Proceedings of the ASME 2012 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference
IDETC/CIE 2012
August 12-15, 2012, Chicago, IL, USA

DETC2012-70272

DEPENDENCY MODELING AND MODEL MANAGEMENT IN MECHATRONIC DESIGN

Ahsan Qamar
Department of Machine Design
School of Industrial Engineering & Management
KTH Royal Institute of Technology
Stockholm, Sweden

Christiaan J. J. Paredis
Model-Based Systems Engineering Center
G. W. Woodruff School of Mechanical Engineering
Georgia Institute of Technology
Atlanta, Georgia, United States of America

ABSTRACT

Mechatronic design is traditionally supported through domain-specific design activities throughout the product development process. The partitioning into domain-specific problems leads to a situation where product properties influence each other, hence giving rise to dependencies. These dependencies play a key role in prediction of properties and as a result, in the decision making process. The important question is: how to manage the dependencies for an efficient and effective decision making? The aim of this paper is threefold. Firstly, we investigate the nature of dependencies and study how to model them. The paper proposes appropriate language constructs taking into account synthesis and analysis nature of properties and dependencies. The concepts related to the dependency modeling are then illustrated through a simple robot design example, where the creation and importance of a dependency model are explained. Secondly, we study practical approaches for consistency management and model management in the presence of dependencies. Six levels-of-detail in modeling dependencies are presented; emphasizing that modeling at higher level-of-detail ensures that more inconsistencies are avoided. Available languages such as OMG SysML™ are evaluated for a possible creation of the dependency models leading towards executable dependency networks. However, at present, SysML does not provide sufficiently rich language constructs to model dependencies. Thirdly, we compare our dependency modeling approach to the other state-of-the-art approaches such as dependency modeling with a Design Structure Matrix, and highlight the benefits of the language constructs proposed in this paper. We aim to convince the reader that there is a substantial value in modeling dependencies explicitly, especially to avoid inconsistencies, which is not the current state of the practice. However, an overall value from dependency modeling can only be obtained if the cost of creating the dependency model is reasonable. Issues such as human interaction/effort and model management through PLM are discussed.

1 INTRODUCTION

Mechatronic design is a multi-disciplinary activity performed by multi-disciplinary design teams. In managing the design of such complex products, a model-based approach promises better complexity management, improved design quality, better knowledge reuse and improved communication [1]. These advantages are however also accompanied by challenges such as model management, interoperability, and consistency management. The ultimate goal in employing a model-based approach over a document-based approach is to make better decisions as early as possible (effectiveness) while utilizing fewer resources (efficiency).

Using formal models enables use of computational power in predicting the outcomes. As complexity increases, the information which is typically part of a design specification becomes large, and computers can handle such large specifications much better than humans do through documents. Typically a model repository enables employment of version and model management approaches over different sets of models. A model is only useful if it is consistent with the rules of the underlying formal system, with the beliefs and preferences of the designer, and with the laws of nature [2]. Ensuring consistency across models is a challenging issue. Among inhibitors to consistency management, humans who create models can be prime suspects, such as not conforming to the rules of the language, or laws of nature, or introducing changes in a model [2]. The underlying dependencies between modeled properties are usually only implicitly known. By implicit, we mean that engineers in an organization possess knowledge about dependencies (at coarse-grain level) in their minds. However, these dependencies are never captured formally or explicitly in a document or a model. Because of only implicit knowledge of dependencies, it is difficult to deduce the effects of changes introduced in one model on other models, resulting in inconsistencies. The focus of this paper is to investigate to what extent we can capture the dependencies explicitly.

Herzig et al. [2] discuss fundamental concepts in consistency management along with deducing which types of inconsistencies can be rectified. The aim of this paper is to take a step further and study the nature of dependencies at a fundamental level. Insufficient management of dependencies is a key inhibitor against ensuring consistency [3, 4]. Therefore, it is important to study the nature and role of dependencies existing both inside a model and in between models. In current industrial practice, workflow management techniques are employed to synchronize and streamline the design activities, with an implicit management of dependencies. A process describing the activities to be performed is used, and specifications are used to direct the design process and to set targets for the predicted outcomes.

The main hypothesis this paper aims to address is that *making dependencies explicit facilitates consistency management and adds value*. In trying to validate this hypothesis during the course of this research, the following questions are asked:

- How detailed should the dependency representations be?
- How to manage the dependency models and how will humans utilize these models?
- How to quantitatively compare the value gained against the effort required in creating and managing dependency models?
- What type of support is available in current modeling languages to model dependencies explicitly?

The remainder of this paper is structured to answer the above questions. Section 2 discusses the fundamental concepts for understanding and representing dependencies. An illustration to the dependency modeling fundamentals is provided in section 3. Approaches for consistency management are discussed in section 4. The developed ideas are compared against the current state of the art in section 5. Section 6 presents a discussion on key issues, followed by concluding remarks and future work proposal in section 7.

2 FUNDAMENTALS OF DEPENDENCY MODELING

In order to understand the characteristics of dependencies in engineering design, it is important to form a fundamental basis for properties (among which dependencies exist), concept creation, design space exploration and decision making. Hazelrigg [5] considered the design as a decision-making process. Models aid in the decision-making process provided that they are based on rational beliefs and are consistent with preferences of the designer and observations of nature [2]. Rational Design Theory (RDT) explains the relationship between properties in design and the decision making process [6]. This theory introduced a conceptual model for design where properties are used to define concept specifications, concept predictions, and decision making criteria. In the following, a short description on property and property spaces is provided based on RDT [6]. A *property* is any descriptor of an artifact. Mathematically, a property is a function defined over the artifact set: $\boldsymbol{p}: \boldsymbol{A} \rightarrow \boldsymbol{Y}$, where $\mathbf{A} \subset \boldsymbol{S'}$: the domain of the property, and with $\boldsymbol{Y}$ a topological space that is its range (Figure 1). For a given realization $\boldsymbol{s} \in \mathbf{A} \subset \boldsymbol{S'}$, a well-defined property has a specific, unique value $\boldsymbol{y} \in \boldsymbol{Y}$, that is: $\boldsymbol{p(s)=y}$. The Cartesian product of all the property ranges is called the *property space* $\boldsymbol{P}$. $\boldsymbol{P'}$ is the subset of a property space and is called the *property projection*, which appears when designers select a finite number of properties that make up a system specification. When the designers select relevant properties for an artifact, it forms the basis for defining a product concept.

In design, we use properties for describing constraints (*specification)*, or for communicating designer's belief regarding what the value of the property is, given a particular specification (*prediction*). Properties can have a logical or numerical nature. For example a car's property *having an engine* to be true to false, or property *horsepower* with a value of 300 hp.

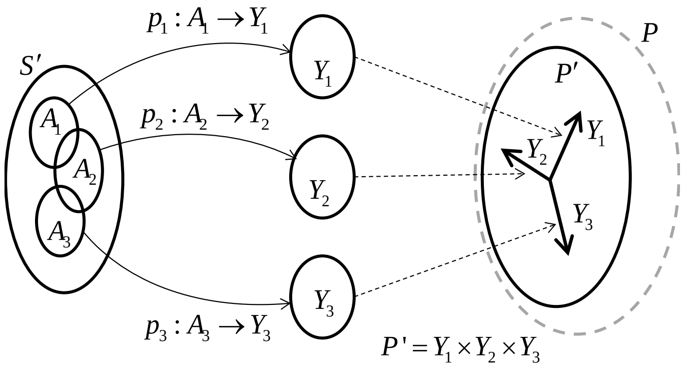

$p_1 : A_1 \rightarrow Y_1$
S'
A_1
A_2
A_3
$p_2 : A_2 \rightarrow Y_2$
$p_3 : A_3 \rightarrow Y_3$
Y_1
Y_2
Y_3
P
P'
Y_1
Y_2
Y_3
$P' = Y_1 \times Y_2 \times Y_3$

Figure 1. Artifacts and property spaces [6].

We make a fundamental distinction between properties in terms of *synthesis* and *analysis* properties. Synthesis properties (SPs) are used to define system alternatives. The specification of a system alternative consists of a set of constraints imposed on SPs. The values of SPs are typically specified in terms of *property ranges* — for instance, tolerance constraints on the specified geometry. Multiple SPs together represent a specification of a system alternative considered in a design decision. Through analysis and optimization, the designer will ultimately choose one of the specified alternatives as the most preferred alternative. SPs could be either specified directly or *computed* in a scenario where a system alternative is defined parametrically. Here a system alternative is characterized by some SPs that are freely chosen and other SPs that are defined as parametric functions of the freely chosen ones.

Analysis properties (APs) capture the beliefs of the designer, and constitute predictions rather than specifications of system alternatives. For instance, given a specified geometry (using SPs), the designer may predict the cost or the mass of a component — cost and mass are Analysis Properties in this example. Since APs are predictions about the future, which is inherently uncertain, they should be expressed in terms of *probabilities* or probability density functions (as opposed to property ranges for SPs). APs are often computed by using predictive models which mathematically capture the beliefs of the designer. Although it is not absolutely necessary for the mathematical relationship between APs and SPs to be modeled explicitly, the APs must be kept consistent with the SPs that define the corresponding system alternative. We illustrate the process of choosing SPs and predicting APs through a robot example in section 3.

For a property to have an unambiguous meaning, it needs to be defined in a *semantic context*. This context is established through relationships with *concepts*, giving rise to a *network of properties*. For example, consider a system alternative for a car defined through the following constraints on SPs: 4 wheels, an engine, engine with 4 cylinders, cylinder bore diameter 0.100 +/- 0.0001m. The SP for the diameter of the cylinder is a value associated with the concept “Cylinder”. The cylinder in turn is a *part of* the engine, which is a *part of* the car. Through semantic relationships such as “part of” or “value of”, the SPs and APs are ultimately all networked to each other. Similarly, the APs for a system alternative also need to be related to the network of SPs which provide the semantic context for the APs.

In addition to the semantic relationships, we use the term *dependency* to indicate that the value of a property depends mathematically on values of other properties. For example, a mathematical equation that expresses how an AP depends on an SP. As with synthesis and analysis properties, we make a distinction between *synthesis dependencies* and *analysis dependencies*. The term Synthesis Dependency (SD) is used to represent the choice made by the designer or the parametric function (in case SD is a computation). The output of a SD always leads to a SP (Figure 2). An example of a SD is the use of controller design heuristics in choosing *controller gains*. As a SD reflects only a heuristic, it is possible that while selecting a SP, a human overrides a value obtained from the heuristic. The choice could be based on a new experiment or observation, rendering the heuristic inapplicable. In this case, the choice is still considered rational and the chosen SP is considered consistent with the current SD. In order to show that a human can override a heuristic, a cross is shown on the representation of SD in Figure 2.

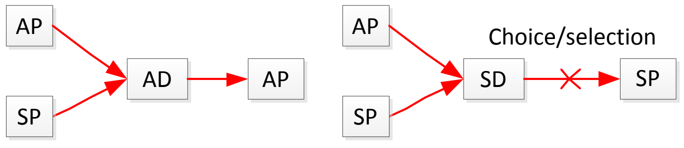


Figure 2. Analysis dependency (left), Synthesis dependency (right).

On the other hand, an analysis dependency (AD) represents the mathematical relationship between a set of APs and SPs to derive a new AP (Figure 2). The prediction of the resulting *rise time* due to chosen controller gains is an example of an AD. Another example of an AD is when a CAD tool provides the analysis that when performed on the modeled geometry, results in prediction of mechanical properties. However, as we have earlier pointed out, it is not necessary, nor always possible to formulate a mathematical relationship for an AD. The mathematical relations might be known at later design stage, and it might be good enough to formulate that an AD simply exists. A discussion in section 4.1 on level-of-detail in relation to dependency modeling elaborates this further.

2.1 Nature of properties, relationships and dependencies

Properties can be part of the same semantic context and they might not affect each other’s value (Figure 3), e.g. semantic relationship between a car, an engine, and a pair of seats. Expressing such relationships semantically is important in addition to representing dependencies where two properties depend upon each other’s value.

In relation to abstraction levels, fewer SPs and APs are identified during the initial design stages (higher abstraction level), and later - with more knowledge – additional SPs and APs are known (lower abstraction level). It is also possible that a property is uncertain initially and certain at the later design stages. Hence it is necessary to keep a version management over the property values, to allow for returning to a previous design stage if required. Model management aims to address this issue, discussed further in section 6.

The semantic relationship between properties is what creates a meaningful context. Modeling languages such as SysML [7] help in representing these semantic relationships efficiently. However, it has not been thoroughly investigated whether SysML provides adequate language constructs to model the dependencies explicitly. A further discussion about this is presented in section 4.

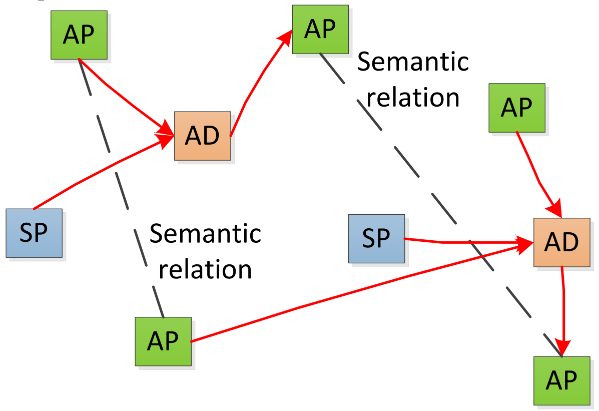


Figure 3. Semantic relationships between properties in addition to dependencies.

So far we have discussed properties, semantic relationships and dependencies from the perspective of a single domain. In model-based mechatronic design, design activities are spread across different domains such as mechanics, electronics, and software. The modeling activities performed within each domain lead to the creation of *domain-specific models*. Within a domain-specific model, there are dependencies (SDs or ADs) between SPs or APs. For instance, the analysis supported by a mechanical CAD tool constitutes an AD to predict the inertial properties of an object. At the same time, there are dependencies in between domain-specific models. For instance, an AP in one domain-specific model can become an AP in another model, or in other words, the two APs in different domain specific models are bound by an equality dependency. Figure 4 illustrates this situation where two domain-specific models: Model A and Model B are shown. Based on chosen SPs, Model A provides a prediction of an AP, which is bound to an AP in Model B (Figure 4). A similar situation can happen in case of SPs. Furthermore, the cross-domain dependencies could be numerous, leading to a multitude of properties being shared between two or more models. In this case, instead of a property, it is a *property file* (APF) containing a list of properties shared between two domain-specific models. The cross-domain dependencies exist between specific properties of domain-specific models, whereas other properties may not be affected by them. Figure 4 illustrates this situation where two portions of Model B are shown separately. One portion has dependencies with Model A (shared AP), and the other portion does not.

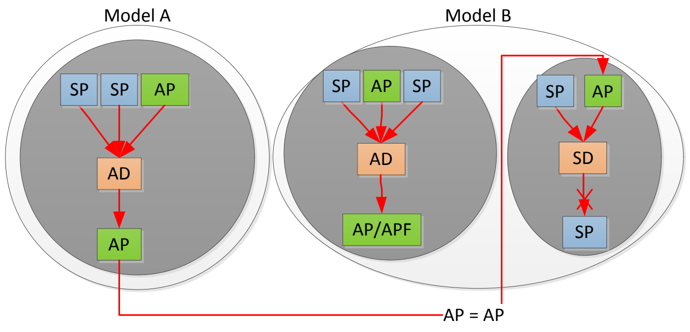


Figure 4. Visualization of a dependency network within and across two domain-specific models.

Both the SD and the AD could either have explicit formal representation through a mathematical function or an analytical model; however, this is usually not the case in current industrial practice. It is common that the dependencies are only implicitly known, e.g. in the mind of a designer.

Through a multitude of properties with dependencies in between them, a network of dependencies comes into existence. This network could be formally captured provided that the dependencies are modeled explicitly. Doing so will provide an option of propagating dependencies while moving in between abstraction levels. Section 4 further highlights the advantages of modeling the dependency networks.

3 ILLUSTRATION

This section aims to illustrate the dependency modeling concepts (presented in section 2) through a concrete design example. The design is based on the problem description, where it is required to pick and place an object in a three-dimensional environment with known obstacle locations. For the sake of simplicity, we have only considered a two-dimensional movement, ignoring the motion in the third dimension. A fairly simple two-degree-of-freedom robot is one of the alternative concepts selected to solve this problem.

The robot design process begins by synthesizing the *basic structures* [8], one of which is chosen as a quantified structure. Figure 5 shows the basic structure of a robot, the workspace environment, the pick and place point, and the known obstacle location. It is required to control the position of the robot with certain accuracy referred to as Controlled Position Accuracy (CPA), and avoid the obstacle. Some design variables that are considered part of the specification for the robot design are: link length (L_A, L_B), link width (W_A, W_B), material density (ϱ), range of joint motion (Θ_A, Θ_B), origin point (O), torque of motor (M_A, M_B), resolution of sensor (S_A, S_B), maximum distance between pick and place point (PE). We assume the ability to grab an object of any size, hence no gripper is considered for this example.

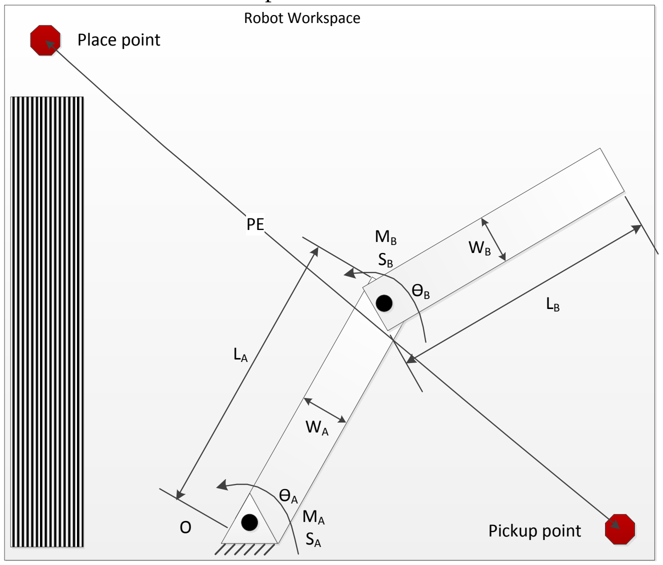


Figure 5. A design concept for the robot.

The robot design activities are partitioned into two domains: mechanical design and controller design. Mechanical design is performed to satisfy the workspace requirement and predict mechanical properties such as inertia. Using the mechanical properties of the robot, it is possible to perform a kinematic analysis that defines the end position (EP) of the robot. Knowing the characteristics of the robot design concept and the requirement CPA, controller gains (CG) will be selected.

Based on the design concept and specifications of the robot in Figure 5, it is now possible to represent the dependencies between robot properties through the concepts developed in section 2. The robot properties are differentiated as synthesis and analysis properties with synthesis and analysis dependencies in between them. Figure 6 shows the dependency network showing analysis and synthesis dependencies (in orange) between analysis properties (in green) and synthesis properties (in blue). The network spreads across the two domains- mechanical design, and controller design. As an example for SD, consider the SD_1 between WS and PE which leads to L_A and L_B. This shows that both link lengths are *chosen* based on workspace requirement (WS) and the distance PE. On the other hand, the analysis dependency (AD_1) between L_A, W_A, and ϱ represents an analysis performed through a mechanical CAD tool which *predicts* the inertia property I_A based on the chosen geometry (L_A, W_A) and material,

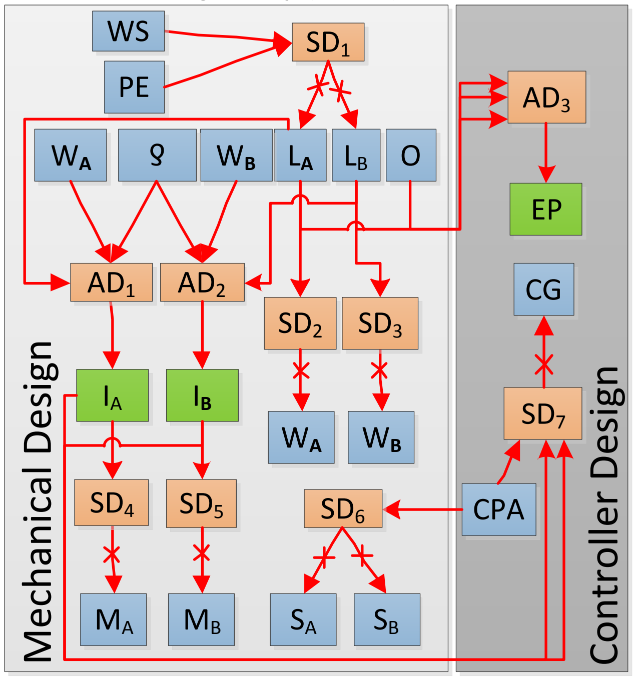


Figure 6. Dependency network for the robot example.

There could be a possibility for an existence of a loop in the dependency network, where a property could both be chosen and predicted. This situation is illustrated in Figure 7, where the property "controlled position accuracy" (CPA) is shown twice. One of the CPAs (in blue) is a SP which is constrained by the specification. The other CPA (in green) is an AP, which is predicted by selecting sensor resolution S_A and S_B (based on the selected CPA), and propagating the mechanical properties through a kinematic model (AD_4) to predict the end position EP (Figure 7). The CPA could then be predicted by finding the error between predicted position and the reference position (AD_5). Until the predicted CPA is within the bounds of specified CPA, this loop is repeated. Therefore, this process is similar to an optimization process in which some SPs are chosen to optimize the objective, CPA.

In certain situations, solving algebraic loops, such as the one shown in Figure 7, requires simultaneous solving of a system of equations. In such cases, the solution algorithm could break the loop using a technique called "tearing." A tearing algorithm can be employed, which performs iteration on reduced number of unknowns to find a solution [9]. Since SDs represent a choice made by the designer, it is most desirable to tear the loop at a SP. For instance, tearing the loop shown in Figure 7 after SD_6. The crosses shown between the SPs and the SD represent tearing or an override by the designer.

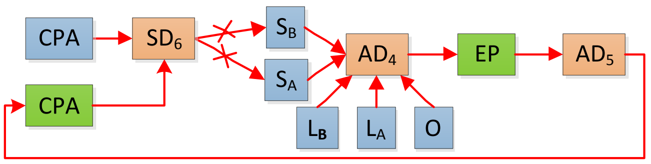


Figure 7. An illustration of a causal loop within the dependency network of the robot example.

It must be noted that illustration of the network shown in Figure 6 only serves the purpose of improving the understanding of the reader of this paper. In reality, such a network will most commonly be created through a modeling language such as SysML. Using SysML, it will also be possible relate the SPs and APs semantically, such as declaring L_A and W_A to be value properties of link A. Representing semantic relationships is important to gain sufficient understanding of the problem and will add further value to a dependency network as in Figure 6.

4 APPROACHES FOR CONSISTENCY MANAGEMENT

4.1 Modeling dependencies at different levels of detail

In the context of modeling, dependencies can be formally represented at different levels of detail. Which level of detail should be included in the dependency model depends on the context. For small design problems in which the number of tools, models, and stakeholders is relatively small, it may be too costly to maintain all the dependencies in great detail. Instead, one could rely on maintaining the dependencies manually. For larger efforts, manual updating becomes too labor-intensive and error-prone, so that detailed dependency modeling is most valuable. To guide the choice of which level of detail should be used, consider the following levels.

Level 0: Dependencies are not modeled explicitly; not all dependencies may be known, or they may only be captured implicitly in the mind of one of the stakeholders.

Level 1: The existence of dependencies is modeled, but without specifying what the dependencies are, for example, formally stating that properties of an analysis model depend upon a CAD model. Typically, PLM/PDM systems capture logical relationships between modeled properties at level 1, where they provide information about which model elements are related to each other. By expressing level 1 dependency, it is possible to maintain some traceability, where a human is reminded of which other objects to investigate/update in a change management scenario. It also aids in analysis of a dependency network, e.g., to determine the optimum workflow for the activities. For instance, based on the structure of the dependency network, one could decompose the network into parallel work flows for concurrent engineering. However, in many cases, it is not enough to know only that a dependency exists, since managing that dependency requires information about what is the actual dependency (level 2).

Level 2: Dependencies are formally captured through a model showing both the existence of the dependency and what the dependencies are. Modeling correspondence relationships between a system model and an analysis model is an example of modeling dependencies explicitly. The analysis model could be executed to find the property values, and automatically update these values between system model and analysis model, resulting in fewer inconsistencies. The challenge is that models exist in different tools, i.e., there is a tool-integration problem.

Level 3: The applicability of dependencies is modeled. A model is only valid under certain conditions. A cantilever beam bends under a load, and the bending can be explained through a linear model assuming a small load. A larger load causes higher bending which is better explained through a non-linear model, and the linear model is no longer valid. Therefore, apart from modeling dependencies, it is also required to mention when the dependency is applicable. The applicability conditions could be described through constraints, meta-data, or validation data set. Modeling dependencies at this detail provides an added advantage of alerting the human (user) if the applicability conditions are violated.

Level 4: Modeling what is the dependency pattern. An important question that comes to mind while considering dependency models is: what happens when structure of the model changes? For example, adding a third link to the robot. In this case, the structure of the dependency network changes. For the robot case, it is possible to deduce the change in dependency network by knowing what object were created for the previous two links, called as a *dependency pattern*. An example pattern is to define an inertia property (I_C) for the third link (L_C) automatically, and creating a synthesis dependency between I_C and M_C. Similar dependency patterns exist across different tools e.g. between system hierarchy and CAD assemblies, between system structure and analysis model structure. Again, tool interoperability is a challenge here which can be addressed through model transformations. The transformation also plays a key role in automating the generation of dependency patterns, in providing consistency checking, and in propagating dependencies in a change management scenario.

Level 5: The applicability of a dependency pattern is modeled, e.g., modeling the applicability condition for a model transformation supporting a dependency pattern.

In order to avoid as many inconsistencies as possible, it is important to maintain level-5 dependency management as the design process proceeds. However, as we have stated earlier, choosing which level of detail should be included in the dependency model depends upon the nature of design problems and organizations. Never the less, the aim is to avoid the situation where modeling becomes *too expensive*. Although having some level of dependency management is imperative, further research is needed to find methods for reducing the cost. There is a strong need for a capability in the form of a tool that aids in management of dependencies. Such a tool should support modeling dependencies between disparate models, apply model transformations (using the dependency patterns), aid in evaluating consistency checks, and support in managing product variants.

4.2 Modeling dependencies in SysML

A key question to modeling dependencies is whether there should be one language where the all the dependencies are captured or whether the dependencies should be represented in a distributed fashion across several modeling languages. In relation to the six levels-of-detail for dependency management, using a single language will be restrictive in the sense that only a certain level-of-detail would be possible to manage. Handling broader levels-of-detail will require the use of different modeling languages. This is also clear from Figure 4, provided that each language contains suitable language constructs to create the required dependency models.

For the purpose of defining vocabularies and their relationships, catering to the processing of information by computers, the Web Ontology Language (OWL) [10] has been proposed. In Computer Aided Design (CAD), ISO 10303 commonly known as STEP standard [11] aims to describe the product data throughout its life cycle, for sharing of information between different CAD systems in a neutral file format. Together, OWL and STEP can be used to address the model interoperability issues.

In model-based design, several dependencies may appear in-between disparate models. Therefore, the required language to model the dependencies should support modeling across different languages and tools (schema). The dependency model can be created in generic languages such as OWL, STEP, or UML [12]. Reference [13] explains how to use a generic modeling language as UML to support multi-domain modeling and data-integration. In order to support modeling across different languages, it is required that one can access and refer to model elements defined in disparate languages. For this reason, languages such as UML and SysML are most suitable. Both of these languages are also supported in the Eclipse Modeling Framework (EMF) [14], hence providing an effective combination to capture the dependencies.

SysML is a language for formally capturing the system specifications and the design concepts (meeting the specifications) in a descriptive fashion and could therefore serve as a modeling language to create dependency models. The advantage of using SysML is the capability to represent the semantic relationships between properties. Figure 8 shows a SysML view of the robot mechanical assembly, describing the semantic relationships between a value property *Link Length* and the block *Arm*.

Creating a dependency network as in Figure 6 in SysML is currently best represented using a parametric diagram. In this case, synthesis and analysis dependencies are represented through constraints. However, SysML does not provide sufficiently rich language constructs to model choice or selection for a synthesis property, nor does it support differentiating between synthesis dependencies and analysis dependencies, and between synthesis properties and analysis properties. Such differentiations play a key role in understanding the effect of dependencies in a change management scenario, and making it clearly visible to the modeler which properties are chosen and which ones are predicted.

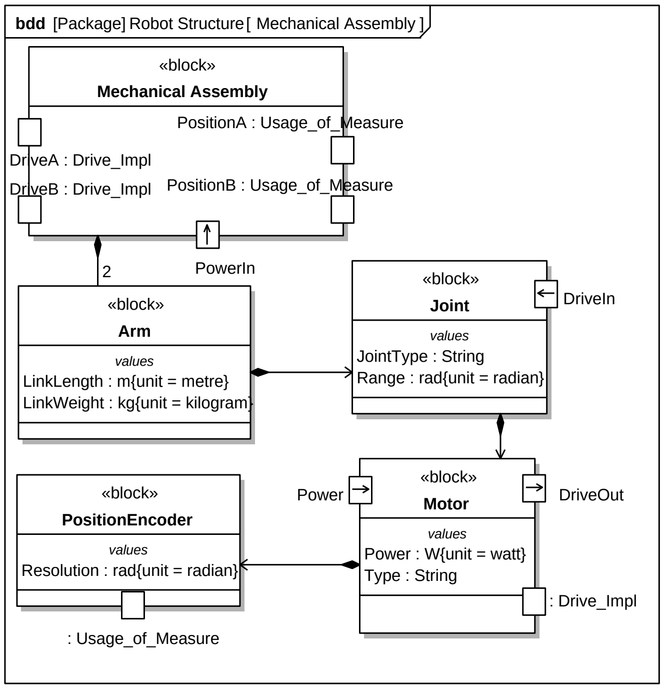


Figure 8. Robot mechanical structure modeled in SysML.

The reason that SysML is not sufficiently expressive is due to the current practice where it is not required to model dependencies explicitly. Provided that the future SysML revisions provide such semantics, it will lead towards a creation of a common product model containing a dependency network, along with individual dependency networks existing among domain-specific models (Figure 9). The advantage of such a configuration is again to represent semantic relationships between properties alongside the dependencies between them.

With the dependency patterns (level 4) discussed earlier, it will be possible to model dependencies between properties of domain-specific models by utilizing patterns, such as between the structure hierarchy (in the system model) and the corresponding CAD assembly models. Figure 10 shows an actual example pattern where the robot CAD assembly is transformed to provide a system hierarchy in SysML (Figure 10, pattern A). The other pattern (pattern B) is a conceptual representation of a pattern between system design and controller design for the robot example in Figure 6. By utilizing such patterns, it will be possible to model APs (and SPs) which become APs (and SPs) in another model. For example: *Link1_length* to L_A, *Origin* to *O* (Figure 10). Many of such patterns are recurring, hence providing benefits of pattern reuse, and reducing the human effort. Model transformations are essential for the application of such dependency patterns, and modeling of dependencies.

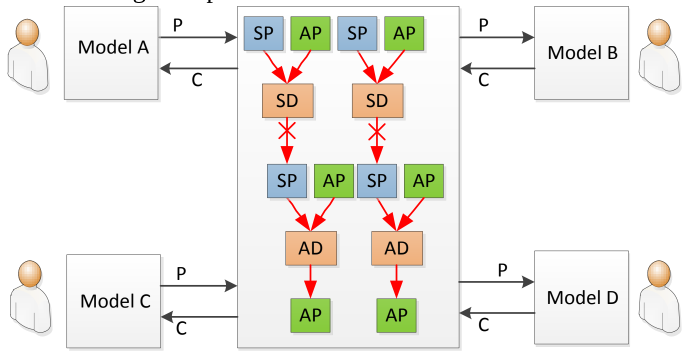


Figure 9. Topological configuration with SysML as a common product model among domain-specific models. *P* stands for a Producer and C stands for a Consumer of a property.

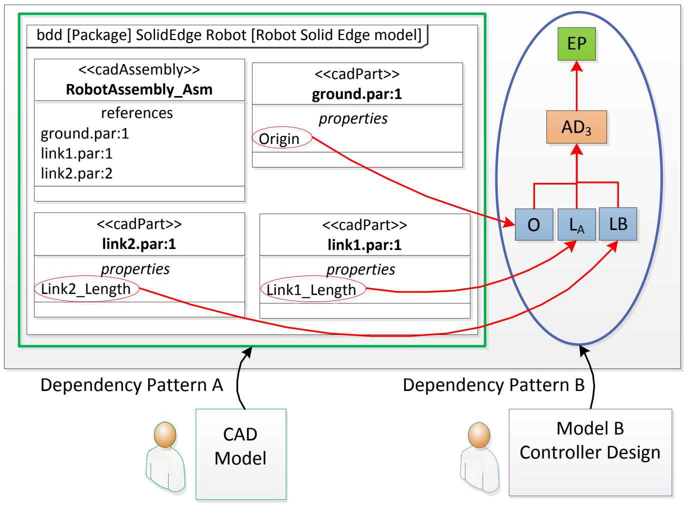


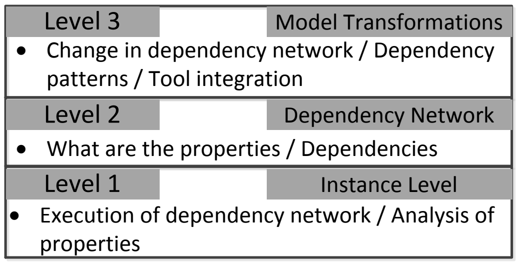
Figure 10. A visualization of dependency patterns. Pattern-A: CAD-SysML. Pattern B: Controller Design-SysML. Properties shared between two domain-specific models are also shown.

4.3 Executing dependencies

After building the dependency network, the next step is to execute it to find the property values. As discussed in section 4.1, it is required to verify that all the dependency patterns and dependencies are still valid. Based on validation, the network execution requires that each individual dependency to be executed to perform property analysis. For the robot example, this means that the CAD model and the kinematic model to be executed after a selection of relevant synthesis properties have been made. It has to be determined which dependencies to be executed before the others. It is possible that values to some of the properties or certain dependencies are not known at the time of execution. In this case, assumptions can be made about the valid range of property values without waiting for a particular dependency to be executed (providing these values). The steps for executing each dependency in the network have to be determined, requiring a supervisor that takes control of this process.

Among the set of known dependencies between domain-specific models, it is not sufficient to model dependencies only between two domain-specific models (leaving out the rest). The global effect of all the dependencies is a key element for understanding the influence of various properties and dependencies (after a network execution). Therefore completeness and validity of the network are important in order to gain value form it.

The information gained from executing the network might also lead towards suggestions for change in network (both dependency structure and properties). Model transformations should support in automation of such changes. Therefore, the creation of dependency network and its execution can be seen as a three-layered architecture. Figure 11 shows the three-layered architecture with dependencies modeled at level 2, model transformations at level 3, and execution of dependencies at level 1.



Level 3	Model Transformations
• Change in dependency network / Dependency patterns / Tool integration	
Level 2	**Dependency Network**
• What are the properties / Dependencies	
Level 1	**Instance Level**
• Execution of dependency network / Analysis of properties	

Figure 11. Meta-levels in relation to modeling/executing dependencies.

In terms of propagating the dependencies in a change management scenario (e.g., change in specifications), the structure of dependency network needs to be updated by the stakeholder having the authoring control over relevant dependencies. If the dependencies are still valid, an execution of dependencies provides new property values. It must be remembered that when the value of property changes, consistency can only be checked if the dependencies remain the same.

4.4 Human effort

A few important questions were asked in section 1 about how humans will utilize the dependency network, and how to maximize the value gained compared to the effort required to build and manage dependency models. Since the dependencies are spread all across different models built by various stakeholders, building the dependency model requires efforts from each stakeholder. Dependency patterns are especially helpful in reducing human effort towards the creation of dependency models; however, these patterns themselves also have to be created. On top of that, each stakeholder needs to validate and execute the dependencies under his or her control in order to gain value from the network. There can be situations demanding change in dependency structure, e.g., adding a third link to the robot. In such cases, dependency patterns will support automation of propagating the change to the dependency network. However, without a dependency pattern, the change has to be done manually by each stakeholder who is affected by it.

The role of a human decision maker within the network is inevitable as there are decision nodes (SDs) requiring a selection or decision by a human. A situation can occur where a SP is determined by multiple SDs, i.e., by making multiple selections (Figure 12). For instance, the geometry for link A of the robot is initially chosen while performing CAD modeling, and later a different geometry is chosen after the FEM analysis has been performed. In this case, a different value of the same synthesis property (L_A or W_A) is selected between initial and detailed analysis, representing multiple selections.

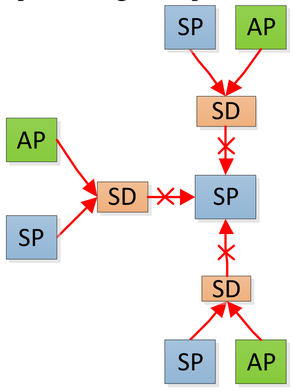


Figure 12. Multiple SDs leading to a single SP, requiring multiple selections.

In order for a human to gain value from the network, network representation should be simple and precise for better understandability, which is a challenging problem due to the large number of properties and dependencies typically part of the dependency network. Views for exposing only the relevant dependencies are vital for reducing the complexity. For instance, when having alternative concepts, only the dependencies related to the current alternative should be exposed. The steps for executing the dependency network are also decided and managed by humans. Network analysis techniques could be used to streamline the execution of dependencies, however the final decision is made by a human. The important thing to remember is to make design decisions effectively and efficiently. Hence the focus should always be on the value gained as compared to effort. Finding methods that help reduce human effort requires further research.

4.5 Consistency checking

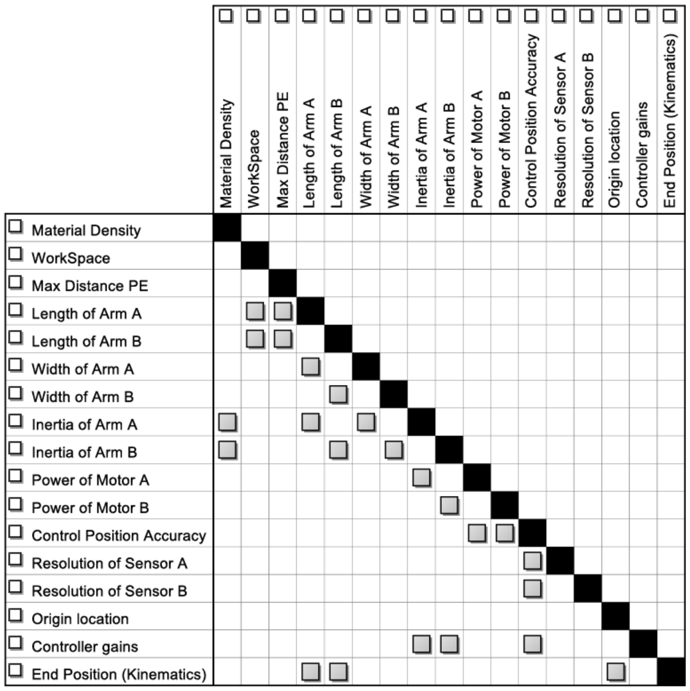
Consistency can never be fully ensured [2]. The best that can be done is to avoid as many inconsistencies as possible. As the level-of-detail of the dependency models increases, more inconsistencies can be avoided. Consistency also requires that each stake holder has built his or her model conforming to the rules of the language, conforming to the laws of nature, and according to rational preferences [2]. The same applies to modeling dependencies. The validity of a dependency and a dependency pattern must always be checked in order to be consistent (internal consistency). As the design process moves on, the structure of the dependency network and the property values change. Ensuring consistency through a dependency network means that the results obtained are in accordance with the current knowledge of dependencies. The network should support in highlighting inconsistencies in situations where a value of a property changes, through the addition or deletion of model elements, or during a change in specifications. There can be situations where there are multiple producers for the same SP or an AP. Human intervention is required in such cases in order to ensure consistency.

5 STATE OF THE ART

In the area of mechatronic design, many research efforts have focused on the development of methods and tools to support the design activity. These efforts include research on managing the complexity of mechatronics systems [15] and the communication between product development groups [16]. Braun and Lindemann [17] described how to consider cross-domain product and process interfaces during cost estimation of mechatronic products. Buremester et al. [18] proposed the Hybrid UML modeling language so that both the structure and the hybrid dynamic behavior of mechatronic design concepts can be investigated together. A similar approach was presented by Cao et al. [19] extending SysML to model and analyze hybrid dynamic behavior of mechatronic systems, in order to aid in the decision making process. Gausemeier et al. [20] proposed a language called Semi Formal Specification Language to specify a mechatronic system from different domain viewpoints, supported by analyses and simulations within each domain. To the best of knowledge of the authors, the issue of dependency modeling has not been discussed in the current state of the art in mechatronic design.

For the topic of consistency management, most of the work originates from the area of software engineering. Egyed [21] described methods for detection and tracking of inconsistencies in software design. Mens et al. [4] described how to avoid inconsistencies using dependency analysis, and also how to manage consistency of UML models [3]. These research efforts are mostly based on rule-based consistency checking methods. Here, the changes in a model are observed, and corresponding to a particular change, a consistency rule is evaluated. The rules are pre-defined, based on the knowledge of consistent patterns. For systems with physical nature, Adourian and Vangheluwe [22] discussed how to maintain consistency between a geometric model and a dynamic analysis model. Gausemeier et al. [23] also studied consistency checking between a system model and domain-specific models based on Triple Graph Grammars (TGG). Hehenberger et al. [24] discussed an approach to detect changes in a model to trigger a consistency checker, which then evaluates a relevant consistency rule for the performed changes.

Among the state of the art in dependency modeling approaches, the Design Structure Matrix (DSM) has gained wide popularity with many documented applications such as in product development [25] and software design [26]. In order to highlight the differences between the dependency modeling approach presented in this paper and the DSM, a DSM is constructed for the robot example (presented in section 3). Figure 13 highlights the dependencies between properties of the robot represented through *parameter-based DSM*.



	Material Density	WorkSpace	Max Distance PE	Length of Arm A	Length of Arm B	Width of Arm A	Width of Arm B	Inertia of Arm A	Inertia of Arm B	Power of Motor A	Power of Motor B	Control Position Accuracy	Resolution of Sensor A	Resolution of Sensor B	Origin location	Controller gains	End Position (Kinematics)
Material Density	■																
WorkSpace		■															
Max Distance PE			■														
Length of Arm A		▢	▢	■													
Length of Arm B		▢	▢		■												
Width of Arm A				▢		■											
Width of Arm B					▢		■										
Inertia of Arm A	▢			▢		▢		■									
Inertia of Arm B	▢				▢		▢		■								
Power of Motor A								▢		■							
Power of Motor B									▢		■						
Control Position Accuracy										▢	▢	■					
Resolution of Sensor A												▢	■				
Resolution of Sensor B												▢		■			
Origin location															■		
Controller gains								▢	▢			▢				■	
End Position (Kinematics)				▢	▢										▢		■

Figure 13. DSM for the robot example built in CAM [27].

Each box inside a cell of a DSM (in Figure 13) indicates a dependency between the corresponding properties. The left diagonal of the DSM is highlighted. All the dependencies for the robot example are shown below the diagonal, meaning that the dependencies can be solved for to find a solution. To solve for mathematical dependencies represented in a DSM, a mathematical solver is needed. However, any dependency marked above the diagonal represents a dependency loop, where solving requires values of properties that are still to be determined. In such cases, reorganization of the DSM is needed in order to avoid loops and achieve a lower-triangular DSM, e.g. through clustering, tearing or sequencing algorithms. If the loops cannot be avoided, then mathematical root finding is required. The DSM in Figure 13 represents the same dependencies as shown in Figure 6. The loop shown in Figure 7 is not represented in the DSM; hence no dependencies appear above the diagonal.

It is clear that a DSM does not differentiate between synthesis and analysis nature of properties and dependencies. Not having this differentiation may lead to confusion among stakeholders where a property could both be a synthesis property and an analysis property (as shown in Figure 7). Moreover, considering a small change in specification e.g. workspace (WS) of the robot, it is possible to deduce from Figure 6 that since there is a synthesis dependency with WS, a small change in workspace does not require propagating this change through all the dependencies and update the analysis properties. This is due to the fact that a small change in WS may still be catered for by the current value of L_A and L_B (selected). On the other hand, deducing such a result from a DSM is not possible since no distinction is made between properties, and a small change in workspace will require corresponding changes in L_A and L_B, along with propagating all the dependencies to make sure the new properties conform to specifications. Furthermore, a DSM does not show semantic relationships between properties which play an important role in understanding the design problem and the nature of dependencies. As stated earlier, the semantic relationship can be made part of the dependency network modeled in SysML.

Among other dependency modeling approaches, Process Integration Design Optimization (PIDO) approaches are commonly used. ModelCenter [28], modeFRONTIER [29], Isight [30], Comet Workbench [31] are well known tools in this area. These tools are helpful in integrating disparate models into a workflow process and perform design optimization and trade-off studies. Therefore, they can be considered as potential candidates for modeling and executing the dependency network. For instance, it is possible to model the dependency network of Figure 6 in ModelCenter, keeping the differentiation between synthesis and analysis properties and dependencies. However, ModelCenter only shows the dependencies between properties, and not the semantic relationships between them. Furthermore, as the dependency network grows, the view in ModelCenter becomes too cluttered, and the issue of human understanding comes into play.

6 DISCUSSION

In terms of implementation, dependency modeling and model management are correlated areas. For some time now, PDM and PLM systems have been employed for model management. Without an explicit model of dependencies, PDM systems only provide information about logical relations between model elements, e.g., maintaining correspondence between SysML Robot and CAD Robot (level 1 dependency management). The influence of a network of properties is not addressed by PDM/PLM systems. However, as we have presented in this paper, modeling and managing both the properties and the dependencies is important for model management.

The efforts in the SysML community so far have been towards integrating SysML within the model based development process, leading towards integration of domain-specific models with SysML. This has led towards a hub-spoke topological structure with SysML as the logical hub and domain-specific models as spokes. It is important to understand that SysML's purpose is not to provide model management capabilities, these capabilities should rather be a part of the PDM/PLM system. Therefore, in terms of implementation, the PDM/PLM system would be a hub for all other models (including SysML) as shown in Figure 14. There are dependencies within and across domain-specific models, and these dependencies can be modeled in SysML leading towards a dependency network (Figure 14). The dependency models discussed in this paper and the corresponding model management through PDM/PLM should provide a sound foundation for consistency management leading towards effective decision making. In order to address the tool integration problem, a tool integration framework enabling information transfer between design and analysis tools to perform model transformations is essential. The iFEST project [32] aims for the provision of such a framework.

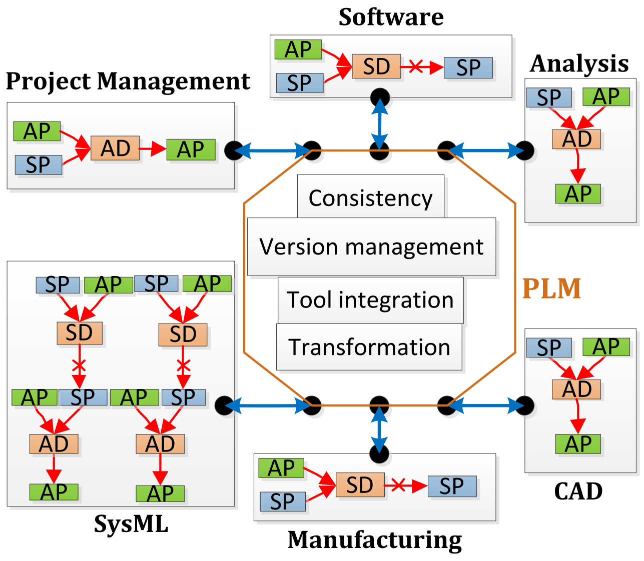


Figure 14. PLM as a core for model management, dependency network represented in SysML with other dependencies spread cross domains. Figure adapted from [33].

7 CONCLUSION

In this paper, an approach for modeling dependencies is presented, based on the hypothesis that modeling dependencies explicitly adds value by increasing the efficiency and effectiveness of the decision making process in engineering design. We establish the relationship between properties and decision making, and build fundamental concepts for modeling dependencies. A fundamental distinction is made between properties that are chosen and the ones that are predicted. We present semantics for modeling properties and dependencies. Provided that SysML will support the necessary language constructs, it is possible to model and manage the dependency networks through a SysML model. The dependency network for the robot example provides an illustration of how to construct the network, and how the network supports decision making. A direct comparison to the DSM is provided in order to highlight the vital differences and the advantages of differentiating between synthesis and analysis nature of properties and dependencies. Six levels of detail for modeling dependencies are discussed, and it is argued that avoiding inconsistencies requires dependency modeling and management at all these levels. However, a user should verify whether the cost of implementation (for avoiding inconsistencies) is small compared to the expected benefits. Although the role of a human decision maker in dependency modeling and management is inevitable, a considerable effort may be required in order to build and manage dependency models. Dependency patterns and model transformations should aid in reducing the human effort, however further efforts are required to improve the human interaction with the network. Therefore, at the present moment, the hypothesis that modeling dependencies explicitly adds value is not verified. Never the less, dependency modeling at one of the levels (1-5) is necessary in order to manage consistency problems, and improve the design process workflow. By utilizing network analysis techniques, the resulting dependency network can support streamlining the design activities, which is beneficial in approaches such as concurrent engineering. Future work aims to incorporate the developed semantics in SysML, and perform case studies to analyze the usability of the approach. The role of PDM/PLM tools in relation to dependency networks is also a future research area.

ACKNOWLEDGMENTS

We wish to thank Axel Reichwein and Sebastian Herzig from MBSE center for valuable discussions during the development of dependency concepts. The lead author is thankful to the Signeuls Foundation and KTH-Royal Institute of Technology for providing travel grant to carry out this research.

REFERENCES

1. Sanford Friedenthal, Alan Moore, and Rick Steiner, *A practical guide to SysML, the systems modelling language*. 2008: Morgan Kaufmann. ISBN 978-0-12-374379-4
2. Sebastian J. I. Herzig, Ahsan Qamar, Axel Reichwein, and Christiaan J. J. Paredis. *A conceptual framework for consistency management in model-based systems engineering*. in 2011 ASME International Design Engineering Technical Conferences & Computers and Information in Engineering Conference IDETC/CIE 2011. 2011.
3. Tom Mens, Ragnhild Van Der Straeten, and Jocelyn Simmonds, *A framework for managing consistency in evolving UML models.* Software Evolution with UML and XML, 2005. Vol.: p. 1-31.
4. Tom Mens, Ragnhild Van Der Straeten, and Maja D'Hondt, *Detecting and resolving model inconsistencies using transformation dependency analysis.* Lecture Notes in Computer Science, 2006. Vol. 4199/2006: p. 200-214.
5. George A. Hazelrigg, *A framework for decision based engineering design.* Journal of Mechanical Design, 1998. Vol. 120: p. 653-658.
6. Stephanie C. Thompson, *Rational design theory: a decision-based foundation for studying design methods*, PhD. Thesis, GW. Woodruff School of Mechanical Engieering, Georgia Institute of Technology, Atlanta, Georgia, USA, 2011, Available from: http://smartech.gatech.edu/jspui/bitstream/1853/39490/1/thompson_stephanie_c_201105_phd.pdf
7. Object Management Group. *OMG system modeling language specification V1.2*. 2010; Available from: http://www.omg.org/spec/SysML/1.2/PDF/.
8. Eskild Tjalve, *Systematic design of industrial products*. 2003: Institute of Product Development, The Technical University of Denmark.
9. Hilding Elmqvist and Martin Otter. *Methods for tearing systems of equations in object-oriented modeling*. in European Simulation Multiconference. 1994.
10. W3C OWL Working Group. *OWL web ontology language overview*. 2002; Available from: http://www.w3.org/TR/2004/REC-owl-features-20040210/#s1.2.
11. Michael J. Pratt, *Introduction to ISO 10303 - the STEP standard for product data exchange.* Journal of Computing and Information Science in Engineering, 2001. Vol. 1(1).
12. Object Management Group. *OMG Unified Modeling Language (UML) specification V2.4.1*. 2011; Available from: http://www.omg.org/spec/UML/2.4.1/.
13. Axel Reichwein, *Application-specific UML profiles for multidisciplinary product data integration*, PhD. thesis, Univeristy of Stuttgart, Germany, 2011, Available from: http://elib.uni-stuttgart.de/opus/volltexte/2012/6949/pdf/Axel_Reichwein_PhD_Thesis_2011.pdf
14. Eclipse Foundation. *Eclipse Modeling Framework (EMF)*. 2011; Available from: http://www.eclipse.org/modeling/emf/.
15. Tetsuo Tomiyama, Valentina D'Amelio, Jill Urbanic, and Waguih ElMaraghy, *Complexity of multi-disciplinary design.* Annals of the CIRP, 2007. Vol. 56(1): p. 185-188.
16. Niklas Adamsson. *Model-based development of mechatronic systems - reducing the gap between competencies?* in Tools and Methods of Competitive Engineering. 2004.
17. Stefanie C. Braun and Udo Lindemann. *A multilayer approach for early cost estimation of mechatronical products*. in International Conference on Engineering Design (ICED07). 2007.
18. Sven Buremester, Holger Giese, and Oliver Oberschelp, *Hybrid UML components for the design of complex self-optimizing mechatronic systems.* Information in Control, Automation and Robotics I, 2006. Vol. 4: p. 281-288.
19. Yue Cao, Yusheng Liu, and Christiaan J. J. Paredis, *System-level integration of design and simulation for mechatronic systems based on SysML.* Mechatronics, 2011. Vol. 21(6): p. 1063-1075.
20. Jurgen Gausemeier, Ursula Frank, Jorg Donoth, and Sascha Kahl, *Specification technique for the description of self-optimizing mechatronic systems.* Research in Engineering Design, 2009. Vol. 20: p. 201–223.
21. Alexander Egyed, *Automatically detecting and tracking inconsistencies in software design models.* IEEE Transactions on Software Engineering, 2011. Vol. 37(2): p. 188-204.

22. Chahe Adourian and Hans Vangheluwe. *Consistency between geometric and dynamic views of a mechanical system*. in Proceedings of the 2007 Summer Computer Simulation Conference. 2007. San Diego, CA, USA.
23. Jurgen Gausemeier, Wilhelm Schafer, Joel Greenyer, Sascha Kahl, Sebastain Pook, et al. *Management of cross domain model consistency during the development of advanced mechatronic systems*. in International Conference on Engineering Design (ICED09). 2009.
24. Peter Hehenberger, Alexander Egyed, and Klaus Zeman. *Consistency checking of mechatronic design models*. in Proceedings of the 2010 ASME International Design Engineering Technical Conferences & Computers and Information in Engineering Conference IDETC/CIE 2010. 2010. Montreal, Quebec, Canada.
25. Mike Danilovic and Tyson R. Browning, *Managing complex product development projects with design structure matrices and domain mapping matrices.* International Journal of Project Management, 2007. Vol. 25(3): p. 300-314.
26. Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. *Using dependency models to manage complex software architecture*. in Object Oriented Programming, Systems, Languages & Applications (OOPSLA). 2005. ACM Press.
27. David C Wynn, Seena M. T. Nair, and P. John Clarkson. *The P3 platform: an approach and software for developing diagrammatic model-based methods in design research*. in International Conference on Engineering Design (ICED09). 2009. Stanford, USA.
28. Phoenix Integration. *ModelCenter*. 2012; Available from: http://www.phoenix-int.com/software/phx_modelcenter_10.php.
29. ESTECO. *modeFRONTIER*. 2012; Available from: http://www.esteco.com/home/mode_frontier/mode_frontier.html.
30. Dassault Systems. *Isight*. Available from: http://www.simulia.com/products/isight2.html.
31. Comet Solutions. *Comet Workspace*. 2012; Available from: http://cometsolutions.com/products/workspace/.
32. Martin Törngren, Hans Petter Dahle, Dagfin Brodtkorb, Jad El-khoury, Ray Chaplin, et al. *Towards an industrial framework for emebdded systems tools*. in HOPES Workshop at ECMFA. 2010.
33. Harald Eisenmann and Joachim Fuchs. *Mutlidisciplinary approach for industrial phases in space projects*. in MBSE Workshop at Incose IW2012. 2012.