An Ontology DSL for the Co-Design of Mechatronic Systems

Milan Cornelis
Milan.Cornelis@uantwerpen.be
Cosys-lab (FTI)
University of Antwerp
Belgium
AnSyMo/Cosys
Flanders Make
Antwerp, Belgium

Yon Vanommeslaeghe
Yon.Vanommeslaeghe@uantwerpen.be
Cosys-lab (FTI)
University of Antwerp
Belgium
AnSyMo/Cosys
Flanders Make
Antwerp, Belgium

Bert Van Acker
Bert.VanAcker@uantwerpen.be
Cosys-lab (FTI)
University of Antwerp
Belgium
AnSyMo/Cosys
Flanders Make
Antwerp, Belgium

Paul De Meulenaere
Paul.DeMeulenaere@uantwerpen.be
Cosys-lab (FTI)
University of Antwerp
Belgium
AnSyMo/Cosys
Flanders Make
Antwerp, Belgium

ABSTRACT

The complexity of mechatronic systems is vastly increasing. Therefore, the design of these systems requires different engineering domains, e.g., the mechanical, electrical, and control domains, to work together. The different domains often work in parallel to gain efficiency in this so-called co-design process. However, the design choices made by engineers in one domain can influence parameters in another domain. Too little or even no knowledge about these cross-domain influences may later lead to system integration problems or to degraded system performance. Solving these problems requires taking steps back in the development process, causing a higher design cost. In order to improve this cross-domain collaboration, we propose using ontologies to assist the co-design process by explicitly capturing the design dependencies, both within and across the engineering domains. However, designing ontologies can be complex and is labor-intensive, especially if one relies on generic ontology languages like the Web Ontology Language 2 (OWL 2). Therefore, we created a Domain Specific Language (DSL) focusing on the essential complexity, which enables engineers to design a cross-domain system ontology in a consistent and straightforward way. We elaborate on the metamodel for this DSL, discuss the realization of a prototype tool, and demonstrate how one can then reason on this ontology to derive new information about the various cross-domain design relationships.

MODELS '22 Companion, October 23–28, 2022, Montreal, QC, Canada
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9467-3/22/10…$15.00
https://doi.org/10.1145/3550356.3561534

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; • **Computer systems organization → Embedded and cyber-physical systems**; • **Information systems → Web Ontology Language (OWL)**; • **Computing methodologies → Ontology engineering**.

KEYWORDS

Mechatronics, Domain-Specific Language, Metamodel, Co-Design, Ontology

ACM Reference Format:
Milan Cornelis, Yon Vanommeslaeghe, Bert Van Acker, and Paul De Meulenaere. 2022. An Ontology DSL for the Co-Design of Mechatronic Systems. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22 Companion), October 23–28, 2022, Montreal, QC, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3550356.3561534

1 INTRODUCTION

The design of mechatronic systems like machines or vehicles entails the collaboration between different engineering domains, such as the mechanical, control, and embedded domain. As these mechatronic systems are becoming more and more complex due to the increasing demand for more functionalities, these engineering domains become increasingly more intertwined, with their interactions becoming more and more complex. While decomposing the design problem into different disciplines in a true divide-and-conquer approach helps master the complexity of the design, it might neglect important cross-domain system properties that are hard to decompose, such as system performance, system safety, system reliability, etc. Therefore, the classical design process risks losing efficiency or even consistency, and the resulting mechatronic product risks becoming less effective.

Therefore, besides splitting the design problem into sub-problems, one needs to adhere to a co-design strategy as well. Choices made by the engineers in the different domains strongly influence each other. For example, when a mechanical engineer modifies the design, it affects the transfer function of the system. As a result, the control engineer must modify the control algorithm, which in turn impacts the embedded engineer, who must ensure that this control algorithm can still run on the embedded platform. In practice, however, engineers are not sufficiently knowledgeable about the influence of their design choices on the other engineering domains. They simply "throw their design over the wall". E.g., not communicating the mechanical adaptation, using different units or value ranges, assuming infinite resources of the embedded platform for the execution of a control algorithm, may result in errors or suboptimal systems when integrating the total system. The only solution is to iterate the design process, which costs time and money.

In this paper, we explore how ontologies assist in the co-design of mechatronic systems and to what extend it can be automated. By making implicit knowledge explicit by modeling it in an ontology, engineers from different domains can find out what influences their choices have on other domains, derive engineering contracts, find for which competing design parameters trade-offs must be made, guard consistency during the design, etc. This will both endorse better communication between engineering domains and indicate which design choices can safely be made without interference with other disciplines.

A significant advantage of relying on ontologies for modeling the cross-domain system design dependencies is that we can use already existing ontology reasoners, e.g., HermiT [6]. In the current work, we use the second version of the Web Ontology Language (OWL 2) [25] to model ontologies. OWL 2 is the extension of the OWL language and is developed by the World Wide Web Consortium (W3C) to describe data for the Semantic Web. All kinds of tools to develop OWL 2 ontologies have already been designed, of which Protégé [14] is one of the most popular. However, the genericity of such tools also brings a lot of accidental complexity for the mechatronic design engineers. To put the focus on modelling the essential complexity, i.e., making the dependencies between design parameters from different engineering domains explicit, we choose to develop a Domain-Specific Language (DSL), lowering the complexity to model and to reason on the co-design problem.

The rest of this paper is structured as follows. First, Section 2 presents some related work. The running example during this paper is discussed in Section 3. Then, Section 4 explains the DSL we created, the model-to-model transformation to OWL 2, and the capabilities of the reasoning tool. Finally, Section 6 concludes our work and discusses future work.

2 RELATED WORK

As previously mentioned, the design of mechatronic systems typically involves different engineering domains, leading to a division into domain-specific problems. This division, however, leads to situations where properties in different domains influence each other, giving rise to dependencies. Qamar *et al.* [17] describe the nature of these dependencies and how to model them, proposing language constructs that take into account the nature of properties and their dependencies. They make the fundamental distinction between synthesis properties (SP), which are used to define design alternatives, and analysis properties (AP), which constitute predictions. In other words, a distinction between properties that are chosen (SP), e.g., component dimensions, and properties that are predicted (AP), e.g., component mass. They determine that while dependency models add value, a considerable effort may be required to build and manage them. As such, this effort needs to be compared to the expected benefits. Additionally, they evaluate the systems modeling language (SysML) [8] for the creation of dependency models, concluding that it lacks the necessary complexity for this purpose. They later propose a domain-specific language (DSL) to address this problem [16]. Their proposed dependency modelling language (DML) allows engineers to model the dependencies between properties referenced in different views on a system. However, the proposed language did not yet support modeling these dependencies at different levels of abstraction. Additionally, they did not define the operational semantics of the language formally. As such, analyzing how dependencies propagate, for example, is not possible.

Törngren *et al.* [19] present a viewpoint integration approach where these dependency models are used together with viewpoint contracts and tool integration models, collectively referred to as "support models". The viewpoint contracts allow them to define the vocabulary, assumptions, and constraints required for the communication between different stakeholders, while the tool integration models capture the interrelations between different tools, making it possible to express tool interrelations, such as data exchange, traceability, etc. This unified approach allows them to deal with viewpoint interrelations at three distinct levels: people, models, and tools.

Several studies have also explored specifically how ontologies can support the (co-)design of mechatronic or Cyber-Physical Systems (CPS). Vanherpen *et al.* [22] introduce the Contract-Based Co-Design (CBCD) method as an extension to the already existing Contract-Based Design (CBD) method introduced by Benveniste *et al.* [1]. Here, ontologies are used to make implicit knowledge explicit, enabling to relate different engineering domains to each other and also their design variables. They show how, with these relationships and a predefined mapping contract, it can be determined how design variables relate, making it possible to derive Assume/Guarantee (A/G) contracts between different engineering domains.

In the product-assembly domain, ontologies have also been used to overcome "interfacing problems", e.g., error-prone manual data handling, error-prone manual algorithm configuration, and limited traceability. Van Acker *et al.* [20] attempt to address these problems by creating a framework making use of a Knowledge Base (KB) to assist in front-loading assembly knowledge to early stages of the product design process. This KB is used to capture cross-domain knowledge in the product-assembly design. A gearbox example is used as a use case to validate the proposed method. Similar to Qamar *et al.*, they conclude that, while there are benefits, there is still a trade-off between the time spent to build the KB and the advantages gained by using it.

Lastly, Sales *et al.* [18] present the systems' architecture ontology (SAO), which allows users to define not only the architecture of the system itself, i.e., hardware, software, and system components, but also the relationships between these different parts. This allows them to identify inconsistencies in architectural design and component compatibility. Additionally, they present a tool which enables the generation of architecture analysis and design language (AADL) [5] models from a system ontology.

3 RUNNING EXAMPLE

Throughout this paper, we use the example of an Adaptive Cruise Control (ACC) that employs a Model Predictive Controller (MPC). MPC is a control algorithm that uses a (reduced) model of the system under control in combination with an optimization algorithm to predict the optimal control value at a certain time interval. The optimization algorithm calculates a sequence of inputs over a control horizon C after which the sequence is held constant for P−C−1 samples, where P is the prediction horizon. This sequence of input values is then fed into the model, which predicts the corresponding output from the plant. The input values during the control horizon are calculated such that the error between the predicted system output and reference is minimal, i.e., minimizing a cost function [15]. This principle is shown in Figure 1.

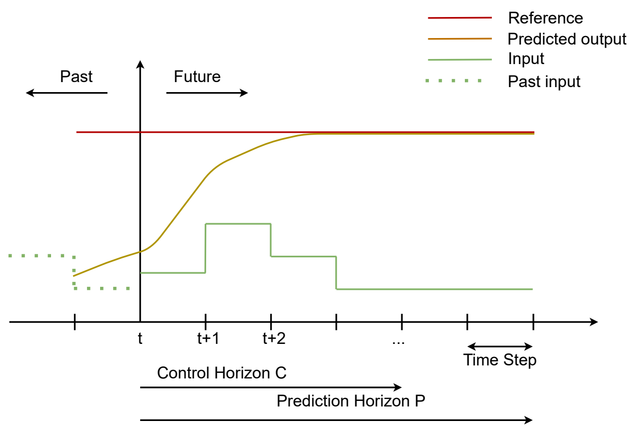

Figure 1: The basic principle of Model Predictive Control. A sequence of control inputs is determined by an optimization algorithm over a control horizon C. This sequence is then held constant for the remaining prediction horizon P. Only the first input of the sequence is sent to the plant.

To obtain an ACC that conforms to its objectives, it must meet a number of predefined requirements. To keep the complexity manageable in the context of this paper, we limit ourselves to the following requirements:

(1) The tracking performance of the ACC must have an Integral Square Error (ISE) of at maximum 0.5
(2) A comfortable system must be guaranteed by keeping the jerk standard deviation below 0.75 m/s^3.
(3) The system must react in 20 ms to a change in distance or velocity.
(4) After 20 s, the reference distance or velocity must be reached.
(5) The total cost of the system is at most €150.

Note that these values are chosen for demonstration purposes and may not be representative of real ACC systems. Also, the maximum value of the jerk_std and ISE only counts for a single test scenario in this case.

4 EXTENDING THE ONTOLOGY CONCEPT

Modeling an ontology directly in OWL 2 would be far too cumbersome. First, the designer would need to have an extensive knowledge of the language and its tools. This makes it difficult for beginners to get started, as they first need to learn the OWL 2 language. Second, everyone creates an ontology in their own way. If two people build an ontology for the same problem, it may well be that these ontologies look completely different. Finally, the manual construction of ontologies in OWL 2 requires a lot of work. Each new type of property has to be defined manually, characteristics have to be added, a domain and range have to be assigned, etc.

In order to overcome these problems, we decided to create a DSL of which the abstract syntax is given via the metamodel (MM) shown in Figure 2. The DSL ensures that the modeling capabilities are limited to the essential complexity. This hides the redundant expressiveness of OWL 2 making it possible to build an ontology model with no knowledge about OWL 2 itself. Also, this allows relationships at different levels of precision, e.g., $\mathbb{L}$1-Fuzzy, $\mathbb{L}$2 and $\mathbb{L}$3, to be modeled directly without requiring workarounds. Using the DSL also ensures that we can build all ontologies with the same structure. This allows reasoning queries to be used consistently, as their names are derived from the MM. All necessary relations are already available along with their semantics.

The root element of the Domain-Specific Language is the 'DesignProblem', which represents the problem that engineers need to solve. This also refers to the system under design, e.g., an ACC or a drone. As said before, developing mechatronic products requires the cooperation of different engineering domains. Therefore, the class 'EngineeringDomain' is included. With this class, we can define all different engineering domains incorporated in the project. Note that the system-level itself can also be an engineering domain, as there might be some over-arching parameters related to the overall system under design. On the left and right side of the figure, we see the 'PropertyOntology' and the 'DependencyModel'. These are the two main building blocks, as they contain all information about the system under design. They are further discussed in Section 4.1 and Section 4.2 respectively. Figure 3 shows a graphical representation of the ontology for the running example. Please note that such graphical representation is for illustration purposes only, and does not contain all details stored in the ontology. Also, it is not currently a feature of the presented prototype tool.

4.1 Property Ontology

Properties can be seen as ontological concepts of a (mechatronic) product. They tell something about the product as a black box, i.e. without the need to know its implementation details. Properties can therefore be regarded as requirements. For example in Figure 3, the property *Safe?* expresses whether or not the full system complies to a set of functional safety requirements. This may entail subtending properties, such as *Schedulable?*, which expresses whether or not the total set of software tasks can be scheduled on the embedded system.

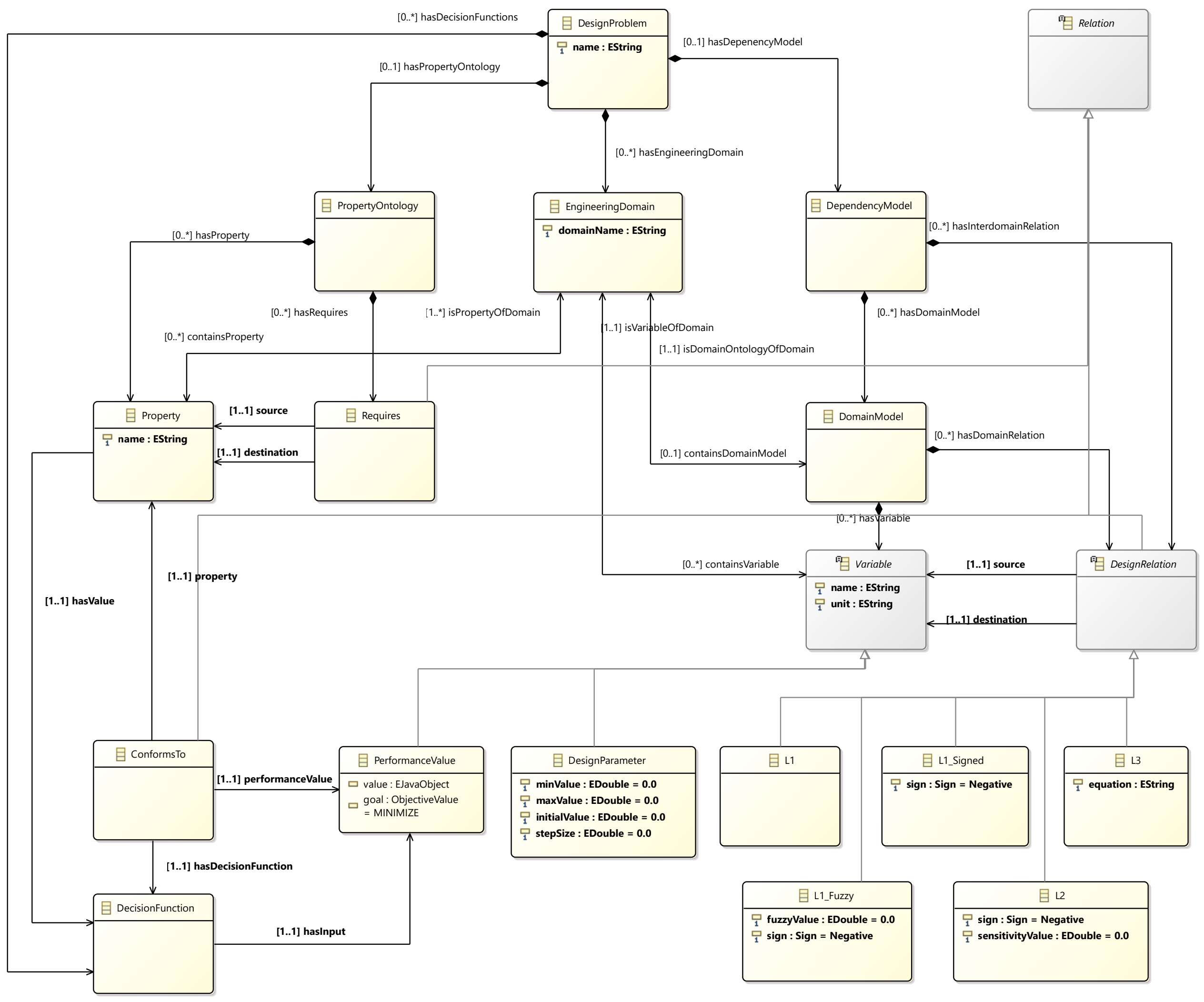

Figure 2: The metamodel of the Domain-Specific Language. It contains two main parts: the Property Ontology and the Dependency Model. The Property Ontology consists of high level properties and the 'requires' relations between these, whereas the Dependency Model consists of the low level design variables and the relations (i.e. dependencies) between these.

This may on its turn depend on the requirement whether or not the processor bears an acceptable load, i.e. *AcceptableProcessorLoad?*.

As we can see, one property can depend on several other properties. The above example is only within one domain. However, it is important to note that these dependencies can also go across different domains. The ACC has to react within 20 ms response time (control domain). In order to achieve this *ResponseTime?*, the system requires that it can execute *FastEnough?* which again requires a *PerformantProcessor?* (embedded domain). Because these properties and their dependencies are often implicit knowledge of an engineer, other engineers from possibly other domains are unaware of them. As a result, unrealistic assumptions are often made, causing problems during implementation [12, p. 31].

We see that the system can be divided into properties, belonging to one or more domains, and a requires-relation between these properties. The principle of connecting properties with requires-relations and the semantics behind it is based on the Contract-Based Co-Design (CBCD) method created by Vanherpen [21]. The resulting model is the so-called 'PropertyOntology'.

As mentioned, properties can be derived from the system requirements, or they are new properties that are required by other properties in order to be true. Indeed, if all properties are true, i.e., the root property (*ControlledSystem?*) is true, then all system requirements should also be met. Generic requirement engineering techniques might help to identify the properties that must be incorporated in the ontology. How the properties obtain their Boolean value is further elaborated in Section 4.3.

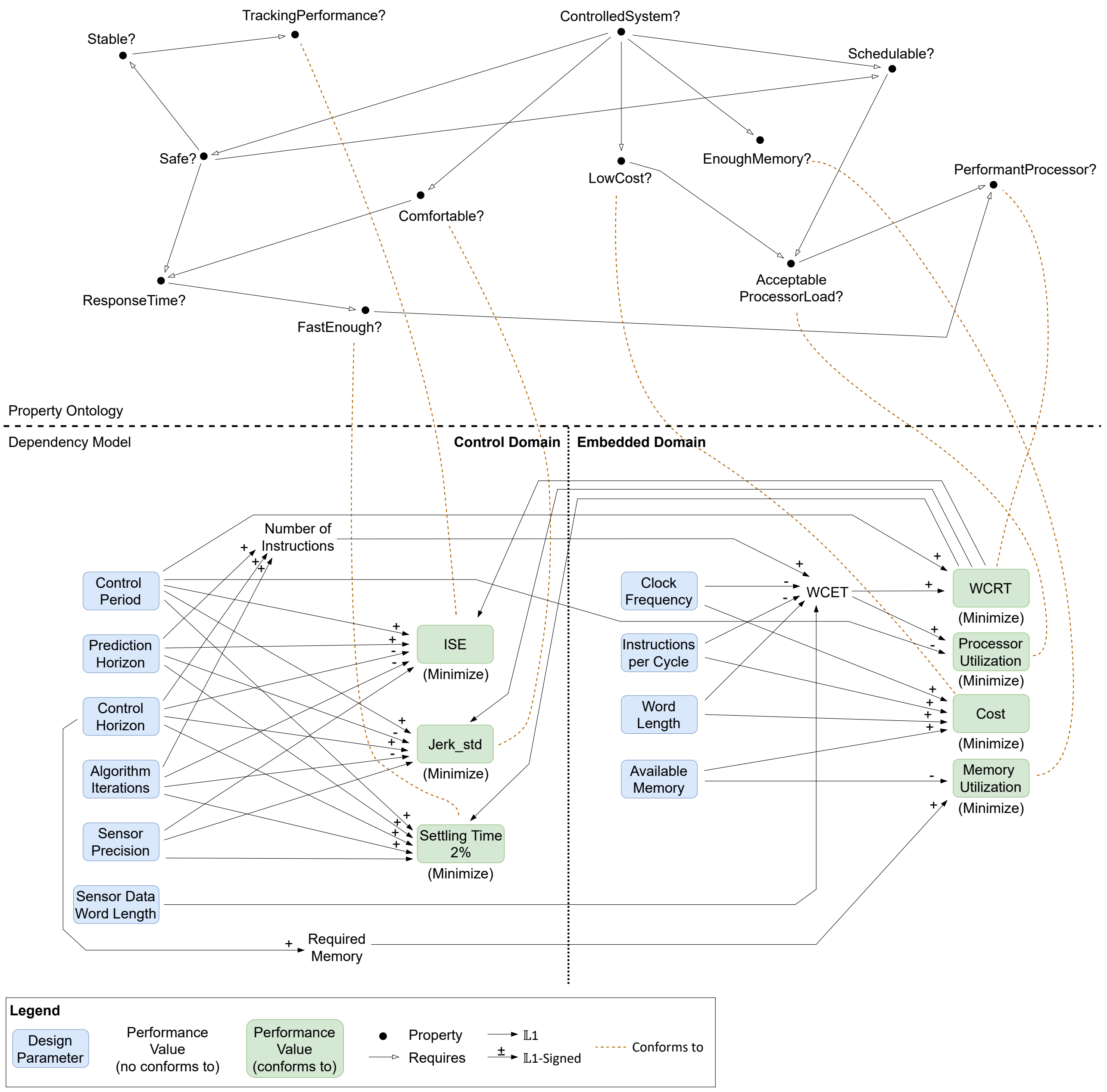

Figure 3: The total ontology of the MPC for ACC running example. On the top, the property ontology is shown, while on the bottom, the dependency model is shown. Only two engineering domains are considered, which are the control domain and the embedded domain.

When only provided with the property ontology, engineers can already infer which domains are going to communicate with each other and possibly also about which system properties. This helps with the further design of certain parts of the system. The inverse relationship of 'requires' is 'influences', meaning that if a property A requires a property B, then property B influences property A. Via this inverse relationship, engineers can obtain hints about their own influence on other domains. Knowing in advance which properties depend on each other can also allow for better synchronisation between different engineering teams during the design phase.

4.2 Dependency Model

The dependency model is based on the work of Vanommeslaeghe *et al.* [23]. It corresponds to the dependency model as proposed in [17]. The right part of the MM, Figure 2, shows the dependency model, and the bottom of Figure 3 shows the example instance for the ACC.

As can be seen, the dependency model consists of different domain models, e.g., the control domain model. Each domain model in itself consists of design parameters (DPs), performance values (PVs) and their relations. As this dependency model describes system concepts, like DPs and PVs, and the relations between them, it can also be regarded as an ontology.

First, DPs, shown in blue in Figure 3, are the different "tuning knobs" for the design of the system. The prediction- and control horizon are variables tuned by the engineer to alter the behavior of the MPC, and therefore, they are considered as DPs. Regarding the embedded domain, engineers can change the clock frequency, word length, etc., by choosing a different processing platform. DPs all have a certain range of values with an associated step size. This can be used later in, e.g., consistency checks and DSE.

Second, PVs, shown in white and green in Figure 3, are variables which result from simulation or measurements with as input one or more DPs and/or other PVs. The resulting values of the PVs can be evaluated with a decision function (DF), to check if the PV is within a specified range. The output value is a Boolean, representing whether the DF is satisfied. As some DFs are derived from the system requirements, we can compare whether a performance value meets its corresponding system requirement or not. E.g., the requirements in Section 3 stated that the total cost of the system must be less than €150. As such, the PV 'Cost' from Figure 3 can be compared with a DF 'Cost ≤ €150'. Section 4.3 further elaborates on this.

Last, we have relationships between the different variables. These relationships can be of different levels of precision [4, 23]:

- $\mathbb{L}1$: only the fact of influence is known.
- $\mathbb{L}1$-Signed: $\mathbb{L}1$ relations are extended with a direction of influence.
- $\mathbb{L}1$-Fuzzy: $\mathbb{L}1$ relations are extended with a fuzzy value in the range [-1,1]. This value is rather intuitive and stems from experience or discussion between engineers.
- $\mathbb{L}2$: sensitivity information is known. This value is obtained via sensitivity analysis.
- $\mathbb{L}3$: an exact mathematical relationship is known.

The current ACC dependency model only contains $\mathbb{L}1$ and $\mathbb{L}1$-Signed relations corresponding to the available reasoners in our current implementation.

For example, looking at the control domain model, we can see that the control period is connected via a positive $\mathbb{L}1$-Signed relation to the ISE indicating that an increase in the control period does increase the ISE. The opposite can be observed with the connection between the prediction horizon and the jerk standard deviation. An increase of the prediction horizon results in a decrease of the jerk standard deviation. In other cases, we know that certain variables influence each other, but not how. This is the case with the sensor data word length and the worst-case execution time (WCET). Therefore, these variables are connected with a $\mathbb{L}1$ relation.

The precision level of the relations is not required to be fixed. As the development process continues, engineers also obtain more knowledge and details about the system, allowing them to further refine the ontology. Some relationships can be given a higher level of precision, while others can even be omitted completely because they are negligible.

Similar to the property ontology, we can already infer a number of things even if we only have the dependency model. Firstly, we are able to check which engineering domains influence each other and also in which way. This allows engineers to better cooperate. Secondly, when we change the value of a DP, the dependency model allows checking which other variables are influenced by this change. The ontology can find new relationships through the transitivity of the known relationships. Hereby, one gains more insight in what other parameters are affected, which may be from other domains. Thirdly, if a precision level larger than $\mathbb{L}1$ is used, the impact on the system due to a change in a parameter can be assessed. $\mathbb{L}1$-Signed relations indicate if other parameters will increase or decrease, $\mathbb{L}1$-Fuzzy and $\mathbb{L}2$ relations can tell us something about the order of magnitude in which the other parameters will increase or decrease, and $\mathbb{L}3$ relations provide us with exact numbers. Fourthly, this type of ontology can also give hints to Design Space Exploration [23] and the automatic generation of workflows [24]. Fifthly, when sufficient $\mathbb{L}1$-Signed information is provided, trade-offs between conflicting PVs can be derived automatically. Finally, the dependency model also allows us to derive A/G contracts between engineering domains. This defines to which extent engineers from different domains can work concurrently even though they depend on each other. If the parameters are adjusted in such a way that certain defined limits are exceeded, the ontology must be used to find out who is affected and then also to inform the affected parties correctly. Section 5.2 further elaborates on this.

4.3 Combining the Property Ontology and Dependency Model

After modeling the discussed ontologies, we can connect them via a 'conforms to'-relationship between a property and (the evaluation function of) a PV. This is in particular the case if the property and the PV are related to the same system requirement. The PV is the measurable quantity behind that property. Via the Decision Function DF, one can confirm whether the property is satisfied or not. It is also possible that a certain property is not directly derived from the requirements, e.g., 'PerformantProcessor?'. Such properties are then necessary conditions for parent properties to become true. These can also be connected to a particular PV and its DF.

Some PVs may not have a 'conforms to'-relation to a property. They are nevertheless useful for assessing certain performance values at the design level, or may serve as intermediate parameter between two other parameters. These are shown in white in the example of Figure 3. Such PVs can be seen as an intermediate performance variable. Nevertheless, this does not mean that the PV can be omitted. In many cases, they provide a clearer connection between certain variables in the dependency model, e.g., 'Number of Instructions' which provides a clear connection between DPs in the control domain and PVs in the embedded domain. They are also useful when generating contracts which is discussed in Section 5.2.

It is important that all leaf nodes from the property ontology are connected to a PV. Otherwise, this property nor its parents can ever be assigned a Boolean value, and it will remain unknown whether the corresponding system requirements are met. The ontology reasoner can support this check for completeness. If such dangling leaves are found, the engineer should either evaluate whether that leaf property might be dropped, or add a missing PV and connect to it, or connect child properties to it such that it is no longer a leaf. Properties that are not leaves may have a 'conforms to'-relationship to a PV, but not necessarily. An example of such a property is 'AcceptableProcessorLoad?'. In order for this property to be true, both its required properties, i.e., 'PerformantProcessor?', and the decision function to which it relates need to be true.

The property ontology and the dependency model complement each other. If no direct 'conforms to'-relationship can be made, the ontology reasoner can still support engineers to think further about the system to discover new properties or PVs. We clarify this with an example. Suppose that in the embedded domain of the dependency model, we forgot to add the PV 'Memory Utilization'. This introduces several problems. First, we now have an empty leaf: 'EnoughMemory?'. We assume that the embedded engineer thought about this property while reasoning about the system. Second, in the control domain, 'Required Memory' no longer has a relation to a PV. We can solve these problems by reasoning about both ontologies. If the property 'EnoughMemory?' is considered to be important, we cannot simply omit it from the property ontology. Therefore, we need to check either whether there is possibly a (new) PV that we can connect to it or whether we can add a new child property to the current property which relates to an existing PV. In the current example, we choose to add a new PV. The property 'EnoughMemory?' is clearly related to the embedded platform memory which should be large enough to run the control algorithm. This requires a relationship between 'Required Memory' and 'Available Memory'. The latter is however a DP and not a PV. Adding the 'Memory Utilization' PV solves both problems. The property 'EnoughMemory?' gets a conforms to relation to the PV, so it is no longer a blank leaf and 'Required Memory' now also gets a relation to a PV.

The combination of the two graphs can also be used to validate whether all connections are present. Relationships that exist at the dependency model level should also exist at the property ontology level, otherwise, one of the ontologies is incomplete. We therefore rather need influence relationships at property ontology level instead of 'Requires'-relationships. These influence relationship can simply be achieved by reversing the arrows of the 'Requires'-relationships. In order to know if the relations exist at both ontology levels, we again use the 'conforms to'-relationships. However, this check for consistency between both levels still has to be validated in future work.

In summary, the proposed system ontology that consists of a property ontology and dependency models allows for system-level analyses such as completeness, consistency and decomposition. At the same time, isolated analysis on the property ontology alone or on the dependency model alone remains possible. To achieve these analyses, we can rely on elementary operations from the world of ontologies or other graph-related models. Table 1 shows a summary of the above discussed analyses types and the required reasoning operations.

5 PROTOTYPING TOOL

After we created an ontology model, it should be possible to extract additional information from it by using an ontology reasoner. In order to automate and facilitate this retrieval of information, we developed a reasoning tool.

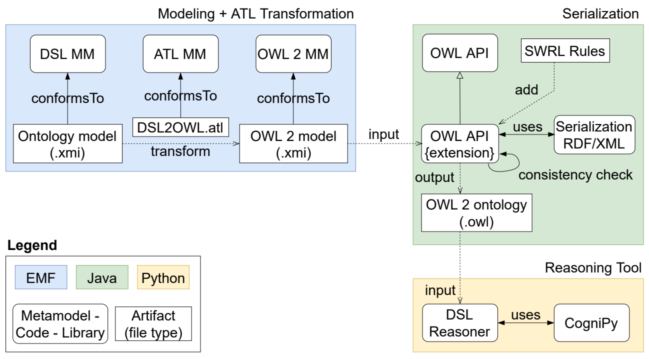

Figure 4: The architecture of the prototyping tool. Creating the ontology from the ontology metamodel and transforming it to an instance of the OWL 2 metamodel is done in the Eclipse Modeling Framework (EMF). This instance is then serialized to the RDF/XML format, which can be used by the DSL reasoning tool. Based on [7, 10].

5.1 DSL to OWL 2 Transformation

As discussed in Section 4, we created a DSL to facilitate the modeling of ontologies. Of course, this does not come without a drawback. The ontology created with the DSL does not have the proper semantics to be used by OWL 2 tools. They require the ontology format to be in an OWL 2 format, e.g., RDF/XML or OWL/XML. Therefore, to allow the use of existing reasoners, the ontology first needs to be transformed to an OWL 2 ontology. Transforming from the DSL to OWL 2 also means that we go from a Closed World Assumption (CWA) to an Open World Assumption (OWA). When modeling our ontology in the DSL, we assume the data to be complete, i.e. CWA. However, we transform to OWL 2 which means going to an OWA. Here, the provided data is not assumed to be complete. Therefore, we can derive new information from the ontology by using the reasoners. The two upper rectangles in Figure 4 show the architecture for the DSL to OWL 2 model transformation. The different steps in this process are further described below.

We start with an instance of the DSL MM, i.e., the ontology model, which needs to be transformed to an instance of the OWL 2 MM. For this model-to-model (M2M) transformation, we base ourselves on the much more common UML to OWL 2 transformation. More specifically, on the work by Haasjes [7], who describes in detail how a UML model can be transformed to an OWL 2 model and vice versa. However, to perform this transformation, we need a MM of the OWL 2 language. As the one listed on the W3C Wiki [2] does not conform to the latest specifications of the language, we use a more recent version obtained from [7], which incorporates the latest (at the time of writing) structural specification of December 11, 2012 [13]. To define the transformation rules between the two MMs, we use the ATLAS Transformation Language (ATL) [10].

Except for the classes representing relationships, the general structure of the MM is adopted when we transform to OWL 2.

Table 1: Required information in the system ontology and the resulting analysis type

Required ontology relations	Elementary Operations	Analysis Results
$\mathbb{L}1$	Transitivity	(In)direct (cross-domain) dependencies
$\mathbb{L}1$-Signed	Fuzzy Cognitive Map sign rules	Trade-offs between PVs
Between Properties	Boolean logic	Requirements compliance
'conforms to'-relations	Graph theory	Consistency and completeness
Domain relations	Contract theory	System decomposition

Thus, the MM classes are transformed to classes in OWL 2. Relationship classes as well as the connections between the different MM classes, on the other hand, are transformed to object properties. Furthermore, all instances of the MM classes, i.e. the objects of the instance model, are transformed to instances of the corresponding OWL 2 classes. The appropriate instances are also connected with each other via the proper object properties. Their attributes are transformed to data properties.

After the M2M transformation, an instance of the OWL 2 MM is obtained. However, this is still an instance of an Ecore model. To make this model compatible with existing reasoning tools, it needs to serialized to an accepted OWL 2 syntax, such as RDF/XML, OWL/XML, Turtle, etc. We opt to use RDF/XML as, as it is mandatory for OWL 2 tools to support this syntax [25], ensuring compatibility with a variety of tools.

The serialization is implemented in Java, making use of an extended version of the OWL API [9] created in [7]. We also add Semantic Web Rule Language (SWRL) rules when mapping the ontology to the OWL API. SWRL rules allow us to make the OWL ontology even more expressive. An example of this is the combination of $\mathbb{L}1$-Signed relations. A SWRL rule can define that a direct positive relation followed by a direct negative relation is combined as an indirect negative relation.

$$\mathcal{X} \xrightarrow{+} \mathcal{Y} \xrightarrow{-} \mathcal{Z} \Rightarrow \mathcal{X} \xrightarrow{-} \mathcal{Z}$$

Another use of SWRL rules is to identify trade-offs. By using SWRL rules, we can check how—positive or negative—DPs influence PVs. If a DP has conflicting influences on PVs (i.e., desirable/undesirable), then it is a trade-off parameter. In this case, several SWRL rules are used in combination to find out if a DP is a trade-off parameter or not.

Although it may seem that this transformation brings additional complexity at first glance, it does not. The model transformation and serialization is generic because the transformation rules are specified at the MM level of the DSL and OWL 2. Therefore, one does not have to touch these for each ontology.

5.2 Reasoning Tool

After modeling, transforming and serializing the ontology, a reasoner can be used to obtain information about the modeled knowledge. In order to make this reasoning more easy and user-friendly, we developed a reasoning tool, shown at the bottom of Figure 4. Currently, the tool can be used via a command line interface (CLI) and is implemented in Python. In order to load and reason about ontologies, the CogniPy library is used [3].

CogniPy, a Python implementation of the Fluent Editor tool created by Cognitum Services S.A., can be used to create and reason about ontologies described in a controlled natural language (CNL). This makes CogniPy intuitive and user-friendly. Creating queries is also done using this CNL, meaning that there is only a small learning curve for inexperienced users to create them themselves if necessary. Additionally, as it is OWL 2-based, it must, as already mentioned, support RDF/XML files. This allows us to load the serialized ontology without requiring additional transformations.

The current version of the reasoning already provides some basic functionality:

- A user interface providing some basic features, e.g., loading the RDF/XML file or creating a graph from the ontology (in OWL 2 format).
- Showing which engineering domains influence each other.
- Showing to which variables a chosen variable has a relation.
- Obtaining the trade-offs per variable.
- The generation of engineering contracts between two domains.

The following sections describe these options more in depth.

5.2.1 Showing Domain Influences. As the description implies, this shows the user which engineering domains influence each other. A SWRL rule is used to check for the condition where a variable belonging to domain A has a relationship to a variable belonging to domain B. If this condition is met, a new relation is automatically established between the two domains. As such, we only need to query for this newly added relationship.

Taking a look at the MPC for ACC case, we can see that there are some relations crossing the domain boundary at the dependency model level. This means that variables from one domain influence variables from the other domain, thus, the domains influence each other. As the relations go in both directions, we determine that the control domain influences the embedded domain and vice versa.

Note that while we currently only use the dependency model for this, it should be possible to also use the property ontology. As the properties also belong to one or more domain, and the inverse relation of requires is influences, enough information is available to derive domain influences. These to methods would provide the same results, provided the two ontologies are consistent.

5.2.2 Showing Variable Relations. This allows the user to query which variables are influenced by a variable of interest.

5.2.3 Deriving trade-Offs. When sufficient $\mathbb{L}1$-Signed relations are available, we are able to derive trade-offs between PV from the dependency model. This is based on the sign of the relations and the goal associated with the PVs. Indeed, based on the $\mathbb{L}1$-Signed relations, we can determine whether a change in value of a specific DP would affect certain PVs desirably or undesirably. E.g., if the goal of the ISE is to minimize it, we know that a rise in prediction horizon (P) is undesirable as it will increase the ISE. However, this increase in P also decreases the jerk_std, which is desirable. As such, there is a trade-off between ISE and jerk_std when changing P. Several SWRL rules use are used to determine these trade-offs. As such, we only need to query for them. In the ACC example, there are such trade-offs available. A few of them are shown below:

1. *Clock Frequency* has trade-off with:
   - Settling Time
   - WCRT
   - Processor Utilization
   - Cost
2. *Control Horizon* has trade-off with:
   - Jerk_std
   - WCRT
   - Integral Square Error
   - Processor Utilization
   - Settling Time
   - Memory Utilization

Currently, we only support minimize/maximize goals. However, in future work we intend to extend this to also work with constraints.

5.2.4 *Generating Assume/Guarantee Contracts.*

Engineering contracts can be used to enable engineers to work concurrently on different system components in a consistent way. It solves the problem that engineers make incorrect assumptions about parameter values from other domains and avoids consistency problems. Assumptions define under which conditions a component operates, i.e., the environment (preconditions). The Guarantees, however, specify what conditions the component must meet (postconditions) [1]. An example for the embedded domain would be that a software program is maximum 1000 instructions long (assumption). The embedded engineer must guarantee that this code is run in 10 ms.

The generation of contracts by making use of ontologies is based on the principles of the CBCD method [21]. First, we need to derive the guarantees of each contract. The guarantees of a contract are all the variables of the domain to which that contract belongs. This is based on the fact that these variables, along with their chosen range of values, ensure that the implementation meets the requirements of the system. During a negotiation phase between the different domains, these values were agreed upon. In the case of the running example, it was agreed that the ACC is implemented using MPC, running on a single embedded platform. Therefore, each domain must guarantee that they meet their own domain related conditions. As can be seen in Table 2 and Table 3, the guarantees correspond to their respective variables in Figure 3.

Vanherpen stated that "deciding upon the assumptions of a viewpoint-specific contract depends on how guarantees from one viewpoint influence the guarantees of another viewpoint" [21, p. 65]. Our derived methods is based on this. As the variables in the domain ontologies are guarantees in their respective contracts, we have to look for variables from one domain that have a direct relation to variables of another domain. These relations are the inter-domain relations, as they cross the boundary between domains. When there is such an inter-domain relation, the source variable becomes an assumption of the contract belonging to the domain of the destination. Taking a look back at the contract and ontology of the running example, we also see that this is true. The WCRT has a direct connection to all the PVs in the control domain. Hence, it is an assumption for the control domain, and is listed as such in the contract. For the embedded domain, we see that the required memory, number of instructions, control period and sensor data word length in the control domain all have a direct relationship to variables in the embedded domain. Therefore, they are assumptions of the embedded domain.

Table 2: Control Contract

Assumptions	WCRT ≤ 20 ms
Guarantees	Jerk_std ≤ 0.75 m/s^3
	Required Memory ≤ 5 MB
	Integral Squared Error ≤ 0.5
	Algorithm Iterations [30:1:50]
	Control Period [5:1:15] ms
	Number of Instructions ≤ 5000
	Settling Time ≤ 20 s
	Prediction Horizon [5:1:30] time steps
	Sensor Data Word Length [8:8:32] bit
	Control Horizon [1:1:5] time steps
	Sensor Precision = 10 cm

Table 3: Embedded Contract

Assumptions	Sensor Data Word Length [8:8:32] bit
	Control Period [5:1:15] ms
	Required Memory ≤ 5 MB
	Number of Instructions ≤ 5000
Guarantees	WCRT ≤ 20 ms
	Memory Utilization ≤ 75%
	Cost ≤ €150
	Processor Utilization ≤ 69%
	Available Memory [3:1:10] MB
	Clock Frequency [250:50:750] MHz
	Instructions Per Cycle [1:1:2] bits
	WCET ≤ 15 ms

6 CONCLUSIONS AND FUTURE WORK

In this paper, we discussed how ontologies can assist in the co-design process of mechatronic systems and how we can automate this. Because OWL 2 does not have the essential complexity we need, a DSL is created to solve this. The DSL consists of two main parts: the property ontology and the dependency model. With the property ontology, system properties and the way they are related to each other are expressed. The property ontology also provides a view on the system requirements. With this information, it can already be determined which engineering domains will have to communicate in later stages of the design process, and about which concepts they need to synchronize. The dependency model further refines this property ontology. It consists of design parameters and performance values, which are connected to each other via relations with different levels of precision. By relating properties with performance values via a 'conforms to'-relationship, we can check whether the ontologies are consistent and complete. The dependency model can again be used to determine which domains will need to communicate, but also to find out which variables influence each other, to derive trade-offs between performance values, and to generate engineering contracts.

Currently, we are only able to reason about $\mathbb{L}1$ and $\mathbb{L}1$-Signed relations in the dependency models. In future work, we will also implement relations with higher levels of precision in order to extract additional detailed information from the ontology. Because some of these relations have numerical weights, they are not directly implementable in OWL 2. Therefore, other formalisms than OWL 2 might be used, such as Fuzzy Cognitive Maps [11] to reason about this information.

As consistency and completeness checks are not automated yet, we will also implement this in future work. The consistency checks can be categorized into two main classes. The first class is a consistency check for the ontology design itself, checking if there are no opposing relations, e.g., a direct positive and indirect negative relation, verifying that no relationships have been overlooked (only if both ontologies are present), etc. The second class involves the values of the DPs and PVs. If $\mathbb{L}3$-relations are present, we can verify if the chosen value ranges are still a valid implementation for the system. E.g., if we have a software program with 1000 instructions (I) that must be completed in 1 ms (t), and the clock frequency (f_{clk}) has a range [500 750] kHz, we know that this system is not suitable due to the $\mathbb{L}3$-relation: $I = f_{clk} * t$. Regarding completeness, we can check whether the relations in one ontology also exist in the other ontology.

The DSL is still only a prototype and therefore evolving regularly. By applying the DSL to multiple and more comprehensive industrial cases, we can check for usability, scalability and validity for these design problems. This also allows us to better assess where and when the tool can be used in, possibly different, design methods, e.g. the V-model. Furthermore, these cases allow the DSL to become more comprehensive by unveiling potential inconsistencies and/or problems.

ACKNOWLEDGMENTS

This research was supported by Flanders Make, the strategic research center for the manufacturing industry in Belgium, within the Flexible Multi-Domain Design for Mechatronic Systems (FlexMoSys) project. This work was conducted using the CogniPy package, which is an open-source project maintained by Cognitum Services S.A. (https://www.cognitum.eu).

REFERENCES

[1] Albert Benveniste, Benoît Caillaud, Dejan Nickovic, Roberto Passerone, Jean-Baptiste Raclet, Philipp Reinkemeier, Alberto Sangiovanni-Vincentelli, Werner Damm, Tom Henzinger, and Kim Guldstrand Larsen. 2015. *Contracts for Systems Design: Theory.* Research Report [RR-8760]. Inria Rennes Bretagne Atlantique; INRIA. 86 pages.

[2] Saartje Brockmans, Peter Haase, and Boris Motik. 2010. *OWL 2 Web Ontology Language MOF-Based Metamodel (Second Edition).* World Wide Web Consortium. Retrieved May 2022 from https://www.w3.org/2007/OWL/wiki/MOF-Based_Metamodel

[3] Cognitum Services S.A. 2020. *CogniPy for Pandas - In-memory Graph Database and Knowledge Graph with Natural Language Interface.* Cognitum Software House. Retrieved March 2022 from https://cognitum.eu/cognipy/

[4] István Dávid, Joachim Denil, and Hans Vangheluwe. 2015. Towards Inconsistency Management by Process-Oriented Dependency Modeling. (2015). https://www.researchgate.net/publication/308967849

[5] Peter H Feiler, Bruce Lewis, Steve Vestal, and Ed Colbert. 2004. An overview of the SAE architecture analysis & design language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering. In *IFIP World Computer Congress, TC 2.* Springer, 3–15.

[6] Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. 2014. HermiT: An OWL 2 Reasoner. *Journal of Automated Reasoning* 53 (10 2014), 245–269. Issue 3. https://doi.org/10.1007/s10817-014-9305-1

[7] Ruben E.Y. Haasjes. 2019. *Metamodel Transformations Between UML and OWL.* Master's thesis. University of Twente, Enschede, The Netherlands. http://essay.utwente.nl/79481/

[8] Matthew Hause et al. 2006. The SysML modelling language. In *Fifteenth European Systems Engineering Conference*, Vol. 9. 1–12.

[9] Matthew Horridge and Sean Bechhofer. 2011. The OWL API: A Java API for OWL ontologies. *Semantic Web* 2 (2011), 11–21. Issue 1. https://doi.org/10.3233/SW-2011-0025

[10] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. 2008. ATL: A model transformation tool. *Science of Computer Programming* 72 (6 2008), 31–39. Issue 1-2. https://doi.org/10.1016/j.scico.2007.08.002

[11] Bart Kosko. 1986. Fuzzy Cognitive Maps. *International Journal of Man-Machine Studies* 24 (1986), 65–75.

[12] Edward Ashford Lee. 2017. *Plato and the Nerd The Creative Partnership of Humans and Technology.* The MIT Press.

[13] Boris Motik, Peter F. Patel-Schneider, and Bijan Parsia. 2012. *OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition).* World Wide Web Consortium. Retrieved May 2022 from https://www.w3.org/TR/2012/REC-owl2-syntax-20121211/

[14] Mark A. Musen. 2015. The protégé project: a look back and a look forward. *AI Matters* 1, 4 (2015), 4–12. https://doi.org/10.1145/2757001.2757003

[15] Khaled Nassim and Pattel Bibin. 2018. *Practical Design and Application of Model Predictive Control MPC for MATLAB® and Simulink® Users.* Mara Conner. 249 pages.

[16] Ahsan Qamar, Sebastian Herzig, and Christiaan J.J. Paredis. 2013. A Domain-Specific Language for Dependency Management in Model-Based Systems Engineering. (2013).

[17] Ahsan Qamar and Christiaan J.J. Paredis. 2012. Dependency modeling and model management in mechatronic design. *Proceedings of the ASME Design Engineering Technical Conference* 2, 1205–1216. Issue PARTS A AND B. https://doi.org/10.1115/DETC2012-70272

[18] Diego Camara Sales, Leandro Buss Becker, and Cristian Koliver. 2022. The systems architecture ontology (SAO): an ontology-based design method for cyber–physical systems. *Applied Computing and Informatics* (2022). https://doi.org/10.1108/ACI-09-2021-0249

[19] Martin Törngren, Ahsan Qamar, Matthias Biehl, Frederic Loiret, and Jad El-Khoury. 2014. Integrating viewpoints in the development of mechatronic products. *Mechatronics* 24 (10 2014), 745–762. Issue 7. https://doi.org/10.1016/j.mechatronics.2013.11.013

[20] Bert Van Acker, Joachim Denil, Alexander De Cock, Hans Vangheluwe, and Moharram Challenger. 2021. Knowledge Base Development and Application Processes Applied on Product-Assembly Co-design. *Companion Proceedings - 24th International Conference on Model-Driven Engineering Languages and Systems, MODELS-C 2021*, 327–335. https://doi.org/10.1109/MODELS-C53483.2021.00055

[21] Ken Vanherpen, Paul De Meulenaere, and Hans Vangheluwe. 2018. *A Contract-Based Approach for Multi-Viewpoint Consistency in the Concurrent Design of Cyber-Physical Systems.* Ph. D. Dissertation. University of Antwerp.

[22] Ken Vanherpen, Joachim Denil, Paul De Meulenaere, and Hans Vangheluwe. 2017. Ontological reasoning as an enabler of contract-based co-design. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10107 LNCS (2017), 101–115. https://doi.org/10.1007/978-3-319-51738-4_8

[23] Yon Vanommeslaeghe, Joachim Denil, Jasper De Viaene, David Ceulemans, Stijn Derammelaere, and Paul De Meulenaere. 2021. Ontological reasoning in the design space exploration of advanced cyber–physical systems. *Microprocessors and Microsystems* 85 (9 2021). https://doi.org/10.1016/j.micpro.2021.104151

[24] Yon Vanommeslaeghe, Joachim Denil, Bert Van Acker, and Paul De Meulenaere. 2021. Automatic Generation of Workflows for Efficient Design Space Exploration for Cyber-Physical Systems. *2021 IEEE International Conferences on Internet of Things (iThings) and IEEE Green Computing & Communications (GreenCom) and IEEE Cyber, Physical & Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics)*, 346–351. https://doi.org/10.1109/iThings-GreenCom-CPSCom-SmartData-Cybermatics53846.2021.00062

[25] W3C OWL Working Group. 2012. *OWL 2 Web Ontology Language Document Overview (Second Edition).* World Wide Web Consortium. Retrieved May 2022 from https://www.w3.org/TR/owl2-overview/