Weaving system-level properties with architectural decomposition for mechatronic co-design

Weaving System-Level Properties with Architectural Decomposition for Mechatronic Co-Design

Milan Cornelis, Yon Vanommeslaeghe, Bert Van Acker and Paul De Meulenaere
Cosys-Lab (FTI)
University of Antwerp
AnSyMo/Cosys
Flanders Make
Antwerp, Belgium
{milan.cornelis, yon.vanommeslaeghe,
bert.vanacker, paul.demeulenaere}@uantwerpen.be

***Abstract*—As mechatronic systems become more complex, their design requires different engineering teams, each specialized in their own domain, to cooperate. This is often accomplished through a co-design process, which aims to parallelize the engineering work and therefore improve the time-efficiency and cost of system development. However, the integration step following the concurrent design often fails due to incorrect or incomplete assumptions made regarding other domains. In this paper, we propose to use a combination of two different views on the system. One view is the traditional system design methods that aim at system decomposition. The other view aims at defining the system-level properties and their interrelations. By means of a drone case study, we demonstrate how to decompose the system into manageable components, while still capturing relationships between these components and their related properties. For the drone modeling and decomposition, we use ARCADIA, whereas for the management of the cross-domain system relations, we use a knowledge model and tool that we presented in previous work.**

***Index Terms*—Mechatronics, MBSE, Co-Design, Knowledge Base**

I. Introduction

Mechatronic systems, such as vehicles, machines, medical devices, etc., are systems that consist of various components, each of them associated with one or more engineering disciplines. A typical example of such a system is a robot arm. Indeed, it has several components belonging to different engineering disciplines. The mechanical team is responsible for designing the robot arm's physical structure, the control team develops the control algorithm for moving the arm with the required accuracy, and the embedded engineers provide a sufficiently embedded platform to execute this control algorithm. As the amount of functionalities these systems must perform rises, the involved domains become even more intertwined, making the design even more complex. Therefore, new design techniques are needed to deal with this complexity. Model-Based Systems Engineering (MBSE) is one such technique which has already been proven to be successful, whereby models are used as a base for designing a system.

MBSE can be combined with a large variety of traditional development processes, ranging from waterfall or V-model type of processes that start from a fixed set of requirements, to more agile-like development processes that are characterized by smaller system increments. All of these processes adhere to the central paradigm of system decomposition, federated component development and finally system integration. In this 'divide-and-conquer'-approach, MBSE largely supports the functional decomposition of the system into components and the system integration thereafter.

For example, a waterfall-like development process starts from system requirements to design the system. During the design, the system is decomposed into smaller, more manageable subsystems which can then be further broken down into components with their corresponding interfaces. Eventually, we come up with a system architecture model usually created with a standard formalism or method such as UML [1], SysML [2], ARCADIA/Capella [3], etc. These components can then be assigned to a specific engineering team. To accelerate the development process, the individual engineering teams can work in parallel on the detailed design and implementation of the system components for which they are responsible. This split out over different engineering teams allowing for parallel development is also known as a co-design process. Next, all components are integrated into a single system that is then verified and validated. It is clear that during such a process, the engineering viewpoint is primarily based on decomposing the system. We call this the decomposition view on mechatronic design in Figure 1.

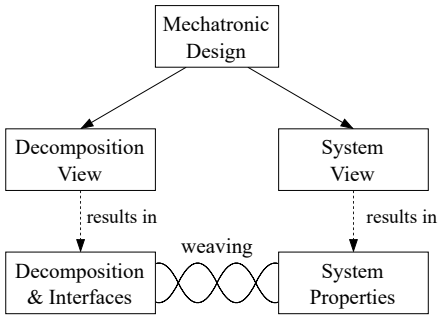

Fig. 1. Representation of the decomposition view and system view on mechatronic system design and their related resulting artifacts.

At first glance, this seems like a valuable process in terms of time efficiency and cost reduction. However, during parallel development in the co-design process, design choices made in one engineering team might influence several other elements of the system assigned to other engineering teams. Also, faulty assumptions might be made about other engineering teams' designs. Implementing these possibly wrong designs may cause the system to work suboptimally or, even worse, not work at all. Revisiting the example of the robot arm further explains this. The mechanical team decides, e.g., to change the material type of the robot arm, also causing its weight to change. This weight change also affects the behavior of the control algorithm. Thus, if this design change is not communicated between the design teams, it causes the control engineers to make an incorrect assumption of the robot arm's weight, which may cause the algorithm to perform suboptimal, as it is designed for the old weight. It might also be that multiple design changes lead to a change in the same system parameter. E.g., the total weight of the system depends on the weight of the individual components. Thus, changing these components might lead to a drastic change of the total system weight such that the system requirement regarding the total system weight might not be satisfied anymore.

Part of the problem thus lies in the fact that engineers often lack awareness of how their design choices affect properties of the system across different engineering domains. While certain influences might be obvious, such as the example of the robot arm weight, others can be more intricate. As the system decomposition does not cover such influences between design properties, it is valuable to look at the system from a different viewpoint. We refer to this new viewpoint as the system view in Figure 1. Within this view, we explicitly note the implicit relationships between the engineers involved, enabling us to identify how their domain-specific design choices can affect those of other domains. Eventually, this results in a cross-domain knowledge model (CDKM) containing system-level properties and their interrelations. We presented a tool to model this in previous work [4]. In the current paper, we propose to use the CDKM to support the design process by providing information about dependencies between system components. We examine how to weave the information about the system decomposition obtained from the decomposition viewpoint, with the system-level properties managed in the CDKM obtained from the system viewpoint. Ultimately, the goal is to obtain a system design with as little implementation errors as possible at the time of system integration.

The remainder of this paper is organized as follows. In Section II, we discuss related work. Then, in Section III, we present a drone as a case study. Next, in Section IV, we elaborate both the decomposition view and the system view on this drone. The decomposition view is elaborated with ARCADIA. For the system-level properties, we apply our previous work to the drone case study. Then, Section V presents the weaving of the two views. Finally, we conclude our work and discuss future work in section VI.

II. Related Work

Several attempts have already been made regarding dealing with the complexity and consistent design of mechatronic co-design. Vanherpen et al. [5] introduced a Contract-Based Co-Design (CBCD) method to support the consistent design of Cyber-Physical Systems (CPS) as an extension to the Contract-Based Design (CBD) method of Benveniste et al. [6]. By using an ontology, implicit domain knowledge can be made explicit in the form of properties, which enables relating different engineering domains to each other. Besides the ontology, they also make use of a mapping contract containing the design variables and their required value (or range). By relating the design variables in the mapping contract with the corresponding properties in the ontology, they can derive which design variables have an influence on each other. This way, Assume/Guarantee (A/G) contracts can be derived for each involved engineering domain, to support a co-design process. The concepts and ideas presented in this work are used as a base for our work.

In the openCAESAR initiative [7], they try to deal with the rising system complexity by providing a rigid modeling and analysis methodology that is tool-neutral. For realizing this modeling and analysis method, they found that Semantic Web technologies, such as ontologies, were the most appropriate and flexible formalism. However, they also found that using the Web Ontology Language v2 (OWL 2) [8] for this purpose was too cumbersome regarding the added accidental complexity and patterns one has to know. Therefore, they created the Ontological Modeling Language (OML) [9], which serves as the core of the openCAESAR project. OML is an ontology language designed for systems engineering that fully translates into OWL 2-DL and the Semantic Web Rule Language (SWRL) [10], which enables existing reasoners to check for logical consistency of the ontology. Other analysis tools can also be used with OML by mapping the information from OML to the tool-specific format [11]. OML consists of four kinds of ontologies: vocabulary, vocabulary bundle, description, and description bundle. With a vocabulary, one can define the architectural framework of the system. It describes the terms of a domain or methodology such as types, properties, and interrelations. Reasoning about vocabularies uses the Open World Assumption (OWA). An OWA states that information that is not modeled is assumed to be unknown, thus it is not true or false. In contrast, a Closed World Assumption (CWA) states that everything that is not modeled is assumed to be false. Vocabulary bundles combine one or more vocabularies and use this CWA for reasoning. With a description, we can instantiate the defined vocabulary/vocabularies to describe the system itself. As with a vocabulary bundle, the description bundle is a combination of one or more descriptions using the CWA for reasoning, whereas the OWA is used for a description [12]. In relation to the topic of the current paper, we observe that openCAESAR provides a formalism which could potentially be used to model the system-level properties and their interrelations.

The Embedded Systems Institute (ESI) has also noticed that systems are becoming increasingly complex to design. Therefore, they have developed a methodology, the DAARIUS methodology, which is based on domain knowledge of the different involved stakeholders in the design [13]. With this, they want to better handle the complexity regarding the system design. By using explicit reasoning, it also ensures that the impact of a design alteration on the system can be better understood. Also, it ensures better communication and collaboration between the different stakeholders as their interrelations are modeled. The DAARIUS methodology consists of four main elements: a method, a structure, a language, and a tool. The first element, the method, is concerned with obtaining information about the stakeholders' needs, the involved Knowledge Domains (KDs) in the design, early definable relations between KDs, etc. Different concepts are explored and it is found where the trade-offs lie. At the end, a decision is made regarding the design choices for the system configuration. The second part of the methodology, the structure, establishes the different abstraction layers in which to model the knowledge and information. The first level, i.e., system level, contains the information required to verify that the requirements are met. The lowest level, i.e., realization level, contains the components of the system and their corresponding properties. These two levels can then be connected to an intermediate level, called the quality level. This contains the realized values obtained by the configuration at the realization level. These realized values can then be compared to the values needed to validate if the stakeholders' concerns are met [14]. To effectively express the system in this structure, a language is needed. This language serves as the formalism, consisting of eight language elements, whereby the KDs and their interrelations are modeled, following the previously discussed method and structure. E.g., it can be defined which relations exist between parameters, how these parameters are related, or how to validate them [15]. The final element of DAARIUS is a web-based tool that supports the structure and language [16]. The current paper takes a similar approach regarding the different abstraction layers present in the structure. Some elements are related to the requirements, as some other elements are realization-oriented. However, the DAARIUS methodology is more focused on the conceptual design of a product whereas we are also concerned with the detailed design and development itself as we focus on co-design.

III. Drone Case Study

Throughout this paper, we use a drone as a case study. A drone was chosen as it is considered a complex mechatronic system where engineers of multiple domains need to collaborate to design the full system. Indeed, a drone contains components belonging to several engineering domains, e.g., a battery (electrical domain), a frame (mechanical domain), a control algorithm (control domain), etc. We identified five main engineering domains on which we focus. These are the mechanical, electrical, control, embedded and software domain.

When designing a system, we typically start with a set of system requirements and specifications. In this paper, we assume that the necessary system requirements for the drone are already available. Their acquisition or the format in which they are presented is not pertinent to the scope of this research. The mission the drone has to perform is to take-off and hover as long as possible at a height of 1.5 m. This hover height may not be exceeded by more than 50 mm in both directions. During its operation, the drone must be able to carry a payload which weighs maximum 1 kg. Table I shows an overview of the requirements.

TABLE I
System requirements for the drone.

ID	System Requirement
$R1$	The drone shall take off and hover at a height of 1.5 m.
$R2$	The drone shall not exceed the maximum absolute distance error of 50 mm relative to the hovering height.
$R3$	The time duration at hovering height shall be maximized.
$R4$	The drone shall be able to carry a payload of 1 kg.

IV. Modeling the Mechatronic Design Views

A. Modeling the System Decomposition

In this section, we present ARCADIA/Capella as the methodology we used to create an architectural decomposition of the drone. ARCADIA is a model-based engineering method to design systems and is inspired by SysML, UML and NAF standards [3]. The method consists of four different engineering phases, all with a variety of diagrams that can be used to model the system under design. The first phase, called *Operation Analysis*, considers what the user should be able to do with the system, i.e., defining user operations. "Place the drone in position" and "switch on the drone" are such examples of functions a user can perform. The second phase is called *System Analysis*. Here, we focus on what functions the system itself must carry out to achieve the desired behavior and meet the requirements, e.g., "provide thrust", "calculate movement action" and "receive data". After this, we can divide the system into logical components in the third phase called *Logical Architecture*. We can link the functions previously defined in the *System Analysis* to a logical component which is responsible for it. It enables us to think about how the system will work in order to meet the expectations. Important to note is that we only think about **what** the system should perform in this phase. In the final phase, *Physical Architecture*, we break down the system into its physical components and the required behaviors these components perform. E.g., the drone contains a propulsion system which consists of a propeller and BLDC (brushless direct current) motor. Also, a flight controller unit is included on which a control algorithm runs to let the drone fly. As we can see, in this final phase, we focus on **how** the system is built. With this information in mind, we can better support and explain the concepts and ideas we discuss in Section V.

It also allows us to identify which information is available or missing in each design phase and how the system-level properties could help with this.

B. Modeling the System-Level Properties

In the previous section, we focussed primarily on the traditional system decomposition. Now, we switch our viewpoint and focus on the properties related to the full system. In order to formally capture these properties, we already created a tool in previous work [4]. As we use this tool to also model the CDKM of the drone, we first give a brief overview of its structure and possibilities. As shown in Figure 2, the tool consists of two main parts: a modeling part and a reasoning tool. With the modeling part, which is fully developed with the Eclipse Modeling Framework (EMF) [17], we can create the CDKM instance model. Modeling can also be done partly graphically with Sirius [18] for easier and faster population. After modeling the CDKM, we want to reason about it to derive new information by using the reasoning tool. To achieve this, we use an ontology as the fundamental formalism such that we can use already existing reasoners such as HermiT [19]. The Web Ontology Language v2 together with additional SWRL rules are used to realize this. The reason we do not directly use OWL 2 as the modeling language lies in the accidental complexity that comes with it. The modeling part of the tool hides this ontology-specific complexity by only focussing on the essential complexity related to modeling system-level properties of mechatronic systems. After transforming and serializing the CDKM to an OWL 2 ontology and other artifacts, they can then be used in the reasoning tool. The reasoning tool is a GUI developed in Python (with accompanying API for tool interfacing), with which the user can reason on the CDKM by using an ontology reasoner as a base in the background. Some reasoning functions are, but not limited to:

- Deriving the interdependencies between various engineering domains and their scopes.
- Deriving trade-offs between design variables, assuming sufficient detail about the data is available.
- Generate Assume/Guarantee engineering contracts to enable co-design between different engineering domains.

We can now start modeling the drone CDKM with the modeling part of the tool. The final model, which we will gradually build up, is shown in Figure 3. Note that we only included the necessary amount of details to explain the concepts and to keep a clear drawing. After obtaining the system requirements, engineers start thinking about how they can realize these. Hereby, they ask themselves questions with regard to what the system design must do to satisfy the system requirements. These questions can be referred to as properties that are ontological concepts of a (mechatronic) product. They can be regarded as requirements, as they tell something about the system as a black box, and the system obviously also has to fulfill these ontological questions, i.e., these properties. This concept is based on the work of Vanherpen [20]. This brings us to the first part of the model,

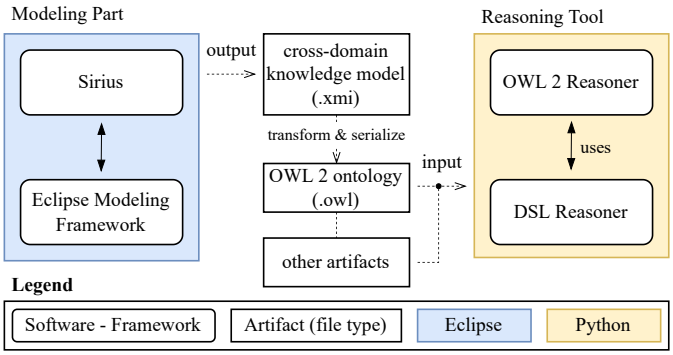

Fig. 2. A visual representation of the different elements of the tool from [4]. It contains a modeling part, shown on the left, as well as a reasoning tool, shown on the right. A cross-domain knowledge model can be created with the modeling part, containing system-level properties and their interrelations. This model is then transformed and serialized to an OWL 2 ontology, which can be used by the reasoning tool to derive information.

which is called the Property Ontology. Here, we typically start by defining a root property, e.g., *ValidSystem?*. This property tells something about the validity of the system, i.e., whether the system meets all requirements. To determine if this is the case, we need to break down this property into additional properties that can be connected via a requires-relationship. These additional properties are directly derived from the system requirements, therefore, we classify these as properties at the requirements-level. Indeed, in order for the system to be valid (*ValidSystem?*), it also *requires* that it satisfies all requirements. Deriving these for the requirements $R1$–4, we come up with the properties *HoverHeightOK?*, *IsMaxDistanceErrorSatisfied?*, *MaximizedHoveringTime?* and *CarryEnoughPayload?* respectively. As one can see, properties represent questions, meaning that if we can answer "yes" to these properties, we can say that these properties are satisfied. As the answer to these questions propagate upwards, we can infer that the property *ValidSystem?* is also satisfied when the properties it requires are also satisfied. Important to note is that properties at requirements level are always assigned to the system domain as they directly relate to the entire system.

Next, engineers can model their implicit viewpoint-specific knowledge on the system. This can also be done with properties in the Property Ontology, however, these properties now represent domain-specific knowledge of the different involved engineers about their viewpoint on the system. These properties tell something about what the system must do in order to meet the requirements. Consequently, during this thinking process, the engineers already start thinking about the domain-related elements (i.e. conceptual system functions or components) the system could potentially involve. Given that this is related to the conceptual architecture of the total system, we classify these properties at the architectural level. The properties can then be connected to either the corresponding properties at requirements level or to other properties at architectural level. Moreover, these properties can also belong to one or more engineering domains. Some of

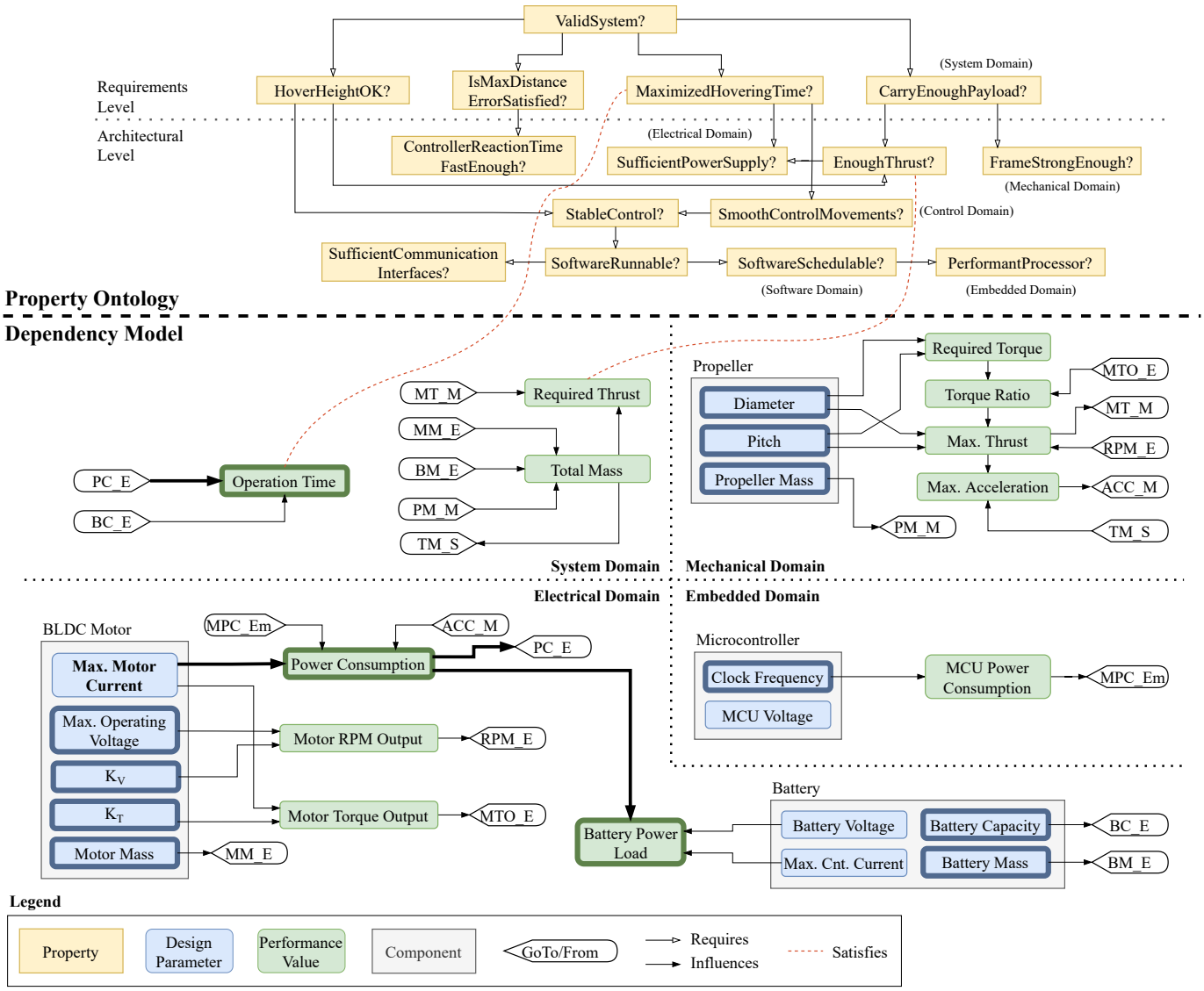

Fig. 3. A (reduced) ontology for the drone case. The Property Ontology is shown at the top and the Dependency Model is shown at the bottom. Currently, only three engineering domains are considered and not all satisfies-relations are shown in order to keep a clear drawing.

the domains are shown in the example. Taking a closer look at *MaximizedHoveringTime?*, we see that it requires another property *SufficientPowerSupply?*, belonging to the electrical domain, and *SmoothControlMovements?*, belonging to the control domain. These properties in itself can also require other properties, e.g., *SmoothControlMovements?* requires *StableControl?* which also requires *SoftwareRunnable?* and so forth.

At the bottom of Figure 3, one can see the second main part of the CDKM, called the Dependency Model. This is based on the work of Vanommeslaeghe et al. [21] and corresponds to the Dependency Model as proposed by [22]. This part of the model is focused on the design parameters and performance values per engineering domain, and their mutual relationships. These design parameters and performance values are now also related to the components obtained from the system decomposition mentioned in Subsection IV-A. As can be seen, the Dependency Model consists of various elements. Firstly, there are domain models that contain the specific elements associated with a domain. These domain models are indicated with the dotted lines, e.g., *Embedded Domain*. Secondly, there are Components, e.g., *Battery*, that contain Design Parameters (DPs). These DPs can be seen as the tuning knobs of the system design. Typically, a DP is a setting that is decided by the designer, e.g., the DP *Clock Frequency* of the Component *Microcontroller*. It may also be possible that a DP changes when changing the component itself. E.g., by changing the battery, we also change (some of) the values of its DPs, such as *Max. Motor Current*, K_V, K_T, etc. Thirdly, we have Performance Values (PV). These are values that result from measurements, calculations, simulations, etc. The determination of PVs is not entirely

arbitrary. As they will be used to validate whether the system meets the requirements, we need to select them such that they provide a value that gives a meaningful result for the validation. E.g., $R3$ stated that the hover duration shall be maximized. This requirement has to be validated somehow. By evaluating the PV *Operation Time*, we can indeed validate that the hovering time is maximized for the specific drone configuration. We can formally model this by using a satisfies-relation, to connect a PV of the Dependency Model with a property in the Property Ontology. Two examples for the satisfies-relation are shown to illustrate this concept. Finally, we have the influence-relation, with which we indicate which DPs or PVs affect other PVs. E.g., the PV *Power Consumption* is influenced by the DP *Max. Motor Current* but also other PVs like *Max. Acceleration* and *MCU Power Consumption*. Note that the influence-relation is transitive. Thus, because *Max. Motor Current* influences *Power Consumption* and *Power Consumption* influences *Operation Time*, the DP *Max. Motor Current* also influences the PV *Operation Time*. In practice, these influence-relations can show different levels of detail, but this is out of the scope of this paper.

V. Weaving the Decomposition and System View

So far, we have defined the drone as a case study, for which we then created a physical architecture, containing the systems' components, using the ARCADIA method. In addition, we also created a CDKM containing the system-level properties and their relations. In this section, we explore how we can weave the information from the CDKM with the system decomposition from ARCADIA such that the resulting development process endorses a more collaborative system design. Also, we try to formalize which information can be used from a specific design phase to set up the CDKM. Comparing the CDKM with the models obtained by following ARCADIA serves as a mechanism to explore how these two can be interleaved. The final objective is to ensure that the information in the CDKM can be used with diverse system design approaches, such as the V-model, Agile, DevOps, etc. Therefore, we try to generalize everything such that we come up with Figure 4. This figure summarizes the final result with the general design steps. These steps will guide us through the explanation that follows. We are aware that the design phases may vary among different types of design methodologies. However, it can be noted that the design phases included in Figure 4 are found in the most common design methodologies. This way, we want to enable one to easily apply the ideas we present to their own design process used for their mechatronic system, assuming that their design process incorporates similar design phases as the ones presented here. Also, note the dotted arrows at the start and end of the figure, suggesting that the design process does not necessarily start or stop at these phases and may even circulate between end and start in case of incremental design processes. Requirements may indeed be subject to change, causing the design phases to be executed again. Also, system design is often considered as an iterative process whereby more detailed information is obtained later in the design process, which can further refine requirements or design choices [23].

A. System Requirements

As shown in Figure 4, the initial phase of the design process is called *System Requirements*. During this phase, the system's requirements are identified and defined. The specific details on the elicitation of these requirements and how they are represented are not in the scope of this paper. Our only concern is that these files are accessible by the designers/engineers. Subsequently, these requirements can be used as input for the initial part of the Property Ontology. Indeed, recall that when we set up the CDKM for the drone case in Section IV-B, we initially derived properties that correspond directly to the system requirements we imposed. These requirements are typically associated with the "system domain" as they concern the entire system. Hence, we obtain a Property Ontology featuring properties at the requirements level.

In addition to the information obtained from the *System Requirements* phase, there is a reciprocal exchange of information that can occur between the Property Ontology and the requirements. Indeed, properties allow us to verify that the system meets the requirements. This verification process is complemented by a satisfies-relation, on which we will elaborate later. Thus, by evaluating a property positively, we can infer that the corresponding requirement is fulfilled as the property is directly derived from this requirement. For instance, if the property *CarryEnoughPayload?* receives the response "yes", it indicates that the system fulfills requirement $R4$.

B. Architectural Design

After deriving and defining the requirements, we arrive at the *Architectural Design* phase. This phase in itself consists of several smaller steps to be performed, as we gradually go from conceptual ideas to a physical architecture of the system under design. Therefore, the *Architectural Design* also includes most of the information required to build the CDKM.

Firstly, a system engineer or engineering teams starts analyzing the requirements for their feasibility and think about the functions that must be performed by the system. Taking a look at ARCADIA, this corresponds to the *Operational Analysis* and *System Analysis* phases. Remember that in these phases, we derive the user needs, that is, what the user should be able to do with the system and what functions the system itself should perform, respectively. With this, we can analyze the requirements for their feasibility and already think about some preliminary concepts for the design. Currently, we don't see a direct relation between this functional analysis of the system and the CDKM, in which they could support each other.

However, the derived system functions can serve as input to the conceptual design of the system. Here, engineers think about what elements of the system are needed to perform the predefined functions. We can model these elements in ARCADIA in the *Logical Architecture* phase. Here, we divide the system into logical components, to which the system functions

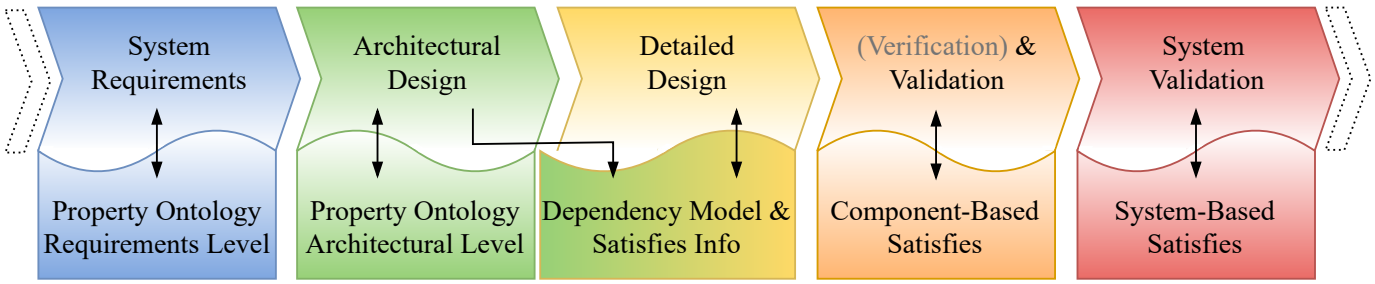

Fig. 4. Weaving of the knowledge information (bottom row) with the different stages of a design process (top row).

are assigned. In addition, we can also add more information to the Property Ontology in this stage of the process. As engineers think about what logical or conceptual components are needed to perform the functions, they also implicitly think about how they could realize this within their own domain. This implicit knowledge can be modeled as properties at the architectural level, as they are related to the architecture of the system. This leads indirectly to traces between their domain-specific design to ensure that the requirements are met by executing the correct functions. Therefore, these architectural properties are refinements of the requirements properties. Thus, these architectural-level properties need to be satisfied to also satisfy their upper requirement-level property(s). An example makes this more clear. The requirement $R3$ states that the hovering duration of the system shall be maximized. This is indicated by the property *MaximizedHoveringTime?*. We can break this property further down into architectural properties that represent the knowledge engineers think of within their own domain about how to realize this. E.g., the electrical domain knows that, in order to fly as long as possible, a power supply is needed that provides enough energy to enable this, indicated by *SufficientPowerSupply?*. Also, control engineers know that smooth control operations typically require less power than jerky movements. Therefore, a property *SmoothControlMovements?* is included. These properties themselves can also be further refined. At this phase of the design, there is often a lack of available system details to establish a comprehensive Property Ontology right at once. However, it should be noted that the Property Ontology is not static but rather dynamically evolves with the system design. As the system design progresses, additional knowledge is obtained, which can be used to further update and detail the Property Ontology. How and when to update the Property Ontology, however, falls out of scope of this research and is considered as future work.

The conceptual designs as well as the information in the Property Ontology can now be used in the development of the physical architecture. This physical architecture contains the different components the system is made out of, with their interfaces, as well as which functions they perform. This is done in the *Physical Architecture* phase of ARCADIA. We can use the reasoning tool to derive information from the Property Ontology that can help design the physical architecture. E.g., we can ask which engineering domains influence each other and also about which concepts of the system. Hereby, the different teams can collaborate on the possibilities of the component choices and what they need from each other. For example, the architectural property *PerformantProcessor?* influences the properties *SoftwareSchedulable?*, *SoftwareRunnable?*, *StableControl?*, *SmoothControlMovements?*, *HoverHeightOK?*, *MaximizedHoveringTime?* and *ValidSystem?*. If we use the reasoning tool to find the path between these properties, we obtain Figure 5 representing a graphical notation of the path. With this information, we not only know which domains (and related properties) are influenced, we also know something about the path between them. Hence, we know that there is an intermediate domain between *PerformantProcessor?* and *SmoothControlMovements?*, i.e., the software domain, for which it might also be convenient that they participate in the negotiations between the engineering domains since they are also influenced in this property chain. Indeed, when an advanced control algorithm is used, it means that it also needs a processor that runs the software fast enough to obtain control actions on time.

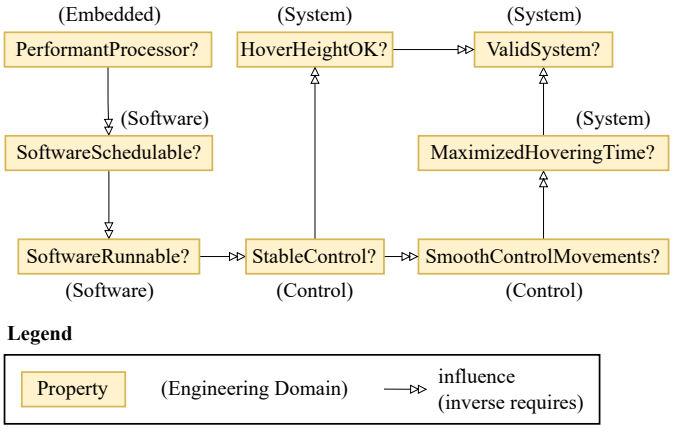

Fig. 5. Here, it is graphically shown which properties are influenced by *PerformantProcessor?* and to which domains they belong.

After carefully analyzing the system, the engineers come up with a physical architecture that consists of the different components of the system and their interconnections. These components are also assigned a set of adjustable parameters, referred to as the Design Parameters. This system decom-

position is used as input to create the Dependency Model. Previously, we have modeled some of the drone components in its Dependency Model in Figure 3. Additionally, it can be seen that the components have been assigned to specific engineering domains. E.g., the electrical domain is responsible for the *BLDC Motor* and *Battery* components, while the embedded domain is responsible for the *Microcontroller*. In addition to the DPs, there are Performance Values. As already mentioned in Section IV-B, these PVs are chosen such that they can represent a value that is used to validate a certain property. This is done by connecting the appropriate PV with the corresponding property through a satisfies-relation with an evaluation function. This evaluation of PVs via such satisfies-relation can be used in the validation stages to verify that the system meets its requirements. As we can see, we essentially "close the modeling process" with the satisfies-relation. As we previously modeled the properties regarding the requirements in the Property Ontology and the components with their DPs plus PVs in the Dependency Model, we now connect these two leading to trace back the physical system itself to the corresponding requirements. While there is still some potential for deriving additional information with this satisfies-relation, such as reasoning about completeness of the CDKM, it is beyond the scope of this paper. When all components, with their corresponding DPs, and PVs are known, they can be connected using the influence-relationship.

C. Detailed Design

At this stage, we have obtained a physical architecture that divides the system into its components, on the one hand, and a Dependency Model that relates these components and their impact on the system, on the other hand. The Dependency Model can now support the next phase of the design process, which is the detailed design. As the architecture has been defined at this point, each engineering team knows what they have to develop. Therefore, they can start working in parallel in order to accelerate development, i.e., work in a co-design process. Engineering contracts are crucial for this co-design in order to keep consistency, by specifying which values for the variables they can assume and which values they have to guarantee within their own design. We can use the reasoning tool to generate these contracts. It will analyze how two domains are related to each other via the relations in the Dependency Model and provide us with an A/G contract for each domain accordingly.

During the detailed design phase, the domain engineers proceed to refine their part of the system's design. E.g., the hardware team designs the Printed Circuit Board (PCB), while the mechanical team is responsible for constructing the frame of the drone. It is possible that design changes made to the components may lead to the violation of the predetermined values outlined in the engineering contract. This could occur because the modification may impact some of the performance values, resulting in one or more properties not being satisfied anymore. In such cases, the reasoning tool uses the information in the Dependency Model to determine which domains are affected and also proposes a possible solution to solve this. Suppose the electrical engineering team opts for a different *BLDC Motor*, which lead to a different value for *Max. Motor Current*. With the Dependency Model, we can observe the impact of this change on different PVs. This is illustrated in Figure 3, where the bold arrows indicate the affected PVs, which are also framed in bold. If the change of this DP results in requirements not being met anymore, we can consider potential modifications to the design to resolve the issue. To do so, we examine other DPs that have a common influence to the violated PV. For example, adjusting the value of *Max. Motor Current* causes *Operation Time* to change, resulting in the hovering duration time not being maximized anymore. To resolve this issue, we need to make a change in the design to keep the hovering time maximized. However, we wish to keep the new choice of the *BLDC Motor*. Therefore, another DP has to be changed. First, we determine which components are suitable for this purpose. To do this, we examine which other DPs influence *Operation Time* by following the influence relations in the opposite direction until we reach a DP. All DPs that meet these criteria are framed in bold. These also indicate potential solutions to our problem. For example, to resolve the issue, we could change the value of *Battery Capacity*. Another potential solution is to change the value of *Clock Frequency*. However, it is worth noting that a microcontroller typically consumes much less power than a BLDC motor, so this change may not have a significant impact, but it is worth to be informed about the possibility.

D. Verification & (System) Validation

The last two steps of the design process are the verification and validation of the individual components and/or the system itself. During the verification process, we check if the product we built actually is the right product. Currently, the CDKM contains no information that can support this verification process. Therefore, we do not further elaborate on this. Nevertheless, we can assist the validation process by using the satisfies-relations that we defined in the previous design phase. Through this relation and its evaluation function, we know which values of the system we have to analyze. If these values are within the range specified by the evaluation function, the corresponding property is fulfilled, which is desirable. If this is not the case, we can use the reasoning tool to identify the components that influence this PV and uncover potential causes for this deviation from the required value. The reasoning tool can also suggest potential solutions to solve the issue by following a similar approach as previously discussed with the example of selecting a new *BLDC Motor*.

VI. Conclusion & Future Work

In this paper, we proposed to integrate a mechatronic cross-domain system design viewpoint with the viewpoint of the traditional system decomposition. Practically, we presented how we can capture engineering knowledge and system knowledge as system-level properties in a CDKM and how this system-wide information can be used to augment cross-domain

consistency. Essentially, we showed that this is a reciprocal process between setting up the CDKM with information obtained from the design process and providing information about dependencies from the CDKM to the design process. Through this study, we discovered how the different elements of the CDKM can be woven with the different design phases of system design and system decomposition. From this analysis, we also hinted to other more incremental design processes to generalize our results as much as possible.

Although we have already discussed the ideas presented in this paper on a small drone case, there is still a need to thoroughly validate these in future work. This involves further extending the CDKM with more components and relationships between the different parameters. This allows us to assess the scalability of the proposed ideas and how they are practically applicable to larger, more complex systems. Also, we see a lot of potential in the satisfies-relationship between the Property Ontology and the Dependency Model. By making this connection, we can potentially derive additional information that further supports the design and evaluation of mechatronic systems. This can include checking for the completeness of the CDKM, identifying potential inconsistencies, and providing feedback on the correctness of the model. We also want to examine how the information in the CDKM evolves during the development. In the early design phases, the CDKM contains less information, rather of a conceptual nature, as not many details are already known about the system under design. However, when more design decisions are made during the further development stages, the model can be continually updated. It is still a question how this evolution actually takes place and from which point onwards the CDKM can provide useful information to the system design.

ACKNOWLEDGMENT

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.

REFERENCES

[1] OMG, *OMG Unified Modeling Language (OMG UML), Version 2.5.1*, Object Management Group Std., 2017. [Online]. Available: https://www.omg.org/spec/UML/2.5.1/

[2] ——, *OMG Systems Modeling Language (OMG SysML), Version 1.5*, Object Management Group Std., 2017. [Online]. Available: https://www.omg.org/spec/SysML/1.5/

[3] P. Roques, "MBSE with the ARCADIA Method and the Capella Tool," in *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, Toulouse, France, Jan. 2016. [Online]. Available: https://hal.science/hal-01258014

[4] M. Cornelis, Y. Vanommeslaeghe, B. Van Acker, and P. De Meulenaere, "An ontology dsl for the co-design of mechatronic systems," in *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, ser. MODELS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 633–642. [Online]. Available: https://doi.org/10.1145/3550356.3561534

[5] K. Vanherpen, J. Denil, P. D. Meulenaere, and H. Vangheluwe, "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)*, vol. 10107 LNCS, pp. 101–115, 2017.

[6] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K. G. Larsen, "Contracts for systems design: Theory," Inria Rennes Bretagne Atlantique; INRIA, Research Report [RR-8760], 2015.

[7] California Institute of Technology. (2023) openCAESAR. [Online]. Available: https://www.opencaesar.io

[8] W3C OWL Working Group. (2012) OWL 2 web ontology language document overview (second edition). World Wide Web Consortium. [Online]. Available: https://www.w3.org/TR/owl2-overview/

[9] M. Elaasar and N. Rouquette. (2023) Ontological modeling language 2.0.0. California Institute of Technology. [Online]. Available: https://www.opencaesar.io/oml/

[10] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. (2004) SWRL: A Semantic Web Rule Language Combining OWL and RuleML. World Wide Web Consortium. [Online]. Available: https://www.w3.org/Submission/SWRL/

[11] D. Wagner, S. Y. Kim-Castet, A. Jimenez, M. Elaasar, N. Rouquette, and S. Jenkins, "CAESAR model-based approach to harness design," in *2020 IEEE Aerospace Conference*, 2020, pp. 1–13.

[12] A. Zindel, S. Feo-Arenis, P. Helle, G. Schramm, and M. Elaasar, "Building a semantic layer for early design trade studies in the development of commercial aircraft," in *2022 IEEE International Symposium on Systems Engineering (ISSE)*, 2022, pp. 1–8.

[13] Embedded Systems Institute. (2023) DAARIUS methodology. [Online]. Available: https://esi.nl/research/output/methods/daarius-methodology

[14] T. Bijlsma, B. van der Sanden, Y. Li, R. Janssen, and R. Tinsel, "Decision support methodology for evolutionary embedded system design," in *2019 International Symposium on Systems Engineering (ISSE)*, 2019, pp. 1–8.

[15] T. Bijlsma, W. T. Suermondt, and R. Doornbos, "A knowledge domain structure to enable system wide reasoning and decision making," *Procedia Computer Science*, vol. 153, pp. 285–293, 2019, 17th Annual Conference on Systems Engineering Research (CSER). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050919307409

[16] H. Moneva, R. Hamberg, and T. Punter, "A design framework for model-based development of complex systems," in *32nd IEEE Real-Time Systems Symposium 2nd Analytical Virtual Integration of Cyber-Physical Systems Workshop, Vienna*, 2011.

[17] Eclipse Foundation. (2023) Eclipse Modeling Framework (EMF). [Online]. Available: https://www.eclipse.org/modeling/emf/

[18] ——. (2022) Sirius. [Online]. Available: https://www.eclipse.org/sirius/

[19] B. Glimm, I. Horrocks, B. Motik, G. Stoilos, and Z. Wang, "Hermit: An owl 2 reasoner," *Journal of Automated Reasoning*, vol. 53, pp. 245–269, 10 2014.

[20] K. Vanherpen, P. De Meulenaere, and H. Vangheluwe, "A contract-based approach for multi-viewpoint consistency in the concurrent design of cyber-physical systems," Ph.D. dissertation, University of Antwerp, 2018.

[21] Y. Vanommeslaeghe, J. Denil, J. De Viaene, D. Ceulemans, S. Derammelaere, and P. De Meulenaere, "Ontological reasoning in the design space exploration of advanced cyber–physical systems," *Microprocessors and Microsystems*, vol. 85, 9 2021.

[22] A. Qamar and C. J. Paredis, "Dependency modeling and model management in mechatronic design," vol. 2, 2012, pp. 1205–1216.

[23] B. Nuseibeh, "Weaving the software development process between requirements and architectures," in *Proceedings of the 23rd International Conference on Software Engineering, International Workshop on Software Requirements to Architectures*, 2001.