Model consistency as a heuristic for eventual correctness

Reference:
David Istvan, Vangheluwe Hans, Syriani Eugene.- Model consistency as a heuristic for eventual correctness
Journal of Computer Languages - ISSN 2665-9182 - 76(2023), 101223
Full text (Publisher's DOI): https://doi.org/10.1016/J.COLA.2023.101223
To cite this reference: https://hdl.handle.net/10067/1992120151162165141

Model consistency as a heuristic for eventual correctness

Istvan David[a,*], Eugene Syriani[a], Hans Vangheluwe[b,c]

[a] *DIRO, Université de Montréal, Canada*
[b] *University of Antwerp, Belgium*
[c] *Flanders Make, Belgium*

Abstract

Inconsistencies between stakeholders' views pose a severe challenge in the engineering of complex systems. The past decades have seen a vast number of sophisticated inconsistency management techniques being developed. These techniques build on the common idea of "managing consistency instead of removing inconsistency", as put forward by Finkelstein. While it is clear what and how to do about inconsistencies, it is less clear why inconsistency is particularly useful. After all, it is the correctness of the system that should matter, as correctness is the end-user-facing quality of the product. In this paper, we analyze this question by investigating the relationship between (in)consistency and (in)correctness. We formally prove that, contrary to intuition, consistency does not imply correctness. However, consistency is still a good heuristic for eventual correctness. We elaborate on the consequences of this assertion and provide pointers as to how to make use of it in the next generation of inconsistency management techniques.

Keywords: model consistency, heuristics, correctness, model-based systems engineering, collaborative modeling

---

1. Introduction

Properly managing inconsistencies—that is, situations when two or more statements can be made that are not jointly satisfiable [1]—has been a grand challenge in software and systems engineering for decades. This challenge is vastly exacerbated in the engineering of heterogeneous systems which requires a coordinated interplay among stakeholders of disparate domains. In such settings, the lack of common vocabulary and modeling languages renders the detection of inconsistencies a particularly challenging task. The inappropriate management of inconsistencies, in turn, leads to incorrect products, potentially resulting in costly and even catastrophic results [2]. In this context, inconsistencies at any point of the engineering process indicate potential risks to the correctness, and great effort is invested into their resolution. Typically, inconsistencies are considered parts of the verification and validation (V&V) process of systems engineering [3]. The extended time between the introduction of inconsistencies and V&V activities adversely affects the cost factors of repairing inconsistencies.

As put forward by Finkelstein [4] over twenty years ago: «*Rather than thinking about removing inconsistency we need to think about "managing consistency"*». Promoting inconsistency as a first-class notion in distributed engineering settings facilitates explicit reasoning about the nature, causes, and implications of inconsistency before deciding how to treat them. This is contrary to simply removing inconsistency as close to the source and as soon as possible. There are obvious benefits to such a mindset, as evidenced by the numerous techniques of inconsistency tolerance, analysis, and the wide array of holistic management techniques [5, 6, 7].

While the body of knowledge on inconsistency management is clear about what and how to do about inconsistencies, it is not entirely clear *why* inconsistency is particularly useful. After all, what matters is the *correctness* of the product to be delivered. It is correctness that has to be ensured, and it is the lack of correctness that makes the end-user question the quality of the product.

---

[*] Corresponding author
Email address: `istvan.david@umontreal.ca` (Istvan David)

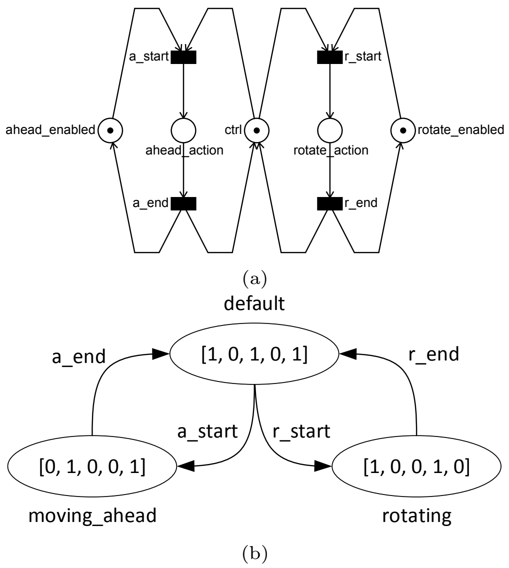
In this paper, we investigate the relationship between (in)consistency and (in)correctness and shed light on why and how the notion of inconsistency should be used to create more efficient engineering processes while still converging to an eventually correct product. We show that **consistency does not imply correctness**, that is, consistent models can still produce incorrect results. Rather, consistency—for the better or worse—is a mere **heuristic** to eventual correctness. We provide formal proof of both of these assertions and discuss their implications. We conclude that overcommitting to retaining consistency in the hope to ensure eventual correctness needlessly impairs the performance of the underlying engineering process. Our observations provide formal validation of the above-quoted proposition by Finkelstein [4].

The rest of this paper is structured as follows. In Section 2, we present the running example we use throughout the paper for demonstration purposes. In Section 3, we give a brief overview of the background relevant to our work. In Section 4, we formalize the concepts of correctness and consistency in terms of ontological properties and prove that consistency does not imply correctness. In Section 5, we map the formal concepts of correctness and consistency onto the definition of heuristics by Romanycia and Pelletier [8] and show that consistency is indeed a heuristic to eventual correctness. In Section 6, we discuss some of the consequences and results of our formal framework. Finally, in Section 7, we draw the conclusions and identify potential future directions.

2. Running example

To illustrate our points throughout this paper, we rely on the running example of an industrial line follower robot.

Line follower robots [9] are autonomous vehicles that move along a line, typically drawn on the floor. Line follower robots are frequently used in industry settings, especially in plants and production facilities, to carry heavy or dangerous payloads between two locations. In our example, the robot has two movement modes: (i) move ahead and (ii) change direction by rotating on omnidirectional wheels. The engineers of the robot are required to build a *safe* robot, which entails three required properties.



(a)

(b)

Figure 1: Petri net model of the robot and its reachability graph as viewed by the safety engineers.

Motion safety. The robot will typically carry large amounts of payloads on it, with a high center of gravity. Executing the two movement modes simultaneously might render the payload unstable. Thus, the engineers must ensure that the two movement modes are not executed simultaneously.

Mission safety. The robot alternates between the two motion modes automatically, according to the line it follows. If the robot gets stuck in one motion mode, it may abandon the line, resulting in hazardous and costly situations. Thus, the engineers must ensure that the robot cannot get stuck in one motion mode.

Interface safety. Superfluous states in the robot's state space allow for later design phases to introduce unwanted behavior and jeopardize the integrity of the behavioral model of the robot. Thus, the engineers must ensure that only the required functionality is present in the models of the robot.

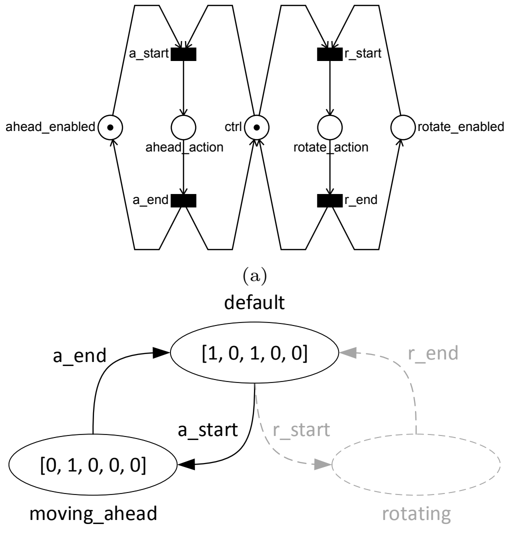
Modeling and analysis. The engineers decide to use Petri nets [10] to model the behavior of the robot and check the three required properties. They model the behavior of the robot as shown in Figure 1a. The *ahead_ enabled* and *rotate_ enabled* states of the net denote the states of the robot in which it can perform the ahead and rotate moving modes, respectively. The *ahead_ action* and *rotate_ action* states denote the states of the robot in which it moves ahead and rotates, respectively. The *ctrl* state controls which action is performed by allowing the firing of the *a_ start* ("start ahead movement") and *r_ start* ("start rotate movement") transitions. A marking of the Petri net is the distribution of its tokens in its states. The marking is given by the vector in which the *n*th element denotes the number of tokens in *n*th state of the Petri net. The initial marking in Figure 1a is [1, 0, 1, 0, 1], modeling the default configuration of the robot in which both move modes are enabled (i.e., the controller can activate any of them).

To be able to express the three required properties, the engineers construct the *reachability graph* of the Petri net, as shown in Figure 1b. The reachability graph of a Petri net is a directed graph $G = (V, E)$ in which each vertex $v \in V$ represents a marking of the Petri net, and each edge $e \in E$ represents a transition between two markings [10]. The marking in vertex v is stored in a vector $[v_1, v_2 ... v_n]$ with each element of the vector corresponding to one particular place of the Petri net and representing its current marking. For example, in Figure 1b, the *default* state represents the *default* marking of the Petri net in Figure 1a. Firing transition *a_ start* brings the marking [1, 0, 1, 0, 1]—i.e., the *default* state of the robot—to marking [0, 1, 0, 0, 1]—i.e., the *moving_ ahead* state of the robot. For convenience, we refer to the marking of place i as the ith element of v and denote it as $v[i]$. For example, in Figure 1b, *moving_ahead*[2] = 1 and *moving_ahead*[4] = 0. This allows us to express properties about the state of the robot in an algebraic way.

Following this formalization, the properties are formulated as follows.

p_1 – *motion is safe.* This property is expressed as the inability to exhibit the *ahead_ action* and *rotate_ action* states simultaneously. That is, there must not exist any vertex in the reachability graph R that encodes a marking in which the second and fourth places are both marked. Formally, $\nexists v \in V(R) : v[2] + v[4] > 1$, where *V(R)* denotes the set of vertices of reachability graph R.

p_2 – *mission is safe* This property is expressed by the lack of deadlock in the Petri net. A Petri net is deadlocking if it exhibits a state in which no transitions can fire. The reachability graph encodes such states as a vertex without an outgoing edge. Thus, the property is formally expressed as $\forall v_i \in V(R)\, \exists v_j \in V(R), v_i \neq v_j : (v_i, v_j) \in E(R)$, where *V(R)* and *E(R)* denote the set of vertices and edges of reachability graph R, respectively; and $(v_i, v_j) \in E(R)$ denotes an edge between vertices v_i and v_j.



(a)

(b)

Figure 2: Petri net model of the robot and its reachability graph as viewed by the configuration engineer.

p_3 – *interface is safe.* This property is expressed by the lack of places in the Petri net that are never marked. Such places can be identified by checking whether there exists at least one state in the reachability graph in which the place is marked. If there is no such state, the place is indeed never marked. Thus, the property is formally expressed as $\forall i \in \mathbb{N}, 1 \leq i \leq |S(P)| : \exists v \in V(R) : v[i] > 0$, where $S(P)$ denotes the set of places of Petri net P and *V(R)* denotes the set of vertices of reachability graph R.

Inconsistencies. The configuration engineer working in parallel with the safety engineers changes the default configuration of the robot. In the new configuration, only the *ahead* movement mode is enabled. The resulting Petri net and its reachability graph are shown in Figure 2.

The new model does not satisfy p_2 – as states *rotate_ enabled* and *rotate_ action* can never be marked; and p_3 – as these states are superfluous in the model. At this point, the model of the configuration engineer in Figure 2 and the model of the safety engineers in Figure 1 are inconsistent with each other with respect to properties p_2 and p_3.

In Sections 3-4 we elaborate on these inconsistencies in detail.

3. Background and related work

In this section, we overview the background of our work. We discuss the notion of inconsistency are the typical techniques to manage its undesired effects (Section 3.1). Then, we discuss the notion of *ontological property* which is the basis of reasoning about inconsistencies in our work (Section 3.2). Finally, we briefly overview model-driven engineering (MDE), a domain particularly vulnerable to inconsistencies (Section 3.3).

3.1. Inconsistencies and their management

Inconsistency is a state in which elements of different models make assertions that are not jointly satisfiable [1]. Manipulating models in multi-view and multi-paradigm settings naturally causes inconsistencies in models due to the *overlap* between the *shared concerns* of stakeholders, and the resulting overlap between their views and models [11]. As one view changes a shared element, the change has to be propagated to the other views that share the same element, otherwise, an inconsistency will occur [12]. These shared elements are not necessarily of syntactic nature. Often, they can only be observed in the *semantic domain* of the union of models as the *ontological properties* of the system—especially in multi-domain settings where different stakeholders operate with vastly different languages [13]. In the running example, such ontological properties are the three safety properties of the system. They are not expressed syntactically at the level of Petri nets, but rather, as structural properties of the reachability graph.

The two main types of inconsistency management approaches are prevention and the allow-and-resolve. Prevention aims to avoid inconsistent situation altogether. The applicability of preventive techniques has been demonstrated in the engineering of complex heterogeneous settings, e.g., by means of design contracts and ontological reasoning [14]. Lately, preventive techniques have been proven effective in real-time collaborative modeling settings as well [15]. Furthermore, preventive inconsistency management techniques have been well-researched in database systems [16] previously. A more permissive approach to managing inconsistencies is allowing them to emerge, and treating inconsistencies with the subsequent activities of detection and resolution [17].

Various forms of graph-based reasoning are a natural choice for inconsistency detection and resolution in MDE, where models typically adhere to graph semantics. Correspondence models are often used to relate elements of two or more models. Once a correspondence model is established, inconsistencies between the two graphs can be detected and in more advanced scenarios, repair actions can be put in place as well. The utility of correspondence models has been demonstrated in multi-disciplinary settings [18, 19]. Triple Graph Grammars (TGG) [20] improve on correspondence models by supporting bi-directional synchronization, with the possibility of incremental model updates [21]. TGG have seen success in cross-domain consistency management as well [22]. Such techniques are important enablers in the development of multi-disciplinary engineering tools [23, 24]. Fully automated model synchronization is not always feasible and human involvement is required. In such cases, the human stakeholder can be assisted by automatically generated editing hints [25] or quick-fixes [26]. Rule-based approaches are often used in combination with correspondence models [27] with the added benefit of utilizing rule engines [28], declarative languages [29, 30], and logic solvers [31, 32] to automate detection and synchronization. Design-space exploration (DSE) has been used as a more complex form of rule-based model repair, in which optimal sequences of model repair actions are identified by smart search heuristics [33].

In some cases, additional inconsistency tolerance techniques are employed between detection and resolution. Inconsistencies might be transient by nature, i.e., can get resolved naturally as the engineering process evolves. Equipping inconsistencies with state [5] and representing models as a sequence of operations [29] are the most typical approaches. The benefits of temporal inconsistency tolerance in MVM have been demonstrated by Easterbrook et al. [6]. Tolerating inconsistencies decouples the viewpoints and introduces flexibility in the design process as deciding upon when to resolve inconsistencies is the responsibility of the owner of the view.

While the state of the art of inconsistency management is substantial, the vast majority of approaches operate at the level of syntax. This is especially clear in graph-based approaches, in which the basis of reasoning is the abstract syntax. In contrast, semantic approaches rely on the assumption that inconsistencies may not surface at the level of syntax in time and therefore, treating them at level of syntax might not be feasible. Therefore, the semantics—the "meaning"—of models needs to be externalized and promoted to a first-class citizen. This is typically achieved by employing various forms of ontologies [34]. Ontologies are structured and organized representations of domain knowledge and enable reasoning over multiple domains. As a consequence, ontologies are especially useful in multidisciplinary settings [35]. Tagging model elements with their domain-specific interpretation has been suggested by Spanoudakis et al. [1] to enrich models with semantic elements and establish an ontology for the engineering endeavor. By that, overlaps across domain concepts can be detected irrespective of the (modeling) language in which they are primarily expressed. More advanced approaches automate the extraction of ontological concepts, e.g., Bayesian inference [3]. Once an ontology is established, automated reasoners can be used to detect inconsistencies [14].

3.2. Ontological properties

The imprecise or vague semantics of modeling languages are often to blame for unnoticed overlaps between concerns [36]. Ensuing inconsistency often does not manifest at the level of syntax, but remains hidden in the *semantic domain* [37]. In the running example, the inconsistency between the configuration engineer and the safety engineers remains hidden at the level of the Petri net models. The actual inconsistency is discovered only when investigating the meaning of the two Petri nets, e.g., by translating them to their respective reachability graphs. In practical scenarios, checking a property often requires more costly property checks, e.g., building a physical prototype of the system and testing its behavior under realistic physical conditions.

Apparently, detecting inconsistencies at the level of syntax might not be sufficient and often, the management of inconsistencies must be approached at the level of *semantic properties*.

The term *property* is vastly overloaded already in computer science. UML[1] considers properties a mere named "structural feature". Some object-oriented languages (such as C#) consider class members with a purpose between an attribute (or field) and a method a property.[2] In our terms, a property is a descriptor of a materialized object or concept that can be used to classify the said object or concept into ontological classes. In the running example, p_1 can be used to classify line follower robots into the *safe* and *not safe* classes. It is then expected, that two objects in the same class are similar in terms of the classifying property[14]. For example, a company might be interested in acquiring only safe line followers; but it does not matter which specific instance they acquire as long as the instances belong to the same class of *safe* line followers.

Throughout the paper, we maintain the view that properties are strictly categorical (i.e., they concern what something is like in their materialized self), and every dispositional property (i.e., what something can be or what abilities something possesses) can be reduced to categorical ones [38, 39]. That is, classifying an object by a property does not require a disposition to decide whether the property holds, but rather, properties are unconditional within a specific validity frame [40]. For example, the safety properties in the running example are all categorical properties of the system, because their satisfaction does not depend on any specific disposition—cf. "the system is safe when the weather is sunny". Should there exist a safety property related to the weather, that property can be turned into a categorical property by extending the validity frame of the model to entail additional physical conditions, such as temperature and precipitation, and positing the property in this new validity frame. This convention allows for describing properties in linguistic terms and evaluating the belonging of an object to a specific ontological class by a function that maps to a Boolean algebra.

3.3. Model-driven engineering

Model-Driven Engineering (MDE) [41] advocates *modeling* the system before it gets realized. This way, the relevant properties of the eventual system can be computed beforehand, allowing for improved design quality.

MDE aims to leverage the mechanism of abstraction to provide succinct representations of the underlying phenomena. Models are typically devel-

---

[1] `https://www.omg.org/spec/UML/2.5.1`

[2] `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/`

oped by means of general-purpose modeling languages (such as UML [42]) or domain-specific modeling languages (DSL) [43]. Models are used for the validation and verification of specific properties, such as safety, security, and performance before the system is assembled. Specifically, this assembly step is largely automated by code generation [44]. Recent improvements in MDE, such as low-code [45] and no-code platforms [46] can even generate the full code base from models.

Due to the complexity of nowadays engineered systems, their modeling is not an individual endeavor anymore but rather, a collaborative effort by multiple stakeholders [47, 48]. Such collaborative endeavors typically involve stakeholders from vastly different domains, who approach the modeled system from their own viewpoints. *Multi-view modeling (MVM)* advocates decomposing models into multiple views that are concerned with specific aspects of the system [49]. The ISO/IEC/IEEE 42010:2011 standard [50] defines a view as a set of concerns of specific stakeholders and viewpoints as the specification of conventions utilized to construct a view. This standard has been heavily relied on in MDE [51]. In the running example, the *safety* view supports a select group of stakeholders to reason about the safety properties of the system. This view includes three specific concerns of safety (motion, mission, interface), and defines methods to reason about these concerns (Petri nets and their properties). Another view could be, for example, the performance view. Such a view could be concerned with the behavioral characteristics of the line follower and could be supported with stochastic Petri net models (Petri nets augmented with statistical distributions on their transitions).

MVM has been shown to be an effective approach in several complex domains, such as cyber-physical systems [52]. Views can belong to different domains, i.e., they may represent various aspects of the single underlying model in different formalisms and on different levels of abstraction. The usage of multiple views fosters collaboration among multiple stakeholders. However, they introduce the threat of stakeholder views diverging and becoming inconsistent [53]. By the classification of Corley et al. [54], inconsistencies in MVM settings can manifest between views or between models to which the views correspond. The synchronization of views has been traditionally approached using correspondence models, such as pivot models [23] and bidirectional model transformations by triple-graph grammars [20]. This paper provides a general formal framework to reason about consistency and correctness in MVM settings.

4. Correctness and consistency

In this section, we provide a formal definition of correctness and consistency, in terms of ontological properties. Our formal system relies on first-order logic. However, as remarked at multiple points, extensions, such as intuitionistic logic [55] and description logic [7] often allow for different interpretations of correctness and consistency.

As outlined in Section 3, requirements are used to obtain the properties the final product must satisfy. From this point on, we assume an appropriate mapping from requirements to the properties and approach the problem of (in)consistency management in terms of properties only. To do so, we will use the concepts shown in Figure 3.

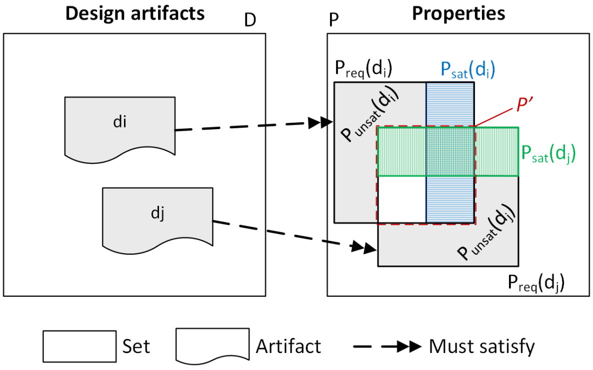


Figure 3: The relationship between properties and design models.

4.1. Preliminaries

Let P denote the set of properties a system must satisfy in order to consider it correct. For our purposes, we consider two design artifacts, $d_i, d_j \in D$. $P_{req}(d_i) \subset P$ and $P_{req}(d_j) \subset P$ denote the subsets of properties required to be satisfied by design artifacts d_i and d_j, respectively.

4.1.1. Design and completeness

Definition 1 (Design) *The collection of design artifacts* $(d_n)_{n \in N}$ *models is said to be design* D. *That is,* $D = \bigcup d_n$.

The design, sometimes called the virtual product or the single underlying model (SUM) [56], is the overall abstract representation of the eventual system. In this work, we assume an ideal assembly process that translates the design to the eventual system and consider any correctness-related issues in this assembly process out of the scope. This allows us to treat the design as the faithful proxy of the eventual product and to investigate the correctness of the eventual system by the correctness of the design. We make no assumptions about the overlap between the design artifacts.

Example. The design of the system in the running example (Section 2) is the collection of design artifacts in Figure 1a and Figure 2a, i.e., the two Petri nets.

Definition 2 (Complete design) *Design* $D = \bigcup d_n$ *is said to be complete iff* $P \setminus \bigcup_{\{1..n\}} P_{req}(d_n) \equiv \varnothing$.

That is, there are no properties of the system that are not required to be satisfied by at least one design artifact. Only by a complete design can one prove the correctness of the system. This definition is not to be confused with Gödel's notion of syntactic and semantic (in)completeness of formal systems [57] that are concerned with provability. Our definition is a mere reflection on the quality of the design and whether it addresses every stakeholder concern—i.e., required property.

Example. The design in the running example is considered a complete design by Definition 2, because properties p_1, p_2, and p_3 are all required to be satisfied by at least one design artifact. In fact, each of these properties are required to be satisfied both by the safety design (i.e., the Petri net in Figure 1a) and by the configuration design (i.e., the Petri net in Figure 2a).

Corollary 1. $\forall p \in P \, \exists d \in D : p \in P_{req}(d)$.

That is, for every property $p \in P$ exists at least one design artifact $d \in D$ of which p is a required property.

Example. In the running example, properties p_1, p_2, and p_3 are required properties of both the safety and the configuration design artifacts (shown in Figure 1a and Figure 2a, respectively).

Requirements management tools, such as Rational DOORS [58] and the IBM Engineering Requirements Quality Assistant [59] leverage this proposition when checking for completeness and calculating various completeness metrics.

Hereinafter in this paper, we consider a complete design. This allows us to assume that the correctness of the eventual system is equivalent to the correctness of the models.

4.1.2. Satisfaction of properties

Definition 3 (Satisfaction of a property) *A design artifact* $d \in D$ *is said to satisfy a property* $p \in P$ *iff* $[\![d]\!] \models p$, *where* $[\![\cdot]\!]$ *denotes the semantics.*

We assume that the satisfaction of a property per Definition 3 maps to the Boolean field, i.e., $\models : D \times P \rightarrow \mathbb{B}$, where $\forall b \in \mathbb{B} : \neg\neg b = b$ (excluded middle).

4.1.3. Satisfied and not satisfied properties

Let $P_{sat}(d) \subseteq P_{req}(d)$ and $P_{unsat}(d) \subseteq P_{req}(d)$ denote the satisfied and not satisfied required properties of d, respectively.

Definition 4 (Satisfied properties of design artifacts) $\forall p \in P_{sat}(d) \subseteq P_{req} : [\![d]\!] \models p$. *That is, every property* $p \in P_{sat}(d)$ *is satisfied by design artifact* d.

Example. In the running example, property p_1 is satisfied by both the safety design artifact d_s in Figure 1a and the configuration design artifact d_c in Figure 2a, as the invariant specified in the definition of the property $\nexists v \in V(R) : v[2] + v[4] > 1$ holds in both cases. Thus, $[\![d_s]\!] \models p_1$ and $[\![d_c]\!] \models p_1$.

Definition 5 (Not satisfied properties of design artifacts) $\forall p \in P_{unsat}(d) : [\![d]\!] \not\models p$. *That is, every property* $p \in P_{unsat}(d)$ *is not satisfied by design artifact* d.

Example. In the running example, property p_2 is not satisfied by the configuration design artifact d_c in Figure 2a, as the invariant specified in the definition of the property $\forall v_i \in V(R) \, \exists v_j \in V(R), v_i \neq v_j : (v_i, v_j) \in E(R)$ does not hold. Similarly, property p_3 is not satisfied by d_c either, as the invariant specified in the definition of the property $\forall i \in \mathbb{N}, 1 \leq i \leq |S(P)| : \exists v \in V(R) : v[i] > 0$ does not hold. Thus, $[\![d_c]\!] \not\models p_2$ and $[\![d_c]\!] \not\models p_3$.

Some key properties of property satisfaction include completeness and unambiguity.

Definition 6 (Completeness of property satisfaction) $P_{req}(d) \equiv P_{sat}(d) \bigcup P_{unsat}(d)$.

That is, every required property of $d \in D$ is either satisfied or not satisfied by d.

Definition 7 (Unambiguity of property satisfaction) $P_{sat}(d) \bigcap P_{unsat}(d) \equiv \varnothing$.

That is, a property cannot be satisfied and not satisfied by $d \in D$ simultaneously.

Hereinafter, we consider complete and unambiguous property satisfaction of design artifacts.

4.2. Correctness

Definition 8 (Correctness of a design artifact) *Design artifact d is said to be correct with respect to its set of required properties $P_{req}(d)$ iff $\forall p \in P_{req}(d) : [\![d]\!] \models p$.*
We use the notation $\rho(d)$ to denote the correctness of a design artifact.

Example. In the running example, the safety design artifact in Figure 1a, here denoted as d_s is a correct design artifact because it satisfies every required property. However, the configuration design artifact in Figure 2a is incorrect, as it does not satisfy properties p_2 and p_3.

We extend Definition 8 to the overall design. We consider a design correct if and only if it meets all the requirements. If at least one requirement is not met, the design is considered an incorrect product.

Definition 9 (Correctness of a design) *Design D is said to be correct with respect to its set of required properties $\bigcup_{\{1..n\}} P_{req}(d_n)$ iff $\forall p \in \bigcup_{\{1..n\}} P_{req}(d_n)\, \forall d \in D : p \in P_{req}(d) \Rightarrow [\![d]\!] \models p$.*
We use the notation $\rho(D)$ to denote the correctness of a design and we assume $\rho : D \times P \rightarrow \mathbb{B}$, i.e., it evaluates to boolean.

That is, the overall design is correct if every design artifact satisfies its required properties.

Example. In the running example, the overall design is composed of the design artifacts in Figure 1a and Figure 2a, here denoted by d_s and d_c, respectively. While d_s satisfies every required property, d_c does not (see the example under Definition 8) and therefore, the overall design is incorrect.

4.3. Consistency

Consistency is inextricably linked to (i) at least two assertions that disagree about (ii) a property. Thus, we formalize consistency as follows.

Definition 10 (Consistency of two design artifacts w.r.t. a property) *Design artifacts $d_i, d_j \in D$ are said to be consistent w.r.t to $p \in P' \equiv P_{req}(d_i) \bigcap P_{req}(d_j)$ iff $[\![d_i]\!] \models p \Leftrightarrow [\![d_j]\!] \models p$.*
If it is needed, we use the notation $\sigma_p(d_i, d_j)$ to denote the mutual consistency of design artifacts per property p and we assume $\sigma : D \times D \times P \rightarrow \mathbb{B}$, i.e., it evaluates to boolean.

Example. In the running example, the safety model and the configuration model are consistent with respect to p_1, as they both satisfy it.

The above definition can be generalized to the set of overlapping properties P'.

Definition 11 (Consistency of two design artifacts w.r.t. a set of properties) *Design artifacts $d_i, d_j \in D$ are said to be consistent w.r.t to the set of properties $P' \equiv P_{req}(d_i) \bigcap P_{req}(d_j)$ iff $\forall p \in P' : [\![d_i]\!] \models p \Leftrightarrow [\![d_j]\!] \models p$.*
If it is needed, we use the notation $\sigma^*_P(d_i, d_j)$ to denote the mutual consistency of design artifacts per the set of properties P. Again, we assume $\sigma^* : D \times D \times P \rightarrow \mathbb{B}$, i.e., it evaluates to boolean.

That is, two design artifacts are said to be consistent *with respect to a set of properties* if they satisfy exactly the same properties of the set. Due to Definition 7, either both design artifacts satisfy the property or jointly do not satisfy it. An inconsistency arises when exactly one of the two artifacts satisfies the property.

It is easy to see that Definition 10 is a special case of Definition 11 with $P' = \{p\}$.

4.4. Consistency $\Rightarrow$ correctness?

Table 1 shows how the satisfaction and dissatisfaction of the required properties $p \in P_{req}(d_i) \bigcap P_{req}(d_j)$ by two design artifacts d_i and d_j can lead to their (in)consistency, and the (in)correctness of the overall design $D = \{d_i, d_j\}$.

Table 1 yields four cases we investigate below.

Table 1: Consistency does not imply correctness.

	$[\![d_i]\!] \models p$	$[\![d_j]\!] \models p$	$\sigma_p(d_i, d_j)$	$\rho(D)$
(1)	✓	✓	✓	?
(2)	✓	×	×	×
(3)	×	✓	×	×
(4)	×	×	✓	×

Inconsistent and incorrect (Cases 2-3). If d_i satisfies p and d_j does not (case 2), or the other way around (case 3), the two design artifacts are inconsistent w.r.t p. This also means that there is at least one required property $p \in P'$ that is not satisfied by d_j (case 2) or d_i (case 3), and therefore, the overall design D is in an incorrect state.

Consistent and potentially correct (Case 1). If both d_i and d_j satisfy p, they are consistent as per Definition 10. This, however, does not guarantee correctness, unless $P_{req}(d_j) \setminus P' \equiv \emptyset \equiv P_{req}(d_j) \setminus P'$, i.e., if $P_{req}(d_i) \equiv P_{req}(d_j)$. Apart from this corner case, in which the two design artifacts have to satisfy exactly the same set of properties, neither correctness or incorrectness can be proved from the premise $[\![d_i]\!] \models p \wedge [\![d_j]\!] \models p$. The proof is trivial as from $P_{req}(d_i) \setminus P' \not\equiv \emptyset$ it follows that a property $p \in P_{req}(d_i) \setminus P'$ may exist such that $[\![d_j]\!] \not\models p$, rendering design D incorrect. However, this is still the *only* case that can lead to a correct product.

Consistent but incorrect (Case 4). Perhaps the most interesting case is the last one. If d_i and d_j both do not satisfy p, they are still considered consistent. This follows from Definition 10. However, both design artifacts are incorrect, and consequently, design D is incorrect. In this case, even though the models seem to be consistent, at the end of the development process, the resulting product will be incorrect.

Example. In the context of the running example, consider now a configuration model M_2 which is similar to M_1 shown in Figure 2, except let the initial marking of M_2 be [0, 0, 1, 0, 1]. That is, only the *rotate* motion mode is enabled by default, the *ahead* motion mode is not. M_2 would not satisfy p_2 and p_3, due to the reasons M_1 does not satisfy them (explained in Section 2). The not satisfied properties would render both models incorrect. However, the two models would be consistent with each other with respect to p_2 and p_3 (Case 4), and also with respect to p_1 (Case 1).

4.5. Consequences

The following conclusions can be drawn from Table 1.

Theorem 1. *Consistency is a necessary but not sufficient requirement for correctness.*

Formally:

$$\rho(D) \Rightarrow \sigma(d_i, d_j)\ (necessity);$$
$$\sigma(d_i, d_j) \nRightarrow \rho(D)\ (insufficiency).$$

We use Lemma 1 to prove Theorem 1.

Lemma 1. *Logical implication evaluates to false iff the antecedent is true and the consequent is false, i.e., true → false.*

Proof 1. *To prove $\rho(D) \Rightarrow \sigma(d_i, d_j)$ (necessity), we remark that there is only one case in Table 1 where $\rho(D)$ can be true, and that is case (1). However, the $\sigma(d_i, d_j)$ relationship, in this case, is true, and with a true consequent, the implication cannot be false.*
To prove $\sigma(d_i, d_j) \nRightarrow \rho(D)$ (sufficiency), it is enough to show that there is at least one case in Table 1 where the antecedent is true and the consequent is false. Case (4) is such a case. □

Theorem 2. *Inconsistency is a sufficient requirement for incorrectness.*

Formally:

$$\neg\sigma(d_i, d_j) \Rightarrow \neg\rho(D).$$

For the proof, we use Lemma 2.

Lemma 2. $\neg X \vee Y \vdash X \rightarrow Y$.

Proof 2. *Due to Lemma 2, $\neg\neg\sigma(d_i, d_j) \vee \neg\rho(D) \vdash \neg\sigma(d_i, d_j) \Rightarrow \neg\rho(D)$. Due to Definition 7, $\sigma(d_i, d_j) \vee \neg\rho(D) \vdash \neg\neg\sigma(d_i, d_j) \vee \neg\rho(D)$. We now show that $\sigma(d_i, d_j) \vee \neg\rho(D)$ always holds.*
From Definition 10, it follows that if either $[\![d_i]\!] \models p \wedge [\![d_j]\!] \models p$ (Case 1 in Table 1) holds or $[\![d_i]\!] \not\models p \wedge [\![d_j]\!] \not\models p$ (Case 4) holds, $\sigma(d_i, d_j)$ holds and consequently, $\sigma(d_i, d_j) \vee \neg\rho(D)$ holds.
From Definition 8, it follows that if either $[\![d_i]\!] \models p \wedge [\![d_j]\!] \not\models p$ (Case 2) holds or $[\![d_i]\!] \not\models p \wedge [\![d_j]\!] \models p$ (Case 3) holds, $\neg\rho(D)$ holds and consequently, $\sigma(d_i, d_j) \vee \neg\rho(D)$ holds. □

5. Consistency as a heuristic to correctness

While consistency does not imply correctness, it is still useful to think of consistency as a heuristic to correctness.

5.1. A definition of heuristic

Romanycia and Pelletier [8] define a heuristic as «any device, be it a program, rule, piece of knowledge, etc., which *one is not entirely confident will be useful* in providing a practical solution, but which *one has reason to believe will be useful*, and which is added to a problem-solving system in expectation that on average the performance will improve».

In this context, consistency is the device that, when added to the one problem-solving system, i.e., the engineering process, might be useful in achieving a practical solution, i.e. a correct system.

On the one hand, *one has a reason to believe consistency will be useful* in achieving correctness, because Theorem 2 states that the lack of consistency surely results in incorrectness. On the other hand, *one cannot be entirely confident consistency will be useful* in achieving the desired correctness, because, as Theorem 1 states, consistency alone is not a sufficient requirement for correctness. Formal evidence follows from the conditional probability of correctness under the condition of consistency. Based on Table 1:

$$0 < P(\rho(D) \mid \exists\sigma(d_i, d_j)) \leq 1, \textit{however} \qquad (1)$$
$$P(\rho(D) \mid \nexists\sigma(d_i, d_j)) = 0 \qquad (2)$$

Equation 1 corresponds to cases described either by row 1 or 4 in Table 1. Since row 1 *may* yield a correct design (the $\rho(D)$ column is not false or true), the probability of a correct design is greater than 0. The probability of correct design is still strictly less than 1, due to row 4 in Table 1 certainly yielding an incorrect design. In contrast, Equation 2, corresponding to cases in rows 2 and 3 in Table 1, shows that the probability of arriving at a correct product in inconsistent cases is 0.

5.2. Leveraging consistency as a heuristic

Treating consistency as a heuristic to correctness motivates and justifies putting regular consistency checks in place. Consistency checks, although often limited in effectiveness [60], are less costly to implement than correctness checks. Upon detecting inconsistencies among design artifacts, incorrectness can be assumed and proper mechanisms can be triggered. Since repair costs tend to increase sharply when incorrectness is addressed at later stages of a project [61, 62], the lower cost of occasional or even regular consistency checks is justified. Thus, by adding consistency to the *problem-solving* system, i.e., the engineering process, the *performance* of the engineering process is expected to improve on account of eliminating lingering errors early on and allowing for better economic outlooks.

5.3. Admissible and consistent heuristics

Admissibility and consistency are two key properties of heuristics.

A heuristic is said to be admissible if it never overestimates the goal. In our context, consistency is an admissible heuristic to correctness if it never overestimates the degree of correctness. Indeed, the admissibility of consistency as a heuristic to correctness follows from Theorem 1 as even a fully consistent design does not guarantee a correct design.

A heuristic is said to be consistent if it exhibits the trait of monotonicity. That is, by continuously improving consistency, correctness improves continuously as well. Unfortunately, since consistency is no guarantee of correctness, consistency is typically not a consistent heuristic to correctness. This follows from Theorem 1: even if consistency is fully restored, the system may remain in an incorrect state.

Thus, it can be concluded, that consistency is an admissible but not consistent heuristic to correctness. The benefit of consistency being admissible is that it can serve as a lower bound estimation of the effort needed to restore correctness. This allows defining quality gates that are operationalized through consistency metrics as thresholds. In the following, we show two of such consistency metrics.

5.4. Some examples

Here, we provide some typical examples of consistency models and metrics.

Heuristic 1: Number of inconsistent properties. The number of inconsistent properties is an admissible heuristic h to the correctness of the design. Formally, following the notations in Figure 3:

$$h(D) = \left|\left(P_{sat}(d_i) \ominus P_{sat}(d_j)\right) \bigcap P'\right|,$$

where $\ominus$ denotes the symmetric difference of two sets and $P' \equiv P_{req}(d_i) \bigcap P_{req}(d_j)$. This follows from the fact that an inconsistent property implies incorrectness (per Theorem 2) and therefore, restoring correctness takes *at least* as many steps as restoring the consistency of the properties. In practical terms, however, restoring correctness usually takes more steps, e.g., due to the challenges of resolution scheduling [63, 64].

This heuristic can be used as a lower bound estimation of the effort needed to restore correctness, and repair actions can be triggered after the heuristic crosses a predetermined threshold.

Heuristic 2: Trace distance of views. Heuristic 1 is based on counting binary satisfaction relationships: the heuristic is the sum of the number of inconsistent properties. Richer basis of reasoning and a more precise lower bound can be provided by quantified consistency measures, e.g., based on behavioral similarity [65] or domain-specific distance metrics [66]. Following our previous work [65], the *trace distance* of two properties p_1 and p_2 over a time window of length λ can be defined as

$$h(D) = \delta_\lambda(p_1, p_2) = \sum_{i=0}^{\lambda-1} \delta(p_1(i), p_2(i)),$$

where $p(i)$ denotes the ith observation of p.

Such a heuristic estimates incorrectness in a quantified fashion and gives hints about *how hard* it may be to restore correctness. In contrast, Heuristic 1 only gives hints about *how many* steps it may take to restore correctness, but not about the severity of those steps. Therefore, heuristics based on quantified (in)consistency metrics allow for better decisions as of when to execute repair actions. Furthermore, the temporal dynamics of Heuristic 2 also allow for *tolerating* inconsistencies as discussed in previous work [65]. Tolerance of inconsistencies, in turn, allows for engineering processes to temporarily deviate from overall correctness and incorporate such temporal deviations into the overall engineering endeavor.

6. Discussion

We now discuss some implications of Section 4, especially Theorem 1 and Theorem 2. Some important tooling aspects have been described previously by Finkelstein [4]. Here, we focus on the conceptual aspects of inconsistency management and their implied language aspects.

6.1. When and how to use these results?

The most important takeaway of this paper is that promoting (in)consistency to a first-class citizen in engineering processes allows for better management of (in)correctness. Although consistency does not imply correctness, it is still an admissible heuristic for it and as such, it allows for putting proper quality checks and repair actions in place. This result is best used in engineering processes in which V&V activities are particularly resource-intensive and costly, such as the engineering of mechatronic and cyber-physical systems. While the costs of regular correctness checks often cannot be justified in such settings, consistency-based quality checks offer a viable alternative. Such techniques can be used at various points of the systems or software engineering process. Perhaps the best example is the V-model [67], in which artifacts of the design phase are used in the system construction phase as well, allowing for consistency checks to be put in place throughout the entirety of the process. Its derivations, such as the Y-model [68] rely on automated correspondence between design and construction, further improving the utility of consistency checks along the process. Therefore, we advocate experts and business stakeholders, especially of such complex domains to incorporate regular and frequent consistency checks and correlate their results with the correctness of the system.

As shown in Heuristic 2 in Section 5.4, tolerance is a powerful enabler to better scaling engineering processes. However, tolerance is the most overlooked aspect of inconsistency management [13] and its support should improve by a large margin in the next generation of inconsistency management frameworks. We argue that tolerance is implicitly present in current inconsistency frameworks, as deciding about *when* to carry out a repair action inherently encodes some level of tolerance. By treating inconsistency as a first-class citizen, its tolerance aspect becomes more feasible to reason about and the enactment of inconsistency treatments can be further optimized [69]. Recent trends in model-driven software engineering, such as blended modeling [70] have highlighted the need for such techniques. Therefore, we recommend prospective researchers focus their attention on the various models and tooling aspects of inconsistency tolerance, especially in relation to system correctness.

6.2. Language requirements

To fully leverage the potential of promoting inconsistency to a first-class citizen, modeling and programming languages need to embrace this idea as well.

At the syntactic level, language features can be introduced that are suitable for expressing consistency rules. Such ideas have been explored in contract-based design [71], most notably in languages such as Eiffel [72] and the FOCUS method [73], with each of the approaches rooted in Hoare's axiomatic basis for computer programming [74]. However, such techniques are still sporadically used. Most languages provide contract-like features, such as assertions in Java and Python, but these elements are optional and cannot capture complex consistency rules. Additional syntactic facilities can be introduced to define tolerance rules and resolution procedures. However, these languages have to work at different meta-levels of the linguistic stack, and their usability would challenge current systems engineering methodologies. For example, it is not clear who should be responsible for capturing such consistency constraints. Due to the most concerning inconsistencies being situated in overlaps of views [11], it is also far from given that complex consistency rules can be fully understood and mapped by just one stakeholder.

Semantic techniques offer solutions to this problem. Thus, languages need to improve at this level as well. Ontologies [75] collect and organize concepts and allow for expressing relationships among them and properties in terms of description logic. Due to their domain-agnostic nature, ontologies are especially suitable for capturing complex concepts that give rise to inconsistencies in the overlaps of domain-specific views [11]. The integration of language engineering and ontology engineering has been first discussed by Kühne [76] in the context of separating the notion of linguistic and ontological conformance in multi-level modeling. Multi-layer ontologies allow for reusing general knowledge (e.g., laws of physics) and gradually augmenting those with more domain-specific knowledge (e.g., laws of mechanical engineering, laws of electrical engineering), while still allowing for identifying related concepts in different domains (e.g., an "engine" in the mechanical domain describes the same real concept as the "motor" in the electrical domain, w.r.t. a set of properties that, in turn, constitute the overlap between views). Lifting properties to the syntactic level has been shown to be an effective technique in the design of complex heterogeneous systems [77]. Such ontological facilities must remain hidden behind the syntax of languages and the related mechanisms (such as consistency checks) should be operationalized in the background, preferably without requiring human input or interaction. Given the computational complexity of such mechanisms, incremental linguistic structures are needed that ensure a swift evaluation of inconsistencies upon changes in the model or program.

6.3. Alternative formal frameworks

Throughout this paper, we have relied on first-order logic (FOL). However, other frameworks can be considered as the formal underpinning to inconsistency management, each with different benefits and challenges.

Description logic is a provable subset of FOL. While satisfiability is undecidable in FOL [78], description logic provides inference mechanisms that are decidable. The increased provability comes at the cost of expressiveness: the expressive power of description logic is situated between those of FOL and propositional logic. Still, this trade-off is often beneficial in consistency problems, as demonstrated, e.g., by Van der Straeten et al. [7] who define a subconcept-superconcept classification mechanism that is decidable and complete. A particularly useful feature of description logic is the distinction between statements on concept hierarchy—captured in terminological boxes (*TBox*)—and statements on relationships between concepts and individuals—captured in assertion boxes (*ABox*). This distinction enables the reasoner to be operationalized only on TBoxes (typically for classification reasoning), only on ABoxes (typically for instance reasoning), or both. The separation of terms also allows treating the inherent complexity of TBoxes separately and reusing TBox information with different ABoxes. An additional benefit of description logic is the lack of unique name assumption that allows for concepts with different names to be equivalent by inference. This aligns very well with stakeholders that possess different vocabularies, such as the ones in the engineering of complex heterogeneous systems. Finally, description logic assumes an open world, i.e., it does not assume the excluded middle (see Definition 3). While this property improves expressive power, it also increases the complexity of reasoning.

Modal logic encompasses multiple logic frameworks with the common trait of being able to distinguish between necessity and possibility. By unary modal operators $\Diamond p$ – possibility, and $\Box p$ – necessity, modal logic improves the expressiveness of first-order logic. This allows for the useful distinction between *knowing p* and *p being true.* Many inconsistency cases can be traced back to the lack of knowledge, e.g., due to miscommunication and misaligned vocabularies. The ability to explicitly denote awareness of axioms even without the ability to evaluate them improves the understanding of how knowledge is accessible to stakeholders [79] and as a consequence, improves the robustness of the engineering setting [80]. Furthermore, modal logic, and specifically, dynamic epistemic logic [81] naturally promotes the evolution of the knowledge base as new axioms are encountered [82]. This is a substantial improvement over first-order logic that aligns logic-based reasoning with realistic engineering settings better. However, the improved expressiveness comes at the price of computational complexity. Due to this complexity, modal logic, especially temporal logic frameworks—such as linear temporal logic (LTL) [83] and computation tree logic (CTL) [84]—are primarily used in verification, i.e., in proving correctness. We foresee future research focusing on extending modal logic to inconsistency management based on the vast body of knowledge available on verification.

Intuitionistic logic. Although its discourse is largely missing from inconsistency management, intuitionistic logic [55] aligns well with our understanding of knowledge in engineering processes. Intuitionistic logic rejects the excluded middle of classical logic, i.e., does not assume that $\neg\neg p = p$. In classical logic, such as first-order logic, if a proof exists that p is true, the interpretation of $\neg p$ is ambiguous. Both the interpretation of "*there is no proof of p*" and the interpretation of "*there is proof of not-p*" are acceptable. To properly distinguish between the two cases, intuitionistic logic only accepts assertions as true that can be proved as such. That is, p being provably true does not automatically imply *not-p* being false. Rather, *not-p* has to be proven on its own right, i.e., *not-p* has to evaluate to true.

This distinction cleans up the semantics of negation and works well with modal propositions, in which often one only *knows p*, but cannot decide its truth value. Similarly, in inconsistency management, it is often the case that "provably consistent" does not imply "not inconsistent". In our formal framework, we defined consistency of models *with respect to* a set of properties. In intuitionistic logic, even if a proof of consistency exists, one cannot be entirely sure that two models are not inconsistent w.r.t. another set of properties. This, in turn, aligns well with dynamic epistemic logic [81] and forces the user of the framework to maintain an open world assumption: since the set of axioms is subject to change, all that current provability of consistency buys is $\Diamond p$ (possibly consistent), but not $\Box p$ (necessary consistent). Again, the improved expressiveness comes at the price of computational complexity. The lack of excluded middle eradicates the mechanism of proof by contradiction from the formal framework, and by extension, widely used reasoning and explanation techniques such as the generation of counterexamples are unavailable.

Summary. A frequent criticism against inconsistency management frameworks tapping into the semantic domain of models is their cumbersome usability and limited applicability [60]. The frameworks presented in this section provide substantially increased expressiveness to describe sophisticated consistency mechanisms and by that, they can contribute to the better applicability of the next generation of inconsistency management frameworks. However, as emphasized, with the improved expressiveness, reasoning mechanisms become more computationally demanding as well. We advocate future research focusing on (i) the trade-off between expressiveness and computational complexity, and (ii) multi-paradigm methods in which different formal frameworks can be used to underpin inconsistency management systems.

7. Conclusion

In this paper, we have validated the generally accepted philosophy of consistency management, that instead of simply removing consistency from an engineering process, one should reason about properly managing inconsistency. We have shown formal proofs of consistency being an insufficient indicator of eventual correctness, and inconsistency being a sufficient indicator of eventual incorrectness. We have drawn the conclusion that over-committing to consistency might not be the best strategy in terms of costs and the end-to-end performance of the underlying engineering process. We suggested future directions to researchers of the topic, tool builders, and language engineers.

Future work will focus on the modalities of the presented formal framework under open-world and closed-world assumptions [85] and gaining a better understanding of modeling under uncertainty [86].

References

[1] G. Spanoudakis, A. Zisman, Inconsistency management in software engineering: Survey and open research issues, in: Handbook of Software Engineering and Knowledge Engineering: Volume I: Fundamentals, World Scientific, 2001, pp. 329–380.

[2] R. Lloyd, C. I. S. Writer, Metric mishap caused loss of nasa orbiter, CNN Interactive (1999).

[3] S. J. Herzig, C. J. Paredis, A conceptual basis for inconsistency management in model-based systems engineering, Procedia CIRP 21 (2014) 52–57, 24th CIRP Design Conference.

[4] A. Finkelstein, A foolish consistency: Technical challenges in consistency management, in: Database and Expert Systems Applications, 11th International Conference, DEXA 2000, Vol. 1873 of LNCS, Springer, 2000, pp. 1–5.

[5] R. Balzer, Tolerating inconsistency, in: Proceedings of the 13th International Conference on Software Engineering, IEEE Computer Society / ACM Press, 1991, pp. 158–165.

[6] S. Easterbrook, A. Finkelstein, J. Kramer, B. Nuseibeh, Coordinating distributed viewpoints: the anatomy of a consistency check, Concurrent Engineering 2 (3) (1994) 209–222.

[7] R. Van Der Straeten, T. Mens, J. Simmonds, V. Jonckers, Using description logic to maintain consistency between UML models, in: «UML» 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference, Vol. 2863 of LNCS, Springer, 2003, pp. 326–340.

[8] M. H. Romanycia, F. J. Pelletier, What is a heuristic?, Computational Intelligence 1 (1) (1985) 47–58.

[9] S. Sedhumadhavan, E. Niranjana, An analysis of path planning for autonomous motorized robots, International Journal of Advance Research, Ideas and Innovations in Tech6nology 3 (6) (2017) 1234–1257.

[10] W. Reisig, Petri Nets: An Introduction, Vol. 4 of EATCS Monographs on Theoretical Computer Science, Springer, 1985.

[11] J. Reineke, S. Tripakis, Basic problems in multi-view modeling, in: Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Vol. 8413 of LNCS, Springer, 2014, pp. 217–232.

[12] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, B. Nuseibeh, Inconsistency handling in multiperspective specifications, IEEE Transactions on Software Engineering 20 (8) (1994) 569–578.

[13] W. Torres, M. G. J. van den Brand, A. Serebrenik, A systematic literature review of cross-domain model consistency checking by model management tools, Softw. Syst. Model. 20 (3) (2021) 897–916.

[14] K. Vanherpen, et al., Ontological reasoning for consistency in the design of cyber-physical systems, in: 1st International Workshop on Cyber-Physical Production Systems, CPPS@CPSWeek 2016, IEEE Computer Society, 2016, pp. 1–8.

[15] I. David, E. Syriani, Real-time collaborative multi-level modeling by conflict-free replicated data types, Software & Systems Modeling (2022).

[16] P. R. Johnson, R. Thomas, Maintenance of duplicate databases, RFC 677 (1975) 1–10.

[17] I. David, A Foundation for Inconsistency Management in Model-Based Systems Engineering, Ph.D. thesis, University of Antwerp, Belgium (7 2019).

[18] A. Qamar, J. Wikander, C. During, A mechatronic design infrastructure integrating heterogeneous models, in: 2011 IEEE International Conference on Mechatronics, IEEE, 2011, pp. 212–217.

[19] A. Bhave, B. H. Krogh, D. Garlan, B. R. Schmerl, View consistency in architectures for cyber-physical systems, in: 2011 IEEE/ACM International Conference on Cyber-Physical Systems, ICCPS 2011, IEEE, 2011, pp. 151–160.

[20] A. Schürr, Specification of graph translators with triple graph grammars, in: Graph-Theoretic Concepts in Computer Science, 20th International Workshop, WG '94, Vol. 903 of LNCS, Springer, 1994, pp. 151–163.

[21] H. Giese, S. Hildebrandt, Incremental model synchronization for multiple updates, in: Proceedings of the Third International Workshop on Graph and Model Transformations, ACM, 2008, pp. 1–8.

[22] J. Gausemeier, W. Schäfer, J. Greenyer, S. Kahl, S. Pook, J. Rieke, Management of cross-domain model consistency during the development of advanced mechatronic systems, in: DS 58-6: Proceedings of ICED 09, the 17th International Conference on Engineering Design, Vol. 6, Design Methods and Tools (pt. 2), Palo Alto, CA, USA, 24.-27.08. 2009, 2009.

[23] A. A. Shah, A. A. Kerzhner, D. Schaefer, C. J. J. Paredis, Multi-view modeling to support embedded systems engineering in sysml, in: Graph Transformations and Model-Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday, Vol. 5765 of LNCS, Springer, 2010, pp. 580–601.

[24] A. Egyed, Automatically detecting and tracking inconsistencies in software design models, IEEE Trans. Software Eng. 37 (2) (2011) 188–204. `doi:10.1109/TSE.2010.38`.

[25] A. Hessellund, K. Czarnecki, A. Wasowski, Guided development with multiple domain-specific languages, in: Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Vol. 4735 of LNCS, Springer, 2007, pp. 46–60.

[26] Á. Hegedüs, Á. Horváth, I. Ráth, M. C. Branco, D. Varró, Quick fix generation for dsmls, in: 2011 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2011, Pittsburgh, PA, USA, September 18-22, 2011, IEEE, 2011, pp. 17–24. `doi:10.1109/VLHCC.2011.6070373`.

[27] S. M. Becker, A. Körtgen, Integration tools for consistency management between design documents in development processes, in: Graph Transformations and Model-Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday, Vol. 5765 of LNCS, Springer, 2010, pp. 683–718.

[28] Y. Xue, B. Feng, Checking validity of topic maps with drools, in: The 2nd International Conference on Information Science and Engineering, 2010, pp. 174–177. `doi:10.1109/ICISE.2010.5689569`.

[29] X. Blanc, I. Mounier, A. Mougenot, T. Mens, Detecting model inconsistency through operation-based model

construction, in: 30th International Conference on Software Engineering (ICSE 2008), ACM, 2008, pp. 511–520.

[30] J. Le Noir, O. Delande, D. Exertier, M. A. A. da Silva, X. Blanc, Operation based model representation: Experiences on inconsistency detection, in: Modelling Foundations and Applications - 7th European Conference, ECMFA 2011, Vol. 6698 of LNCS, Springer, 2011, pp. 85–96.

[31] C. Quinton, A. Pleuss, D. L. Berre, L. Duchien, G. Botterweck, Consistency checking for the evolution of cardinality-based feature models, in: 18th International Software Product Line Conference, SPLC '14, ACM, 2014, pp. 122–131.

[32] A. Rauzy, Y. Dutuit, Exact and truncated computations of prime implicants of coherent and non-coherent fault trees within aralia, Reliability Engineering & System Safety 58 (2) (1997) 127–144, eSREL '95. `doi:https://doi.org/10.1016/S0951-8320(97)00034-3`.

[33] I. David, J. Denil, K. Gadeyne, H. Vangheluwe, Engineering Process Transformation to Manage (In)consistency, in: Proceedings of the 1st International Workshop on Collaborative Modelling in MDE (COMMitMDE 2016) co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2016), Vol. 1717 of CEUR Workshop Proceedings, CEUR-WS.org, 2016, pp. 7–16.

[34] N. Guarino, D. Oberle, S. Staab, What is an ontology?, in: Handbook on ontologies, Springer, 2009, pp. 1–17.

[35] P. Hehenberger, A. Egyed, K. Zeman, Consistency checking of mechatronic design models, in: ASME 2010 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, American Society of Mechanical Engineers, 2010, pp. 1141–1148.

[36] Z. Huzar, L. Kuzniarz, G. Reggio, J. Sourrouille, Consistency problems in UML-based software development, in: UML Modeling Languages and Applications, «UML» 2004 Satellite Activities, Vol. 3297 of LNCS, Springer, 2004, pp. 1–12.

[37] D. Harel, B. Rumpe, Meaningful Modeling: What's the Semantics of "Semantics"?, Computer 37 (10) (2004) 64–72.

[38] D. M. Armstrong, The causal theory of properties: Properties according to shoemaker, ellis, and others, Philosophical Topics 26 (1/2) (1999) 25–37.

[39] J. Schaffer, Quiddistic knowledge, Philosophical Studies 123 (1) (2005) 1–32.

[40] B. P. Zeigler, Theory of Modeling and Simulation, John Wiley, 1976.

[41] D. C. Schmidt, Model-driven engineering, Computer-IEEE Computer Society 39 (2) (2006) 25.

[42] M. Fowler, UML distilled: a brief guide to the standard object modeling language, Addison-Wesley Professional, 2004.

[43] M. Fowler, Domain-specific languages, Pearson Education, 2010.

[44] J. Herrington, Code generation in action, Manning Publications Co., 2003.

[45] J. Cabot, Positioning of the low-code movement within the field of model-driven engineering, in: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, 2020, pp. 1–3.

[46] T. C. Lethbridge, Low-code is often high-code, so we must design low-code platforms to enable proper software engineering, in: International Symposium on Leveraging Applications of Formal Methods, Springer, 2021, pp. 202–212.

[47] I. David, K. Aslam, S. Faridmoayer, I. Malavolta, E. Syriani, P. Lago, Collaborative model-driven software engineering: A systematic update, in: 24th International Conference on Model Driven Engineering Languages and Systems, MODELS 2021, Fukuoka, Japan, October 10-15, 2021, IEEE, 2021, pp. 273–284.

[48] I. David, K. Aslam, I. Malavolta, P. Lago, Collaborative Model-Driven Software Engineering – A Survey of Practices and Needs in Industry, Journal of Systems and Software (2022).

[49] R. von Hanxleden, E. A. Lee, C. Motika, H. Fuhrmann, Multi-view Modeling and Pragmatics in 2020 – Position Paper on Designing Complex Cyber-Physical Systems, in: Large-Scale Complex IT Systems. Development, Operation and Management, LNCS.

[50] ISO/IEC/IEEE, Systems and software engineering – architecture description, ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000) (2011) 1–46.

[51] D. Broman, E. A. Lee, S. Tripakis, M. Törngren, Viewpoints, formalisms, languages, and tools Cyber-Physical Systems, in: Proceedings of the 6th International Workshop on Multi-Paradigm Modeling, MPM@MoDELS 2012, ACM, 2012, pp. 49–54.

[52] L. Zhang, Multi-view approach for modeling aerospace cyber-physical systems, in: 2013 IEEE International Conference on Green Computing and Communications (GreenCom) and IEEE Internet of Things (iThings) and IEEE Cyber, Physical and Social Computing (CPSCom), Beijing, China, August 20-23, 2013, IEEE, 2013, pp. 1319–1324. `doi:10.1109/GreenCom-iThings-CPSCom.2013.229`.

[53] D. Bork, A development method for the conceptual design of multi-view modeling tools with an emphasis on consistency requirements, Ph.D. thesis, University of Bamberg (2015).
URL `https://opus4.kobv.de/opus4-bamberg/frontdoor/index/index/docId/44613`

[54] J. Corley, E. Syriani, H. Ergin, S. Van Mierlo, Modern Software Engineering Methodologies for Mobile and Cloud Environments, no. 7, IGI Global, 2016, book section Cloud-based Multi-View Modeling Environments, pp. 120–139.

[55] D. van Dalen, Intuitionistic logic, in: Handbook of philosophical logic, Springer, 1986, pp. 225–339.

[56] C. Atkinson, D. Draheim, Cloud-aided software engineering: evolving viable software systems through a web of views, in: Software engineering frameworks for the cloud computing paradigm, Springer, 2013, pp. 255–281.

[57] T. Franzén, Gödel's theorem: an incomplete guide to its use and abuse, AK Peters/CRC Press, 2005.

[58] E. Hull, K. Jackson, J. Dick, Doors: a tool to manage requirements, in: Requirements engineering, Springer, 2002, pp. 187–204.

[59] A. Post, T. Fuhr, Case study: How well can ibm's" requirements quality assistant" review automotive requirements?, in: REFSQ Workshops, 2021.

[60] R. Jongeling, F. Ciccozzi, J. Carlson, A. Cicchetti, Consistency management in industrial continuous model-

based development settings: a reality check, Software and Systems Modeling (2022) 1–20.

[61] G. Tassey, The economic impacts of inadequate infrastructure for software testing, National Institute of Standards and Technology (2002).
URL `https://www.nist.gov/system/files/documents/director/planning/report02-3.pdf`

[62] K. Beck, Extreme programming explained: embrace change, addison-wesley professional, 2000.

[63] T. Mens, G. Taentzer, O. Runge, Detecting structural refactoring conflicts using critical pair analysis, in: Proceedings of the Workshop on Software Evolution through Transformations: Model-based vs. Implementation-level Solutions, SETra@ICGT 2004, Rome, Italy, October 2, 2004, Vol. 127 of Electronic Notes in Theoretical Computer Science, Elsevier, 2004, pp. 113–128. `doi:10.1016/j.entcs.2004.08.038`.

[64] H. Liu, Z. Ma, W. Shao, Z. Niu, Schedule of bad smell detection and resolution: A new way to save effort, IEEE Trans. Software Eng. 38 (1) (2012) 220–235. `doi:10.1109/TSE.2011.9`.

[65] I. David, E. Syriani, C. Verbrugge, D. Buchs, D. Blouin, A. Cicchetti, K. Vanherpen, Towards inconsistency tolerance by quantification of semantic inconsistencies, in: Proceedings of the 1st International Workshop on Collaborative Modelling in MDE (COMMitMDE 2016), Vol. 1717 of CEUR Workshop Proceedings, CEUR-WS.org, 2016, pp. 35–44.

[66] E. Syriani, R. Bill, M. Wimmer, Domain-specific model distance measures., J. Object Technol. 18 (3) (2019) 3–1.

[67] S. Balaji, M. S. Murugaiyan, Waterfall vs. v-model vs. agile: A comparative study on sdlc, International Journal of Information Technology and Business Management 2 (1) (2012) 26–30.

[68] L. F. Capretz, Y: New component-based software life cycle model, Journal of Computer Science, Science 1 (1) (2005) 76.

[69] I. David, B. Meyers, K. Vanherpen, Y. V. Tendeloo, K. Berx, H. Vangheluwe, Modeling and enactment support for early detection of inconsistencies in engineering processes, in: Proceedings of MODELS 2017 Satellite Event: Workshops co-located with ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), Vol. 2019 of CEUR Workshop Proceedings, CEUR-WS.org, 2017, pp. 145–154.

[70] I. David, M. Latifaj, J. Pietron, W. Zhang, F. Ciccozzi, I. Malavolta, A. Raschke, J.-P. Steghöfer, R. Hebig, Blended Modeling in Commercial and Open-source Model-Driven Software Engineering Tools: A Systematic Study, Software & Systems Modeling (2022). `doi:10.1007/s10270-022-01010-3`.

[71] A. L. Sangiovanni-Vincentelli, W. Damm, R. Passerone, Taming dr. frankenstein: Contract-based design for cyber-physical systems, Eur. J. Control 18 (3) (2012) 217–238.

[72] B. Meyer, Eiffel: A language and environment for software engineering, J. Syst. Softw. 8 (3) (1988) 199–246.

[73] M. Broy, K. Stølen, Specification and Development of Interactive Systems - Focus on Streams, Interfaces, and Refinement, Monographs in Computer Science, Springer, 2001.

[74] C. A. R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (10) (1969) 576–580.

[75] S. Grimm, A. Abecker, J. Völker, R. Studer, Ontologies and the semantic web, in: Handbook of Semantic Web Technologies, Springer, 2011, pp. 507–579.

[76] T. Kühne, Matters of (meta-)modeling, Softw. Syst. Model. 5 (4) (2006) 369–385.

[77] I. David, J. Denil, H. Vangheluwe, Process-oriented Inconsistency Management in Collaborative Systems Modeling, in: J. Machado, L. Mendes Gomes, H. Guerra, A. Abelha (Eds.), 16th International Industrial Simulation Conference 2018, ISC 2018, Eurosis, 2018, pp. 54–61.

[78] R. M. Smullyan, First-order logic, Courier Corporation, 1995.

[79] A. Fraga, J. L. Morillo, G. Génova, Towards a methodology for knowledge reuse based on semantic repositories, Inf. Syst. Frontiers 21 (1) (2019) 5–25.

[80] A. Fraga, J. L. Morillo, L. Alonso, J. M. Fuentes, Ontology-assisted systems engineering process with focus in the requirements engineering process, in: Complex Systems Design & Management, Proceedings of the Fifth International Conference on Complex Systems Design & Management CSD&M 2014, Springer, 2014, pp. 149–161.

[81] H. Van Ditmarsch, W. van der Hoek, J. Y. Halpern, B. Kooi, Handbook of epistemic logic, College Publications, 2015.

[82] H. Kannan, Formal reasoning of knowledge in systems engineering through epistemic modal logic, Syst. Eng. 24 (1) (2021) 3–16.

[83] M. Y. Vardi, An automata-theoretic approach to linear temporal logic, in: F. Moller, G. M. Birtwistle (Eds.), Logics for Concurrency - Structure versus Automata (8th Banff Higher Order Workshop, Banff, Canada, August 27 - September 3, 1995, Proceedings), Vol. 1043 of LNCS, Springer, 1995, pp. 238–266.

[84] M. Reynolds, An axiomatization of full computation tree logic, J. Symb. Log. 66 (3) (2001) 1011–1057.

[85] A. Motro, Integrity = validity + completeness, ACM Trans. Database Syst. 14 (4) (1989) 480–502.

[86] M. Famelis, R. Salay, M. Chechik, Partial models: Towards modeling and reasoning with uncertainty, in: 34th International Conference on Software Engineering, ICSE, IEEE Computer Society, 2012, pp. 573–583.