:domain rdf:resource="PersonClassURI"/>
    <rdfs:range rdf:resource="PersonClassURI"/>
</owl:ObjectProperty>
```

Listing 3.3: Snippet of OWL code showing how `owl:inverseOf` *can be used. It also uses vocabulary defined by RDFS. Note that an explicit* `owl:inverseOf` *definition was not needed for* `hasAFather` *as it can be also be inferred using Description Logic.*

3.6.4 Structure of Ontologies in Protégé

Protégé is a free, open source ontology editor and a knowledge management system. It is created by Stanford Medical School for developing intelligent systems. Protégé provides a graphical interface to define ontologies which helps various stakeholders to think in terms of concepts and relations in the domain. Protégé is used as the ontology building tool and to demonstrate knowledge exchange across CPS modules later in this thesis work.

Entities are the atomic constituent of ontologies [**w3c2009owl**]. Typically, entities are an encompassing concept for classes, object properties, data properties or individuals. Entities are discussed in detail here:

Classes

Classes are the center of an ontology as they represent concepts in most ontologies. Classes are concepts in a domain. Classes can have hierarchy in the form of subclass and superclass. A subclass inherits from a parent class or superclass. The idea behind inheritance is that subclass acquires all the properties and behaviours of parent class. Thus, it saves the effort to define similar classes again. Classes can be created in a particular ontology or can be imported from a different ontology. Figure 5.18 shows `WeighingSensor`, `Container` and `RFID` are the classes created in the weighing module system ontology. The classes correspond to different concepts of weighing module. `WeighingSensor` reports the raw weighing sensor data, `RFID` gives information read from RFID tags and `Container` has static values about container box.

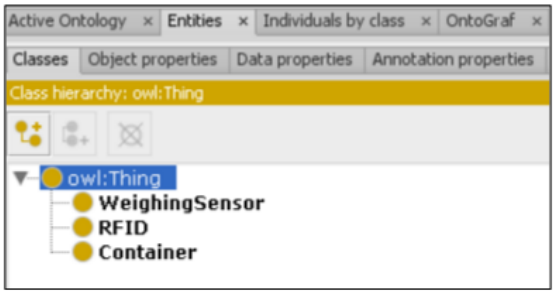

Figure 3.9: Shows the classes (`WeighingSensor`, `RFID` and `Container`) defined in weighing module ontology.

Further, axioms are defined for classes. They can be understood as facts about a concept. According to the W3C's OWL specifications, axioms are as the basic statements that an ontology expresses and asserts as true. They can be explicitly stated or inferred from given knowledge. For example, given that `John-isA-teacher` and `teacher-isA-person`, the axiom thus inferred is `John-isA-person`.

Object properties

Object properties define relationships between classes and, therefore, between individuals. Object properties can have certain attributes themselves (e.g. `owl:FunctionalProperty`) and can be related to other properties (e.g. using the `owl:inverseOf` relationship). For example, if there is an object property defining John-`likes`-Music, then the inverse of the object property can be Music-`isLikedBy`-John.

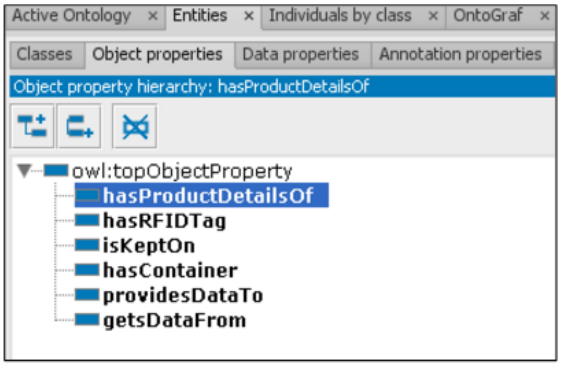

Figure 3.10: Figure shows relationships between objects `WeighingSensor`, `RFID` *&* `Container` *defined in weighing module ontology in Protégé.*

Figure 3.10 shows both direct and inverse relations defined in weighing module ontology between the classes: `Container`, `WeighingSensor` and `RFID`. Following are the object properties defined in Protégé for weighing module ontology:

`WeighingSensor-RFID` relationship:

- `WeighingSensor` **getsDataFrom** `RFID`
- `RFID` **providesDataTo** `WeighingSensor`

`WeighingSensor-Container` relationship:

- `WeighingSensor` **hasContainer** `Container`
- `Container` **isKeptOn** `WeighingSensor`

`RFID-Container` relationship:

- `RFID` **hasProductDetailsOf** `Container`
- `Container` **hasRFIDTag** `RFID`

Data properties

These are used to connect individuals (see Section 3.6.4) to literal values. Literal values are defined by domain and range it has and can be of various data types like string, float, integer. Figure 3.11 shows data properties corresponding classes: WeighingSensor, RFID and Container. WeighingSensor has data properties: `hasTotalWeight`, RFID

has data properties: {hasPartName, hasPartType, hasPartWeightTolerance} and Container has data properties: {hasX, hasY, hasZ, hasLength, hasBreadth, hasHeight, hasInventory} as shown in the figure.

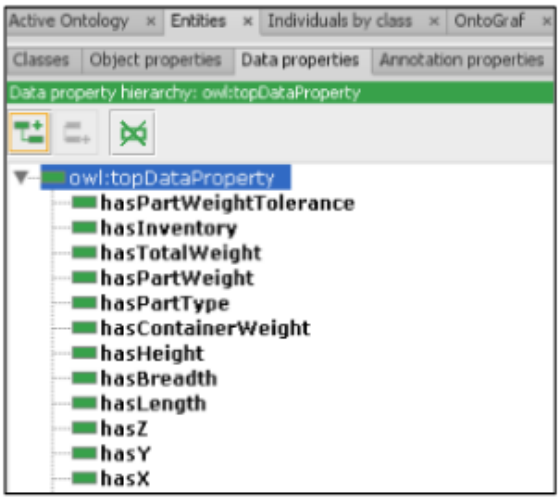

Figure 3.11: Shows data properties of classes `Container`, `WeighingSensor` *and* `RFID` *in weighing module ontology.*

Individuals

Individuals are *instances* of the aforementioned classes. Thus, individuals contain properties of their classes and further assign values to the properties to describe that instance of class. Figure 5.26 shows three individuals of classes `Container`, `WeighingSensor` and `RFID`.

Individuals further assign values of data and object properties for each instance of that class. Figure 3.13 shows an example of individual `Container1` of the class `Container` created in the weighing module ontology. The individual describes data properties: `hasContainerWeight`, `hasLength`, `hasBreadth`, `hasHeight`, `hasX`, `hasY`, `hasZ` and `hasInventory` for a particular container. Similarly, individual `Container1` defines its relationships with individuals of other classes. Container1 `hasRFIDTag` RFID1 and `isKeptOn` WeighingSensor1.

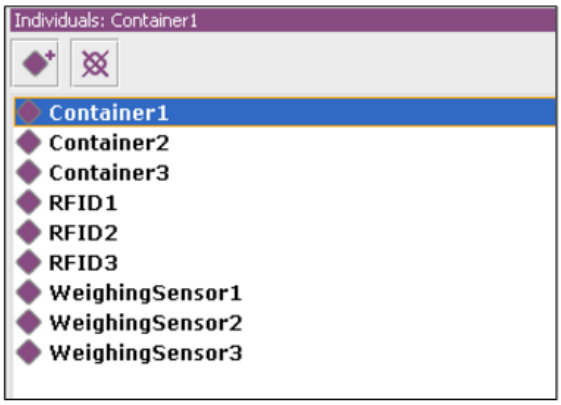

Figure 3.12: Shows three individuals of classes `Container`, `WeighingSensor` *and* `RFID` *each in weighing module ontology.*

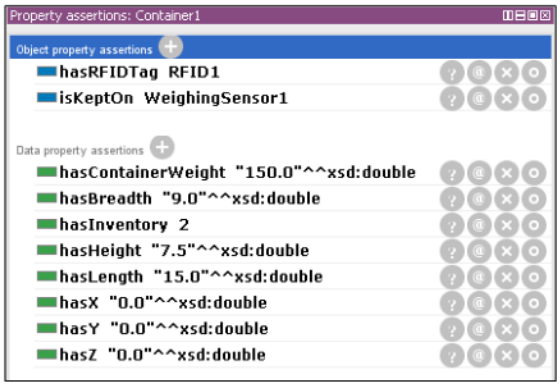

Figure 3.13: Figure shows values of data properties of `Container1` *and its relation with* `RFID1` *&* `WeighingSensor1`.

SPARQL Query

SPARQL Protocol and RDF Query Language (SPARQL) is a semantic query language for databases. It retrieves and manipulates data stored in RDF format. SPARQL 1.0 is a W3C recommendation since 2008. Later in 2013, SPARQL 1.1 was added as W3C recommendation. Listing 3.4 shows an example of a SPARQL query. Prefixes `rdf`, `owl`, `rdfs` and `xsd` (all are standards of W3C) are added if their vocabularies are required to make the query. The namespace/prefix `arbeitsplatz` is defined by the author in the process to build Protégé model for the use case.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX arbeitsplatz: <http://www.semanticweb.org/SmartFactoryKL/
Arbeitsplatz/BaseOntology/>

    SELECT ?partName ?name
    WHERE { ?partName arbeitsplatz:hasPartType ?name  }
```

Listing 3.4: Shows an example of SPARQL query where prefixes `rdf`, `owl`, `rdfs` *and* `xsd` *are defined by W3C and prefix* `arbeitsplatz` *is defined by the author. The prefixes identify namespaces which contain the vocabulary to run the query.*

3.6.5 Ontology Integration

Ontology development can be seen as defining structure, constraints and data for other programs to use. Software agents and other problem solving methods can use these ontologies as ready-made data that can be fed to the program in order to understand the axioms and basic principles of the domain. The independently developed ontologies need to join to exchange data.

Figure 3.14 depicts the idea behind ontology integration. It is the process of finding commonalities between two ontologies, for example Ontology A and ontology B, and a third ontology C is derived from it. This new ontology C facilitates interoperability between software agents based on ontologies A and B. The new ontology C may replace the old ontologies or may be used as only an intermediary between systems based on ontologies A and B are merged in a third ontology C as shown in the figure. Ontologies can be integrated primarily in three ways depending on the amount to change required to derive the new ontology [**sowa2000knowledge**]:

By alignment

It is the weakest form of ontology integration. This requires minimal change, but supports only limited kinds of interoperability. It is generally used for information retrieval, but does not support deep inferences. For example, alignment maps concepts and relationships between ontologies A and B such that it partly preserves ordering by subtypes in both

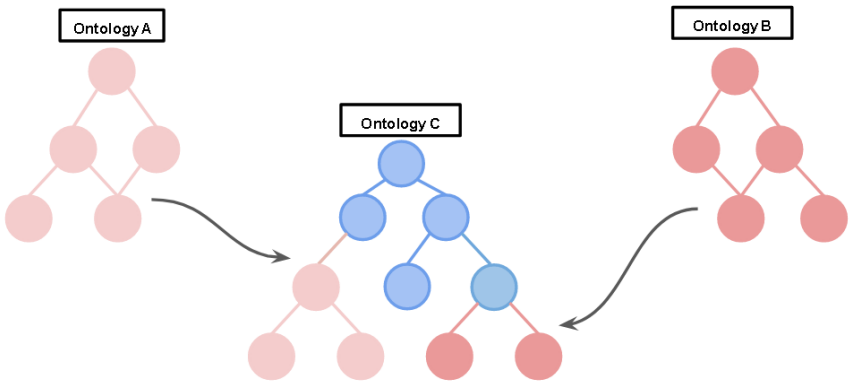

Figure 3.14: Shows schematic description of ontology merging. In the presented example, `Ontology A` *and* `Ontology B` *are merged with an existing* `Ontology C`.

ontologies. If an alignment maps a concept or relation x in ontology A to a concept or relation y in ontology B, then x and y are equivalent. The mapping of concepts is not complete, therefore there can be a concept or relation in ontology A that has no equivalent in ontology B.

Before the two ontologies A and B are aligned, it may be necessary to introduce new supertypes and subtypes of concepts and relations in one of the two ontologies. No other changes to the axioms, definitions or computations in either A or B are made during the process of alignment [**sowa2001building**].

By partial compatibility

It is more interoperable than by the alignment but also requires extensive changes as compared to alignment. It can be defined as an alignment of ontologies A and B that supports equivalent inferences or computation on all equivalent concepts and relations. For example, if ontologies A and B are partially compatible then any inference or computation that can be expressed in one ontology using only the aligned concepts and relations can be translated to an equivalent inference or computation in the other ontology.

By unification or total compatibility

This is also known as ontology merge. It gives complete interoperability between the data of ontologies, but may require significant changes. For example, if partial compatibility of two ontologies A and B is extended to a total compatibility in the new ontology C then ontology C includes all concepts and relationships of both ontologies A and B. Any inference or computation that can be expressed in either one ontology can be mapped to an equivalent inference or computation in the other ontology [**pinto1999some**].

3.6.6 Ontology Conflicts

Ontology merging for integration of heterogeneous data sources is a complex activity that involves data reconciliation at various levels of conflicts. These heterogeneous conflicts need to be resolved before the data can be integrated. **ram2004semantic** categorize heterogeneity conflicts in the following abstraction levels, each discussed separately.

Data

Conflicts arise due to discrepancy in the underlying data values across multiple sources. This conflict arises at instance level and are related to the representation or interpretation of data values. This includes type, incorrect names, unit, precision, allowed data and missing data. Examples of these can be "km" and "metre", "dollar" and "$".

Structural

Conflicts arise due to discrepancy in the underlying schema. This means different alternatives are provided by one data model to develop schema for same reality.For example, what is modelled as an attribute in one relational schema may be modelled as an entity in another relational schema for the same application domain. "Author" can be an attribute for the entity "book" and "author" can be an entity that has a relationship with "book". Another example two sources may use different names to represent the same concept, "price" and "cost", or the same name to represent different concepts, or two different ways, for conveying the same information, "data of birth" and "age".

Further, conflicts at each level can be categorized into two kinds:

- **Syntactical Conflicts:** refer to discrepancies in the representation of data. For example, "1-54" and "1.54" or "price=100 euros" and "price : 100 euros".
- **Semantic Conflicts:** refer to disagreement about the meaning or interpretation of same or related data. For example, "staff" and "employees".

3.6.7 Temporal Dynamic Ontologies

In dynamic ontologies, time can be added as a variable for restraining the system. CHRONOS can be used as a tool to make temporal ontology in Protégé [**preventis2014chronos**]. The system is to be updated with time. However, time as a variable for the assistance system is not discussed in this thesis. It is one of the areas that might be useful in the development of assistance system ontology and can be explored in future work in Chapter 7.

3.6.8 Importance of Ontologies in Context of Industry 4.0

Semantic description of devices and services play a crucial role to be able to exchange data. Moreover, **grangel2016towards** talk about a common semantic model for components of I4.0. This section deals with the major reasons for using ontology based semantic modeling for I4.0:

- **To share common understanding of data:** Ontologies help in providing formal naming and definition of entities across different domains.

  For example, different airline websites contain information about flights. If these websites share the underlying ontology, then other software agents can extract and index this information. This makes it possible for software agents, like a travel planner website, to query the indexed aggregated information, thus facilitating sharing of knowledge and putting data to more use [**noy2001ontology**].
- **To facilitate interoperability:** I4.0 envisions new ways of managing data, devices and services. These new components are made using different formats of data. Furthermore, there is an existing legacy of production systems need to coexist with the new data and new formats. To meet this demand of interoperability, ontologies have proven to be a successful way to integrating different types of data [**grangel2016towards**].

- **To explicitly state the underlying assumption of domain:** It is easier to define assumptions in ontology and revisit them from time to time. If the domain knowledge changes, these assumptions can be changed accordingly. On the other hand, hard-coding these assumptions might make if difficult for other programmers to search and change them. It becomes particularly more difficult for people who do not have expertise in programming. Further, it helps newbies to understand the scope of domain and the terms associated with it.
- **To enable reuse of domain knowledge:** It is the most important feature of ontologies. Upper ontologies are built to facilitate the reuse of domain knowledge. It allows experts to build semantically rich common knowledge base and a theoretical framework for design. This knowledge base can be imported by other ontologies to represent concepts and relationships in a domain-perspective. Multiple domains can have the same concept but their representation might be different depending on the domain context.
- **To ensure data availability:** In order to build more pervasive context-aware adaptive systems, it is important that data is available to devices in both vertical and horizontal integration (discussed in Section 3.1) of Industry 4.0. I4.0 components should be able to communicate the data generated and interact with other machines. Ontologies can be employed as the standard representation of data. RDF data serialization can be easily done in many formats and SPARQL can be used to query database, thus making the data available through a standard interface [**grangel2016towards**].
- **Providing global identification** Global identification of Industry 4.0 components and a linking mechanism between components and information are of paramount importance for enabling intercommunication between components and their environment. Ontology provides Uniform Resource Identifiers and Internationalized Resource Identifier (URIs/IRIs) as the unique global identifiers, thus ensuring disambiguation of entities. In addition, OWL provides identification capabilities that can be extended by various existing vocabularies and achieve an unambiguous reference to an entity within a given context.

4. Methodology

The focus of this work is to develop guidelines for interoperability between different cyber physical system with heterogeneous interfaces. To facilitate data exchange across CPS modules the data needs to be semantically described. Figure 4.1 shows different languages used to represent data semantically. The languages and their relevance in semantic representation of data in the context of cyber-physical systems are discussed. These languages can be categorized as: controlled vocabularies, taxonomies, folksonomies, thesauri and ontologies. However, not all languages are sufficiently expressive. The more expressive a language is, the more accurately it can capture the specification of terms and relationships between them. This chapter will briefly explore the various languages and discuss their suitability for semantically describing CPS modules.

A controlled vocabulary is a contended list of terms for concepts along with the non-preferred (incorrect variants) terms. Controlled vocabularies are also called authoritative files and may not necessarily have structure or relationships between terms. They are generally used to ensure consistent indexing. Controlled vocabularies are the broadest category which includes thesauri and taxonomies.

A taxonomy is typically a controlled vocabulary with hierarchical structure and relationships between terms. The hierarchy is generally of the form parent/broader, child/narrower or both if the term is at the mid-level hierarchy. Terms with a taxonomy are called nodes. Equivalent synonyms may exist in a taxonomy.

However, taxonomies lack more complex relationships found in thesauri and ontologies. Thesauri are essentially controlled vocabularies following a standard structure and well-defined relationships. The relationships are generally of three kinds: hierarchical, associative and equivalent (synonyms). Thesauri are mostly used to index literature on a specialized subject area.

Traditional languages like UML do not permit sharing of the contextual knowledge between domains. However, such knowledge based semantic modeling with efficient interop-

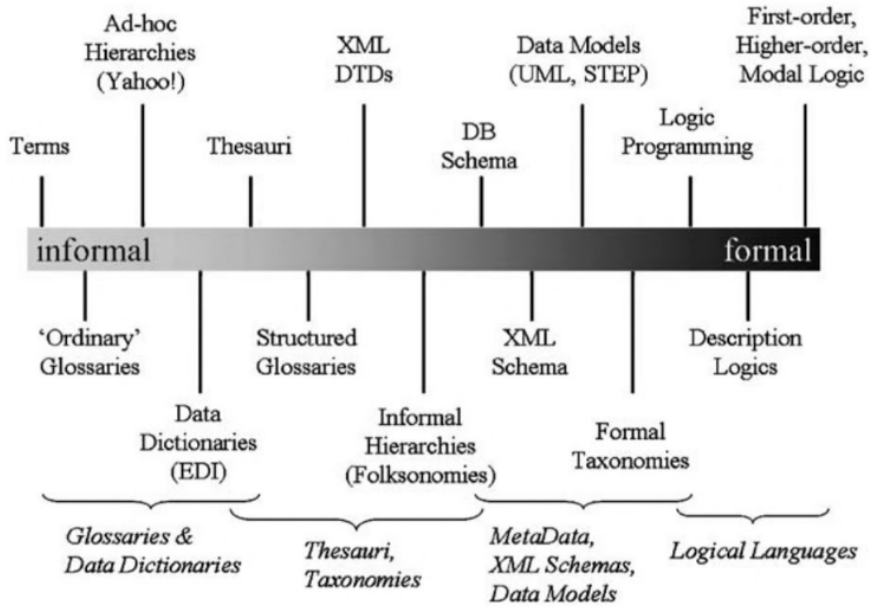

Figure 4.1: These are the different languages that can be used for semantic representation of data. Languages are placed on a scale based on how formally and explicitly they state the terms *and* relations *between them. An ontology is a logical language which uses description logic to represent a data model for a domain. It itself is specified using an XML schema and usually uses XML as the serialization format. Source:* **noy2001ontology**.

erability between heterogeneous modules can be done using ontology [**noy2001ontology**].

An ontology is a kind of taxonomy with structure and specific types of relationships between terms. There are more types of relationships than a taxonomy and they are more specific in their function. An example of such a relationship is the *inverse function*, which cannot be expressed in a taxonomy. Ontological relationships are generally used to describe information systems as they capable of capturing more terms and their relations explicitly and formally.

A folksonomy differs from taxonomy in structure. Folksonomies are typically have categories defined by tags and may not necessarily have a hierarchical parent-child structure. Folksonomies are primarily used by people to apply tags to online terms.

Keeping in mind the task at hand, folksonmies are not a viable option as they serve a completely different purpose: that of generating customized categories for users. Con-

trolled vocabularies and taxonomies lack the structure to depict complex relationships between terms. Furthermore, comparing thesauri and ontologies, ontologies are the better choice as they can specify terms and relations in a manner which allows for intelligent inferences using Description Logic [**baader2003description**].

Ontologies can be categorized in broadly three kinds: upper, mid-level and domain ontologies depending whether they state the general terms and definitions of these terms or can be tailored to domain-specific applications. Upper ontologies are made to define terms and relationships at a high-level and are domain-independent, thus providing a framework by which disparate systems may utilize common knowledge base. Upper ontologies present abstract concepts of systems and facilitate interoperability between domain-specific ontologies by the virtue of shared common terms and definitions. Mid-level ontologies serve as a bridge between upper ontologies and domain-specific ontologies. They provide mechanism to map concepts across domains and present more concrete representations of abstract concepts of upper ontology. Domain-specific ontologies, as the name suggests, are tailor-made for a particular domain.

Since this work deals with an assistance system which has a central system and one or more CPS modules, the design of the semantic structure of the complete system will be considered. Here, upper and mid-level ontologies can be employed to provide exchange and ease of understanding of data among modules: Upper ontologies provide basic vocabulary and mid-level ontologies provide mechanism to map concepts for other domain-specific ontologies. Upper ontologies would provide vocabulary to CPS modules through which they can interact and query the central database for the data which the central system itself uses. If domain-specific ontologies, i.e. the module ontologies, have access to the vocabulary being used by other modules (either via mid-level ontologies or otherwise), they too can query the central database for data being provided by other modules. Sharing of data across the entire CPS system while preserving the semantic meaning attached to it will solve the problem of interoperability, which is the primary goal of the thesis.

This thesis also provides a framework for designing a CPS module and adding it to a central system. Before the start of the designing of the ontology of the module, it is necessary to define the scope of CPS module which in turn decides the CPS intelligence. The role of hardware and software limitations and the optimal representation of the information for the central system while deciding the system boundary will be discussed. Finally, a real world implementation will be used as an example to explore the various issues one may face during implementation and deployment and principles will be discussed which will form a basis for making the design choices encountered.

5. Concept

As discussed in Chapter 2, this thesis deals with an assistance system used to help workers during assembly processes. This system consists of a central system and one or more CPS modules as described in Section 3.5. Presently, the system implemented at *SmartFactory*^{KL} has a hand-tracking module deployed at a manual assembly station. However, to fully exploit the benefits of an assistance system, it is necessary that the CPS modules can exchange data between themselves. The data exchange is non-trivial because these CPS modules may be developed independently, and, therefore, may have different data structures, representation and communication protocols. One way to overcome these barriers is to require modules to be semantic described [**quint2016system**]. This work aims at providing guidelines for designing a CPS module and adding it to an assistance system. A framework will be proposed for designing CPS modules, semantically describing these modules and exchanging of data using ontologies. The framework is then used to show how to design a weighing module for the assistance system, how to semantically describe it, and how to allow interoperability with other modules.

5.1 Framework

This section describes the framework for designing additional modules for an assistance system. These principles can also be applied for designing general CPS, but will concentrate only on the assistance module for ease of discussion. The framework focuses on defining system intelligence and developing semantic models for the systems.

The first step in the design process is to define the objective of the CPS and how the effectiveness of the module can be gauged. Since CPSs should have decentralized intelligence [**saldivar2015industry**], the next step is deciding the intelligence of CPS modules which, in turn, requires deciding their *scope*. Hence, after defining the objective, how to

decide CPS boundary and, consequently, formulation of system intelligence prerequisites is discussed. Following that, an information model, which defines and declares all the relevant data of the CPS, is developed. Further, ontologies are developed based on the information model and a way to integrate ontologies is suggested which addresses the problem of interoperability of heterogeneous data.

5.1.1 Objective

This section deals with the objective of adding a CPS module to the assistance system and discusses a possible ways of assessing the effectiveness of the added CPS.

Since the assistance systems are aimed at helping the worker, it is important to ensure that the system is not too complex for the worker who is using it. Complex-to-use systems may act as inhibitors and cause general discomfort in workers [**villani2017towards**]. Therefore, it is necessary that the interacting interface for the worker, if any, should be easy-to-use. The metric can be: improvement in worker's comfort and job satisfaction and can be gauged through on and off-line surveys and feedbacks. Consequently, the effectiveness of the CPS module and the assistance system as a whole, can also be obtained.

The next step in the design process is deciding the system boundary which is discussed in the following section.

5.1.2 System boundary

The assistance system assists workers by communicating the next step to be taken in the assembly process. It has different CPS modules with sensors and a central unit which decides the next step depending on the data provided by the modules. These modules provide sensory data in different forms to the central unit. At the onset of design, it is often unclear what data the central unit should receive from the sensors. The options range from all raw data to a binary signal (`OK/not-OK`). This design choice decides the level of processing which should happen on the modules. Hence, it becomes important to ascertain the scope of the modules which defines the form in which data is required by the central unit. Here, the use of the notion of *sufficient statistic* to decide the boundary of systems is proposed.

In statistics, a statistic is sufficient with respect to a parameterized statistical model if no other statistic that can be derived from the same sample (e.g. raw sensor data) provides

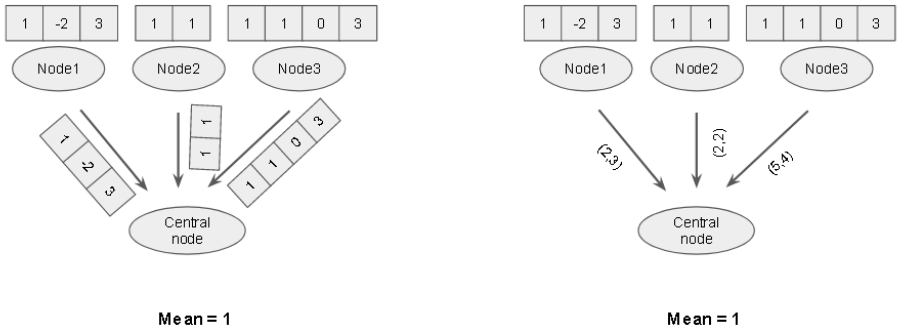

Figure 5.1: Shows data required to calculate mean of values. In the first case, complete raw data is provided to the central node to calculate mean whereas in the second case, only the sufficient statistic is provided.

any additional information as to the value of the parameter [**fisher1922mathematical**]. For example, consider the sufficient statistic to calculate mean of samples which are distributed across multiple nodes as shown in Figure 5.1. Each node only needs to report the sum of its samples and the number of samples to the central node doing the calculations. The central node then can calculate the total sum and the total number of samples and produce the mean without having the complete raw data (thereby saving computation and communication costs). Hence, `(sum of samples, number of samples)` from each node is a sufficient statistic for calculation of the mean on the server. Note that sufficient statistic are not unique for a given model and dataset.

The choice of sufficient static is driven by the data required by central system. In other words, the vocabulary, i.e. the terms defined by the central system, decide the system boundary.

5.1.3 System intelligence

Ideally, the modules should have as few computational/communication requirements as possible. It keeps the module cheap to produce (higher computational power is more expensive), simple to maintain (simpler programs and internal components), and energy efficient (more powerful hardware requires more energy to run) while still assisting the decision-making in the central system.

However, in order to calculate the sufficient statistic discussed above, the module needs

to be able to process the raw data it is receiving from its sensors and perform certain calculations. Hence, the choice of the sufficient statistic will put lower bounds on how much computational power the module needs to have. Further restrictions on the computational ability of the system can be derived from bounds on quality or accuracy of the statistic required by the central system. Higher accuracy may require using more powerful models for approximating the sufficient statistic, which may, in turn, require higher computational/communication overhead.

Therefore, system boundary and system intelligence are coupled requirements which need to be decided simultaneously. Both are key design decisions which will guide the extensibility and deployability of the CPS in real-life.

5.1.4 Developing information model & ontology

Information model represents concepts and the relationships, rules, and constraints to specify data semantics in the domain of discourse. It helps in understanding the structure of information, make domain assumptions explicit and analyze domain constraints. Therefore, it is important to develop an information model of the semantic data before creating an ontology.

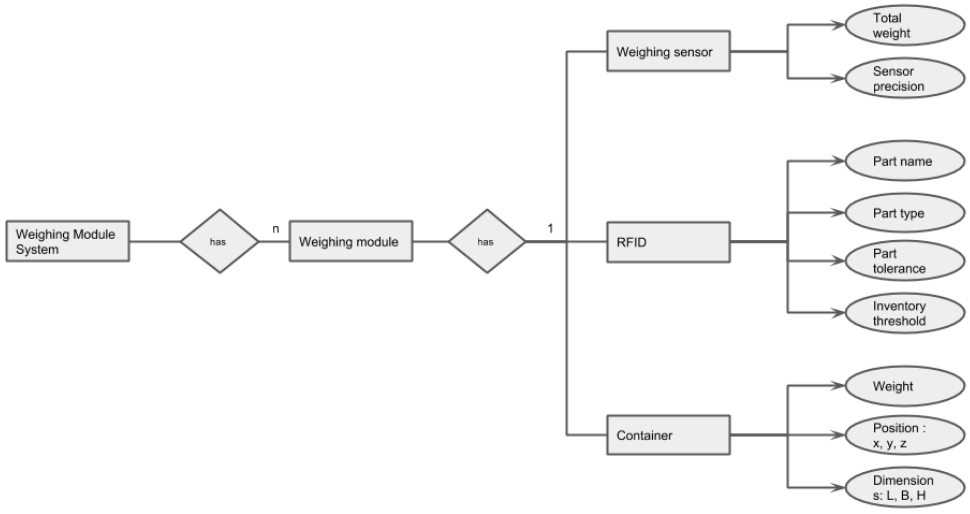

Figure 5.2: An example of information model for a weighing module. It shows the relevant physical entities and relationships between them.

Information model can be seen as an abstraction of the physical layer developed by listing the physical entities present in process. All the physical entities (e.g. actuators, sensors and tags) are listed, irrespective of their role in calculating sufficient statistic. In the second step, relations between physical entities are drawn. A module can have more than one sensor which is represented by `has (n)` statement in Figure 5.2. The end nodes of all branches are the literal values for a parameter, which are usually either a string, an integer, or a floating point number.

Figure 5.2 shows the information model of a weighing module. The model consists of physical entities and relationships between these entities. One or more weighing modules can be attached to the central system, as discussed earlier. The relationships can map object-to-object, e.g. weighing module has *exactly one* {`Weighing sensor, RFID, Container`}, or can map object-to-data properties, e.g. `RFID` has *exactly one* {`part name, part type, part tolerance, inventory threshold`} as shown in Figure 5.2. It is important to map all possible (raw) data in the model so that the relationships are clearly defined. This model will be instrumental in creating the weighing module ontology.

Based on the information model, an ontology is created for the module as shown in Figure 5.3. Information model acts as a lower bound for the ontology, which means entities presented in information model are mapped one-to-one in the ontology. However, sufficient statistic might not be a part of the information model which has to be included in ontology. Thus, sufficient statistic, derived from system boundary and system intelligence, serves as an upper bound of the ontology.

For example, sufficient statistic for a `WeighingModule` can be `hasNumberOfParts` which is not present in the physical layer of information model. Hence, `hasNumberOfParts` is added as a data property in the ontology.

5.1.5 Merging ontologies

Ontology integration is the process of finding commonalities between two different ontologies A and B and deriving a new ontology C that facilitates interoperability between computer systems on the A and B ontologies [**sowa2000knowledge**]. Integration can be done primarily in three ways [**sowa2001building**], namely:

- Ontology alignment
- Partial Compatibility Ontology

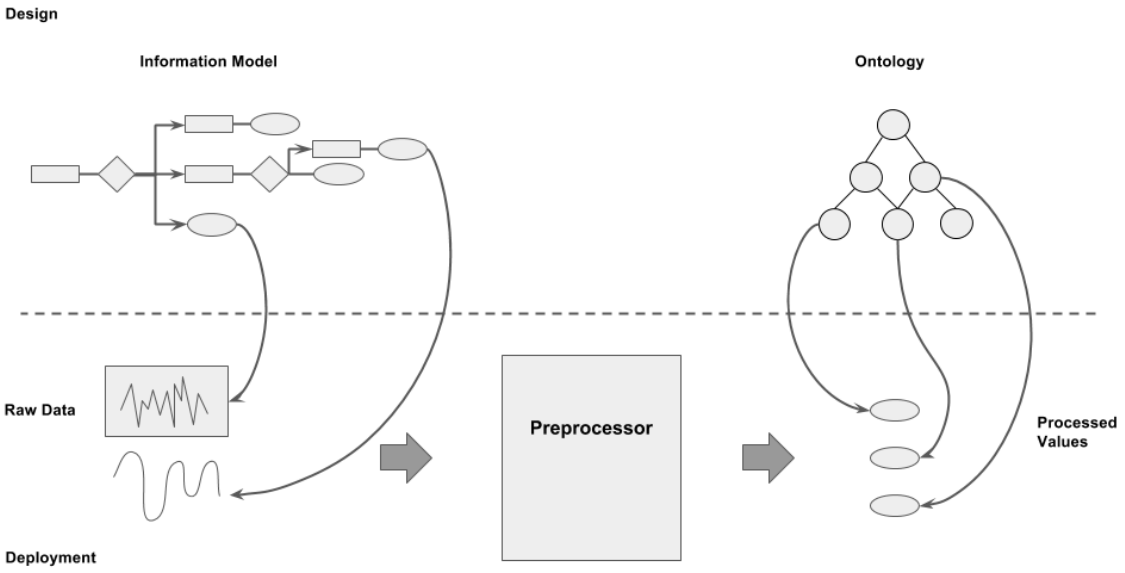

Figure 5.3: The information model (top-left), which is based on the physical setup of the system, is used to design the ontology (top-right) during the design phase. During implementation of the model, in the deployment phase, the sensors in the information model will produce some raw data. This raw data will be preprocessed by the CPS module (this is where system-intelligence comes into play) and will be made ready for the ontology. The preprocessing step may include operations like analog to digital conversion, computing a parameter which is a function of data from more than one sensor (e.g. `numberofParts` *from* `totalWeight` *and* `weightPerPart`*), calculating a moving average of a sensor reading, etc.*

- Unification or Ontology Merge

Ontology merge is chosen as the preferred way of ontology integration because it preserves complete ontologies while collecting data from different parts of the system into a coherent format which is not completely true for ontology alignment and partial compatibility ontology as discussed in Section 3.6.5. Figure 5.4 schematically depicts merging of ontologies. The entities (classes, individuals, data and object properties) used by more than one module are defined and assigned in one module. These entities are imported by other modules when required.

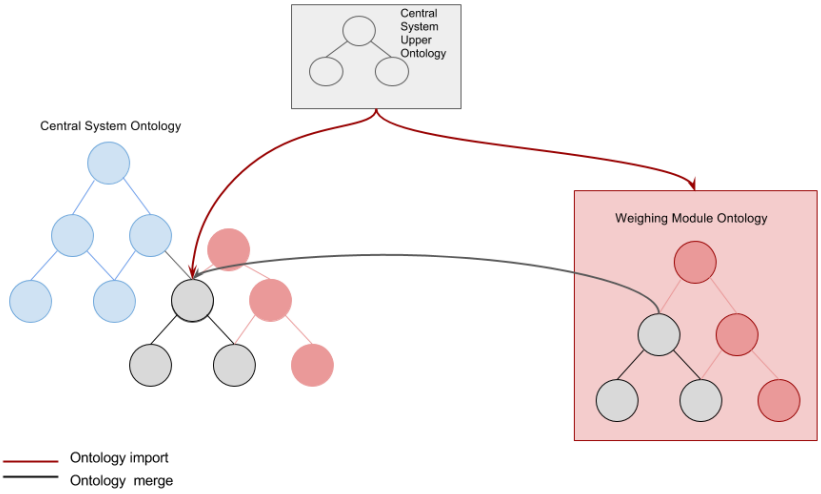

Figure 5.4: Shows schematic description of central system upper ontology import by central system and weighing module ontologies. Further, weighing module ontology merge in central system ontology is indicated.

For example, weighing module ontology and central system ontology use `WeighingSensor` class (refer Figure 3.14). Therefore, these ontologies are developed independently and both import central system base ontology in order to access the same class definitions. The ontologies are then merged to update the values of the instances. Weighing module ontology can be further programmed using an OWL API function for merging ontologies to deal with problems like the latest updated data in case of merging ontologies. Here, entities are identified as IRIs. It should be decided based on the requirements if the complete ontology is merged every time and at what frequency the ontologies should be merged. It is important to update otherwise static values because we do not know whether any value is changed or not. A venue for future work is including (synchronized) time as a variable which can keep track of the last-updated time of various quantities saved on the central system, thereby allowing for selective merges based on which data is new.

To summarize, a framework has been proposed for enabling interoperability of heterogeneous sensory data while designing CPS modules for assistance system. First, the objective of adding the CPS module should be clearly stated which is necessary to set targets and gauge improvements. Then system boundary and system intelligence should be defined which are necessary to decide the scope of modules and decide the hardware to be used. Next, the information model ought to be designed which will serve as the

base for ontology creation. In the final section, the communication between modules and how ontologies should be merged is discussed.

In the next section (i.e. Section 5.2), a deeper look into the design of the information model and the ontology is taken as they leave scope for several design decisions. These are discussed with the help of examples and rationale behind the decisions taken are discussed. Similarly, Chapter 6 will deal with the design choices and problems faced during deployment.

5.2 Model Development

In this section, the top-down (ontology based) and bottom-up (information model) development of models for the CPS modules are discussed. Though the information model should mirror the physical layer as faithfully as possible, there are some room for design in it. Similarly, the ontology designed may differ from the information model in a few places due to requirements of the central system and, hence, while the information model provides a good blueprint for it, some design choices still need to be made.

The issues faced during deployment of the modules will be discussed in Chapter 6.

5.2.1 Information model

In this section, designing of the information model of a CPS is described while using the model of the Weighing Module as an example. Before describing the information model, it will be worthwhile to briefly study the physical layout of the Weighing Module System, which is shown in Figure 5.5. The system may contain more than one Weighing Module with each module further containing an RFID reader, a container and a weighing sensor, as shown in the figure. Information model for this weighing module is developed as follows.

Figure 5.6 shows the information model for a part of the weighing module. The *continuation* (i.e. the dotted lines) denote the places where the model has been truncated and the remaining parts are shown in Figure 5.9, Figure 5.8 and Figure 5.10. The weighing module system may have more than one weighing module and, therefore, the information model allows a weighing module system to have `n` weighing modules. Each weighing module in the model further has *exactly one* container to keep parts or products. It has, among many other properties/entities) *exactly one* RFID *tag* attached to it (see

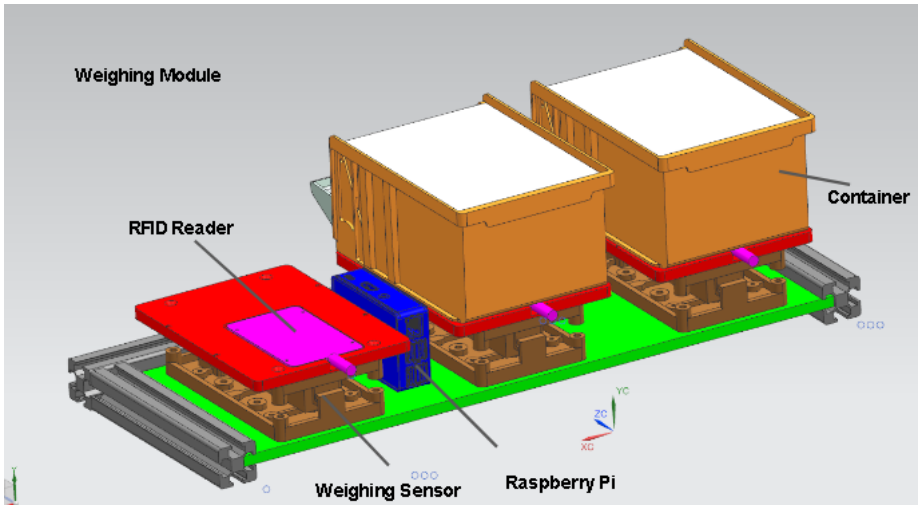

Figure 5.5: Simulated image of the Weighing module used for the implementation. The module consists of three weighing sensors, each with a container and an RFID reader. Source:SmartFactory^{KL}

Figure 5.10). The RFID tag *has* the specifications (static values) of the product (see Figure 5.9). The weighing module further has a weighing sensor to read the weight values (see Figure 5.8). This model is shown in Figure 5.6.

As a side-note, there is no unique information model for the weighing module and a design decision is made to arrange the entities in this manner. If the information model was to reflect physical reality more closely, it would have looked like Figure 5.7 because the RFID tag is attached to the container. The RFID tag in this model is *nested* inside the Container. However, actually, the RFID tag and the Container are independent when it comes to the information they provide to the Weighing Module. Nesting them one inside the other belies their dependence. It also makes the information model more nested and, hence, more complex and constrained. The independent nature of each information source is preserved and the structure of information model is kept more flexible by saying that weighing module has exactly one container, one RFID tag and one weighing sensor.

The purpose of RFID tag is to link a product with its specifications. Therefore, an RFID tag has exactly one product and contains exactly one name, weight and tolerance of weight corresponding to the product it has as shown in Figure 5.9. Here again the information could be structured in the different way by saying that RFID tag has exactly one product name, product weight and product weight tolerance but the former structure is chosen because it emphasizes the objective of attaching RFID to a product which is

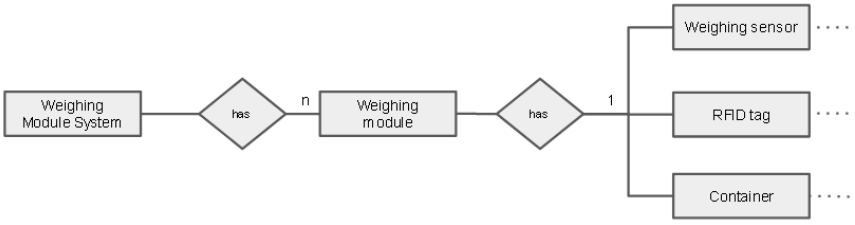

Figure 5.6: Shows configuration of `Weighing module` *which has* `Weighing sensor`, `RFID Tag` *and* `Container`.

also one of the principles of Industry 4.0 as discussed in Chapter 3.

The `Container` has its own weight, dimensions and position on the manual work station. These are mapped in the information model as container has exactly one weight value, one position and one set of dimensions as shown in Figure 5.10. The position has exactly one coordinate which contain exactly one value of x, y, z and dimensions have exactly one length, breadth and height. As the figure depicts, there is one more value attached to the container which is inventory threshold. Inventory threshold can also be attached to the object product but is a part of container for the ease of data handling as RFID tag would then only contain the product specifications and the RFID tag does not need to be altered in case of change of inventory threshold.

The information model described in this section defines the entities and relationships between these entities for a weighing module system. It is important to map all possible (raw) data in the model so that the relationships are clearly defined. This model is then used to create the weighing module ontology in the following section.

5.2.2 Weighing module ontology

An ontology is a formal way of explicitly defining entities and the relationships between them as discussed in Section 3.6. This facilitates discourse among engineers and agents and enables domain knowledge being taken into account while reasoning about the system. Here ontologies and their use are discussed in the context of weighing module.

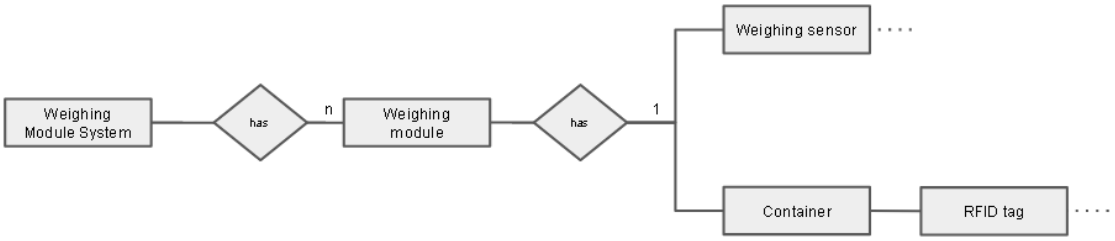

Figure 5.7: Shows the alternate possible configuration, `Weighing module` *contains* `Weighing sensor` *and* `Container`*. The* `Container` *has an* `RFID tag` *nested inside it. Though it depicts the physical setup better, it increases hierarchy, and does not depict the informational independence between the two sources. This model, hence, is unnecessarily more complex and constrained.*

As discussed in Section 3.5, an assistance system consists of a central system and one or more CPS modules. As a part of this thesis, weighing module's ontology is designed and the module is added to assistance system. Information model designed in the previous section describes a weighing module. Weighing module, as shown in Figure 5.5, has one or more weighing sensors with RFID reader attached to each module. A container with parts is kept on each weighing module and an RFID tag is attached to each container: RFID tags link product/part to containers. The RFID tag has the part information as described in the information models (see Figure 5.7).

Ontology building starts with creating an upper ontology for the assistance system. This upper ontology consists of the common knowledge base for all CPS modules attached to assistance system and provides basic vocabulary for the modules. In the presented example, the relationships between upper ontology of the assistance system, weighing module ontology and a basic eye-tracking (as an example of a third party module) are shown.

Upper ontology for the complete assistance system is developed by stakeholders as discussed in Section 3.6.1. Upper ontology can only be changed with the consensus of all stakeholders when the central system grows to accept more kinds of data (through new modules) to augment its intelligence. These are necessary changes in the basic vocabulary required by the system.

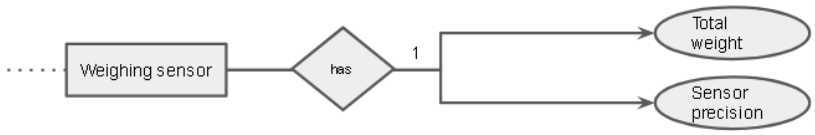

Figure 5.8: Shows configuration of `Weighing sensor`*. `Weighing sensor` has values of `Total weight` reported by sensor and `Sensor precision`.*

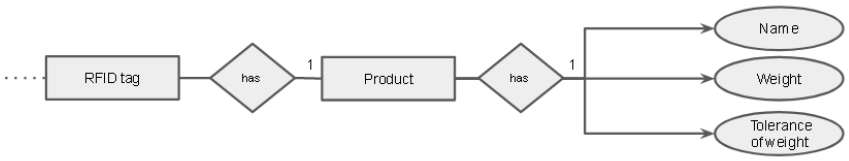

Figure 5.9: Shows configuration of `RFID Tag`*. `RFID Tag` links `Product` to the system. `RFID Tag` contains part `Name`, `Weight` and `Tolerance of part weight`.*

Upper ontology of an assistance system has a minimal set of definitions and axioms relating entities. The entities defined in upper ontology are imported by central system and CPS modules. The import of entities facilitate the exchange and understanding of the (minimal) data between modules through common definitions and IRIs. IRIs, as discussed in Section 3.6, are the Internationalized Resource Identifiers used for unique nomenclature of entities and axioms. This is the minimum data that the central system needs to use the module effectively in assisting the worker. In the next step, ontologies for CPS modules are created using the information models developed. How these ontologies are organized will be discussed in the next section. CPS modules import upper ontology of assistance system to ensure that the interacting variables are defined once in the complete system. For example, weighing module has to at least report the name and number of parts taken-out/kept-in container to the central system. In order to report the change in the number of parts, both central system and weighing module system should have a common vocabulary for the part name and number of parts. These variables are defined in upper ontology of assistance system as `hasPartName` and `hasTotalParts`. Hence, by importing the assistance system upper ontology, weighing module system ontology ensures that it shares the common representation of the interacting variables

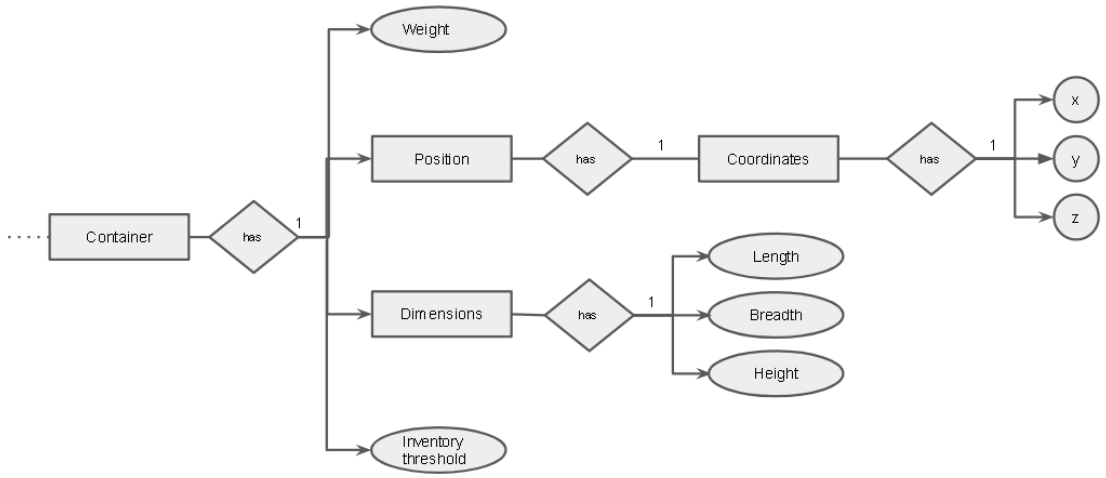

Figure 5.10: Shows configuration of `Container`*.* `Container` *has information about its* `Weight`, `Dimensions`, `Position` *and* `Inventory threshold` *for a part.*

`hasPartName` and `hasTotalParts` and their associated restrictions and relationships. Similarly, interacting variables for other modules are defined in upper ontology. However, it is noteworthy that the variables imported by weighing module can be augmented, i.e., axioms and properties pertaining to the imported variables can be added in the weighing module ontology and this addition will not be reflected in upper ontology. If changes are required in interacting variables by both central system and CPS modules, it has to be changed in upper ontology. Upper ontology changes are infrequent and need consensus of all stakeholders.

Next, interoperability of data between weighing module and hand-tracking module is explained. Two possible ways of sharing data between the two modules are presented and discussed in detail:

Decentralized Organizational Scheme

In this section, a decentralized organizational scheme for the ontologies is described. See Figure 5.12 for a visual description. As shown in Figure 5.11, upper ontology of assistance system is designed. To recap, upper ontologies of modules are created from their information models. Weighing module ontology is described using its information model.

These upper ontologies consists of definitions and axioms of entities and relationships between them. Upper ontologies of all modules import the upper ontology of assistance system as discussed above. So the basic vocabulary described by upper ontology of assistance system is imported by the modules' ontologies. As information model maps all possible data, weighing module upper ontology will contain axioms and definitions which are needed for the weighing module system, but are not required by the central system ontology. An example of this can be position of container `(x, y, z)`. This allows for flexibility in implementation of weighing module system. If another module requires data regarding container position, ontology of that particular module can import the weighing module upper ontology.

Design. Figure 5.11 shows the upper ontologies of assistance system, weighing module and eye-tracking module. Upper ontology of assistance system describe minimal data required for interaction between central system and both the modules. Upper ontologies of weighing module and eye-tracking module describe all relevant data of the modules. Both weighing module, eye-tracking module and central system ontologies import assistance system upper ontology to ensure that the entities used to share data are uniquely defined and all ontologies access the same IRI where required.

As an example, weighing module needs to report the part name and change in number of parts to the central system. Therefore, these data properties, `hasTotalNumberOfParts` and `hasPartType`, are defined in the upper ontology of assistance system. From there, it will be imported into the central system ontology and the weighing module ontology.

To facilitate interoperability of data between modules, both the interacting module ontologies should have definitions and axioms of entities pertaining to the required data. Here eye-tracking module is taken as a sub-system developed by a third party vendor. For example, the eye-tracking module needs the position of containers kept on weighing module to calibrate itself. This information was mapped in the information model of weighing module and weighing module upper ontology describes the position of class `Container` as a data property `(x, y, z)`. Hence, eye-tracking module ontology can get definitions and axioms corresponding to `(x, y, z)` by importing the upper ontology of weighing module system. This is the most crucial feature of this design.

Deployment. Figure 5.12 shows the deployment phase for the assistance system. The weighing module will create instances of the `WeighingSensor` class and assign their `hasTotalWeight` and `hasPartType` properties. Additionally, other properties will also be assigned, e.g. `dimensionsOfContainer`, which are defined in the weighing module upper ontology. During deployment, these instances with their properties will be com-

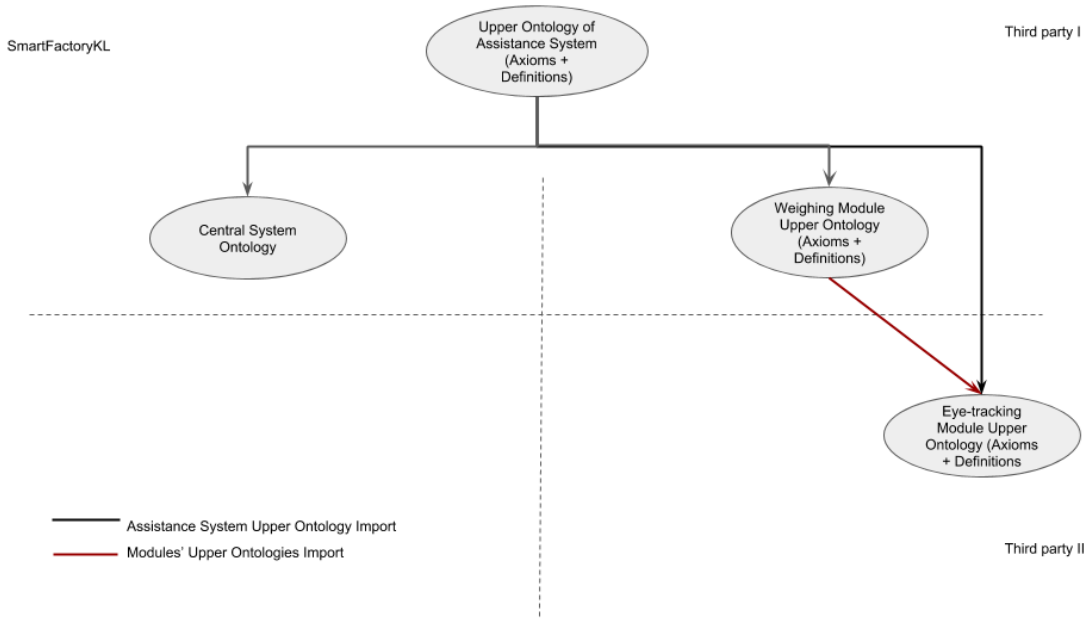

Figure 5.11: Schematic description of upper ontologies of assistance system and modules. All module ontologies import assistance system upper ontology. Eye-tracking ontology imports weighing module upper ontology to access entities/relationships of weighing module.

municated back to the central system and eye-tracking module can query the central system for the individual values for positions of different containers. While deploying the system, individuals are created in weighing module system ontology which in turn imports both upper ontologies of both assistance system and weighing module system as shown in Figure 5.12.

Pros & Cons. This design focuses on building a completely decentralized system. The central system's ontology will only contain the minimal taxonomy of entities and properties which are necessary for the Central System to function, i.e., be able to use the information from the modules effectively (see Section 3.6.1 for how to design the vocabulary). However, the individual modules are free to report any variable which they can measure and to report it to the central system. The central system will store that information even if it may not have explicit uses for the variables but can produce this knowledge if a different third party module requests for it through SPARQL queries.

Hence, if the eye-tracking module requires the container position coordinates `(x, y, z)`

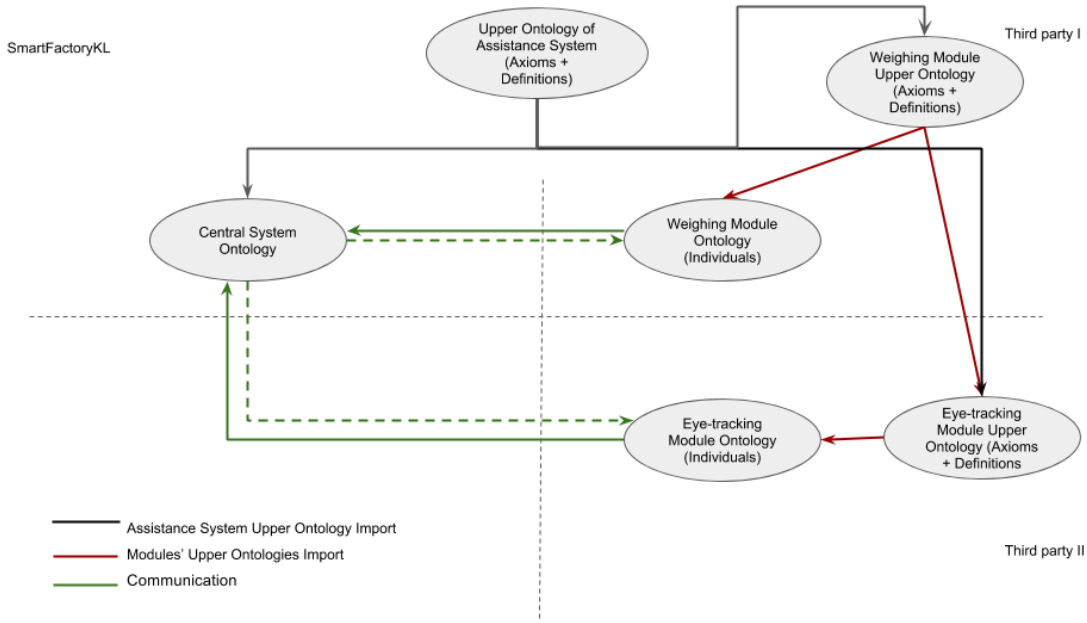

Figure 5.12: Schematic description of ontologies while deployment. Module ontologies import upper ontologies of assistance system and their own upper ontologies. Individuals are created while deployment.

from the weighing module, and it was known during the design of the eye-tracking module that the weighing module's ontology does have this information, the eye-tracking module can write a SPARQL query to ask the central system for these coordinates (using IRIs it imports from the weighing module ontology, i.e. the IRI for the relations `hasX`, `hasY`, `hasZ`). If the weighing module attached to the central system has indeed reported the `(x, y, z)` coordinate values, they will be returned to the eye-tracking module. Otherwise, the eye-tracking module may have to fall back to manual calibration. Eventually, if the properties are found to be useful in general, the upper ontology of the weighing module and made it a standard way of reporting the value. This method, hence, allows any two teams to communicate and to mutually agree on how best they can share data and promotes rapid prototyping.

However, such a setup has the disadvantage that independent teams may reinvent properties independently and since these properties will have unique IRI (e.g. `TeamA:hascoordinateX`, `TeamB:hasX` and `TeamC:hasPositionX`) but with the same semantic meaning. This would complicate interoperability across modules and for the same information from different weighing modules the eye-tracking module will have to query independently. Figure

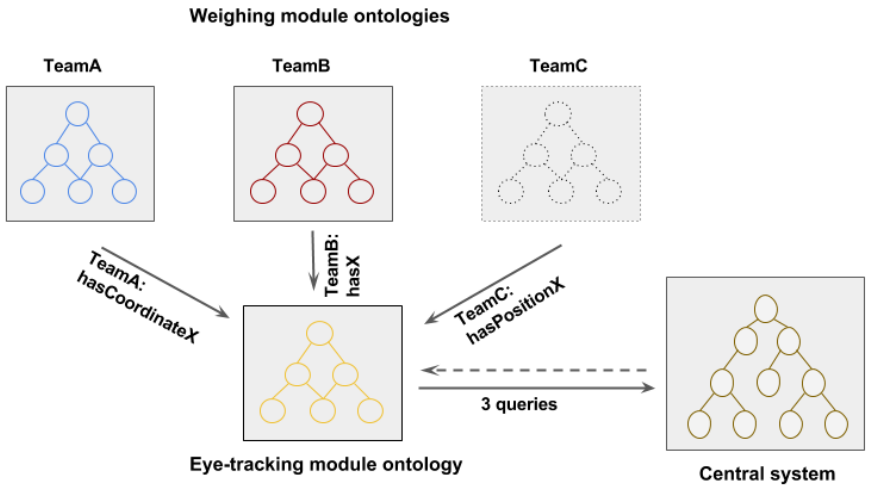

Figure 5.13: Shows an example of reinventing variables that have same semantic meaning. Here container position coordinate `x` *is described by three different teams as* `hasCoordinateX`*,* `hasX` *and* `hasPositionX`*.*

5.13 shows an example of such a situation where teams A, B and C independently define the variable for position of container as `hasCoordinateX`, `hasX` and `hasPositionX`. Eye-tracking module needs the position coordinates then it would be required to import all the three weighing module ontologies and query for them individually which increases its work and making the system more error prone. On the other hand, if the teams follow a particular nomenclature for defining variables would avoid reinventing similar variables which reduces the number of both imports and queries. Consolidation of the property names may also suffer due to the Not-Invented-Here syndrome [**katz1982investigating**].

A more subtle, and potentially more dangerous, side-effect of this design is compromised security of the data stored in the central system. In this design, the central system is completely unaware to the features which are being developed by independent modules. Hence, the central system will need to be excessively permissive when it comes to allowing arbitrary SPARQL queries by third party modules to run against the data stored it has stored. A malicious module can very easily take advantage of vulnerability to obtain data on the central system. An example of such a query is shown in 5.1. This simple query can fetch complete data stored in the central system.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

    SELECT ?subject ?predicate ?object
    WHERE { ?subject ?predicate ?object }
```

Listing 5.1: An example of a SPARQL query which will fetch all triples and hence complete data on the central system.

Keeping these concerns in mind, another solution is proposed which trades-off complete decentralization in favour of a W3C committee like setup where a formal specification (i.e. a mid-level ontology) is maintained by a committee which consists of all stakeholders. Its organizational scheme and benefits (i.e. enhanced security and prevention of re-invention of properties) will be discussed in the next section.

Centralized Organizational Scheme

This section describes a centralized organizational scheme for the ontologies. As shown in Figure 5.14, upper ontology of assistance system is created which consists of the basic vocabulary for the complete system. The upper ontology consists of definitions and axioms of entities and relations between them. Then a mid-level ontology is created. This mid-level ontology imports the upper ontology of assistance system. Further, the mid-level ontology describes the entities of all other modules. Depending on the engineers describing the mid-level ontology, all or some of the significant entities used by other modules are defined in the ontology.

The idea behind creating a mid-level is to create a repository of all relevant entities described in any CPS module. This simplifies the search by engineers for variables required by other modules. Mid-level ontology collects entities and their definitions described by upper ontologies of modules to facilitate exchange of data and this differentiates the approach from the previous approach. An assistance system upper ontology defines the minimal variables requires by modules to send data to the central system. This ontology is governed by the highest level committee and usually changes to it will be made when new modules are attached to the assistance system. On the other hand, modifications can be done easily in mid-level ontology which gives engineers the freedom to extend and access variables.

All modules' upper ontologies import the mid-level ontology. All modules need to import the mid-level ontology only once as it has all entities defined in the complete system. Discussing the previous example in the context of new design, if another module requires position of container `(x, y, z)`, it does not need to import weighing module (in which these variables were defined earlier) upper ontology because mid-level ontology has definitions and axioms of `(x, y, z)`.

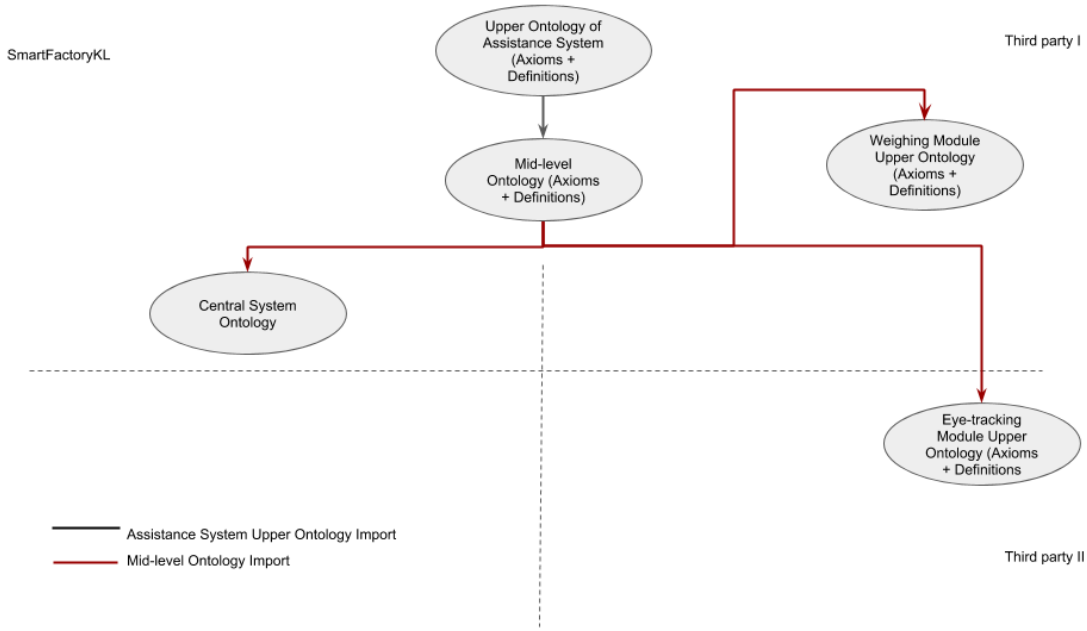

Figure 5.14: Schematic description of upper ontologies and mid-level ontology. Mid-level ontology imports upper ontology of assistance system. Mid-level ontology contains entities present in all modules. Modules' upper ontologies import mid-level ontology.

Design. Figure 5.14 shows the upper ontologies of assistance system, weighing module, eye-tracking module and a mid-level ontology. Upper ontology of assistance system describes minimal data pertaining to basic vocabulary of the complete system. Mid-level ontology is a bigger ontology describing entities of all modules. Thus, the mid-level ontology has definitions of all relevant data present in the complete system. Upper ontologies of all modules import the mid-level ontology. Hence, all module ontologies have definitions of all entities present in the system and all entities are defined only once and thus have unique IRIs.

As an example, weighing module imports the data properties, `hasTotalNumberOfParts` and `hasPartType` defined in the upper ontology of assistance system through import of mid-level ontology. This prevents defining similar entities twice by different teams because there is a centralized list of entities.

Further, eye-tracking module needs the position of containers kept on weighing module for calibration. Though this information was mapped in the information model of weighing module, but the mid-level ontology has definition of class `Container` which contains data properties `(x, y, z)`. Since all modules import the mid-level ontology, all modules have all the data described in the mid-level ontology.

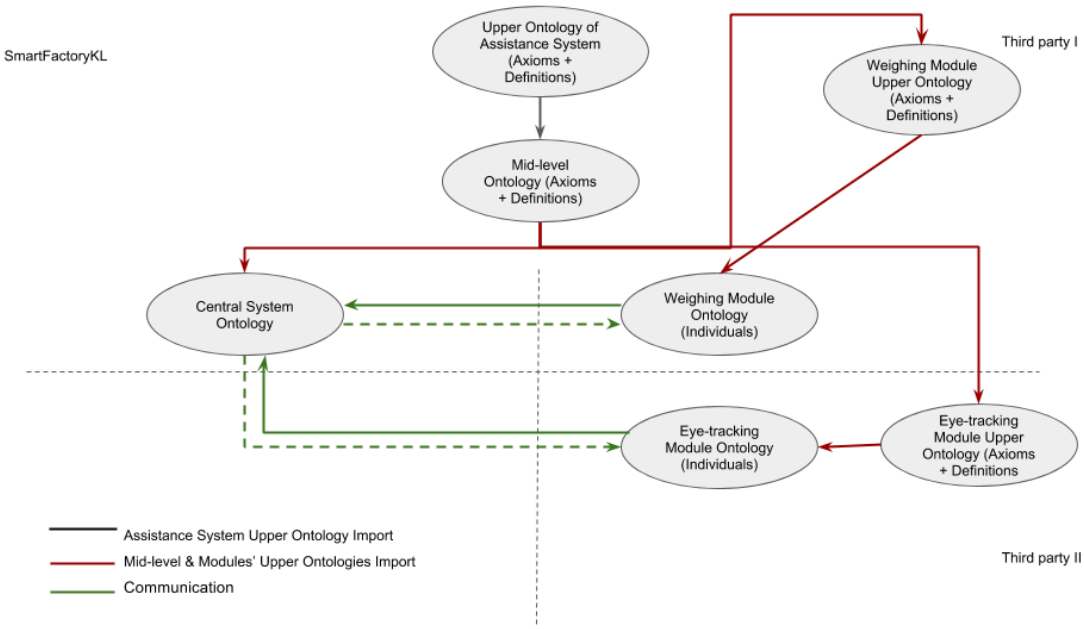

Figure 5.15: Schematic description of ontologies while deployment. Modules import their own upper ontology. Instances are created during deployment.

Deployment. Figure 5.15 shows the deployment phase of the system. Individuals of the `WeighingSensor` class which assign their `hasTotalWeight` and `hasPartType` properties are created during deployment. Additionally, other properties will also be assigned, e.g. `dimensionsOfContainer`, which are defined in the mid-level ontology. During deployment, these instances with their properties will be communicated back to the central system and eye-tracking module can then query the central system for the individual values for positions of different containers. While deploying the system, individuals are

created in weighing module system ontology which in turn imports both upper ontologies of both assistance system and weighing module system as shown in Figure 5.15.

Further, upper ontologies of modules are created which are imported by modules during deployment. As shown in Figure 5.12, modules import upper ontologies and create individuals.

Pros & Cons. In comparison with the more decentralized structure given in the previous section, the benefits and costs of this design are apparent. First of all, the mid-level ontology can be viewed as a *white-list* of entities and properties which can be read from the central system during execution through SPARQL queries. This prevents reinvention because a cursory check through the mid-level ontology will show that the properties already exist for the module being developed. Further, each module can individually extend the entities imported from mid-level ontology. These extensions are local and are not propagated to the mid-level ontology, thus modules may again reinvent variables. These extensions can be made directly in the mid-level ontology to avoid reinvention on the next level. For example, engineers working on weighing module import a class `Container` from the upper ontology of assistance system and further add property `hasContainerColour` pertaining to the colour of container. This property is described locally in the weighing module ontology and is not propagated to upper ontology of assistance system and other modules importing this ontology. However, whether these extensions should be a part of the mid-level ontology is not discussed in this thesis and can be seen as a future work.

Because of the white-list provided by the mid-level ontology, the central system can also put in place a system for authorizing certain (known) modules to have access to information which is not available to other modules. This allows security sensitive data to be *inaccessible* from potentially malicious or unknown modules. The exact authorization and authentication mechanism will depend on public key cryptography [**smith2001authentication**], which is out of scope of this work.

Pair-wise collaboration of teams is not encouraged in this setup. This can be a potential downside of the organizational scheme. This may increase the development time for a particular module if the properties it needs to import are not in the white-list already and the decision and procedure of whether to add these properties might take longer compared to the previous design scheme.

In the next step, weighing module ontology is merged with the central system ontology at the class `WeighingSensor`. Now central system and weighing module ontologies have the same data properties, `hasPartName` and `hasTotalNumberOfParts`, facilitating data exchange and updates between central system and CPS modules. The modules can query

the central system to obtain data about its and about other modules' states.

5.3 Implementation in Protégé

This section deals with implementing the ontology of assistance system discussed so far. The assistance system considered in this implementation has a central system, weighing module and a basic eye-tracking module.

Ontology referred to thus far is a mesh of information read and understood by humans. The need of encoding this information in a machine interpretable language has led to the development of a number of languages. These languages are being developed in various fields and with different goals in mind. Presently, there is no consensus regarding the de facto language to be used for developing ontologies but OWL is regarded as the best language in this regard as it is specifically developed by the W3C for ontologies.

Among the available platforms, Protégé, developed by Stanford Medical School is one platform which can be used while developing ontology for the presented assistance system. It is a free, open source ontology editor and a knowledge management system for developing intelligent systems. Protégé provides a graphic user interface to define ontologies, thus enabling modeling at a conceptual level that allows stakeholders to think in terms of concepts and relations in the domain [**noy2001creating**]. It is an easy-to-use platform with sufficient flexibility and clarity. Based on the information model, ontologies are created in Protégé.

5.3.1 Implementation of decentralized organizational scheme

This section describes the implementation of the decentralized organizational scheme of ontologies. Interoperability between modules is established in Protege through SPARQL queries. First, creation of assistance system upper ontology is discussed followed by creation of modules' ontologies and their imports.

Assistance System Upper Ontology

First step of implementing the system is creating upper ontology of assistance system. It consists of the minimal data required by the central system from various modules. The minimal data is defined as sufficient statistic for a particular module is required to

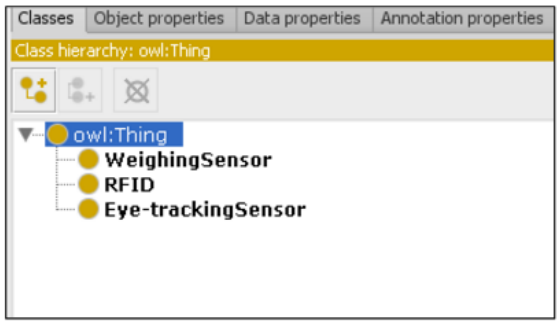

Figure 5.16: Shows classes defined in upper ontology of assistance system. These classes are imported by weighing module, eye-tracking module and central system ontologies.

exchange and understand data between central system and the module. Upper ontology of assistance system is created once at the beginning of design phase of the complete system. However, minimal data for new incoming modules can be added when required. Here an assistance system with a central system, weighing module and a basic eye-tracking module is considered. Weighing module reports part name and change in part to the central system whereas eye-tracking module reports cognitive load of worker to the central system. Thus, upper ontology of assistance system contains these three variables as `hasPartName`, `hasNumberOfParts` and `cognitiveLoadOfWorker`. `hasPartName` is of type `string`, `hasNumberOfParts` is of type `integer` and `cognitiveLoadOfWorker` is of type `boolean` indicating cognitive load level low or high. As described in the information model, `hasPartName` is a data property of RFID tag, `hasNumberOfParts` is a data property derived from `hasTotalWeight` which is defined in weighing sensor and `cognitiveLoadOfWorker` is a data property of eye-tracking module. Thus, `RFID`, `WeighingSensor` and `Eye-trackingSensor` classes as defined in the upper ontology of assistance system as shown in Figure 5.16. Consequently, the data properties discussed above are also defined in this ontology as shown in Figure 5.17.

Modules Upper Ontology

In the next step, upper ontologies of weighing module and eye-tracking module are created. Both modules import upper ontology of assistance system and add entities as required. Weighing module ontology maps all data described in the information model in the previous section. Data properties, `hasPartType`, `hasPartWeight` and `hasPartWeightTolerance` are added in the imported class `RFID`. It is noteworthy that

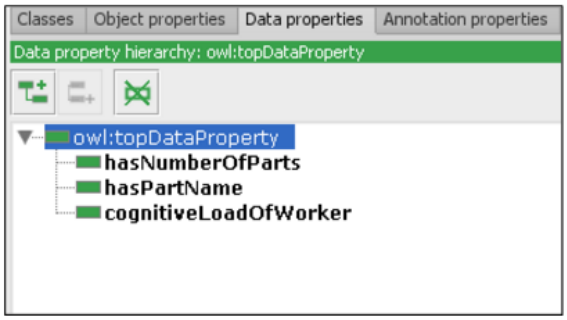

Figure 5.17: Shows data properties defined in upper ontology of assistance system. These data properties are the sufficient statistic required by central system to receive information from modules.

the data added to an imported class is not reflected in the ontology from which the class is imported.

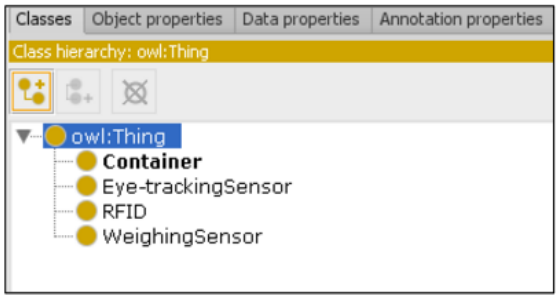

Figure 5.18: Shows weighing module ontology. It imports upper ontology of an assistance system and adds data locally. `Container` *class is defined locally in weighing module ontology.*

Thus, these changes are local and do not affect the upper ontology of assistance system. This is one of the limitations of ontologies that data changed in an imported entity does not transcend to the complete system. This can lead to duplication of data if more than one module augment similar data. This discussion is not in the scope of this work. However, to avoid such situation, teams developing modules need to interact while adding new data to imported entities or they should propose the inclusion of their addition to the maintenance, i.e., stakeholders, of the mid-level ontology.

Similarly, data properties `hasTotalWeight` is added to the class `WeighingModule`. Data properties pertaining to weight: `hasContainerWeight`, dimensions of container:

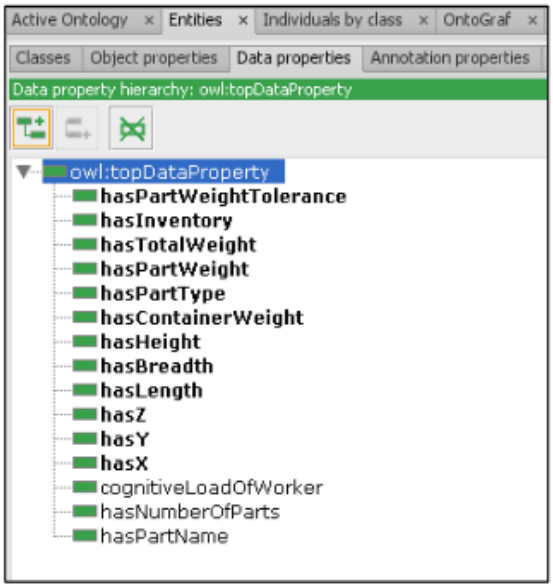

Figure 5.19: Shows data properties of weighing module ontology. Apart from `hasPartName`, `hasNumberOfParts` *and* `cognitiveLoadOfWorker` *all other data properties are defined locally.*

`hasLength`, `hasBreadth`, `hasHeight` and position coordinates: `hasX`, `hasY`, `hasZ` are defined in class `Container` as shown in Figure 5.19. Protégé has visually differentiated the entities described locally using bold font.

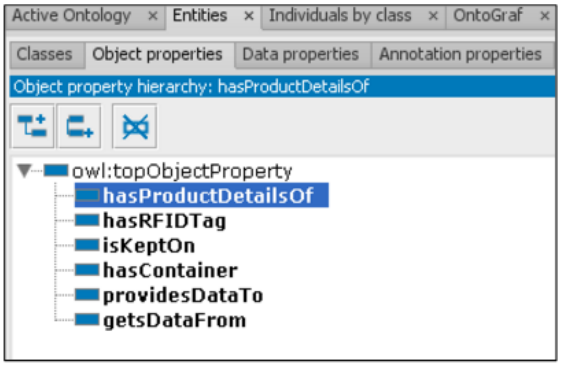

Figure 5.20: Shows relationships between objects `WeighingSensor`, `RFID` *&* `Container`. *These object properties are defined in weighing module ontology.*

WeighingSensor-RFID relationship is described as:

- WeighingSensor **getsDataFrom** RFID
- RFID **providesDataTo** WeighingSensor

WeighingSensor-Container relationship is described as:

- WeighingSensor **hasContainer** Container
- Container **isKeptOn** WeighingSensor

RFID-Container relationship is described as:

- RFID **hasProductDetailsOf** Container
- Container **hasRFIDTag** RFID

Eye-tracking is taken as a dummy module to discuss interoperability between modules. The sufficient statistic for eye-tracking module is the level of cognitive load of worker. This is defined as a boolean where 0 means low and 1 means high. The module should report this boolean variable to the central system. Hence, upper ontology of assistance system defines class `Eye-trackingSensor` and data property `cognitiveLoadOfWorker` as the minimal data required to receive information from eye-tracking module. Eye-tracking module needs the position of containers kept on weighing module for calibration. The position coordinates are defined in weighing module ontology. Therefore, to obtain the position coordinates `(x, y, z)`, eye-tracking module ontology imports weighing module ontology. Figure 5.21 shows data properties imported by eye-tracking module ontology.

5.3.2 Implementation of centralized organizational scheme

Centralized organizational scheme of ontologies described in section solution is implemented here. Since upper ontology of assistance system is discussed in the previous section, it is not defined again in this section. Mid-level ontology, which differentiates the two solutions, is discussed in this section. All ontologies describing both the organizational schemes are created in Protégé. These ontologies are provided in digital appendix attached with the thesis.

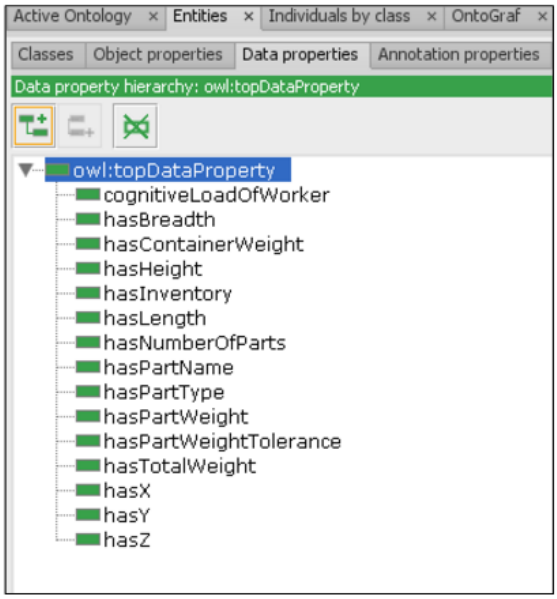

Figure 5.21: Shows data properties of assistance system upper ontology and weighing module ontology imported by eye-tracking module ontology.

Mid-level Ontology

Mid-level ontology serves as a bridge between the assistance system upper ontology and upper ontologies of all modules. Upper ontology of assistance system has classes : `WeighingSensor`, `RFID` & `Eye-trackingSensor` and their data properties : `hasNumberOfParts`, `hasPartName` & `cognitiveLoadOfWorker` respectively as described in Section 5.3.1. Mid-level ontology imports the upper ontology of assistance system and further defines data of other modules in order to provide a repository for other modules to fetch data. Weighing module contains a weighing sensor, an RFID and a container. `WeighingSensor` and `RFID` classes are already imported and `Container` class is defined in the mid-level ontology. Furthermore, all or some data properties described in the information model are defined in the mid-level ontology as decided by engineers developing various modules.

For example, all data properties of weighing module can be defined in mid-level ontology whereas only few data properties of container and RFID as shown in Figure 5.22. Data properties: `hasLength`, `hasBreadth`, `hasHeight`, `hasInventory` of class `Container` are defined in Weighing module upper ontology. Similarly, data property `hasPartWeight` is defined in class `RFID`.

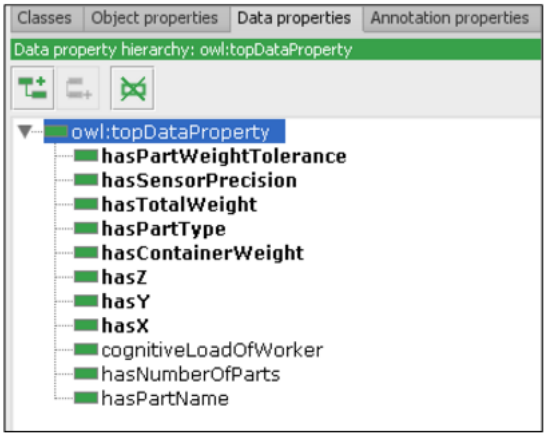

Figure 5.22: Shows data properties of mid-level ontology. Mid-level ontology imports upper ontology of assistance system and defines data properties of modules. Mid-level ontology may define all or some data properties of modules depending on design.

Object properties are defined as shown in Figure 5.20. Modules import mid-level ontology and add data when required. This design helps engineers to build a list of entities which is accessible by all modules. It also helps in building a more secure system using the list of permissible entities for which a module can query. Further, checks can be implemented on the entities a particular module can query. In the next step, deployment is of the ontologies is discussed.

5.3.3 Deployment

During deployment, the ontologies created thus far are imported by modules. At this stage, individuals are created according to the setup of system. As discussed earlier, the assistance system has a central system, a weighing module and an eye-tracking module. Weighing module, further, has three containers, weighing sensors and RFID tags. These individuals are defined as shown in Figure 5.23. Individuals are defined and described in the deployment for both classes imported from other ontologies and classes defined in the particular ontology. Examples of each object are discussed next in the section.

Class `WeighingSensor` has three individuals as shown in Figure 5.23. Each individual is defined and described by assigning values to data properties. Axioms of class `WeighingSensor` are inherited by the individuals of the class. Object properties define re-

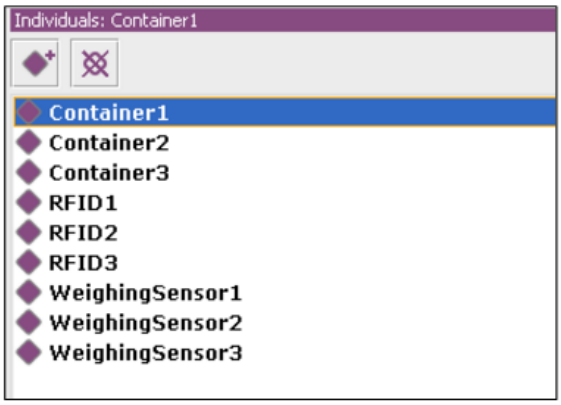

Figure 5.23: Shows individuals created during deployment. Weighing module has three `WeighingSensor`, `RFID` & `Container`.

lations between objects: `WeighingSensor1` **hasContainer** `Container1` and `WeighingSensor1` **getsDataFrom** `RFID1`. Similarly, other individuals of class `WeighingSensor` are defined.

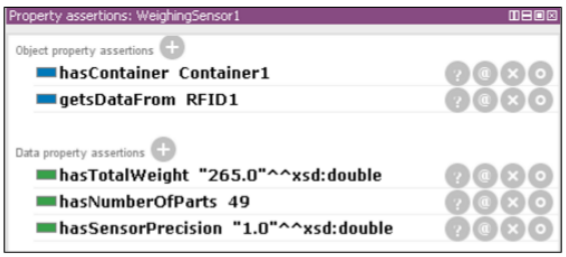

Figure 5.24: Shows values of data properties of `WeighingSensor1` *and its relation with* `RFID1` & `Container1`.

Class `Container` has three individuals `Container1`, `Container2` & `Container3`. Description of `Container1` is shown in Figure 5.25 by assigning data properties to it. Object properties are defined as relationship between classes. `Container1` has two object properties: `Container1` **hasRFIDTag** `RFID1` and `Container1` **isKeptOn** `WeighingSensor1`. Similarly, `Container2` and `Container3` are described.

Class `RFID` has three individuals `RFID1`, `RFID2` & `RFID3` as shown in Figure 5.23. `RFID1` is described by assigning values to data properties. Object properties are defined as: `RFID1` **providesDataTo** `WeighingSensor1` and `RFID1` **hasProductDetailsOf** `Container1`. Similarly, other container individuals are described.

There are data properties whose values are not explicit in the system, i.e., these values

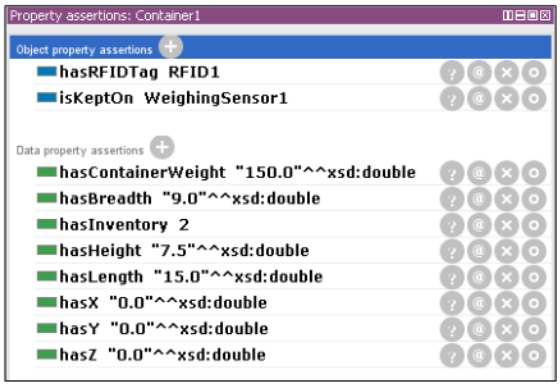

Figure 5.25: Shows values of data properties of `Container1` *and its relation with* `RFID1` *&* `WeighingSensor1`.

Figure 5.26: Shows values of data properties of `RFID1` *and its relation with* `Container1` *&* `WeighingSensor1`.

are not predefined information as RFID data or raw values from sensors. These data properties are processed before merging ontologies. For example, total number of parts in a container is calculated as follows:

$$\texttt{hasTotalNumberOfParts} = \frac{\texttt{hasTotalWeight}}{\texttt{hasPartWeight}} \tag{5.1}$$

where, `hasTotalWeight` is the value provided by weighing sensor and `hasPartName` is the data provided by RFID tag.

Furthermore, the value of `hasNumberOfParts` can be refined to get more accurate result. For example, values of `hasSensorPrecision` and `hasPartWeightTolerance` can be used to error proof the value of `hasNumberOfParts`. Additionally, moving average of sensor values can be used to improve the accuracy of the system. The problems faced during implementation of weighing module are discussed in Chapter 6.

SPARQL Query

SPARQL is a semantic query language for database and is used to retrieve and manipulate data stored in RDF format as discussed in Section 3.6.4. Code Snippet **??** is used by eye-tracking module ontology to query the position of containers `(x, y, z)`.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX central: <http://www.semanticweb.org/thesis/
   UpperOntologyWeighingModule#>

    SELECT ?container ?x ?y ?z
    WHERE { ?container central:hasX ?x .
            ?container central:hasY ?y .
            ?container central:hasZ ?z . }
```

Listing 5.2: Example of SPARQL query. Eye-tracking ontology imports weighing module upper ontology to get definitions of container coordinates and then queries `hasX, hasY, hasZ`.

Prefix `central` is used to import the upper ontology of weighing module. Eye-tracking module ontology gets definitions and axioms of variables `hasX`, `hasY` and `hasZ` by importing `central`. Now it can query the central system for values of these variables. The query has four variables: `?container` denotes container name and `?x ?y ?z` denote x, y, z coordinates of the container. These `(x, y, z)` coordinates can be the coordinates of the upper left corner of container or center of container depending upon the design opted for by engineers. Further, information about length, breadth, height of container can help other modules to ascertain position of workers' hands and eyes in relation to the container's position.

Figure 5.27 shows an example of the values returned by the central system against the SPARQL query 5.2. These values correspond to the position coordinates of individuals of class `Container` created during deployment stage in weighing module ontology. This is the ontology import in decentralized scheme of ontology organization where weighing module upper ontology has all variables defined and described pertaining to weighing module. Eye-tracking module imports the upper ontology of weighing module.

In a centralized scheme of ontology organization, mid-level ontology contains definitions and axioms of all variables present in all modules. The mid-level ontology is imported

container	x	y	z
Container3	"0.0"^	"60.0"^	"0.0"^
Container1	"0.0"^	"0.0"^	"0.0"^
Container2	"0.0"^	"30.0"^	"0.0"^

Figure 5.27: Shows values returned by the SPARQL query for position of containers. Values returned show container names and their corresponding `(x, y, z)` *coordinates.*

by eye-tracking module ontology instead of weighing module upper ontology to access positions of containers. Thus, the prefix in the SPARQL query will change to the address of the mid-level ontology instead of the upper ontology of the weighing module: `PREFIX central: <http://www.semanticweb.org/thesis/midlevelontology#>`.

6. Implementation

This chapter deals with issues that may arise while deploying the system. These are the problems faced during implementing weighing module. The implementation dealt with reading data from weighing sensors and RFID tags, processing the data and sending the required information to central system of assistance module.

First phase of implementation for weighing module is completed. This includes reading data from sensors and recognizing RFID tags through RFID readers. Sensor data and part information from RFID tags are used to provide information about the part in each container. In the present scenario, Rapsberry Pi (RPi) attached to weighing module provides information about part name, total number of parts contained in each container and change in the number of parts for each container to a central system. RPi also sends a message to signal low inventory for parts if number of parts in a container drops below a threshold level.

In the next phase of implementation following features should be added to the system:

- Central system contains data in an ontology. This means that it should be able to receive updates in form of partial / complete ontologies and answer SPARQL queries.
- Weighing module should communicate its readings to the central system through a partial / complete ontology transfer.
- Third party module should be able to query data using SPARQL queries.

Figure 6.1 shows the setup of weighing module used in implementation. Weighing module has three weighing sensors with container kept on each sensor. Further, an RFID tag is attached to each bin. RFID tags contain data regarding parts, for example type of part, part name which can be read by RFID readers kept on weighing sensors as shown in the figure. However, it is noteworthy that there is a scope of human error in this scenario as the part details are entered manually and while filling the container it should be ensured

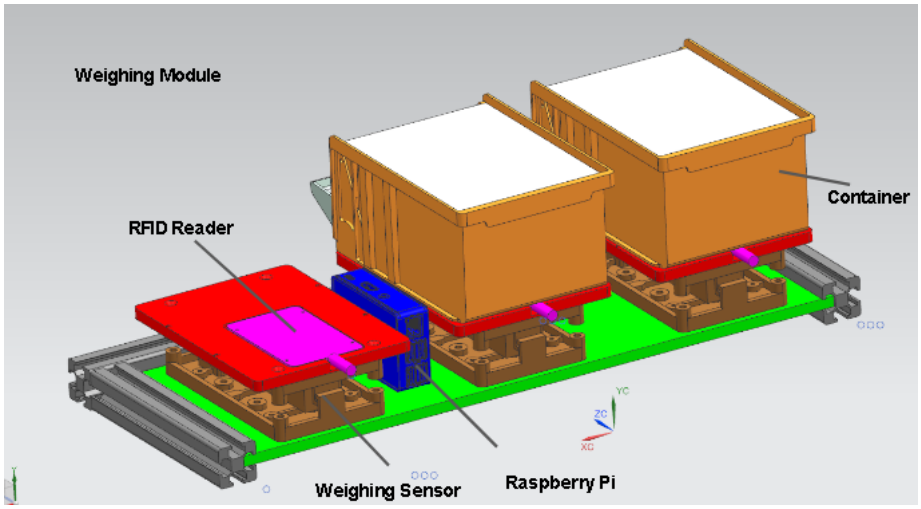

Figure 6.1: Shows weighing module used for implementation. Module consists of three weighing sensors, each with a container and an RFID reader. Weighing sensors and RFID readers are connected to Raspberry Pi. Source: SmartFactory^{KL}.

that container has the corresponding parts. This, indeed, can be one area of further improving the system and making it error proof.

This chapter discusses various hardware and communication choices available for implementation and makes recommendations based on the issues faced chronologically during implementation. First, different hardware options and procedures are discussed followed by communication between the central system and the weighing module. In the end, some implementation recommendations are made.

6.1 Hardware Design

As discussed in Section 3.5, an assistance system consists of a central system and CPS modules. In this implementation, weighing modules are the only kind of module attached to the central system.

In Figure 6.2, each shelf has one weighing module. The weighing module may have one or more weighing sensors depending on its configuration. In this case, each weighing module has three weighing sensors. These sensors can be configured to perform within

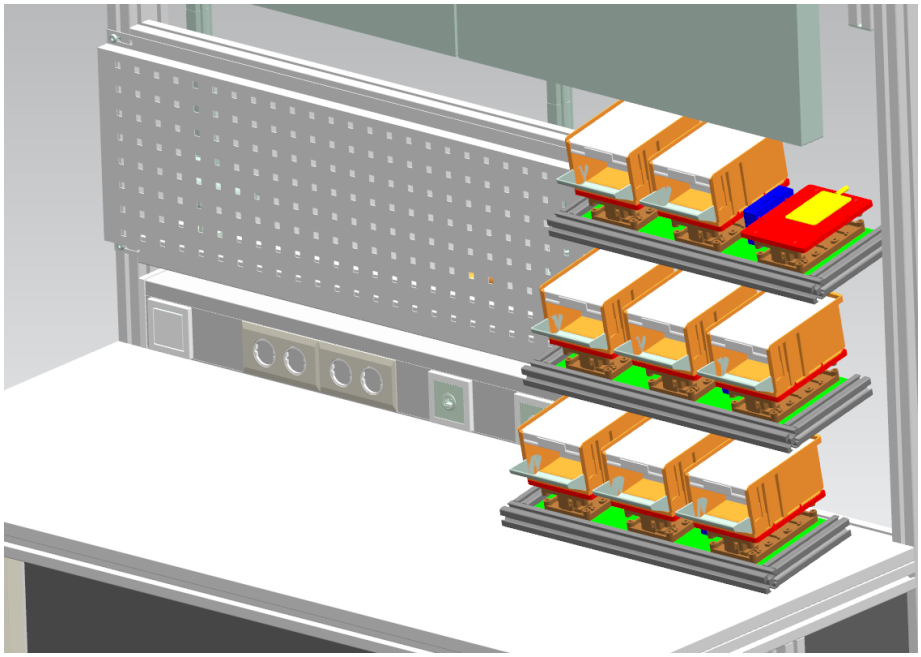

Figure 6.2: Shows an example of a possible setup of weighing modules in an assistance system. An assistance system can have more than one weighing module. Source: SmartFactory^{KL}.

a set range of load capacity and precision. Hence, during configuration each sensor's precision settings can be individually set.

Mettler Toledo provides the weighing module and the tools to interact with the module. Using these tools, weighing sensors can be configured, calibrated and read/tested. Figure 6.3 shows the interface of the tool, named *Speed*, used to configure the weighing sensors. The weighing sensors could be calibrated from 1 gram to 20 gram. In this implementation, sensors have precision of 1 gram, 2 gram and 10 gram.

Calibration of the sensors avoids any discrepancies in weights. Calibration follows a procedure wherein weighing module number and weighing sensor number are given to calibrate the weighing sensor against dead-load. The Weighing module's manual, which describes the calibration procedure in detail, is attached in the digital appendix of the thesis. Dead-load are the loads that remain constant over time including the weight of structure and immovable fixtures. Since each weighing sensor has a standard container and an RFID reader, the weights of these two entities are included in the dead-load of weighing sensor for the ease of calibration procedure. Including the weight of container

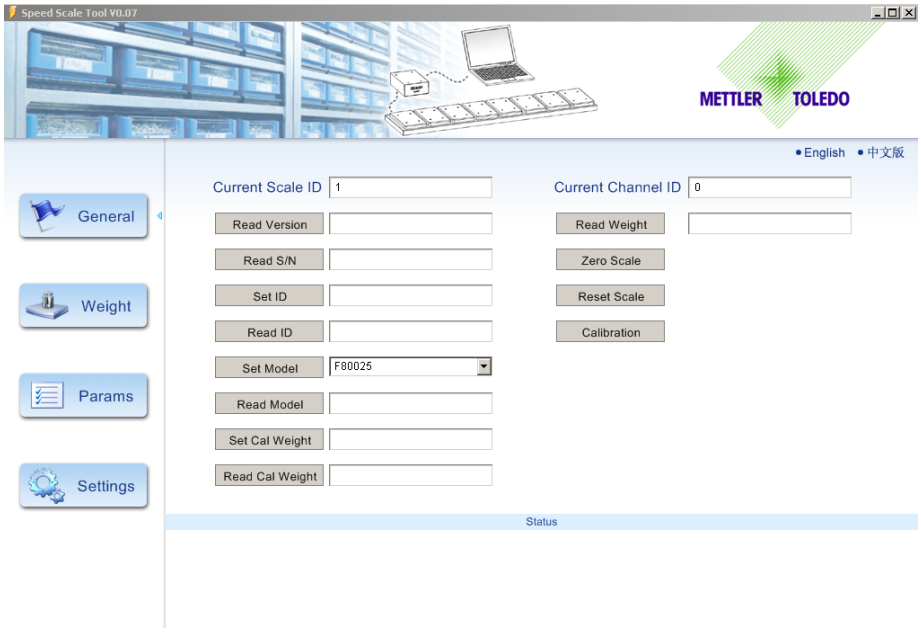

Figure 6.3: Shows UI of Speed *software used to configure and calibrate weighing sensors. Source: Mettler Toledo Software.*

and RFID reader in dead-load would lessen the complication in finding the number of parts as the weighing sensor will report only the weight of parts as opposed to the weight of the whole setup. However, weight of container is still a data node in our information model and ontologies in order to capture as much data as possible.

This software implementation can be done for a micro-controllers, a Raspberry Pi (RPi) or a computer. RPi has more computational power than micro-controllers. It also has lower cost & lower power consumption than computers and is easy to use for programming. Hence, RPi was used for the implementation.

The weighing module uses communication protocol RS485 whereas the de-facto protocol for computer communications is RS232. There are multiple ways of converting RS485 signals to RS232. One off-the-shelf solution is using a converter box provided by Mettler Toledo. Other options are using RS485 to RS232 USB-serial converter or an RS485 shield. A shield is an extension board that sits on RPi, receives voltage corresponding to RS485 and converts it RPi standard I/O voltages. These setups for communication were tried and a RS485 shield was chosen for use in implementation as it does not require any converter to change the voltage and is mounted directly on the RPi. Figure 6.4 shows the final setup used to implement the weighing module.

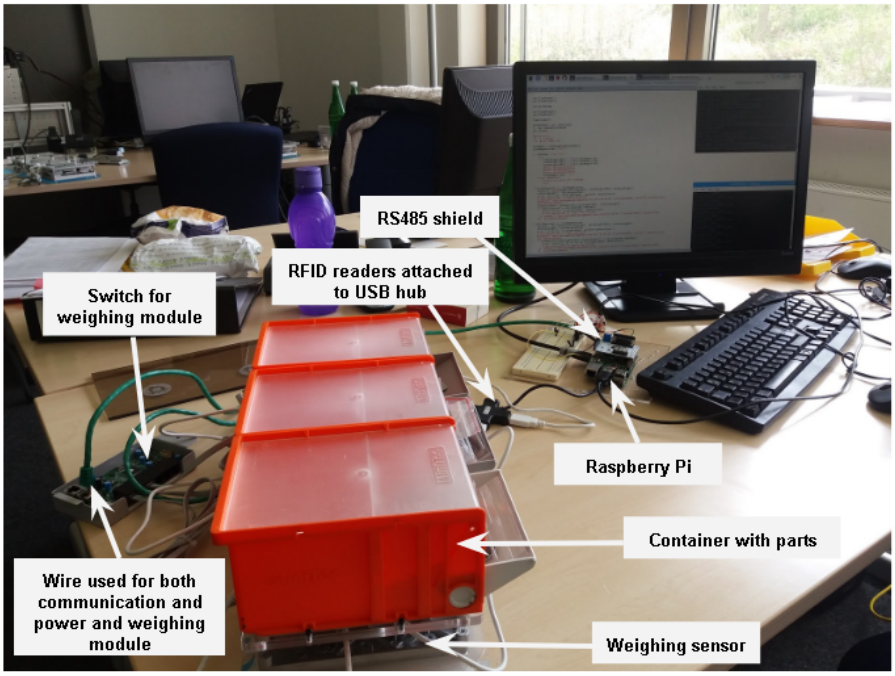

Figure 6.4: Shows weighing module implementation which deals with collecting data from weighing sensors and RFID tags and extracting number of parts in each container. This information is sent to the central system.

A switch is used to power the weighing module that consists of multiple weighing sensors. Figure 6.4 shows three weighing sensors attached to a switch. The weighing module uses a LAN (*RJ45*) cable to power and to send/receive data to RPi. LAN cable contains eight wires in all: two wires for $+V$, two wires for $-V$, two wires to earth and two wires for communication. An external power supply of $12V$ is used to power both weighing module, through the $+V$ and $-V$ wires of the LAN cable, and the RPi. The transmission and reception (Tx-Rx) wires are attached to the RS485 shield at ports A and B (see Figure 6.5).

HEAD | L | x | C | END

Listing 6.1: An schematic layout of an encoded message to the weighing module. See Figure 6.6 for an explanation of each part of the command.

A Python program is written to calibrate the module. The program sends byte-encoded messages to the sensor which responds by sending bytes back. The byte code message for different commands are provided by Mettler Toledo as the documentation for weighing module. The documentation and code written for reading sensor data are provided in the

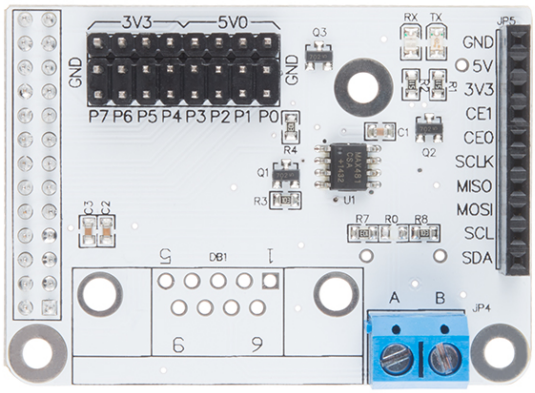

Figure 6.5: An image of the RS485 shield we used. The I/O pins `A` *and* `B` *are the blue pins towards the bottom left (clearly marked). Source:* `https://www.sparkfun.com/products/13706`.

digital appendix attached to the thesis. The module works on the principle that it gets a predefined encoded message from the user/RPi and depending on the value of message it returns the desired bytes. Listing 6.1 shows the general encoding of messages to/from sensors. Definitions regarding the command are provided thereafter in Figure 6.6.

Raw data from weighing sensors are read in hexadecimal form and is converted to decimal for the ease of reading and understanding. Part information is read from RFID tags which are placed at the bottom of containers. RFID readers are attached to RPi through a USB hub as shown in Figure 6.4. RPi collects the data from sensors, part information from RFID tags and extract information regarding total number of parts and change in number of parts for each container. Python code also incorporates the detail of inventory threshold for each part. If inventory for a part goes below this threshold, a flag is raised to signal that refilling of parts is required.

HEAD	0xF2
L	Length in bytes counting from the byte after the L-byte to the end including checksum byte and END
x	Command byte and literal values to the command
C	Checksum byte XOR function on all bytes preceding the checksum byte, not including HEAD byte.
END	0xF3

Figure 6.6: Shows definitions of command bytes in a communication protocol used for weighing module. Commands start with a HEAD character followed by a length byte, the command itself, a checksum byte, and an END character at the end.

6.2 Communication Design

This section describes communication between RFID readers, weighing sensors, RPi & the central system and makes a few recommendations. In the implemented weighing module, communication uses both push and pull modalities. The weighing module is calibrated before use. RPi pulls data from weighing sensors & RPi and RFID readers push data to the central system.

For the ease of understanding, a sequence diagram is drawn showing communication between different entities shown in Figure 6.7. Sequence diagram shows how objects operate with respect to time.

Weighing sensors send encoded bytes to weighing module which is processed to interpret the weight read by the sensor. RFID readers send the part information to the central system and weighing module. Weighing module uses data from RFID readers to map containers, and the sensor weight values, to parts. Weighing module further processes the data using sensor value `hasTotalWeight` and RFID information `hasPartWeight` to get `hasNumberOfParts` for a container.

Communication between RPi and weighing module is written in Python and communication between RFID reader and RPi is done using ZeroMQ. ZeroMQ is an efficient, embeddable library which handles messages asynchronously in background threads [**hintjens2013zeromq**].

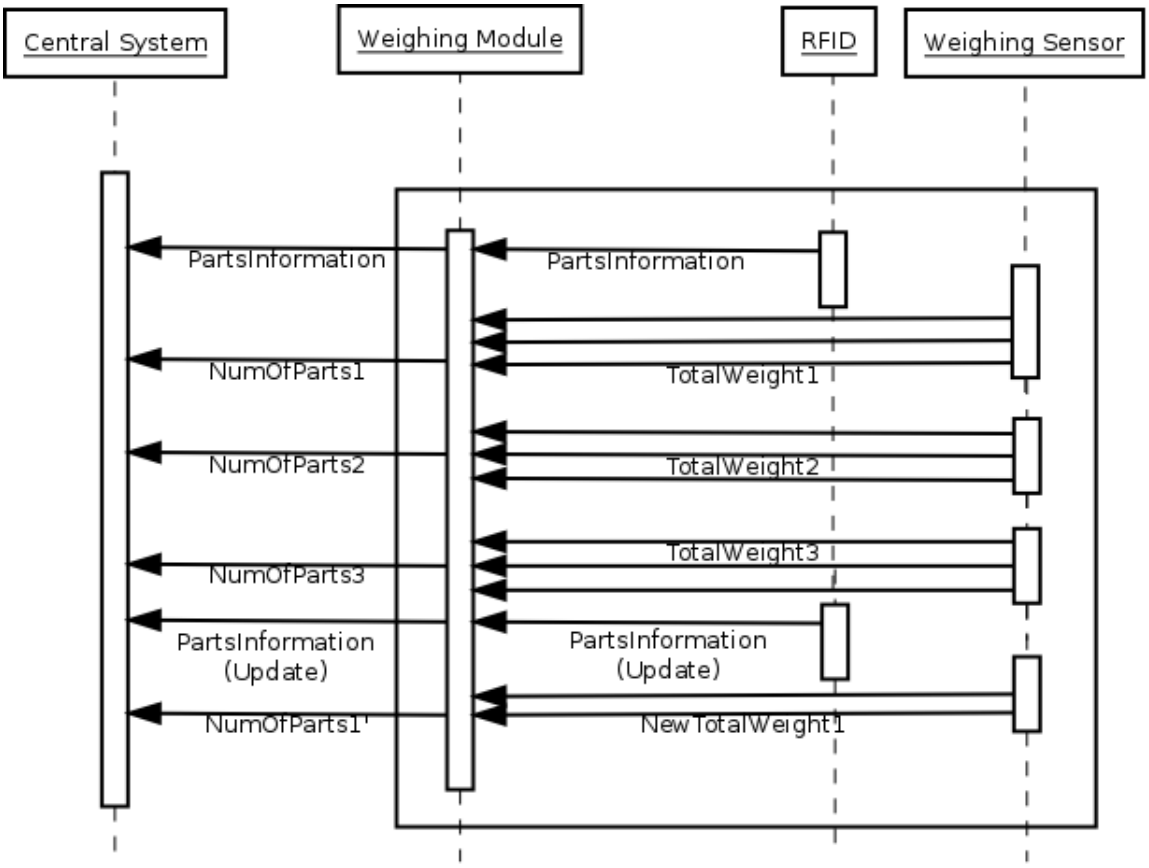

Figure 6.7: Shows sequence diagram of communication between weighing module and the central system. Weighing module consists of weighing sensors and RFID readers. ACK of receiving messages are implicit (not shown).

It does not need locks or semaphores which makes it easy to use. It is highly useful in Service Oriented Architecture where services can join and leave the network any time. ZeroMQ also automatically queues messages when needed. This helps in error proofing the system. For example, messages sent by RFID readers while the module is busy reading the weighing sensor values would not be lost as ZeroMQ queues it. It also handles network errors effectively. In the implementation, RPi sends variables, `hasPartName` and corresponding `hasNumberOfParts`, to the central system. There is a layer of abstraction (implemented in C#) between the code running on the module and the central system.

The time interval to update values is decided depending on the processing power of the machine, amount of data to be transmitted and hardware challenges like heat dissipation.

The degree of system sophistication and intelligence depends on the hardware and algorithms used in the process. In the implementation, RPi sent data to the central system once every two seconds.

6.3 Recommendations

The weighing module must be developed keeping in mind the failures and errors that might happen while deploying the system. Thus, the system must have certain properties which make it error proof. Intertwined with the desirable properties of the system, some practical recommendations regarding implementation are also made in this section. The inspiration for these recommendations comes from design of concurrent systems [**andrews1991concurrent**].

Safety property asserts that *nothing bad* happens. The foremost requirement to implement this property is the system should not be in an invalid state at any point in time. For CPS, it means that the module should never report values which are not computed from its sensor values. A discrepancy may result from the scenario that the module reports a value to the Central System after it has probed one sensor but before probing another sensors, if the report contains values which were computed using both the sensor's readings (one of the sensors may have out-dated value). An invalid state can also be a deadlocked state where there are no outgoing transitions, such as an error state. Semaphores, mutex and locks should be judiciously used during development to avoid such scenarios. As a side-note, handling missing values (which is a subset of error states) gracefully is a potential future extension of this thesis.

Liveness property asserts that the system will perform its intended use *eventually*. In other words, liveness means that the system will continue to make progress. This implies ensuring that the semaphores and mutexes will be unblocked and the module will, eventually, send data to the Central System. Though several race conditions can be avoided simply by using atomic operations exclusively, it is possible to end up in a live-lock. Say the module has a *hard* parameter which controls after how long a sensor's data is considered *stale*. Then say the module reads data from one sensor, and then while it is reading data from another sensor, the data from the first sensor becomes *stale*. So the module will go back to re-reading data from the first sensor and in the meanwhile data from the second sensor becomes *stale*. This could bind the module into a sensor reading infinite loop. During implementation, such situations should be carefully thought about and the liveness of the module should be tested/verified under the most extreme of conditions.

Encapsulation is another way of making system more reliable. Encapsulation is restricting direct access of software components so that they cannot interfere with other subsystems or privileged-level software. It keeps faults from propagating which increases the reliability of the overall system.

Finally, in case everything fails, a **watchdog timer** (or a Heartbeat) can be used to detect the catastrophic failures and recover from it. The timer is regularly reset by computer during normal operation, but it timeouts if there is a hardware or software error. The timeout signal initiates a corrective measure by placing the system in a safe state or restoring normal operation. One of the ways this can be accomplished is by using a Hypervisor [**masmano2009xtratum**] which can simply restart the entire module in case the timer timeouts.

These are some necessary properties that the system must have, but not sufficient to ensure that it functions properly. In the end, the deployment and user feedback would be the final test of the module.

7. Discussion and Outlook

This work has tried to address the complex problem of interoperability among assistance modules, and CPS in general, with the help of ontologies. Along the way, various recommendations about development of the information model have been made along with guidance about how to make certain design choices while using a Weighing Module as an example. However, the proposed solution is best looked at as a framework. It leaves open several avenues for future work. Some of the prominent directions will be discussed in this section.

Section 5.2.2 describes two ways of organizing the ontologies for central systems and CPS modules. The section ends with the recommendation that there should be a committee setup which oversees the maintenance of the mid-level ontology. However, setting up such a committee is a non-trivial task which requires careful choice of who the stakeholders are and who holds the power of arbitrage in case of disputes [**jacobs2004world**]. Authentication and authorization mechanisms also need to be designed as discussed in the section.

Further, Section 5.2.2 also talks briefly about the extension of variables imported from upper ontologies as one of the significant areas where design decisions need to be made. By default, ontologies are static objects: additions made locally to imported variables do not propagate to other teams working with the same imported ontology automatically. If such synchronization were possible, it may prevent reinvention of variables and may save the effort to resolve discrepancies among reinvented variables even more than having a committee make these decisions. This is a feature that designers of Protégé may want to look into. In the present scenario, this issue can be resolved to some extent by informing all stakeholders about local changes made to the ontology via third-party solutions like mailing lists.

Knowledge mapped in ontologies may evolve over time due to modifications in conceptualization and adaptation to incoming changes. Evolution of ontologies cause backwards

compatibility problems and thus hamper its reuse. A versioning mechanism is required to explicitly define versions and relations between versions [**klein2001ontology**]. This is not discussed in this thesis but is a necessary feature to promote adoption of ontologies in real-life settings.

Another feature which is missing from the ontologies is explicit modeling of the notion of time. As noted in Section 5.2, the issue can be addressed to some extent by adding time-stamps to values or through programming outside the ontologies. However, adopting CHRONOS [**preventis2014chronos**] can make the handling of time more elegant and will allow us to include constraints on temporal evolution of values (e.g. value X can only be set after value Y has been set). This is an interesting direction of research which can lead to much more expressive ontologies.

The problem of missing data has largely been ignored in this thesis. A guide is necessary to deal with it gracefully. For example, a module may provide all the necessary information required by the central system but lack data suggested by the mid-level ontology. A protocol should be developed to deal with such errors without halting or rendering the complete system invalid. The protocol should also have a way to communicate these errors/omissions to engineers and the central system. Note that some recommendations are made in Section 6.3 to harden the system against such issues.

An immediate next step in line with the thesis work is to further develop an end-to-end implementation using ontologies as discussed in Chapter 6. Implementation would ensure that the demonstrations made in Protégé actually work with hardware sensors and that the modules can access and exchange data. As such, there are no foreseeable hindrances that one might face during implementation but run-time errors may occur which would need to be resolved as they are witnessed. A prototype implementation in Protégé is done as a part of this thesis.

Section 6.2 discusses selected challenges faced during designing the communication phase of the system. There can be hardware errors as well as run-time errors which might render system in invalid or end state. It is important to work in this area to make the system safe and reliable. A few such errors can be taken care of during programming by providing and filtering data provided by sensors and by queuing the messages sent and received during communication. Nevertheless, this area needs further exploration and, on top of the guiding principles offered in Section 6.3, needs a comprehensive treatment.

8. Summary

The thesis deals with development of a CPS module for an assistance system during manual assembly. Further, a framework for designing and semantically describing CPS modules which allows for interoperability across modules. Semantic descriptions are largely self documentation of modules and abstraction over the low level details of raw data collected through sensors. Ontologies are chosen for semantic description of modules as they formally and explicitly define objects and relationships between them. An upper ontology is a high level, domain-independent ontology which describes framework and basic vocabulary. An assistance system upper ontology is designed at the beginning of the design of assistance system which contains minimal information needed by the central system from *all* modules. It helps other CPS modules to easily access the information collected by processing data from various sensors. Thus, each module collects and processes data and provides information to other modules.

The framework proposed in the thesis provides steps to take in order to design a CPS module and describe it semantically for interoperability of data. At the advent of design phase of a CPS module, its scope and intelligence need to be defined. To define scope of modules, an analogy to the concept of sufficient statistic in mathematics is drawn. Sufficient statistic for a module is defined by the minimal information needed by the central system from modules to understand and exchange data. System intelligence depends on the hardware and algorithms used to process information. In the next step, information model is designed which identifies all input data, both from sensors and otherwise static values of the system. The data is preprocessed to get the minimal information required by the central system to operate. Based on the information models, ontologies of modules are created.

Two approaches for designing ontologies of modules are proposed: a decentralized organizational scheme and a centralized organizational scheme. Decentralized scheme defines upper ontologies of assistance system and modules. Module ontologies import upper ontology of assistance system and any other module they wish to interoperate with. In

contrast, in centralized scheme along with upper ontologies of assistance system and modules, a mid-level ontology is created which defines variables of various modules. The mid-level ontology imports assistance system upper ontology and upper ontologies of modules import the mid-level ontology. Pros and cons of both organizational schemes are discussed.

The development process of a weighing module is used as a case study for the thesis. Sufficient statistic, system intelligence and various design decisions which need to be made while developing the information model and the ontology of weighing module are explored. Ontologies for both decentralized and centralized organizational schemes are created with a detailed step-by-step description. An example ontology of eye-tracking module is also created and data exchange between weighing and eye-tracking modules is demonstrated in Protégé.

On the hardware demonstration front, first phase of a weighing module implementation is completed. In this phase, RFID tags are used to calibrate the position of containers and to get information about the parts stored in containers. Raw data collected by weighing sensors are processed to get information about the total number of parts in each container. Complete information about part and number of parts in each container is sent to the central system of assistance system. Python and ZeroMQ are used to code and send data over wire. Communication is described using sequence diagram and a few recommendations regarding implementation are made. In the second phase, ontologies designed for use case to describe CPS module and query data across modules should be implemented on hardware in real-time.

Finally, discussion of the limitations and future directions concludes the thesis.