RoboCIM: Towards a Domain Model for Industrial Robot System Configurators

Daniella Tola[1], Cláudio Gomes[1], Carl Schultz[1], Christian Schlette[2], Casper Hansen[3], and Lukas Esterle[1]

[1] Aarhus University, Aarhus, Denmark
{dt,claudio.gomes,cshultz,lukas.esterle}@ece.au.dk
[2] University of Southern Denmark, Odense, Denmark
chsch@mmmi.sdu.dk
[3] Technicon ApS, Hobro, Denmark
cha@techicon.dk

Abstract. Determining which components are required for a system configuration, and whether they are compatible, can be a difficult task, especially in an industry with significant amounts of information that resides within a group of experts. In this paper we illustrate some of the main challenges we and our industrial partner (Technicon) face when configuring a robot system (typically consisting of a robotic arm, end effector, coupling device, and a base) and present our domain model, Robot System Configurator Information Model (RoboCIM). We formalise the model within a defeasible reasoning framework, in order to explicitly capture cases where information is missing or is obtained from the system integrators' experience. We provide a prototype implementation of the framework in ASP and evaluate it on a subset of components from Technicon's component catalogue, illustrating the feasibility of the configurator.

Keywords: Robot configurator · Incomplete information · Non-monotonic reasoning · Defeasible reasoning · Knowledge engineering · Answer Set Programming

1 Introduction

In the context of industrial robotics, *integration* is the process of introducing and merging robotics hardware, peripherals, software, and supporting technology into a production, or manufacturing line, to automate it [10]. The rapid re-purposing and reconfiguration of manufacturing cells allows for increased resilience in existing supply chains in response to unexpected disruptions such as shifting policy and market preferences. Robotic system integrators (individuals or businesses) play an essential role in reconfiguration, which still requires extensive expertise and manual effort (e.g. which grippers to use, which robot models to select, or which data connections to rely on).

The challenge of configuration is not new. In other areas such as personal computers (PCs), customisation tools have had tremendous success, thanks to standardised interfaces[4]. These tools are easy to use, contain comprehensive databases, and allow consumers to pick and choose the parts desired based on their performance needs, while ensuring compatibility among the hardware elements.

Our vision, and that of our industrial partners, is that robot configuration tasks should be as easy as their PC configuration counterparts. Recent developments [11,12,14] attempt to simplify robot integration by modelling skills for automation of robots and end effectors. While this is an important milestone, the state of the art assumes the information regarding these skills is available. We focus instead on how to represent compatibility constraints, in the face of incomplete, contradictory, or rapidly changing, information. We sketch a solution to this using Answer Set Programming (ASP).

The challenges described in this paper have been identified in the course of our work with Technicon, while assembling robotic cells in the Aarhus University Digital Transformation Lab, as well as discussions with Technicon's engineers. Technicon is a Danish system integrator, that has been in the market for 7 years, and has made more than 300 robotic system integrations. So far, Technicon has been able to successfully operate due to its engineers' extensive experience and internal documentation, determining compatibility constraints between robotic components. However, it recognises the need to formalise these rules, into tools that facilitate robotic system configuration.
The main contributions of this paper are:

1. (Section 3) We report on some of the challenges we encountered. The main factor in these is the uncertainty and incomplete information.
2. (Section 4) We sketch our open-ended domain model, the Robot System Configurator Information Model (RoboCIM), which describes how to capture knowledge on robot products. We illustrate how to go from product specifications to a formalised domain model, and how to tackle real world challenges using a rule-based approach.

We give a brief description of the robotic domain in Section 2 and demonstrate the results of our prototype implementation and validation in Section 5, concluding in Section 6.

The robot components and challenges we describe originate from real industrial manufacturers. In order to avoid harming the companies, we have anonymised their names.

2 Background

In this section we describe what a robot system configurator is and introduce the main components in a robot system that we will use in such a configurator. For

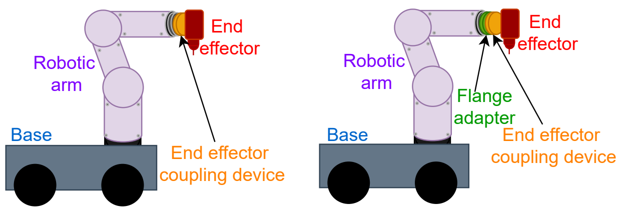
[4] An example customisation tool: https://pcpartpicker.com/list/.

the remainder of this paper we will refer to a robot system configurator simply as a *configurator*.

According to [9], to create a configurator, the knowledge of the domain can be represented, for example, by creating a domain model. Utilising this model, the configuration can be generated using algorithms or rules. Each object in the domain model has properties, which can be *attributes*, *resources*, or *ports*.

A robot system consists of one or more robots, end effectors, and other devices combined with the robot to perform a particular task [2]. A simple example, illustrated in Figure 1, is a robotic arm with an attached end effector, mounted on top of a mobile base. The most common tasks performed by such robot systems are: *Pick and Place*, where the goal is to pick up an object and place it in another location, and *Screwdriving*, where the goal is to tighten screws on a surface.

Without loss of generality, we focus on robot systems consisting of the components between the robotic arm and the tool, i.e. we do not consider the base that the robotic arm is mounted on. The reason for this choice is that these components cover a wide range of 10+ manufacturers. Most of these components have mechanical and data interfaces, which are used to connect them with other components. We briefly describe each of the components and their relevant characteristics below.



(a) Example Setup A (b) Example Setup B

Fig. 1: Example of two robot setups with a different number of components.

Robotic Arm has mechanical, data, and electrical interfaces. The tip of the robotic arm is called the robot flange. The mechanical interface of the robot flange is described using the ISO 9409-1 standard [3]. This can be used when defining the compatibility of the mechanical interface. The supported data interface is described by Industrial Communication protocols [1]. The main property of the electrical interface of a robotic arm is the maximum current that can be drawn by the end effector.

End Effector Coupling Device (EECD) has two mechanical interfaces, one for connecting with the robotic arm, and one for connecting with the end effector. It also has an electrical interface for supplying the current from the robotic arm to the end effector.

Flange Adapter has two mechanical interfaces, as it adapts from one interface to the other in order to be able to connect a robot flange to an EECD with a different mechanical interface.

End Effector is connected to the tip of the robotic arm, and is application specific, i.e. gripper for Pick and Place, screwdriver for Screwdriving applications. The end effector has a mechanical interface that connects to the EECD. Typically, both the end effector and EECD are developed by the same manufacturer.

Data Connection can be provided using different components, such as a Data Cable or a Data Control Box. The Data Control Box supports more communication protocols.

3 Challenges

In this section we describe four main challenges that were found during the modelling phase of the configurator. The underlying source of each of these challenges is incomplete information.

CH1: Unexpected Sources of Information When looking for technical information about a product, it is typical to expect this information to be present in the data sheet or technical report of the product. Figure 2 illustrates an example of unexpected sources of information, where the ISO of the robot flange of a robotic arm was not found in the data sheet of the product. Instead, this information was found in the data sheet of a compatible end effector, which is manufactured by another company.

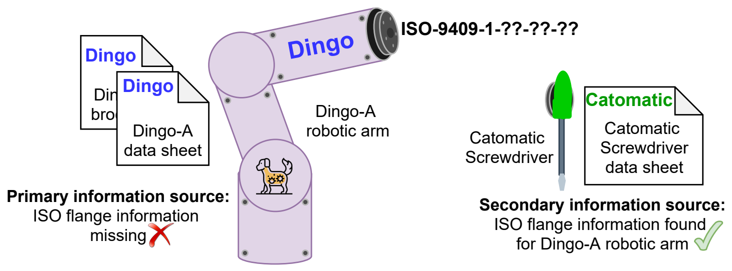

Dingo
Dir
bro
Dingo
Dingo-A data sheet
Dingo
ISO-9409-1-??-??-??
Dingo-A robotic arm
Primary information source:
ISO flange information missing
Catomatic Screwdriver
Catomatic
Catomatic Screwdriver data sheet
Secondary information source:
ISO flange information found for Dingo-A robotic arm

Fig. 2: An example of unexpected sources of information, missing from the primary data sheet of a product, but found in a related product's data sheet.

CH2: Misleading Compatibility When a data sheet states that a product supports an Industrial Communication Protocol [1], it can be expected that, this product port is compatible with other products supporting the same protocol. However, this was not the case for one of the product pairs we worked with, exemplified in Figure 3. The data sheet of an end effector did not state anything about the communication protocol it supports, and after contacting the manufacturer, we obtained a non-public document which specified the end effector supports Modbus TCP, which is also supported by a robotic arm. The document then also specifies that the end effector *requires* a Data Control Box, that also supports Modbus TCP, in order to be compatible with the robotic arm.

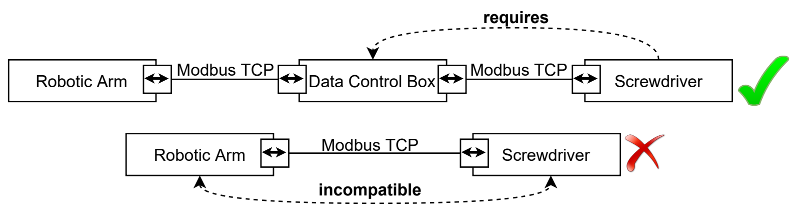


Fig. 3: Illustration of the challenge of misleading compatibility. Note the shown configurations exclude the mechanical parts for simplicity.

CH3: Implicit Information In some cases, component properties were inferred from the name of the component, and not from its data sheet. The example, illustrated in Figure 4, shows that the maximum current load of the *EECD 3A* can be interpreted as 3A, but this information is not in the data sheet. Moreover, it leaves the authors to wonder what the maximum current load of the *EECD* component is. Performing an empirical test of the configuration showed that the *Screwdriver* was incompatible with the *EECD*.

CH4: Misleading Incompatibility Some data sheets specify that two products A and B are incompatible in one part of the document, while revising this by stating that A and B can be made compatible when some precondition is fulfilled. An example is illustrated in Figure 5, describing that the Robotic Arm is incompatible with EECD, unless the Data Control Box is used for the data connection. This challenge example originates from the manufacturer adding the *EECD IO* to their component list in order to support using a Data Cable. This introduced the complexity of supporting both the configuration with the *EECD IO* and the legacy configuration with the Data Control Box.

4 RoboCIM: Robot System Configurator Information Model

In this section we describe the different concepts, layers and rules that we apply in the methodology of developing a configurator, illustrated in Figure 6. We present how RoboCIM has been designed to tackle the challenges presented

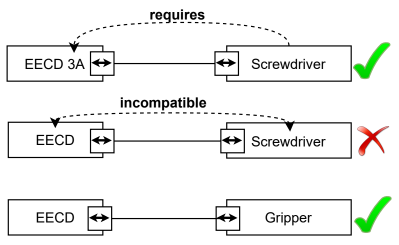


Fig. 4: Example of implicit information.

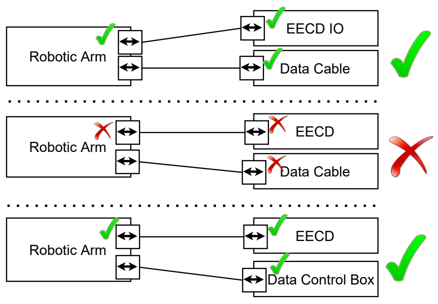


Fig. 5: Challenge of misleading incompatibility.

in Section 3 above, based on the two rule layers illustrated in Figure 6. The Universal Rules layer contains generic rules that can be applied to configurators of various domains, focused on generating valid configurations. The Application Rules layer is domain specific, containing rules that in this case are specific to robot systems. Here, we illustrate some of the main rules used to describe a configurator for robot systems, built on top of the Universal Rules.

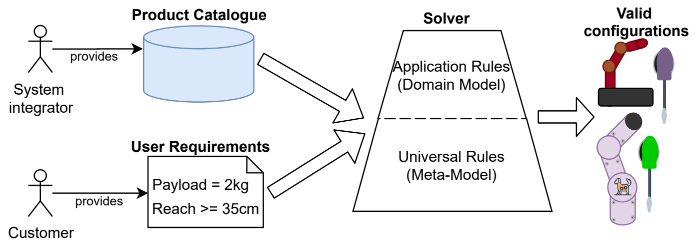


Fig. 6: Overview of the RoboCIM framework and how it works.

4.1 Universal Rules (Meta-Model)

We describe the Universal Rules (Meta-Model) of the configurator to be applied to domains where incomplete information is pervasive. We use similar concepts to the ones defined in [4]: *components*, *connections* and *attributes*, and extend them. Figure 7 shows the developed Meta-Model of the configurator, which includes the main concepts used for configuring products with missing information. White rectangles represent Meta-Model classes, and purple rectangles represent a subset of concrete subclasses of sources and justifications in this first version of RoboCIM. Arrows with triangle arrowheads represent inheritance (subclass) relations, arrows with pointed arrowheads represent association relations, and arrows with filled diamond arrowheads represent composition.

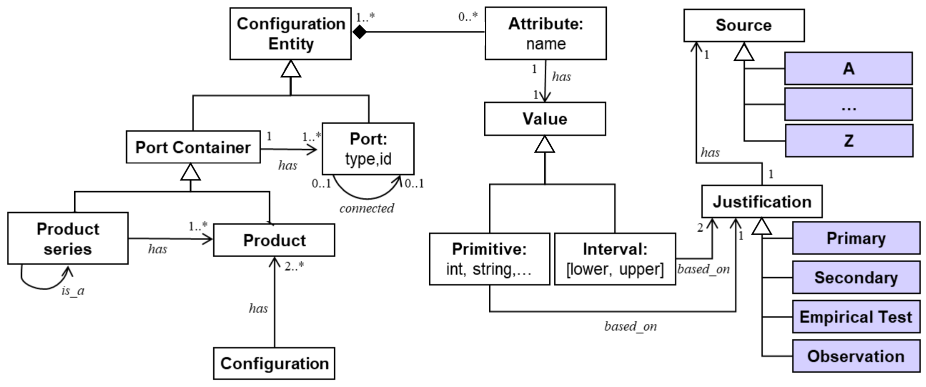


Fig. 7: UML class diagram representing RoboCIM at the *Universal Rules* level.

Product Configuration. A Configuration consists of a number of Products, which are Port containers. Port Containers can be connected to other Port containers through their ports. A Product can be part of a Product series, which is an abstract notion of a product. The concept of Product series represents a group of products that have common attributes, in which a product belonging to this series inherits these attributes. A product series itself can be defined as a subclass of another product series via the *is_a* association relation (transitive, irreflexive). A subset of constraints on Ports and Product series, are shown in Listing 1.1, which are made explicit using Object Constraint Language (OCL)[5].

```
--Constraint 1: A port cannot be connected to itself:
context Port inv: self.connected <> self

--Constraint 2: A product series cannot be its own product:
context Product_series inv: self.is_a <> self

--Constraint 3: A port must have an orientation:
context Port inv: self.attribute.name = 'input' or self.attribute.name = 'output'

--Constraint 4: A port can connect to another port, if they have opposite orientation and
    the same interface (represented by attribute value)
context Port inv:
    self.connected(p2) implies
        Set{self.attribute.name} union Set{p2.attribute.name} = Set{'input', 'output'}
    and
        self.attribute.name.value = p2.attribute.name.value
```

Listing 1.1: Examples of basic Meta-Model axioms described using OCL.

Assuming general properties and compatibility of products in industrial configurators is common, however, special cases exist where the initial assumption about a product may be incorrect, defined by an explicit constraint which underlies hidden information. "*The relationship of support between premises and conclusion is a tentative one, potentially defeated by additional information*" is stated by Koons [7] as one way to describe defeasible reasoning. He also describes defeasible reasoning as "*exception-permitting generalisation*", which is partly how we utilise this form of reasoning.

[5] version 2.4: https://www.omg.org/spec/OCL/

We define default properties of products, which can be defeated by explicit evidence disproving the assumption, based on Nute's defeasible logic framework [8]. This is typically given in forms of incompatibility requirements, that is an extra requirement imposed on the compatibility of a product. We extend the traditional concept of inferring component compatibility in a purely deductive setting, to making contingent inferences about compatibility in a non-monotonic setting.

Listing 1.2 exemplifies a constraint for defining incompatible products, and a constraint on the uniqueness of products in a configuration.

```
--Constraint 5: Two products (productA and productB), known to be incompatible, must not
     exist in the same configuration:
context Configuration inv:
    self.products->excludes('productA') and self.products->excludes('productB')
--Constraint 6: Only one instance of a product can exist in the same configuration:
context Configuration inv: Set{self.products} = self.products
```

Listing 1.2: Examples of configuration related axioms described using OCL.

Tackling Incomplete Information. To formally model cases of incomplete information, the configurator is designed with a notion of information sources from various categories, defined as *justifications*. This provides the basis in RoboCIM for reasoning about defeasible compatibility. Categories that have been used when integrating data from multiple sources (e.g. in [13]), are *primary* and *secondary* information sources.

A *primary* source comes from the manufacturer of the product in forms of data sheets, technical reports or brochures. The *secondary* source comes from another manufacturer of a related product, typically this product can be connected to the referred product. We also add the two categories *empirical test* and *observation*. An *empirical test* can be derived from physical test results of connecting products, which could be performed by system integrators. An *observation* is a property of a product that is defined by a domain expert, which has acquired this knowledge by observing how this product can connect to others.

The veracity of information about (in)compatibility varies for each category. The *primary* source is the strongest justification category, followed by the *empirical test*, and then *observation*. Importantly, RoboCIM is *customisable* such that users can specify which justifications are used to infer defeasible compatibility, e.g. users can setup a RoboCIM solver to generate configurations such that compatibility must be justified by primary sources (thus only delivering strongly justified configurations), as illustrated in Listing 1.3.

```
--Constraint 7: Configurations using information from primary justifications only:
context Configuration inv:
    self.products->forAll(p1 | p1.attribute.value.justification = primary)
    and
    self.products->forAll(product->forAll(
                    port | port.attribute.value.justification = primary))
```

Listing 1.3: Example of justification related axiom described using OCL.

Rationale of Modelling Choices. Both challenges CH3 and CH4 can be dealt with by explicitly defining that these products may not exist in the same configuration. Constraint 5 in Listing 1.2 solves the issue in CH3, and can be extended to include three products for solving CH4. Although CH2 can also be addressed using incompatibility constraints (as CH3 and CH4), this incompatibility constraint differs in the sense that the two products can exist in the same configuration but cannot be directly connected. An incompatibility constraint only on neighbouring products can be used, defining that if two products are directly incompatible, then their ports are also incompatible, shown in Listing 1.4.

```
--Constraint 8: Two directly incompatible products cannot be connected:
context Configuration inv:
    self.products->forAll(p1,p2 | p1 <> p2 and incompatible_neighbours(p1,p2) implies
        p1.ports->forAll(port1 | p2.ports->forall(port2 | port1.connected <> port2)))
```

Listing 1.4: Example of incompatibility axiom described using OCL.

Solving the challenge of CH1 can be done using justification and sources of information. To ensure the certainty of the candidate configurations, it is possible to exclude configurations using knowledge from specific justifications.

4.2 Application Rules (Domain Model)

The Application Rules layer extends the Universal Rules layer with concepts and rules related to the robotics domain, and also incorporates rules to impose the user requirements in the configurator. A robot configuration must contain specific products, as the examples shown in Figure 1. Each of these products must have specific types of ports, for example, a robotic arm must have a robot flange interface. These rules are defined in the Application Rules and illustrated in Listing 1.5. Defining which products must exist in a robot configuration can also be used to solve challenges, such as CH2 described in Section 3 above.

```
--Constraint 9: One product of each type robotic_arm, eecd, end_effector, and
     data_connection, must exist in the configuration:
context Configuration inv:
    let product_types_attr = self.products->forAll(
                                        p->select(attribute | attribute.name = 'type')) in
        product_types_attr->one(value = 'robotic_arm') and
        product_types_attr->one(value = 'eecd') and
        product_types_attr->one(value = 'end_effector') and
        product_types_attr->one(value = 'data_connection')

--Constraint 10: Products of type robotic_arm must have specific port types:
context Product inv:
    self.attributes->exists(attribute | attribute.name = 'type' and attribute.value = '
     robotic_arm') implies
    self.ports->exists(port | port.type = 'robotic_arm_flange')
```

Listing 1.5: Configuration rules on required product and port types in a configuration described using OCL.

Other than rules on the specific products and their ports in a configuration, rules related to user requirements are also implemented in the Application Rules. As illustrated in Figure 6, the user can specify different requirements, such as the required payload of the robotic arm, and the type of application. Different robotic arms have different payload requirements, e.g. some can carry up to 3kg, while others up to 10kg. Here, the user should be able to specify their application requirements, and the configurator should only give the relevant configurations. Examples of rules related to the user requirements are shown in Listing 1.6.

```
--Constraint 11: User requirement on payload of robotic_arm:
context Product inv:
    req_payload <> none and
    self.attributes->exists(attribute | attribute.name = 'type' and attribute.value = '
     robotic_arm') implies
    self.attributes->one(attribute | attribute.name = 'payload') and
    req_payload <= self.attributes->select(attribute | attribute.name = 'payload').value

--Constraint 12: User requirement on application
context Configuration inv:
    req_application <> none and self.applications->exists(req_application) implies
    let end_effector_type = self.applications->select(req_application).end_effector_type
     in
    let end_effector_product = self.products->select(p | p.attributes->exists(attribute |
     attribute.name = 'type' and attribute.value = 'end_effector')) in
    end_effector_product.attributes->exists(attribute | attribute.name = 'subtype' and
     attribute.value = end_effector_type))
```

Listing 1.6: Rules related to user requirements described using OCL.

The concept of Product series allows to easily group products with similar properties. In the robotics domain this can be applied to incorporate product type series, such as robotic arm series, but even extend this to gripper types, by creating product series of electrical grippers, vacuum grippers etc.

Apart from these general domain rules that are defined in the Application Rules, RoboCIM is designed to incorporate knowledge about specific products, even without their existence in the product catalogue. This allows users to build up the knowledge base on robotic products, and later specify which products a company owns, by specifying its product catalogue.

5 Prototype Implementation and Evaluation

In this section we describe the prototype implementation and evaluation of RoboCIM using ASP. We have chosen to use ASP, since it can handle non-monotonic and default reasoning for cases where information is incomplete. We use *clingo* [5] to ground and solve the implementation.

RoboCIM was implemented in ASP, using the UML diagram in Figure 7, and the constraints described above in Section 4. A *product* is defined as an ASP fact, it's belonging to a product series is defined as a rule and a fact on the specific product and series. Listing 1.7 exemplifies how these concepts have been implemented in ASP, and the remaining concepts in RoboCIM were implemented likewise. The complete code can be found in the public Github repository[6].

```
product(koala_a).                % robotic arm
product(catomatic_gripperA).     % end effector (gripper)
series_has_product(koala, koala_a). % koala_a belongs to the product series koala
% A product inherits the same attributes as the product series it belongs to
has_primitive_attr(X,I,V,J,S) :- has_primitive_attr(Y,I,V,J,S), series_has_product(Y,X).
```

Listing 1.7: Product and product series in ASP.

The ASP implementation was validated on a subset of 20 products from 3 manufacturers, from Technicon's product catalogue. The configurator was validated with regards to both product compatibility and user requirements, where

[6] https://github.com/Daniella1/robocim_configurator

the user could specify an application and payload. The time taken to generate all configurations is presented in Table 1. The very short runtimes (within 0.1 seconds) demonstrate the practicality of our approach for defeasible reasoning via ASP to infer plausible configurations in the case of incomplete knowledge.

Table 1: Results of generating valid configurations using RoboCIM and ASP.

Products in Catalogue	**Products in Configuration**	**Configurations**	**Time [s]**
20	4	122	0.255
20	5	11	0.094

6 Conclusion and Future Work

We presented our configurator model, RoboCIM, which was designed to deal with incomplete information and a number of challenges found in the robot system integration domain. RoboCIM uses the concept of products, product series, ports, information sources and justifications which are all used to generate valid configurations composing compatible products.

An initial prototype implementation executed on a real data set of 20 products from 3 manufacturers demonstrates the practicality of our approach, taken within 0.3 seconds to generate all configurations consisting of either 4 or 5 components. This proof of concept operates on about a third of the devices usually offered by system integrators, in future work we will verify our approach in full inventory sets. We plan on working with human expert system integrators to evaluate the usability and impact of RoboCIM. Specifically, we plan to integrate this work with Technicon and perform a small user experiment, where one of the engineers will extend the product catalogue and constraints of the currently developed domain model prototype.

To ease the work of adding information to the knowledge base, an approach of extracting rules from natural language, as the one presented in [6] will be investigated in the future. Rules to extract products with missing information can be added to the framework, and used to identify which products require attention before they can be used in configurations.

Acknowledgement

We acknowledge the Innovation Foundation Denmark for the MADE FAST project. The views and opinions expressed in this paper are those of the authors and do not necessarily reflect the official policy or position of Technicon ApS.

References

1. Industrial robot communication protocols, https://s3.amazonaws.com/RobotiqContent/Documents/Industrial-robot-communication-protocols.pdf, (Accessed: 05-06-2021)
2. Manipulating industrial robots – vocabulary. Standard, International Organization for Standardization, Geneva, CH (2013)
3. Manipulating industrial robots – mechanical interfaces – part 1: Plates. Standard, International Organization for Standardization, Geneva, CH (2004)
4. Felfernig, A., Friedrich, G., Jannach, D.: Uml as domain specific language for the construction of knowledge-based configuration systems. International Journal of Software Engineering and Knowledge Engineering (2001). https://doi.org/10.1142/S0218194000000249
5. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Clingo = asp + control: Preliminary report (2014)
6. Hassanpour, S., O'Connor, M., Das, A.: A framework for the automatic extraction of rules from online text. pp. 266–280 (2011). https://doi.org/10.1007/978-3-642-22546-8_21
7. Koons, R.: Defeasible Reasoning. In: Zalta, E.N. (ed.) The Stanford Encyclopedia of Philosophy. Metaphysics Research Lab, Stanford University (2017)
8. Nute, D.: Apparent obligation. In: Defeasible deontic logic, pp. 287–315. Springer (1997)
9. Sabin, D., Weigel, R.: Product configuration frameworks-a survey. IEEE Intelligent Systems and their Applications **13**(4), 42–49 (1998). https://doi.org/10.1109/5254.708432
10. Sanneman, L., Fourie, C., Shah, J.A.: The state of industrial robotics: Emerging technologies, challenges, and key research directions (2020)
11. Schou, C., Hansson, M., Madsen, O.: Assisted hardware selection for industrial collaborative robots. Procedia Manufacturing **11**, 174–184 (2017). https://doi.org/10.1016/j.promfg.2017.07.222, proceedings of the 27th International Conference on Flexible Automation and Intelligent Manufacturing
12. Schäffer, E., Bartelt, M., Pownuk, T., Schulz, J.P., Kuhlenkötter, B., Franke, J.: Configurators as the basis for the transfer of knowledge and standardized communication in the context of robotics **72**, 310–315 (2018). https://doi.org/10.1016/j.procir.2018.03.190, proceedings of the 51st Conference on Manufacturing Systems
13. Shi, L., Roman, D.: Using rules for assessing and improving data quality: A case study for the norwegian state of estate report (2017)
14. Stampfer, D.: Contributions to system composition using a system design process driven by service definitions for service robotics (2018)