View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Elsevier - Publisher Connector

Available online at www.sciencedirect.com

ScienceDirect

Procedia CIRP 21 (2014) 52 – 57

www.elsevier.com/locate/procedia

24th CIRP Design Conference

A conceptual basis for inconsistency management in Model-Based Systems Engineering

Sebastian J. I. Herzig[a,*], Christiaan J. J. Paredis[a]

[a]*Model-Based Systems Engineering Center, Georgia Institute of Technology, Atlanta, Georgia, United States of America*

* Corresponding author. Tel.: +1-404-247-0290. *E-mail address:* sebastian.herzig@gatech.edu

Abstract

A crucial issue in system architecting is the need to study systems from different viewpoints. These viewpoints are defined by a variety of factors, including the concerns of interest, level of abstraction, observers and context. Views conforming to these viewpoints are highly interrelated due to the concerns addressed overlapping. These interrelations and overlaps can lead to inconsistencies. The challenge is to identify and resolve - that is, manage - such inconsistencies. This paper introduces an approach to managing inconsistencies within the context of Model-Based Systems Engineering (MBSE). In current practice, the management of inconsistencies relies on ad-hoc methods and infrequently conducted activities such as reviews. The result of this practice is that decisions are often made based on inconsistent information, which can lead to costly rework or even mission failure. Therefore, assisting humans by means of a computational method that can continuously identify and aid in resolving inconsistencies adds significant value. In the paper, the hypothesis that pattern matching can serve as a generic means of identifying inconsistencies is investigated. It is shown that graph patterns can be used as a means to capture conditions for and formally reason about the existence of inconsistencies, and to specify resolution alternatives. The paper concludes that using patterns to manage inconsistencies can be very effective and accurate, but it may also incur additional costs that must be carefully balanced with the benefits gained.

© 2014 Elsevier B.V. This is an open access article under the CC BY-NC-ND license
(http://creativecommons.org/licenses/by-nc-nd/3.0/).

Selection and peer-review under responsibility of the International Scientific Committee of "24th CIRP Design Conference" in the person of the Conference Chairs Giovanni Moroni and Tullio Tolio

Keywords: inconsistency management, model-based systems engineering, viewpoint integration, model integration, graph pattern matching

1. Introduction

When designing and developing engineering systems, one common practice of managing the often overwhelming complexity is to study the system from different viewpoints. Such viewpoints are defined by a variety of factors, including the concerns of interest, level of abstraction and context. Different stakeholders study the system from different viewpoints. Consider the design and development of an aircraft system: a project manager, manufacturing engineer and design engineer each address different concerns and have differing interests. However, it is only through their collaborative effort that the overall design and development process of the system can progress. This is due to the concerns addressed by the various stakeholders being highly interrelated. The presence of such interrelations introduces the potential for *inconsistencies* [1].

We say that an inconsistency is present if two or more statements are made that are not jointly satisfiable. From the perspective of logic, an inconsistency is the result of a contradiction. There are many examples for inconsistencies: failure of an equivalence test, non-conformance to a standard or constraint and the violation of physical or mathematical principles. The presence of inconsistencies can have serious consequences, as can be deduced from studying just a few of the catastrophic mission failures in aeronautics [2,3]. This emphasizes the need to identify and resolve inconsistencies — a process we refer to as *inconsistency management*.

In current practice, support for managing inconsistencies is limited. Most modeling tools provide ad-hoc support for checking conformance to syntactical and well-formedness rules. However, no appropriate infrastructure is available that is capable of managing a broad class of inconsistencies. In systems engineering, inconsistencies are identified as part of the process of verification and validation (V&V) [4,5]. Since the time interval between V&V activities is commonly very long, the cost associated with resolving inconsistencies can be very high. Typically, the earlier an inconsistency is identified, the cheaper it is to fix. Therefore, we argue that a continuous and (semi-) automated process of managing inconsistencies positively supports V&V.

This paper provides a conceptual basis for managing inconsistencies in Model-Based Systems Engineering (MBSE). We

2212-8271 © 2014 Elsevier B.V. This is an open access article under the CC BY-NC-ND license
(http://creativecommons.org/licenses/by-nc-nd/3.0/).
Selection and peer-review under responsibility of the International Scientific Committee of "24th CIRP Design Conference" in the person of the Conference Chairs Giovanni Moroni and Tullio Tolio
doi:10.1016/j.procir.2014.03.192

present graph pattern matching as a formal basis for identifying and resolving inconsistencies. Furthermore, we discuss key limitations and practical implications of the approach as well as related challenges, some of which are unique to designing and developing physical systems. The remainder of this paper is structured as follows: section 2 provides a brief overview of the related literature. Our approach to managing inconsistencies is outlined in section 3. Practical considerations for implementing the conceptual approach are discussed in section 4. The paper closes with directions for future research.

2. Related Work

Finkelstein is often credited with being the first person to introduce the notion of inconsistency management [6]. Based on the work of Finkelstein *et al.* [7] and that of Nuseibeh *et al.* [8], Spanoudakis and Zisman propose a general framework for inconsistency management in [1]. The framework defines six distinct activities: *detection of overlaps*, *detection of inconsistencies*, *diagnosis of inconsistencies*, *handling of inconsistencies*, *tracking of inconsistencies* and *specification and application of an inconsistency management policy*. Early work primarily focuses on developing methods and tools for identifying inconsistencies (and overlaps). Recent related literature discusses the process of resolving inconsistencies more elaborately [9,10].

Most related work originates from model-driven software engineering research. In early work, Finkelstein *et al.*, discuss the use of first-order predicate logic to detect inconsistencies through automated theorem proving [6]. A similar approach using propositional logic is followed by Schaetz *et al.* [11]. Both works mention that an inherent limitation of their approach is that both propositional and first-order logic are not sufficiently expressive enough to capture all of the knowledge and information related to software models. Van der Straeten *et al.* explore the use of a description logic to attempt to not only identify inconsistencies, but maintain consistency through logical inference [12,13]. In their work, the authors use (domain-specific) rules to both identify and resolve inconsistencies. Later work by Mens *et al.* describe capturing dependencies between inconsistencies and possible resolution actions (in the form of model transformations), as well as sequential dependencies between resolution rules [9]. This work is complimented by earlier work, in which Mens *et al.* argue that existing formal modeling languages such as UML should be extended to directly incorporate support for inconsistency management [14].

In systems engineering research and, more generally, the field of model-based design and development of (cyber-) physical (as opposed to purely software) systems, particularly fundamental work is lacking. This is often justified by the premise that the fundamental concepts developed in model-driven software engineering research can directly be applied. This is somewhat surprising in that most researchers agree that there are key differences between the design and development of physical systems and of software systems – for example, the broader knowledge required due to the multi-disciplinary nature of most technical systems and the heterogeneity of the multitude of disparate models being used. This resulted in Herzig *et al.* investigating the fundamentals of consistency management in models of complex systems [15]. Qamar and Paredis later propose a conceptual approach to explicitly capturing dependencies across a set of models as an aid for identifying and sequentially resolving inconsistencies [16]. Hehenberger *et al.* [17] suggest the use of domain-spanning ontologies for the purpose of identifying overlaps and use a rule-based approach to identifying inconsistencies. In [18], Gausemeier *et al.* introduce an approach aimed at maintaining consistency between disparate models and a *principle solution* (an abstract model of the system). The approach is based on capturing correspondences between models using triple graph grammars, which are also used to propagate changes. However, this approach is limited in the sense that correspondences cannot be defined across domain-specific models.

3. Conceptual Model

In systems engineering, inconsistencies manifest in a variety of forms: violation of well-formedness rules, inconsistencies in redundant information, mismatches between model and test data, and not following heuristics or guidelines. In previous work, we concluded that it is impossible to identify all inconsistencies [15] — that is, it is impossible to prove (or maintain) consistency. Therefore, the focus must be on identifying and resolving *inconsistencies*.

Inconsistencies can be identified through deductive reasoning [19]. Deductive reasoning is the process of reaching a conclusion starting from a set of premises [20]. Automating this process requires the ability to perform symbol manipulation. A prerequisite to this is that the information being reasoned about must conform to some formalism. In Model-Based Systems Engineering (MBSE), a key principle is that only formal models should be used [5]. Therefore, MBSE provides a suitable basis for reasoning.

3.1. Graphs: a Common Representation for Models

The heterogeneouity of the models used in the process of designing and developing physical systems complicates the identification of a unifying formalism. Here, we consider graphs as a generic enough formalism to represent any model.

Proposition 1. *A model can be represented by a graph.*

Models are abstractions of reality [21] and are used to capture knowledge. Knowledge defines how different pieces of information are related to one another. Therefore, a model can be interpreted as being composed of elements and relations, and it is only natural to think of models as graphs. This supports proposition 1. Similarly, artificial intelligence applications often use graphs to represent information and knowledge through the use of *object-attribute-value* triples or *semantic nets* [20]. In [21] proposition 1 is supported further by the argument that the meta-model of a modeling language, which can be used to specify the abstract syntax of a modeling language, is a type graph.

We argue that in order to express knowledge and information meaningfully in a graph, data and relations must be attributed (e.g., labeled), and relationships among data must be directional. Therefore, a model should be represented by an attributed, directed multigraph, where we use the term multigraph to indicate that the graph may contain cycles.

Definition 1. *A 6-tuple* $\boldsymbol{G} = (V, E, A_V, A_E, m_V, m_E)$ *is an attributed, directed multigraph, where* $V = \{v_1, v_2, ..., v_n\}$ *is a finite set of vertices and E is a set of tuples (ordered pairs denoting edges) over the relation* $E \subseteq V \times V$. A_V *and* A_E *are sets of attributes (e.g., labels) and* $m_V : V \rightarrow A_V$ *and* $m_E : E \rightarrow A_E$ *are partial functions that assign attributes to vertices and edges, respectively.*

A graph-based representation of a model can be constructed by defining a morphism $g : \mathbf{M} \rightarrow \mathbf{G}$ between a model **M** and a corresponding graph representation **G**. Note that this morphism is not restricted to an isomorphism. That is, not all of the information associated with a model must necessarily be translated into graph form. For the purpose of managing inconsistencies, only the information that is considered meaningful should be included. For example, including details about the layout of elements in a SysML or UML diagram is not always meaningful.

3.2. Identifying Inconsistencies

Assuming that all data associated with a model is represented by a graph, any deducible manifestation of an inconsistency must be *contained in* said graph. That is, given additional knowledge (in the form of a negative constraint or a negative conjuncted proposition), a subset of the graph must represent what constitutes a part of a particular logical contradiction.

Proposition 2. *Given a graph* **G** *representing a model* **M**, *a particular instance of an inconsistency manifests as a particular subset of the vertices and edges of* **G**. *Therefore, a particular inconsistency can be represented by a graph* **I**, *of which an isomorphic image exists in* **G**.

Determining whether or not a graph (and hence the associated model) contains an inconsistency **I** is therefore related to the problem of finding a particular subgraph, i.e., the *subgraph isomorphism* problem. For directed multigraphs, the associated decision problem can be formulated as follows [22]: *given two graphs* $\boldsymbol{G} = (V, E)$ *and* $\boldsymbol{I} = (V_I, E_I)$, *is there a subgraph* $\boldsymbol{S} = (V_S, E_S)$ *where* $V_S \subseteq V$, $E_S = E \cap (V_S \times V_S)$ *such that there exists a bijective mapping* $s : V_S \rightarrow V_I$ *and such that for each ordered pair* $(v_1, v_2) \in E_S$*?* In other words, it is checked whether a) all vertices defining the inconsistency can be mapped injectively to the vertex set of the graph and b) at least those edges that are part of the inconsistency are present in **G**. For the case of attributed, directed multigraphs, the definition of the decision problem is similar where, in addition, vertex and edge attributes must be preserved.

In the related literature, finding a particular subgraph is part of the more general area of *pattern matching*, specifically pattern matching in graphs [23]. There, partially defined graph data (a pattern) is used to *query* a *target graph* to determine whether or not a particular subgraph exists. Therefore, a pattern may be interpreted as a specification of the subgraph(s) to be matched in a target graph, where the pattern itself is defined by a graph. Finding an isomorphic image may be classified under *exact pattern matching*. In addition, there is *inexact pattern matching*, where patterns incorporate vertex and edge variables as well as substitution rules [23,24]. Substitution rules are typically composed of a combination of variables and production rules.

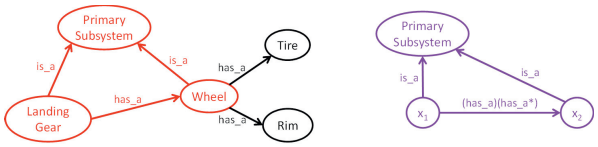

Fig. 1. Left: graph representation of a model of a system with an inconsistency marked in red; right: pattern used to identify the inconsistency associated with the rule *primary subsystems may not contain other primary subsystems*

To illustrate our approach, we introduce a simple means of defining graph patterns. The technique is based on regular expressions and is similar in spirit to [24]. Assuming that $REG(\Sigma)$ is an expression for the set of all regular languages that can be formed over the alphabet Σ, we define a graph pattern as follows:

Definition 2. *A graph pattern* **P** *is defined as an attributed, directed multigraph* $\boldsymbol{P} = (V_P, E_P, A_{V_P}, A_{E_P}, m_{V_P}, m_{E_P})$ *with* $A_{V_P} \subseteq A_V \cup L_{V_{var}}$ *where* $L_{V_{var}}$ *is a set of labels denoting vertex variables that is disjoint from* A_V *(i.e.,* $A_V \cap L_{V_{var}} = \emptyset$*), and with* $A_{E_P} \subseteq A_E \cup REG\,(A_E \cup L_{E_{var}})$ *where* $L_{E_{var}}$ *is a set of labels denoting edge variables that is disjoint from* A_E.

Such patterns are matched using a pair of mappings (p_V, p_E), where p_V defines a surjective, non-injective mapping from a subset of the vertices in **G** to those defined in **P** based on label equality. Vertex variables injectively map to one or more vertices of **G**. The second mapping p_E maps edge labels used in **P** to those in **G**: for every edge $(v_{P_x}, v_{P_y}) \in E_P$ there must exist a path (or walk) between $p_V(v_{P_x})$ and $p_V(v_{P_y})$ whose label is in A_{E_P}. This is illustrated in figure 1, where a graph (left) contains an inconsistency that can be identified using the given pattern (right). Note that, in accordance with definition 1, A_V and A_E are sets of attributes of the multigraph **G** (which represents a model **M**). The elements contained in these sets represent a part of the vocabulary that is available for defining patterns. Therefore, the expressiveness of the patterns depends, in part, on the known vocabulary.

Theorem 1. *Given a graph-based representation of a model, any contained inconsistencies can be identified by querying graph patterns.*

By definition of regular languages, an equivalent finite automaton can be constructed for each pattern that is defined according to definition 2. Therefore, it is decidable whether or not a particular inconsistency is present. This further supports theorem 1.

Most practical patterns will only require vertex variables: for instance, the query *find all elements that are generalizations of themselves*. The use of regular expressions can be useful to define more complex patterns, e.g., to make use of *negative matching* (negative lookahead). For example, *find all abstract classes that have no concrete implementations*.

3.3. Resolving Inconsistencies

Once an inconsistency has been identified, an action must be taken. Like design [25], inconsistency resolution can be considered a *decision making process*. For each inconsistency,

multiple alternative courses of action may exist. Relevant alternatives must be enumerated (e.g., based on the original pattern that has been matched) and evaluated before a decision is made. Thinking of the process of resolving inconsistencies in this manner allows one to determine which course of action is the most *valuable*. It is important to recognize that the most valuable course of action is not necessarily to *repair* the inconsistency: in some cases it may be better to simply *ignore* the inconsistency (temporarily or permanently). For example, if the inconsistency is caused by a change to the inputs of a computationally expensive analysis model and it is known that the resulting change in the output is insignificant, the inconsistency may be tolerable.

What makes the process of resolving inconsistencies particularly challenging is that an inconsistency may span multiple views and, hence, models that are owned by separate stakeholders. In order to support the process of resolving an inconsistency, the ownership of certain parts of models (and hence, of the graph) must be explicitly captured. This information can then be used to determine the relevant parties that need to be involved in the inconsistency resolution. In addition to ownership, the sequence in which inconsistencies are resolved is also an important consideration, since the resolution of one inconsistency may result in the introduction of further inconsistencies.

Ideally, inconsistencies should be resolved automatically. However, we argue that this is only possible in trivial cases. If the *certainty* of the applicability of a particular alternative can be computed, the decision whether or not a human should be involved in the resolution can be automated. A transformation of the graph could then be used to resolve a particular inconsistency automatically, which would mean that the inconsistency pattern is treated as the left-hand side and the modifications to the graph as the right-hand side of a rule.

4. Practical Considerations

In the previous section it was shown that any inconsistency can be inferred by means of graph pattern matching. However, if a model contains redundant information, and the existence and nature of such redundancies is not explicitly captured, associated inconsistencies cannot be identified using such graph patterns. Furthermore, in any practical systems engineering scenario, models will be distributed physically and disparate due to their multi-disciplinary nature. This section discusses how these practical considerations can be overcome.

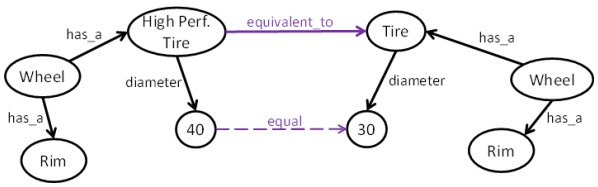

Fig. 2. Explicitly capturing the relationship between two redundantly defined elements. The dashed line marks a relationship implied by a corresponding pattern.

4.1. Redundancies & Semantic Overlaps

A key concept in MBSE is the development of a single model of the system: the *system model*. *Views* are established by referencing elements from the system model. Views conform to *viewpoints*, which are defined by a set of properties that identify the concerns addressed, and languages and methods used to present the view [5]. A system model can be regarded as a *composition* of all views.

In practice, models evolve concurrently. This can lead to redundant information being introduced. For example, modelers could introduce semantically equivalent statements such as equivalent value properties. This issue can occur even in the simplistic case of considering a single isolated model. If redundancies are not explicitly identified, a pattern matching query cannot identify associated inconsistencies.

The problem worsens in a more realistic scenario where multiple, physically distributed and disparate models are considered. Assume that it is possible to define an operator $\otimes$ that composes a set of n models into a single model, i.e.:

$$model_{sys} = model_1 \otimes model_2 \otimes ... \otimes model_n \quad (1)$$

Such an operator must fold all redundant statements into a single statement and, therefore, be surjectively defined. Additionally, the relations among semantically related statements must be made explicit by adding additional information. However, applying such an operator requires a certain level of *understanding* of the information and knowledge captured in the models. Humans use both implicit knowledge and the context of a particular element to infer its meaning. An algorithm that is not provided with this additional knowledge cannot identify redundancies with certainty.

The related literature refers to this problem as identifying *overlaps*, which is considered to be an open and, in terms of automation, largely unsolved problem [1]. Among the alternative approaches mentioned, matching elements by their names and methods based on the use of a unifying ontology are most common. However, it is largely agreed that such mechanisms are not sufficient in any realistic scenario and that a human must at least assist an algorithm in identifying both the existence and nature of overlaps [1]. For example, in figure 2, both instances of *Wheel* and *Rim* can easily be identified as being redundant based on the names and surrounding context. However, the fact that both *Tire* and *High Perf. Tire* are semantically equivalent can only be inferred using additional knowledge.

In our conceptual model, redundancies are identified by adding additional edges to a graph. In practice, capturing all of the required additional relationships manually can be very costly. By defining patterns to identify more complex relationships at a higher level of abstraction (e.g., equivalence of a property implies equality of values, units and quantity kinds), this cost can be significantly reduced. This is similar to the idea behind dependency modeling [16] or relying on a unifying, system-wide ontology [1].

4.2. Distributed Modeling Infrastructures

When multiple stakeholders work in different domains with a disparate set of models, the models typically evolve concurrently. Each model is maintained by a particular person and

using a number of tools. Additionally, multiple copies and versions of each model may exist at various locations.

In practice, models are distributed on a number of physical devices. Relating to the conceptual approach of building a single graph of all models, this means that neighboring vertices could refer to information or knowledge that is stored at a different physical locations. In order to match subgraphs in such a scenario, it must be possible to uniquely identify and access each element. A linked data [26] approach can serve as an enabling platform for this purpose due to it being an environment that was – from the start – meant to be used to access distributed knowledge. *Unified Resource Identifiers* (URIs) serve as strong keys to uniquely identify individual nodes or subgraphs. As a knowledge representation language, the *Resource Description Framework* (RDF) [27] or the *Web Ontology Language* (OWL) [28] could then be employed due to the possibility of encoding information and knowledge using *subject-predicate-object* triples.

Defining how the information and knowledge contained in models should be structured in a graph – that is, how the mapping that is used to construct a graph representation of a model should be defined – can be based on the underlying structure of modeling languages or how data is stored in tools. For example, a meta-model of a particular modeling language is useful for this purpose. The way information and knowledge in a model is structured can often also be inferred from the organization of the application programming interface (API) of a tool. Ideally, concepts (such as modeling constructs) should be differentiated from individuals. *Open Services for Lifecycle Collaboration* (OSLC) [29] defines several guidelines for the purpose of representing models and meta-models from different domains in RDF.

4.3. Versioning

In most practical scenarios, multiple versions and copies of models exist. This raises a set of interesting questions: *which versions should be checked for which particular inconsistencies?* Also, *should inconsistencies between versions of the same model be managed?* The related literature provides a mixed perspective on this problem. Some researchers suggest that inconsistencies must be checked both across and between versions [11,14]. However, this argument is based on the assumption that the evolution of models merely entails refinement of previously made statements. We argue that, within the context of systems engineering, checking for inconsistencies across versions has little value. As the understanding of a system grows, the beliefs of designers are updated (and, additionally, customer requirements may change) and earlier versions of models may have been developed based on premises that are obsolete. Therefore, we argue that inconsistencies should *not* be managed across versions. A special case is that of the *working copy* of a model. A working copy is a copy that is not yet considered complete enough to be a separate version. Therefore, it is likely to contain numerous inconsistencies, most of which a modeler will very well be aware of. Presenting a modeler with an exhaustive list of all of these inconsistencies is not valuable. In agreement with related work [11], we argue that one should differentiate between *invariant* and *variant* inconsistency checks. Invariant checks should be performed irrespective of the specific context or moment in time, while the applicability of variant checks is more restrictive. For example, conformance with the syntax of a language should always be checked. Therefore, it is meaningful to check for invariant inconsistency conditions in the working copy, and both invariant and variant conditions in and among the *head* – that is, latest, committed – versions of models.

4.4. Proprietary & Regulated Data

Another aspect to be considered is that full access to a model may not always be granted to all stakeholders. Such may be the result of the proprietary nature of the data (e.g., supplier data) or because access to the data is regulated (e.g., sensitive material regulated under the *International Traffic in Arms Regulations* (ITAR)). This means that different stakeholders must have different levels of access to portions of the overall data. Therefore, identifying some redundancies (and, hence, managing related inconsistencies) may be difficult, if not impossible in some cases. In principle, access rights can be incorporated into the graph by adding additional nodes and edges. We argue that this practical consideration has no impact on the conceptual approach, given that we acknowledge the facts that a) not all inconsistencies can be identified [15] and b) the mapping $g : \mathbf{M} \rightarrow \mathbf{G}$ is not necessarily an isomorphism (see section 3.1).

5. Discussion

Since the set of all possible inconsistency patterns is likely to be infinitely large [15] and since graph pattern matching is an NP-complete problem [22], both the topic of maintainability and complexity of the approach are discussed briefly in the following.

Careful crafting of inconsistency rules is required to minimize the number of false positives. A consequence of this is that patterns may become very complex. In addition, one may need to define several variations of semantically very similar patterns to more precisely control the context in which the patterns are matched. Therefore, it is conceivable that, in any realistic scenario, maintaining a set of inconsistency patterns can be very costly. Clearly, this cost is proportional to the number of patterns that are defined, but is offset by their benefit. Patterns that require little context to be specified (see e.g., the example related to inconsistencies of value properties in Fig. 4c) and are therefore not very complex, but have the potential to identify a large number of inconsistencies (e.g., because of value properties being very common) are likely to be very valuable. However, inconsistencies that are easily spotted by a human and require complex patterns to be defined may not always be sufficiently valuable to justify the cost involved in defining and querying for the particular pattern. The resulting inherent incompleteness of the set of inconsistencies that are managed versus the complete set of inconsistencies that could conceivably exist is not inconsistent with our conceptual approach, which acknowledges the fact that it is impossible to maintain consistency [15].

Graph pattern matching and hence inconsistency identification is an NP-complete problem [22]. However, due to the labeled nature of the graphs, known heuristics can be employed to increase the performance of pattern matching to less than polynomial time (at least for most practical cases) [23,30,31].

Additionally, for very large graphs, performance can possibly be increased by making use of distributed computing. A third possibility is to enforce an artificial upper bound on the computation time and output a result of *unknown* (i.e., it is unknown whether or not a particular inconsistency exists). This is a strategy similar to that employed by some theorem provers [32].

6. Conclusions

This paper presents a conceptual basis for inconsistency management in Model-Based Systems Engineering. In systems engineering, inconsistencies manifest in a variety of forms: violation of well-formedness rules, inconsistencies in redundant information, mismatches between model and test data, and not following heuristics or guidelines. We argue that a model can be represented by a graph and that inconsistencies manifest as subgraphs. To identify inconsistencies, graphs can be queried using partially defined graph data — in other words, graph patterns. Resolving inconsistencies is, in most cases, a non-trivial problem that requires further analysis and human input.

Future work should focus on two aspects: 1) proving the technical viability and practicality, and measuring the effectiveness of the conceptual approach by implementing a set of supporting tools and 2) investigate the possibility of using stochastic reasoning to allow for partial pattern matches and thereby attempt to mitigate the limitation related to maintainability identified in section 5. The authors are currently actively investigating both of these topics.

Acknowledgements

This work was supported by Boeing Research & Technology. The authors would like to thank Michael Christian (Boeing), Ahsan Qamar (KTH) and Axel Reichwein (Koneksys LLC) for the many valuable discussions.

References

[1] Spanoudakis, G., Zisman, A.. Inconsistency Management in Software Engineering: Survey and Open Research Issues. In: Handbook of Software Engineering and Knowledge Engineering. World Scientific Publishing Co.; 2001, p. 329–380.

[2] Report on Project Management in NASA: Phase II of the Mars Climate Orbiter Mishap Report. 2000. URL: ftp://ftp.hq.nasa.gov/pub/pao/reports/2000/MCO_MIB_Report.pdf.

[3] Nuseibeh, B.. Ariane 5: Who Dunnit? IEEE Software 1997;14(3):15–16.

[4] Qamar, A.. Model and Dependency Management in Mechatronic Design. Ph.D. thesis; KTH Royal Institute of Technology; 2013.

[5] Friedenthal, S., Moore, A., Steiner, R.. A Practical Guide to SysML: the Systems Modeling Language. Morgan Kaufmann Publishers Inc.; 2011.

[6] Finkelstein, A.C., Gabbay, D., Hunter, A., Kramer, J., Nuseibeh, B.. Inconsistency Handling in Multiperspective Specifications. IEEE T Software Eng 1994;20(8).

[7] Finkelstein, A., Spanoudakis, G., Till, D.. Managing Interference. In: Joint Proceedings of the Second International Software Architecture Workshop (ISAW-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints' 96) on SIGSOFT'96 Workshops. ACM; 1996, p. 172–174.

[8] Nuseibeh, B., Easterbrook, S., Russo, A.. Leveraging Inconsistency in Software Development. Computer 2000;33(4):24–29.

[9] Mens, T., Van Der Straeten, R., DHondt, M.. Detecting and Resolving Model Inconsistencies using Transformation Dependency Analysis. In: Model Driven Engineering Languages and Systems. Springer; 2006, p. 200–214.

[10] da Silva, M.A.A., Mougenot, A., Blanc, X., Bendraou, R.. Towards Automated Inconsistency Handling in Design Models. In: Advanced Information Systems Engineering. Springer; 2010, p. 348–362.

[11] Schatz, B., Braun, P., Huber, F., Wisspeintner, A.. Consistency in Model-Based Development. In: Engineering of Computer-Based Systems. IEEE; 2003, p. 287–296.

[12] Van Der Straeten, R., Mens, T., Simmonds, J., Jonckers, V.. Using Description Logic to Maintain Consistency between UML Models. In: UML 2003 - The Unified Modeling Language. Modeling Languages and Applications. Springer; 2003, p. 326–340.

[13] Van Der Straeten, R., D'Hondt, M.. Model Refactorings through Rule-Based Inconsistency Resolution. In: Proceedings of the 2006 ACM Symposium on Applied Computing. ACM; 2006, p. 1210–1217.

[14] Mens, T., Van Der Straeten, R., Simmonds, J.. A Framework for Managing Consistency of Evolving UML Models. 2005.

[15] Herzig, S., Qamar, A., Reichwein, A., Paredis, C.J.. A Conceptual Framework for Consistency Management in Model-Based Systems Engineering. In: Proceedings of the ASME 2011 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference. 2011, p. 1329–1339.

[16] Qamar, A., Paredis, C.. Dependency Modeling and Model Management in Mechatronic Design. In: ASME 2012 Design Engineering Technical Conferences & Computers and Information in Engineering Conference. 2012, p. 1205–1216.

[17] Hehenberger, P., Egyed, A., Zeman, K.. Consistency Checking of Mechatronic Design Models. In: ASME 2010 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference. 2010, p. 1141–1148.

[18] Gausemeier, J., Schäfer, W., Greenyer, J., Kahl, S., Pook, S., Rieke, J.. Management of Cross-Domain Model Consistency During the Development of Advanced Mechatronic Systems. In: Proceedings of the 17th International Conference on Engineering Design (ICED'09); vol. 6. 2009, p. 1–12.

[19] Liu, W., Easterbrook, S., Mylopoulos, J.. Rule-Based Detection of Inconsistency in UML Models. In: Workshop on Consistency Problems in UML-Based Software Development, Dresden, Germany. 2002, p. 106–123.

[20] Giarratano, J.C., Riley, G.. Expert Systems. PWS Publishing Co.; 1998.

[21] Giese, H., Levendovszky, T., Vangheluwe, H.. Summary of the Workshop on Multi-Paradigm Modeling: Concepts and Tools. In: Models in Software Engineering. Springer; 2007, p. 252–262.

[22] Eppstein, D.. Subgraph Isomorphism in Planar Graphs and Related Problems. In: Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms. Society for Industrial and Applied Mathematics; 1995, p. 632–640.

[23] Gallagher, B.. Matching Structure and Semantics: A Survey on Graph-Based Pattern Matching. AAAI FS 2006;6.

[24] Barceló, P., Libkin, L., Reutter, J.L.. Querying Graph Patterns. In: Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. ACM; 2011, p. 199–210.

[25] Hazelrigg, G.A.. Fundamentals of Decision Making for Engineering Design and Systems Engineering. 2012.

[26] Bizer, C., Heath, T., Berners-Lee, T.. Linked Data - The Story So Far. Int J Semant Web Inf 2009;5(3):1–22.

[27] RDF Primer. 2004. URL: http://www.w3.org/TR/rdf-primer/.

[28] OWL 2 Web Ontology Language Primer (Second Edition). 2012. URL: http://www.w3.org/TR/owl2-primer/.

[29] Open Services for Lifecycle Collaboration (OSLC). 2013. URL: http://open-services.net/.

[30] Fu, J.. Pattern Matching in Directed Graphs. In: Combinatorial Pattern Matching. Springer; 1995, p. 64–77.

[31] Lingas, A.. Subgraph Isomorphism for Biconnected Outerplanar Graphs in Cubic Time. Theor Comput Sci 1989;63(3):295–302.

[32] Levesque, H.J., Brachman, R.J.. Expressiveness and Tractability in Knowledge Representation and Reasoning. Comput Intell 1987;3(1):78–93.