A Domain-Specific Language for Dependency Management in Model-Based Systems Engineering

Ahsan Qamar[1], Sebastian Herzig[2], and Christiaan J. J. Paredis[2]

[1] KTH-Royal Institute of Technology, Stockholm, Sweden
{ahsanq}@kth.se
[2] Georgia Institute of Technology, Atlanta, Georgia, USA
{sebastian.herzig,chris.paredis}@me.gatech.edu

Abstract. The varying stakeholder concerns in product development today introduces a number of design challenges. From the perspective of Model-Based Systems Engineering (MBSE), a particular challenge is that multiple views established to address the stakeholder concerns are overlapping with many dependencies in between. The important question is how to adequately manage such dependencies. The primary hypothesis of this paper is that modeling dependencies explicitly adds value to the design process and in addition supports consistency management. We propose a domain-specific language called as the Dependency Modeling Language (DML) to capture the dependencies between multiple views at the appropriate level of abstraction, and utilize this knowledge to support a dependency management process. The approach is illustrated through a dependency model between three views of a robot design example. In addition, we discuss how to analyze dependency graphs for consistency checking, change management, traceability and workflow management.

Keywords: Dependency Modeling Language, Domain Specific Modeling Language, Model Based Systems Engineering, Consistency Management, Change Management.

1 Introduction

Contemporary product development is a complex process. Primarily, this is the case due to a large number of stakeholders being involved, all of which have varying and overlapping concerns. Therefore, adequate methods to manage the consequential conflicts are required. In this sense, the design of mechatronics is a particularly interesting case, since stakeholders from a very diverse set of disciplines are involved. This makes good decision making very challenging. To support a model-based mechatronic design process, different viewpoints are defined, each supported by one or more modeling views. Naturally, multiple viewpoints are supported through multiple modeling languages, where the overlapping stakeholder concerns lead to *dependencies* between the established views. Traditionally, the dependencies are managed in an ad-hoc fashion by mainly relying on the communication between the stakeholders. However, ad-hoc dependency management can prove to be ineffective, especially for complex and large scale systems where there could be a large number of such dependencies.

In earlier work, consistency management was explored in the context of engineering design, and a classification of several distinct type of inconsistencies was identified [1]. It was concluded that no consistency check can ever be complete and that only some inconsistencies can be identified, that too only in the information captured explicitly and formally. Dependencies are interesting because they could be the cause of the arising inconsistency, and hence explicit knowledge of dependencies is vital for consistency management. However, this is not a trivial task since adequate support in terms of a modeling language and a supporting tool for dependency management is currently lacking. In this paper, we present a modeling language to help build a model of dependencies.

The fundamental question to answer is whether it is valuable to model dependencies in contrast to current approaches where dependencies are not captured formally. The work reported in this paper builds on the hypothesis that modeling dependencies adds value to the design process. The value can be measured in terms of support for consistency management, change management, ensuring traceability and managing the design process workflow, each of which can be supported by the dependency modeling approach presented in this paper.

The remainder of this paper is organized as follows: Section 2 builds a notion of dependency. An example use case is described in Section 3. Section 4 introduces a Domain-Specific Modeling Language (DSML) for capturing dependencies, which is illustrated through the example use case in Section 5 and Section 6. Section 7 presents the related work and is followed by a discussion in Section 8. Section 9 presents conclusions and possible future work.

2 Notion of Dependency

Rational Design Theory (RDT) [2] establishes a theoretical foundation for *Artifacts*, *Properties*, *Concept Selection* and *Concept Evaluation*. Based on RDT and on Hazelrigg's decision-based design framework [3], we argue that two types of properties are prevalent in design: one is used to describe constraints (specification), whereas the other is used to communicate the designer's belief regarding the value of the property (a prediction based on a given specification). We call specification properties *Synthesis Properties* (SP) and prediction properties *Analysis Properties* (AP).

To describe an artifact, there could potentially be an infinite number of properties spread across multiple views. In this paper, the term *dependency* refers to a type of model capturing the relationship between the values of input and output properties (regardless of the view they belong to). This is somewhat different to how dependencies are defined in UML [4], where they represent a relation and express the need for a particular element to exist. For example, if an element A depends on element B, and B no longer exists, A is no longer specifiable. In our case, removing a dependency does not invalidate the inputs or the outputs - the dependency is just no longer captured. Causality naturally arises from the fact that a dependency has well defined inputs and outputs.

A dependency is considered to be a model; it is possible that this model is unknown at a given design stage; in this case a dependency can still be specified (along with its input and output properties) in a dependency model. Once the model that specifies a particular dependency is known, references can be created between this model and the dependency model. A natural question to ask is why not utilize currently available languages to model dependencies. In the work reported in [5], different modeling languages were analyzed for dependency modeling (e.g. OMG SysML$^{\text{TM}}$[6]). However none were found to be suitable for capturing dependencies adequately without having to modify them (e.g., profile extension of SysML). In addition, the size of meta-model extensions (when using a general purpose language for many different purposes) adds to the intrinsic complexity of the underlying system [7] and introduces *accidental complexity* [8].

In contrast to a general purpose modeling language (such as SysML), a DSML is restrictive and has a specific purpose, in particular as per the demands of a specific viewpoint [7], and it captures the object of interest at the appropriate level of abstraction and formalism to help minimize the complexity [9]. Based on this motivation, we will - in the following section - introduce a DSML to model dependencies called the *Dependency Modeling Language* (DML).

3 Example Use Case

In order to illustrate the proposals of this paper to the reader, an example use case is considered: a simple two degree of freedom robot. The design problem is formulated as follows: *Design a pick and place robot with Work Space (WS) coverage of 4m^2, with Close loop Position Accuracy (CPA) of at least 5mm, and with the End-to-End Response Time (EERT) of the robot should not be more than 0.5 seconds.* Three viewpoints (one for each stakeholder) are considered for this example: mechanical design, control design, and Hardware/Software (Hw/Sw) design. The three stakeholders - based on the design specifications for each viewpoint - develop disparate models focusing on different aspects of the robot by utilizing different design and analysis tools, such as a CAD tool for mechanical design, a control design tool and a software design tool. The semantic overlaps between the views results in dependencies, which will be the focus of the illustration in section 5. It is worthwhile to mention that gaining knowledge about properties CPA and EERT requires the combined work of the three stakeholders, making it essential to manage dependencies.

4 A Modeling Language for Capturing Dependencies

This section describes the DML which is currently supported in Eclipse Modeling Framework (EMF) [10]. Figure 1 illustrates the abstract syntax of the DML using a class diagram. Any model that conforms to this meta-model is referred to as a *Dependency Model*. In the following, the semantics of the language constructs of the DML are discussed.

A **Concept** is a description of an artifact and can refer to the actual product to be developed, or to any of its sub-components. We say that concepts are formed by constraining some of the properties associated with it. With the passage of time, more constraints are put on properties, hence leading to further refined concepts. For instance, by constraining the number of arms of a robot to two, a two arm robot *Concept* is created. A concept *Contains* zero to many subordinate concepts - by way of example, here are a few that can be considered for the two arm robot: Arm1, Arm2, Controller, Motor1, Motor 2, Sensor1, and Sensor2. All these Concepts are contained under the main concept *two arm robot*. Concepts are related to each other through an *isPartOf* relationship, which creates the semantic context around each concept. Each *Concept* can be looked at from many *Viewpoints*, and is characterized by a number of *Properties*, which are captured in a *Model*.

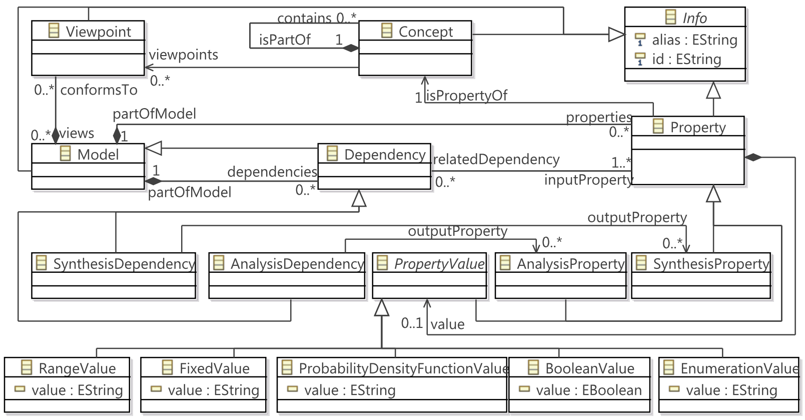

Fig. 1. Abstract syntax (meta-model) of the DML.

A **Viewpoint** refers to the guidelines and conventions used to establish a *View*, where a *View* corresponds to a *Model* or a composition of disparate models: for example, mechanical design viewpoint encompassing a Solid Edge model or the dynamic analysis viewpoint encompassing a Modelica model.

A **Model** is an abstraction of a real-world artifact (described by a *Concept*). Multiple *Viewpoints* may be required to address the stakeholder concerns with respect to an artifact (*Concept*), which can be supported through multiple modeling *Views*. A *Model* contains many *Properties* with multiple *Dependencies* between them.

A **Property** is any descriptor of an artifact. Its value could be numerical, logical, stochastic or an enumeration. In design, two types of properties are used:

properties which are selected or chosen by the designer (Synthesis Properties (SP)) and ones which are predicted through an analysis model or an equation (Analysis Properties (AP)). In order for a property to have an unambiguous meaning, the semantic context around each property should be specified, which is done through *isPropertyOf* relationships to a *Concept*. For example, *EERT isPropertyOf* a *Concept ControlSystem* (see Figure 2), which, in turn, *isPartOf* a *Robot*. A *Property* can influence other properties via a *Dependency*, which is captured through the *relatedDependency* relationship: e.g., SD_4, SD_5 and SD_{13} are related dependencies for the property EERT (see Figure 2).

A **Synthesis Property (SP)** describes the value that a designer has selected for a particular property. SPs are usually defined through a range of values *(RangeValue)*; but they can also be defined through a *FixedValue* or a *BooleanValue*. For example, a load profile could be used as an SP to select the corresponding *actuator power*.

An **Analysis Property (AP)** describes the value predicted as a result of performing an analysis captured in a model, e.g. solving an equation or a constraint. APs are predictions and hence uncertain by definition. Therefore, APs should be specified using a *Probability* value *(ProbabilityDensityFunctionValue)*.

A **Dependency** describes the nature of the relationship between two or more properties. The relationship is assumed to be causal, thereby assuming that some properties are *inputs* while others are *outputs*. The nature of a particular dependency could be known or unknown at a given design stage, and its specifics are described in a number of ways. As per [5], dependencies can be expressed in two forms - a heuristic between two or more properties (*Synthesis Dependency* (SD)), or a constraint, an equation or an analysis model (*Analysis Dependency* (AD)). One particular case is that of an equality binding between two or more properties (e.g., same properties belonging to multiple views). There could be many dependencies within a *Model*, hence a particular *Dependency* can be a part of (i.e. contained within) a particular *Model* (i.e., a model within a model, such as a constraint within a CAD model), or, in other cases, represents a distinct *Model* (e.g., a Simulink model). *Bindings* between properties are captured in the *Dependency Model*.

A **Synthesis Dependency (SD)** refers to the heuristics used in selection of a SP. It is also possible that a modeler uses their experience in making this selection, and overrides the heuristic completely. An SD could have one or more SPs as its output, e.g. SD_4 in Figure 2.

An **Analysis Dependency (AD)** refers to the analysis (present in a model), an equation, or a constraint used to predict the value of an AP. An AD could have one or more APs as its output.

While the dependency models are causal in nature, in many practical scenarios, cycles will be present. For example, an algebraic loop could exist, where a property is both chosen and predicted. From the perspective of structural semantics, cyclic models are valid. However, from the perspective of operational semantics (which are outside the scope of this paper), such loops must be broken during execution - for example, by using the well known tearing algorithm.

5 Illustration: Dependency modeling through the DML

EMF was used to support the DML which we used to construct the dependency model for the example use case. In the following, we will illustrate the equality binding between properties that are part of different views of the robot. Other possible illustrations include (but are not limited to): top-level view of the robot showing the involved *Viewpoints* and *Concepts*, and binding of a *Property* to multiple dependencies. The reader should note that the illustrations we provide are models generated based on the abstract syntax and no concrete syntax was developed at this stage, although graph-based visualizations of the dependency models were built (see Section 6).

Consider the *Synthesis Dependency* SD_4 in the CAD View. Figure 2 shows the dependency SD_4 where Motor A Torque (M_A) is determined based on the information about Inertia of Arm-A (I_A), the requirement for End-to-End Response Time (EERT), and the control system structure (CS).

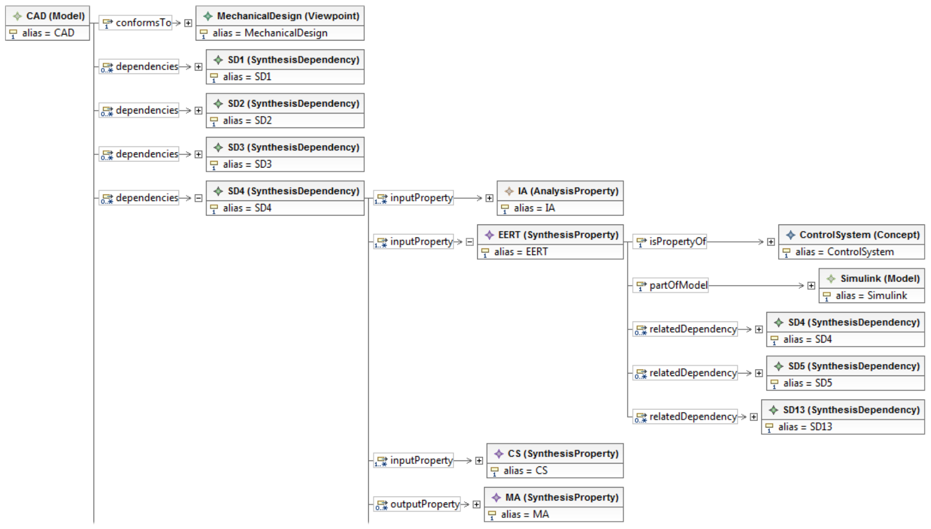

Fig. 2. Contents of the dependency SD_4 (within the CAD View) showing equality binding to the property EERT, which is a property of the Simulink View.

It can be seen that property EERT *isPropertyOf ControlSystem Concept* and is *partOfModel* Simulink, which is a supporting View in the ControlDesign Viewpoint. The resulting property binding relationship is contained under the property EERT, and maintained inside the dependency model. The meta-models of Matlab/Simulink, and MagicDraw SysML (as UML2.1) are available in our Eclipse implementation (Cameo Workbench [11]), and we added the Solid Edge (CAD tool) meta-model to it, hence the models created in these tools can be read as Ecore models and transformations between them can be built.

6 Visualizing dependencies as graphs

The information captured in the dependency model can be visualized by a *dependency graph*. As opposed to a tree-based representation, a graph-based representation is better suited for discussions between different stakeholders (see Figure 2 and Figure 3). We used the tool Graphviz [12], which supports the DOT language [13], to build graphs. Figure 3 shows a directed dependency graph between the three views of the robot. As an example, consider SD_9 which refers to the dependency between inertia of first and second arm of the robot (I_A and I_B in mechanical design view) and the transfer function ($G(s)$) attributes (in the control design view). The figure illustrates that even for a fairly simple robot design example, there are many dependencies between the considered viewpoints, and manual management of such dependencies is either very challenging or often not possible due to a lack of information.

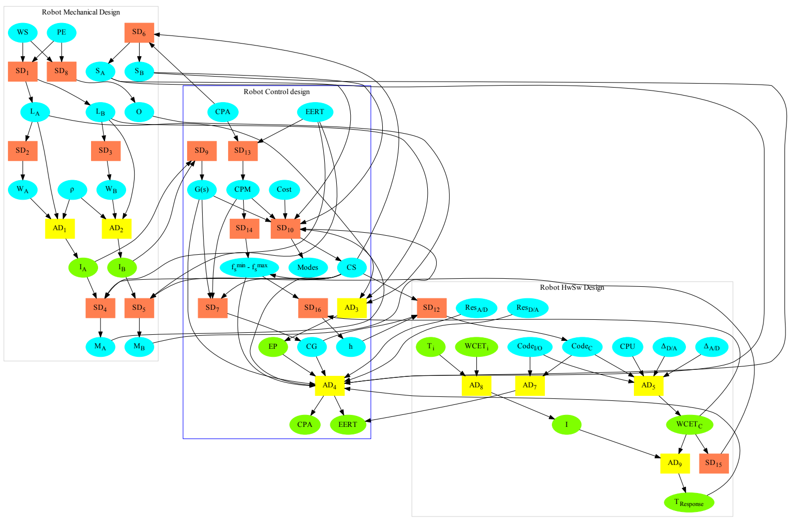

Fig. 3. Dependency graph between three robot views. SPs are shown in Blue, APs in Green, SDs in Orange and ADs in Yellow.

7 Related Work

One popular method of modeling dependencies is the Design Structure Matrix (DSM) [14], which allows relations among properties to be represented in a matrix. DSMs are used in a variety of disciplines - for example, in software engineering [15]. Compared to the DML, DSMs are limited in terms of their expressiveness. For one, it is not possible to differentiate between synthesis and analysis properties, nor between synthesis and analysis dependencies. As discussed in [5], this differentiation is important to keep analysis results separate from selections made by the designer. In addition, this differentiation adds to the semantic richness of a dependency model thereby supporting the change management and inconsistency management scenarios. Furthermore, the semantic context around a property can be shown in a DSM only to a limited degree.

In terms of tools, Product Data Management (PDM) systems are probably among the most widely used systems to manage product-related data. One of the core capabilities of modern PDM systems is allowing users to establish relations between elements that are stored in the repository: for instance, reference and correspondence relationships can be created. Such relationships can typically only be created among files. However, some contemporary PDM systems integrate tool adapters, allowing for certain properties of supported models to be exposed (for example: the part hierarchy in CAD models). While PDM systems implement some of the desired functionality, they are still limited in terms of their capabilities of capturing dependencies. In particular, PDM systems require dependent models to already exist, therefore not enabling one to create a dependency model independently from the corresponding design artifacts.

Modeling dependencies is also supported (at least to some extent) in the Process Integration Design Optimization (PIDO) approach, which is implemented in tools such as ModelCenter [16]. The underlying principle is the integration of disparate models. ModelCenter, for instance, provides several tool connectors, which enable data exchange among disparate models and allow for properties of compatible models to be exposed. As a result, dependencies between properties can be modeled. However, current PIDO tools only provide a black box view for each model and hide most of the semantic context of properties. Furthermore, not all possible properties can be exposed.

To the best of knowledge of the authors, modeling languages intended specifically for the purpose of modeling dependencies have, to the date of writing this paper, not been publicized. While there are some promising methods and tools available, none implement all of the envisioned capabilities. For one, none allow for the definition of a dependency model independent of other domain specific models. Furthermore, of the approaches surveyed, none provided the desired level of depth and access to properties in models.

8 Discussion

The order in which the different views are developed in relation to the dependency model is an important consideration. There are two possibilities here: a bottom-up scenario where the initial design and analysis of design concepts is already captured in multiple views (e.g. a CAD model and a Simulink model), and then the dependency model is built. In this case, the knowledge already present in disparate views can be used to automatically build parts of the dependency model. The other possibility is a top-down scenario where the dependency model is manually created after the requirements and the system architecture are identified, and based on the dependencies captured in the dependency model, other views such as CAD and Simulink models are developed. For the example described in Section 5, we have followed the former approach, where the views supporting mechanical, control and Hw/Sw design of the robot already existed.

Dependency models can be used for more than just one purpose. Given the causal nature, a dependency model can be used for change propagation and consistency management. For example, in Figure 3, a change to the predicted value of $T_{Response}$ triggers the necessity for AD_4 to be refreshed automatically. It can also support traceability in that it is possible to reason about both the existence and nature of certain relationships among models. For example, one useful application is requirements traceability. Dependency models are also useful for the purpose of managing workflow. Given a (causal) network of dependencies, tasks can be parallelized and merge points identified. A dependency model can also be used for the purpose of avoiding certain inconsistencies. Not only can changes be propagated through such a model, but a single source of truth for properties is established. Such is the case because properties in the dependency model are unique, even though these may refer to elements in disparate models.

Modeling dependencies requires additional effort and, hence, additional resources to be allocated. However, any commitment of resources needs to be justified. It is entirely conceivable that in some cases (e.g. very simple or well understood systems) the risk associated with not explicitly capturing dependencies is negligibly low. Similar arguments can be made about the completeness of the dependency model: to what level of detail should dependencies be modeled? As per [5], dependencies can be defined at six levels of detail starting with the level-0 where the dependencies are completely unknown to level-5 where both the dependencies and the transformation models that lead to them are explicitly known. Behind building such transformation models are *dependency patterns* which gather and illustrate known dependencies between specific types of properties under a design context. The use of such patterns would decrease the cost associated with modeling dependencies. Patterns are currently not supported by the introduced DML and their discussion is beyond the scope of this paper.

9 Conclusions

This paper presents a DSML for modeling dependencies between properties. Properties are typically referenced in multiple views on a system. A dependency modeling language allows for the dependencies between these properties to be captured in a single model, as illustrated for a robot example in Section 5.

Future work should includes the provision of additional features, such as supporting modeling at multiple levels of detail and allowing for a variety of stakeholder-specific views to be generated automatically. Furthermore, the operational semantics of the DML should be defined formally. This is particularly important for the purpose of supporting the accompanying dependency management process. For example, an essential task is analyzing how changes propagate.

Since most analysis activities involve some sort of token flow, we suggest to investigate the mapping to the semantic domain of petri-nets as future work.

Acknowledgments. This work was supported in-part by Boeing Research & Technology. The authors would like to thank Michael Christian (Boeing) and Axel Reichwein (Koneksys LLC) for the many valuable discussions.

References

1. Herzig, S.J.I., Qamar, A., Reichwein, A., Paredis, C.J.J.: A Conceptual Framework for Consistency Management in Model-Based Systems Engineering. In: ASME 2011 Design Engineering Technical Conferences & Computers and Information in Engineering Conference IDETC/CIE 2011, Washington, DC, USA, ASME (2011) 1329–1339
2. Thompson, S.C.: Rational Design Theory: A Decision-Based Foundation for Studying Design Methods. Phd. thesis, Georgia Institute of Technology, Atlanta, Georgia, USA. (2011)
3. Hazelrigg, G.A.: A Framework for Decision Based Engineering Design. Journal of Mechanical Design **120**(4) (1998) 653–658
4. Object Management Group: OMG Unified Modeling Language (UML) Specification V2.4.1 (2011)
5. Qamar, A., Paredis, C.J., Wikander, J., During, C.: Dependency Modeling and Model Management in Mechatronic Design. Journal of Computing and Information Science in Engineering **12**(4) (December 2012) 041009
6. Object Management Group: OMG Systems Modeling Language Specification V1.3 (2012)
7. Vallecillo, A.: On the Combination of Domain Specific Modeling Languages. In: Modeling Foundations and Applications, Lecture Notes in Computer Science. Volume 6138. (2010) 305–320
8. Brooks, F.P.: No Silver Bullet Essence and Accident in Software Engineering. IEEE Computer **20**(4) (1987) 10–19
9. Mosterman, P.J., Vangheluwe, H.: Computer Automated Multi-Paradigm Modeling: An Introduction. Simulation: Transactions of The Society for Modeling and Simulation International **80**(9) (September 2004) 433–450
10. Eclipse Foundation: Eclipse Modeling Framework (EMF) (2009)
11. No Magic: Cameo Work Bench (2011)
12. Ellson, J., Gansner, E.R., Koutsofios, E., North, S.C., Woodhull, G.: Graphviz and Dynagraph - Static and Dynamic Graph Drawing Tools. In: Graph Drawing Software, Springer-Verlag (2003) 127–148
13. Gansner, E.R., Koutsofios, E., North, S.: Drawing Graphs With Dot. Technical report (2009)
14. Eppinger, S.D., Browning, T.R.: Design Structure Matrix Methods and Applications. Engineering Systems. MIT Press (2012)
15. Sangal, N., Jordan, E., Sinha, V., Jackson, D.: Using Dependency Models to Manage Complex Software Architecture. In: Object Oriented Programming, Systems, Languages & Applications (OOPSLA), San Diego, CA, USA, ACM Press (2005) 167–176
16. Phoenix Integration: ModelCenter (2012)