Uncertainty-aware consistency checking in industrial settings

Robbert Jongeling
Mälardalen University
Västerås, Sweden
robbert.jongeling@mdu.se

Antonio Vallecillo
ITIS Software. Universidad de Málaga
Málaga, Spain
av@uma.es

***Abstract*—In this work, we explore how we can assist engineers in managing, in a lightweight way, both consistency and design uncertainty during the creation and maintenance of models and other development artifacts. We propose annotating degrees of doubt to indicate design uncertainties on elements of development artifacts. To combine multiple opinions, we use the fusion operators of subjective logic. We show how these annotations can be used to identify, prioritize, and resolve uncertainty and inconsistency. To do so, we identify the types of design uncertainty and inconsistency to be addressed in two concrete industrial settings and show a prototype implementation of our approach to calculating the uncertainty and inconsistency in these cases. We show how making design uncertainty explicit could be used to tolerate inconsistencies with high uncertainty, prioritize inconsistencies with low associated uncertainty, and uncover previously hidden potential inconsistencies.**

***Index Terms*—Uncertainty, Consistency management, Model-Based Development.**

I. Introduction

We see a major challenge of model-based engineering in the management of inconsistencies that occur between models and other artifacts that are iteratively created during the development process. It is well known that inconsistencies between related artifacts must often be tolerated so as not to inhibit engineers in their work [1]. However, there is a moment when certain inconsistencies can no longer be tolerated and must be resolved before continuing with the development process. When is the right time to decide whether to continue to tolerate an inconsistency or to resolve it? The answer to this question is not obvious and often involves a high degree of doubt.

In addition to inconsistency, uncertainty is another unavoidable aspect in the development of model-based engineering of software-intensive systems. Uncertainty can be due to different factors, such as imperfect, incorrect, incomplete, or vague requirements; lack of complete knowledge about the system or its environment; distinct, even conflicting, interpretations of the same evidence by separate parties; unforeseen changes in the specifications; or the inability to determine whether particular design decisions are better than others [2], [3].

In this paper, we consider the connection between uncertainty and inconsistency, and how epistemic design-time uncertainty (see Section II.A) might be an input to help prioritize detected inconsistencies. Our starting assumption is that more uncertainty in elements implies a lower priority to resolve an inconsistency, and conversely, more certainty about inconsistent elements implies a higher priority for resolving the inconsistency. The rationale for this assumption is that we deem it less likely for uncertain elements to be the basis of further development before the uncertainty is resolved. Conversely, we foresee that elements without uncertainty are more likely to be the built further on, and therefore inconsistency in these elements may propagate through the system sooner.

Let us then formulate the research questions that we answer by studying two industrial model-based development settings:

RQ1: What kinds of uncertainty and inconsistency are involved in these two case studies?
RQ2: How can we allow engineers to explicitly represent uncertainty in their models?
RQ3: How do we combine multiple expressions of uncertainty across multiple development artifacts (and how do they relate to the consistency rules)?
RQ4: How do we reason about uncertainty and inconsistency in order to use them for (i) prioritizing inconsistencies and (ii) help engineers in refining both their understanding and their models of the system under development?

We limit our scope to detecting and reporting inconsistencies and do not consider automatically resolving them. This choice is due to two main reasons. First, in the considered settings there is no authoritative source of truth [4], i.e., if an inconsistency between two artifacts is detected, it is not possible to determine which of them shall be changed to restore consistency. Second, even knowing which artifact to change would not be sufficient, since it is not meaningful for the developer to, e.g., add the simple skeleton elements that would make the consistency check to pass. In fact, meaningful changes involve adding more semantics that can not be automatically derived from the consistency rules.

We must also consider the following requirements when addressing these research questions that are induced by our application context. First, the users in the studied settings are software engineers whose models are not diagrams with well-defined semantics, but are often just sketches or are described with informal notations. Second, consistency rules must be specified in a very simple way, so that they can be easily checked and, more importantly, easily specified and maintained. Otherwise, they become another cumbersome artifact added to the development environment. Third, whatever notation we use to express uncertainty, it needs to be simple to be usable. Finally, our proposal must be automatizable and the results of our checks must be readable, easily understandable and machine-processable. In short, our proposal must be lightweight and have minimal impact on the current processes and practices of software engineers in these companies.

The remainder of this paper is organized as follows. First, Section II contains background on uncertainty, subjective logic, and consistency management. Then, Section III introduces the two industrial settings used to motivate, demonstrate, and evaluate our proposal. Section IV presents our approach to making explicit design-time uncertainty at the level of models. After that, Section V details our prototypical implementation and its evaluation in the two industrial settings. Section VI discusses our findings and relates our results to other work. Finally, Section VII concludes with an outlook on future work.

II. Background

A. Uncertainty

Uncertainty can be defined as "the quality or state that involves imperfect and/or unknown information" [2]. The primary classification of uncertainty divides it into aleatory and epistemic uncertainty [5], [6]. *Aleatory uncertainty* refers to the inherent probabilistic variability or randomness of a phenomenon. For example, deciding the result of rolling a die. This type of uncertainty is irreducible, in that there will always be variability in the underlying variables [2]. *Epistemic uncertainty* refers to the lack of knowledge we have about the system (modeled or real) or its elements. For instance, the confidence we have in the actual occurrence of a modeled event. This type of uncertainty is reducible, in that additional information or knowledge may reduce it [7].

A particular type of epistemic uncertainty, called *Belief Uncertainty* [8]–[10], occurs when a user is unsure about any of the statements made in the model about the system or its environment. Belief uncertainty is normally represented in software models by assigning a *degree of belief* to model elements such as classes, relationships, attributes, or their values. It is sometimes called second-order probability or second-order uncertainty in the literature of statistics and economics, and is normally subjective, i.e., it depends on the individual agent holding the belief. Belief uncertainty is normally expressed by probabilities (interpreted in Probability theory [11]), possibilities (in Fuzzy set theory [12])), plausibilities (in the Dempster–Shafer theory [13])), or opinions (in Subjective logic [7]). For a comprehensive survey on the representation of uncertainty on software models, see [9].

B. Subjective logic

Subjective logic [7] is an extension of probabilistic logic to expresses beliefs (opinions) about the truth of propositions under degrees of uncertainty. Opinions can also indicate confidence, or trust, in a given statement and this is what makes them suitable in a context like ours.

Let F be a Boolean predicate stating a fact. Let X be a user or any entity able to express opinions about facts, also called a *belief agent*. A binomial *opinion* by user X about the truth of F is defined as a quadruple $w_F^X = (b, d, u, a)$ where:

- b (*belief*) is the degree of belief that F is true.
- d (*disbelief*) is the degree of belief that F is false.
- u (*uncertainty*) is the degree of uncertainty about F, i.e., the amount uncommitted to belief or disbelief.
- a (*base rate*) is the prior probability of F without any previous evidence.

These values satisfy the constraints that $b + d + u = 1$, and $b, d, u, a \in [0, 1]$. Intuitively, the *base rate* represents the *objective* probability that can be assigned to the fact using *a priori* evidences or statistical estimates, whilst the other elements of the tuple represent the *subjective* degrees of belief, disbelief and uncertainty about the fact assigned by the expert.

In addition to the common logical operators (and, or, implies, etc.) used to combine the opinions of the same expert about different facts, Subjective logic implements *fusion* operators for combining the opinions of different users about the same fact [7]. We will use these fusion operators later to combine multiple opinions about epistemic uncertainty of elements of development artifacts.

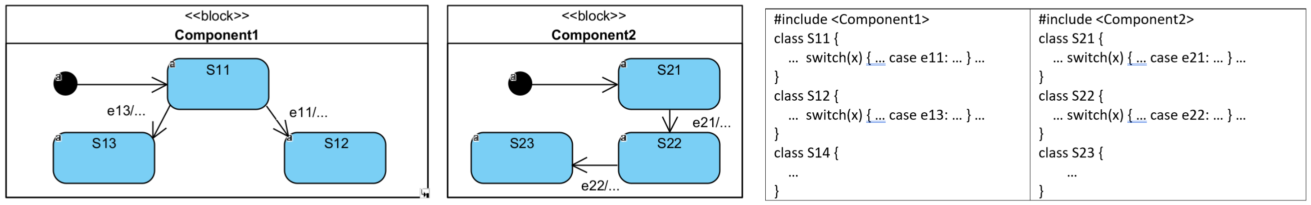
C. Consistency Management

Describing the system using various models and other artifacts induces a subsequent need to manage consistency among these artifacts, since inconsistency across them could lead to unmet requirements, delays, and ultimately even system failures [14], [15]. A commonly accepted idea is that inconsistency shall be tolerated and managed, so that it does not negatively impact development [1]. In this paper, we use the common rule-based approach to consistency management, in which consistency rules are defined and evaluated upon changes to development artifacts [16].

One of the key findings of most works on industrial studies is that, despite the various existing academic proposals for (automated) consistency management, their adoption in the industry is not straightforward [17]. The challenges are multiple: consistency management tasks are costly and often not prioritized; many legacy projects need to be maintained and no consistency checks are defined for them; there are usually many people and processes involved that cannot be easily changed; and the usability of consistency checking processes and tools is not good enough. As a result of these challenges, there is an argument for approaching consistency management in industrial settings in a rather lightweight manner [18].

III. Motivating examples

Our proposal is motivated by previous experience with companies that use model-based software engineering practices in their developments and need to manage inconsistencies between their artifacts. To illustrate our proposal and evaluate it, we will use two real industrial case studies from two companies. To keep their names anonymous, we call them A and B.



```
#include <Component1>
class S11 {
   ...  switch(x) { ... case e11: ... } ...
}
class S12 {
   ...  switch(x) { ... case e13: ... } ...
}
class S14 {
   ...
}
```

```
#include <Component2>
class S21 {
   ... switch(x) { ... case e21: ... } ...
}
class S22 {
   ... switch(x) { ... case e22: ... } ...
}
class S23 {
      ...
}
```

Fig. 1. A SysML model and its related C++ code excerpt.

A. Component-based development in company A

Software architects at company A use SysML to create system models that describe both software and hardware concerns. In this setting, SysML models are used to capture the high-level design of the system, its decomposition into separate components, and their allocation to software or hardware parts. Component behavior is described at a high level by operations, specified by activity diagrams; interfaces, which are specified by proxy ports; and states, specified by state machines. Independent teams of software engineers work on the implementation of the system, which is done in C++.

Figure 1 shows on the left two SysML components and their associated state machines, and on the right an excerpt of the corresponding C++ code (for confidentiality reasons, all examples used in this paper have been anonymized).

The company has adopted consistency checks in its current development processes to improve the overall consistency of its development artifacts and detect potential sources of errors. Consistency is key in this context to verify that the system architecture defined by the software architects was followed by the software engineers and accurately reflected the structure of the code, and vice-versa. Furthermore, any changes made to the code cannot conflict with the architectural requirements set out in the models. In addition, both the models and the code can evolve separately throughout the development process.

In this setting, structural consistency between the system model and the code is defined by the following three rules:

R1 Each system component should be implemented in a separate repository.
R2 Each state should be implemented in a separate class.
R3 Each reception event in the state machines should be implemented as a case in a switch statement.

These rules can be easily checked with automatic means. For example, in Figure 1 some discrepancies can be detected, such as states in the SysML model that do not appear in the code (S13), and vice versa (S14). Although this solution may seem sufficient, it is not. In particular, the size of the artifacts are in the order of thousands of elements and the number of discrepancies between the models and the code is often so large (in the order of hundreds) that they are difficult to manage, and thus become useless [19].

A key solution for the company to successfully address this issue is to prioritize inconsistencies, highlighting those that need to be resolved first. Currently, all inconsistencies are considered equally important, but some of them could be more critical than others, and therefore are more urgent to repair. Similarly, there may be others of little relevance that could be tolerated or even ignored for the time being. In general, it is during the early design phases that the most inconsistencies occur. But it is also the period when there is the most uncertainty in design decisions and implementations. For example, a software architect may have doubts about the constituent states of a state machine, e.g., whether a given state should be part of it or not, or whether a state is missing. Similarly, a software developer may be unsure of the occurrence of a case in a switch statement, or of a particular class in a component.

In this context, it is not only important to highlight information regarding the *doubts* that engineers may have about their designs or implementations. It is also essential to be able to express *certainty* about some of its elements. For example, the fact that some of the requirements have been consulted and confirmed by the customer. Or that an implementation has to include some particular class for security reasons or company implementation policies. We will later show how to make use of this additional information to prioritize inconsistencies and also discover new inconsistencies that were previously hidden.

B. A product line architecture for embedded systems

The second example comes from company B that specializes in the development of embedded systems. Using a product line approach, a team of software architects is responsible for designing and maintaining the reference software architecture for the various variants of the embedded systems that the company produces [18]. This reference architecture is specified by one base model with all possible family *components*, described using an informal notation (similar to UML deployment diagrams) sketched in Draw.io. In addition, a set of informal *relationship diagrams*, written in PlantUML, describe the possible *connections* between these components. The reference architecture for a fictional Family F is shown in the upper part of Figure 2, showing the base model (RefArch) and one associated relationship diagram (RefArch-RD).

The specification of every product of the family is given by one product model (PM) that describes the set of elements from the reference architecture that the product comprises, and a set of associated relationship diagrams (PM-RD). An example of the specification of a product P from family F is shown in the bottom part of Figure 2. When a new product is created, the reference architecture diagram is copied and components are deleted or added as needed for that particular product, analogous to clone-and-use practices common in software product line engineering [20].

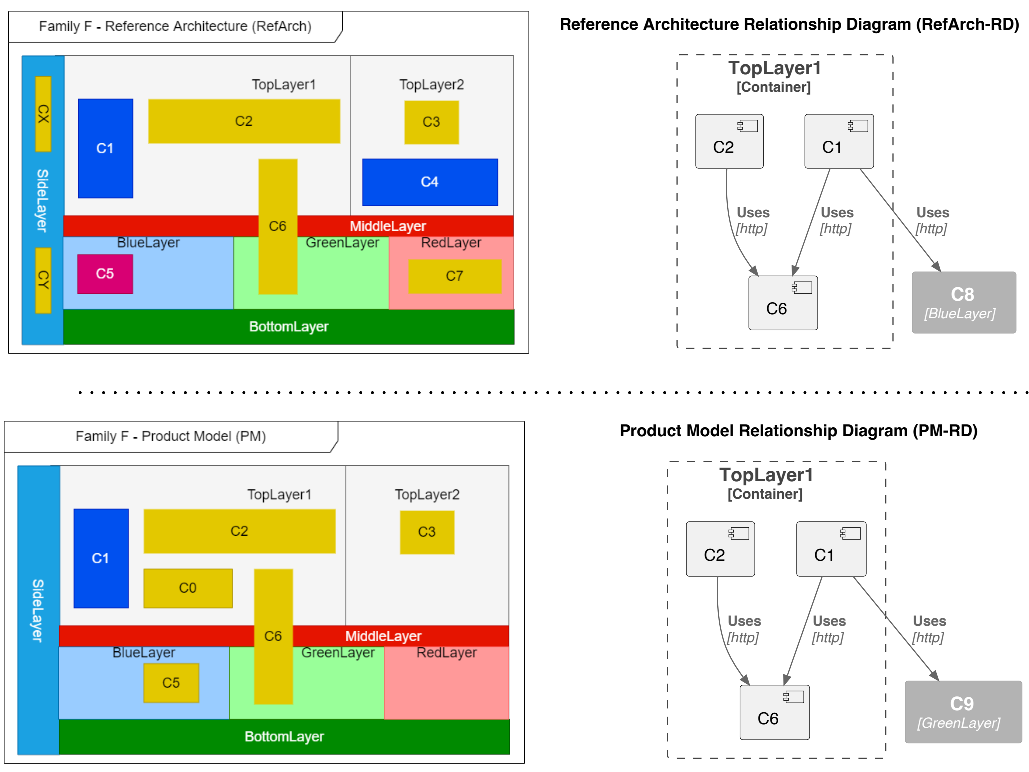


Fig. 2. Reference Architecture of product family F (top left) and Product Model of product P (bottom left), and a Relationship Diagram for each one.

In the reference architecture and product models, components are assigned to layers, which act as containers for them. Both components and layers have properties, such as color, size, or position. Likewise, components and connectors in the relationship diagrams may have properties (expressed within square brackets), such as the protocol a connector uses.

There are different consistency rules that these four types of models (RefArch, RefArch-RD, PM, PM-RD) should respect:

S1 Every component in a product model (PM) should exist in the reference architecture (RefArch).

S2 Every component in a relationship diagram (RD) should be in the corresponding model (RefArch or PM).

S3 Every connector in the RD of a product should be in the corresponding RD of the RefArch.

As in the previous case, in this company one group of engineers works with the reference architecture and another with the product models. This can lead to inconsistencies. For example, in the models and diagrams shown in Figure 2, Component C0 exists in PM but not in RefArch; Component C8 exists in RefArch-RD but not in RefArch; the properties of Component C5 in PM are different from those in RefArch (namely, its color and positions are different); Component C9 exists in PM-RD, but not in RefArch-RD, nor in RefArch.

Moreover, during the design phase of the reference architecture and the product models, different engineers may be uncertain about these models and their elements. A difference with respect to the previous example is that the teams of architecture and product models work closely together, and therefore one engineer could have doubts about any of the model elements, no matter if they belong to the family architecture or to the product model.

For example, Thomas K., a software architect developing the reference architecture, may not be sure if a given component (C1) is part of Family F or not, or whether protocol https should be used instead of http for connecting components C1 and C8. Even worse, for that connection in the relationship diagram of the product model (PM-RD), he thinks IMAP could be a better option. He also doubts that component C6 is always available for use within the family architecture. On the contrary, he is completely sure that component C2 should belong to this family. In turn, Gabriele T., another software architect also working on the project, has serious doubts about the same component C2 that Thomas was certain about. She is not sure if that component C1 is available for product P, and also about the contents of the product model. She thinks that some components could be missing in PM and that component C9, included in PM-RD should be part of product P (the complete lists of doubts is shown in Figure 5).

In this company, more than the number of inconsistencies to manage, they are interested in using them to guide evolution. For example, in situations when a component gets deprecated and needs to be updated. They also need to count on mechanisms for dealing with exceptions to the rules. For example, to state that they want to keep using a version of a given component in a product, even when the reference architecture prescribes a later version, and do not want to be warned about this inconsistency anymore: “Stop telling me this is inconsistent, I know it but I don’t want to change it.”

IV. Making doubts explicit

We propose a lightweight means to annotate development artifacts with design uncertainties, which we will refer to as *doubts* to avoid confusion with the term "uncertainty" as defined by subjective logic. We propose these degrees of doubt as a simplified notation for expressing epistemic design uncertainties. We still rely on the underlying mechanism of subjective logic to (i) allow for customization of the distribution of subjective logic values underlying particular degrees of doubt, and (ii) have a solid formalism underlying any calculations done to combine multiple opinions.

This section describes how we propose the artifacts to be annotated with doubts and how we can make use of them for discovering and prioritizing inconsistencies. In the following subsections, we present four stages of our approach: I–Manually annotating elements in design artifacts with doubts; II–Automatically propagating doubts across the set of development artifacts and their elements; III–Automatically merging multiple doubts on the same elements by different agents; and IV–Automatically prioritizing inconsistencies using subjective logic operators and the consistency rules. Implicitly, a fifth stage can be imagined during which the doubts are resolved. Phase I responds to research questions RQ1 and RQ2. Phases II and III respond to RQ3. Phase IV responds to RQ4.

Phase I: Annotating doubts

Types of doubts: Depending on the intended use of development artifacts and the relationships between them, we have identified different types of doubts that engineers may express: Occurrence, Availability, Contents, and Properties.

- *Occurrence* doubts are those related to the existence of an element in a model or a development artifact. For example, Thomas may have doubts about component C1 being part of the configuration of product P.
- *Availability* doubts occur when the agent is not sure that a component can be available when needed in a given configuration. They are usually related to shared or scarce resources. For example, a component that may not be available under heavy load or high demand.
- *Contents* doubts are associated with container elements. Layers, components or SysML blocks are examples of containers in the previous case studies because they can contain other elements. Doubts about the contents of a container happen when the engineer believes that some of its internal elements may be missing.
- *Property* doubts occur when the engineer is not sure about the current value of a property of one element. For example, its color, version or position.

Degrees of doubt: Expressing an opinion using subjective logic tuples including degrees of belief, disbelief, and uncertainty may be challenging for users [21]. In fact, we do not expect that in practice engineers will be able to quantify all these degrees and, moreover, requiring all this information is not likely to be beneficial to the usability of our approach. Therefore, we propose a very lightweight way to annotate the elements of a model with a single degree of *doubt*, using a notation inspired by [22], with 4 values in a Likert scale:

Symbol	**Meaning**	**Subj. Logic opinion**
!	Very certain	$(1.0,\ 0.0,\ 0.0,\ 0.5)$
$\sim$	Somewhat uncertain	$(0.95, 0.025, 0.025, 0.5)$
?	Uncertain	$(0.8,\ 0.1,\ 0.1,\ 0.5)$
??	Very Uncertain	$(0.5,\ 0.25, 0.25, 0.5)$

To be able to perform calculations with these values in Subjective Logic, we have mapped them to subjective logic opinions, as shown in the table above and, graphically, in Figure 3. Basically, the greater the doubt, the greater the uncertainty and disbelief. For example, the $\sim$ annotation corresponds to little doubt and therefore it has a high degree of belief and low degrees of disbelief and uncertainty. We chose to assign equal values to the degrees of disbelief and uncertainty (since, without more information, both might be equal causes for doubt), but this can be configured.

We also need to give semantics to the elements of the model without annotations about their uncertainty, i.e., most of the model elements. Having no annotations has two possible interpretations. The most obvious one is that we are quite sure about them, i.e., that we have no doubt. This interpretation is the one given, for example, by the creator of the model. However, when reviewing an artifact, expressing no explicit doubt may mean either "no reason to disagree with the original creator" or that the artifact was not actually reviewed. This absence of doubt will also be represented as "_", and its corresponding value in subjective logic is $(1.0, 0, 0, 0.5)$, same as a Bang ("!"). There is no possible confusion between "_" and "!". The former means absence of doubt while the latter means complete certainty and requires the provision of concrete evidence to reinforce the assertion.

Table I shows the operators we will use on doubts. They correspond to the *and* (AND), *implies* (IMP), *weighted belief fusion* (AVG) and *equivalent* (PLUS) operators in Subjective logic [7]. The values in these tables have been computed using the corresponding operators in Subjective logic.

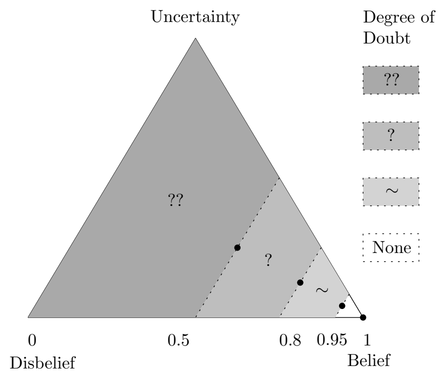


Fig. 3. Mapping degrees of doubt to Subjective Boolean values. Annotation '!' corresponds to an absolute belief, with no uncertainty nor disbelief, thus corresponding to the point on the bottom-right corner of the triangle.

TABLE I
BEHAVIORAL TABLES FOR OPERATORS ON DOUBTS.

AND	–	∼	?	??
–	–	∼	?	??
∼	∼	?	?	??
?	?	?	??	??
??	??	??	??	??

AVG	–	∼	?	??
–	–	∼	?	??
∼	∼	∼	?	?
?	?	?	?	??
??	??	?	??	??

IMP	–	∼	?	??
–	–	∼	?	??
∼	–	∼	?	??
?	–	∼	?	??
??	–	∼	?	??

PLUS	–	∼	?	??
–	–	∼	?	??
∼	∼	∼	?	??
?	?	?	∼	??
??	??	??	??	?

Expressing doubts: User should be able to annotate elements they have doubts about. We propose the following notation:

$$\langle\text{Agent}\rangle\langle\text{Degree of Doubt}\rangle\langle\text{Type}\rangle(\langle\text{element}\rangle) : \langle\text{Rationale}\rangle$$

where:

- *Agent* is the identifier of the belief agent expressing the doubt, e.g, Thomas (TK) or Gabriele (GT).
- *Degree of doubt* can be any of the symbols described above: !, ∼, ?, or ??.
- *Type* is the identifier of the type of doubt: O (occurrence), A (availability), M (missing contents), or P (property).
- *Element* refers to the model element about which the doubt is expressed. For example, C1@RefArch refers to component C1 in the RefArch model. More than one level of nesting is possible, e.g., S13@Component1@SysML. If the element is a property, it is prepended to the identifier of the component, e.g., color#C1@PM.
- *Rationale* is a string with the reason for the doubt, or the evidence that supports the certainty (!).

For example, Thomas might express his doubts on the occurrence of component C1 in the product model PM:

TK?O(C1@PM):"Not sure if this component should be included in this product after the change in requirements."

Another agent, e.g., Gabriele, might independently express another opinion on the same element:

GT!O(C1@PM): "I have confirmed with the customer that this component is needed in this product."

Phase II: Doubt propagation and combination

Propagating doubts: In this phase, the annotated doubts from Phase I are propagated to the related development artifacts. Propagation is achieved through the *correspondences* that relate the elements in the different model views [23]. In our industrial settings, these correspondences are implicitly defined by the consistency rules, and element names are used to identify the related elements. For example, component C1 in the product model PM corresponds to component C1 in the reference architecture RefArch; and State S11 of Component1 in the SysML model corresponds to Class S11 of Component1 in the C++ implementation.

In our Product Line example, the components of the reference architecture are exactly the same as those of the product models. Thus, if Thomas is not sure that C2 exists in the reference architecture, his doubt should propagate to component C2 of all product models. Note that the reverse need not be true. Gabriele may have doubts that component C3 should be part of a certain product model, but this does not mean that she has doubts that C3 is part of the reference architecture. That is, propagation is directional in this case.

Table II shows how doubts propagate in this example. Propagation is defined considering the four diagrams in Figure 2, their relationships as described in Section III-B, and the consistency rules S1–S3. The directions in the table refer to the way the development artifacts are organized in Figure 2. *Estimated* doubts refer to the annotations that modelers are allowed to express, of different types. A "+" symbol means that they are allowed, while a "–" symbol means that they are not, mainly because their values are derived. *Propagated* doubts are derived from the estimated doubts or from other propagated doubts. Propagation directions are defined by the nature of the consistency rules: the occurrence of a component in one diagram often implies it must occur in another one too.

There are two exceptions to propagation rules: (1) We do not propagate "!" or "None" since these are not expressions of doubt. Bangs will be used later, during inconsistency prioritization; (2) we do not propagate a doubt if the target element has the same degree of doubt, because they are already consistent.

Another type of correspondence occurs between elements that are different but related by the consistency rules. For example, states of components in the SysML model and C++ classes in the implementation. In this case, doubts do not propagate because they are independent artifacts.

Combining multiple doubts by the same agent: Our goal is to end up with a single degree of doubt (or none) for each annotated element. Thus, after propagation, we combine the opinions of the same agent on the same elements. If an agent has provided two doubts of the same type, d_1 and d_2, on the same element, that is equivalent to stating the single opinion: $d_1 \wedge d_2$. This combined doubt is computed using the AND operator (see Table I).

Phase III: Doubts merge

It may be the case that multiple agents have expressed doubts of the same type about the same elements. To derive a single opinion per element, in this phase we combine opinions from multiple agents using the weighted belief fusion operator [7], noted AVG in Table I. This operator is commutative, idempotent, and non-associative, and computes a weighted average of the doubts, giving more weight to those opinions with less uncertainty, i.e., the smaller the value of the uncertainty, the higher the weight. Again, we do not include "!" in our calculations, since it is not a degree of doubt.

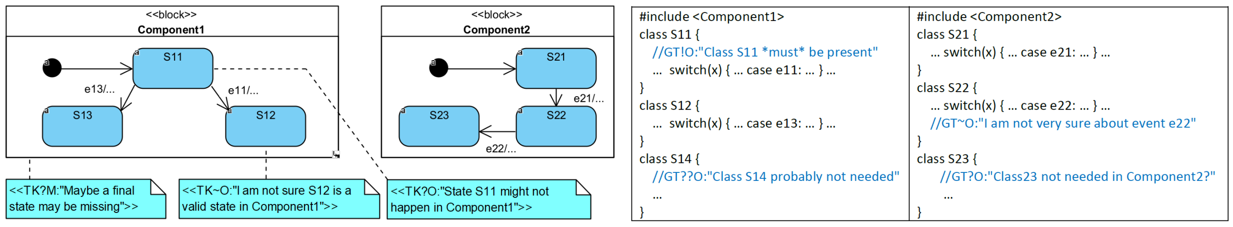
Phase IV: Uncertainties for inconsistency prioritization

Prioritizing inconsistencies: To prioritize the detected inconsistencies, we will assign a degree of doubt to each of them, and rank them according to their uncertainty.

TABLE II

PROPAGATION DIRECTIONS FOR ESTIMATED DOUBTS IN SETTING 2, THE DIRECTIONS REFER TO THE LAYOUT SHOWN IN FIGURE 2.

		Components				Connectors		Containers	
		RefArch	PM	RefArch-RD	PM-RD	RefArch-RD	PM-RD	PM	PM-RD
Estimated	Occurrence	+	+	+	+	+	+	-	-
	Property-value	+	+	+	+	+	+	+	+
	Availability	+	+	+	+	+	+	-	-
	Missing	-	-	-	-	-	-	+	+
Propagated	Occurrence	-	from up	from left	from left	-	from up	-	-
	Property-value	-	from up	-	-	-	from up	from up	from up
	Availability	-	from up	from left	from left	-	from up	-	-
	Missing	-	-	-	-	-	-	-	-


<<block>>
Component1
S11
S12
S13
e13/...
e11/...
<<block>>
Component2
S21
S22
S23
e21/...
e22/...
<<TK?M:"Maybe a final state may be missing">>
<<TK~O:"I am not sure S12 is a valid state in Component1">>
<<TK?O:"State S11 might not happen in Component1">>

```
#include <Component1>
class S11 {
   //GT!O:"Class S11 *must* be present"
   ...  switch(x) { ... case e11: ... } ...
}
class S12 {
   ...  switch(x) { ... case e13: ... } ...
}
class S14 {
   //GT??O:"Class S14 probably not needed"
   ...
}
```

```
#include <Component2>
class S21 {
   ... switch(x) { ... case e21: ... } ...
}
class S22 {
   ... switch(x) { ... case e22: ... } ...
   //GT~O:"I am not very sure about event e22"
}
class S23 {
     //GT?O:"Class23 not needed in Component2?"
     ...
}
```

Fig. 4. A SysML model and its related C++ code excerpt. Both are annotated by their developers with information about their uncertainties.

To assign degrees of doubts to the inconsistencies, we will use the doubts noted on the elements involved in the inconsistency. Three cases may occur. First, if none of the elements involved in the inconsistency has any associated doubt, the degree of doubt associated with the inconsistency is "none" ("_"). Second, if one or more elements have a "!" associated with it, the degree of doubt associated with the inconsistency is "!". Finally, if the elements involved in the inconsistency have associated degrees of doubt, they need to be combined, and the result assigned to the inconsistency. The operator used to combine the doubts associated with the inconsistent elements depends on the directionality of the inconsistency. We previously saw that inconsistencies can be either directional (when they work in one way only) or bidirectional. Then, the implies (IMP) operator will be used in the first case, and the equivalence operator ($\Leftrightarrow$, PLUS) in the second. Examples of these are detailed in Section V.

Uncovering potential inconsistencies: A byproduct of annotating elements with uncertainties is that we can uncover potential inconsistencies that were hidden before, in addition to the "rule-based" inconsistencies that follow directly from the consistency rules. For example, consider the state S11 in the state machine in Figure 1 and the corresponding class S11 in the code. By the consistency rule, occurrence of a state implies occurrence of the class, and vice-versa, and therefore this pair will not be marked as inconsistent. But when we consider the annotated doubts, we see that the state has been annotated with a doubt of occurrence ("?"), while the occurrence of S11 in the code has been confirmed by another agent expressing certainty about its occurrence ("!"). This discrepancy between the degrees of doubts associated with the related elements needs to be signaled too, because it might mean a potential inconsistency. Our reasons for discovering these potential inconsistencies are due the fact that elements explicitly marked with doubts are more likely to change in the future and hence more likely to become inconsistent.

Finally, both detected and uncovered inconsistencies will be ranked according to the degree of doubt assigned to them.

V. IMPLEMENTATION

This section describes the prototype implementation we have developed to demonstrate and validate our approach as well as its application to the two industrial case studies. The description follows the process of application of our approach, as detailed in the previous section. For space and readability reasons, only a few synthetic doubts have been annotated in the models, although they are sufficient to illustrate the proposal. All artifacts used and produced by our algorithm, as well as its implementation, are available in the replication package [24].

Phase I: Annotating doubts. Figures 4 and 5 show the annotated doubts on the models used in the two settings. Engineers can use notes, stereotypes, code comments, text files, or any other ways to denote doubts in their models. These annotations are then extracted from the models and stored in textual files (*.uncertainty). The grammar is defined using TextX [25] (see architecture.tx in the replication package).

Phase II: Doubt propagation. The algorithm automatically propagates the doubts through the consistency rules, and also throughout artifacts, see Section IV. The propagation depends on the relationship between the artifacts, i.e., whether they refer to the same elements or not. In Setting 1, the related artifacts were of different nature, and therefore the doubts are not propagated. In contrast, the elements in the models of Setting 2 refer to exactly the same components, and therefore doubts must be propagated throughout the diagrams following the guidelines given in Table II.

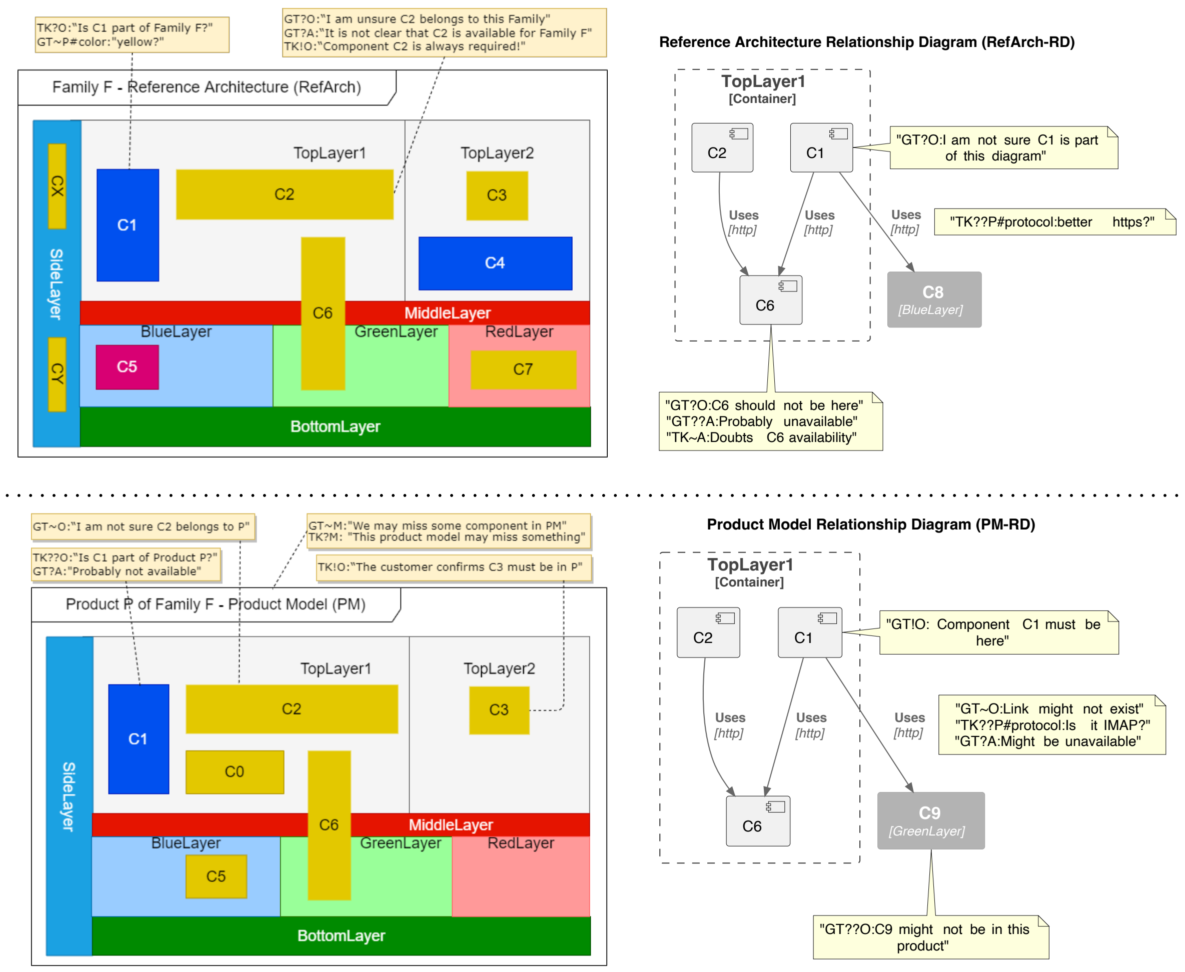

TK?O:"Is C1 part of Family F?"
GT~P#color:"yellow?"
GT?O:"I am unsure C2 belongs to this Family"
GT?A:"It is not clear that C2 is available for Family F"
TK!O:"Component C2 is always required!"
Family F - Reference Architecture (RefArch)
SideLayer
CX
CY
TopLayer1
TopLayer2
C1
C2
C3
C4
C5
C6
C7
MiddleLayer
BlueLayer
GreenLayer
RedLayer
BottomLayer
Reference Architecture Relationship Diagram (RefArch-RD)
TopLayer1
[Container]
C2
C1
C6
C8
[BlueLayer]
Uses
[http]
"GT?O:I am not sure C1 is part of this diagram"
"TK??P#protocol:better https?"
"GT?O:C6 should not be here"
"GT??A:Probably unavailable"
"TK~A:Doubts C6 availability"
GT~O:"I am not sure C2 belongs to P"
TK??O:"Is C1 part of Product P?"
GT?A:"Probably not available"
GT~M:"We may miss some component in PM"
TK?M: "This product model may miss something"
TK!O:"The customer confirms C3 must be in P"
Product P of Family F - Product Model (PM)
SideLayer
TopLayer1
TopLayer2
C1
C2
C3
C0
C5
C6
MiddleLayer
BlueLayer
GreenLayer
RedLayer
BottomLayer
Product Model Relationship Diagram (PM-RD)
TopLayer1
[Container]
C2
C1
C6
C9
[GreenLayer]
Uses
[http]
"GT!O: Component C1 must be here"
"GT~O:Link might not exist"
"TK??P#protocol:Is it IMAP?"
"GT?A:Might be unavailable"
"GT??O:C9 might not be in this product"

Fig. 5. Models and diagrams of Fig. 2, here annotated with uncertainty information.

A concrete example of propagation happens with the doubt by agent TK on C1 in RefArch, which is propagated to the same component in PM, based on the "from up" entry in Table II. Since C1 in PM also had an annotated doubt by the same agent (TK), we obtain a single doubt by applying the AND operator on the two doubts (See Table I). The result is the doubt TK??O(C1@PM). The other propagated doubts in this setting are:

- GT?O(C2@PM) – From GT∼O(C2@PM) and GT?O(C2@PM), propagated from GT?O(C2@RefArch)
- TK??O(C1@PM-RD) – From TK??O(C1@PM)
- GT∼O(C2@PM-RD) – From GT∼O(C2@PM)
- GT?O(C2@RefArch-RD) – From GT?O(C2@RefArch)

These propagated doubts are automatically added to the estimated doubts of each case as listed in the textual files.

Phase III: Doubt merge: In Setting 1, no doubts were propagated during Phase II and no elements in Figure 4 were annotated with multiple doubts (from different agents).

The situation is different in Setting 2. After propagating the doubt TK??O(C1@PM) to TK??O(C1@PM-RD), we now have two opinions associated with this element, the other being GT!O(C1@PM-RD). Since we ignore "!" during merging, we obtain the merged opinion *??O(C1@PM-RD). Note that the asterisk is used to denote that this opinion is originating from multiple agents. Similarly, we obtain *?O(C2@RefArch) from merging GT?O(C2@RefArch) and TK!O(C2@RefArch). Operator AVG is used to perform the merge operations.

Phase IV: Uncertainties for inconsistency prioritization. The last step consists in applying the consistency rules to search for possible inconsistencies. In addition to those found by the rules, other inconsistencies are detected in the case where the consistency rule is not violated, but there is a difference of doubt between the elements. Below, we provide examples to illustrate the final priorizations in the two settings. The complete results can be found in files output-example-1.txt and output-example-2.txt in the replication package.

In Setting 1, we found two rule-based inconsistencies since S13 and S14 occur only in one view and not in the other. In addition, we found four uncovered inconsistencies, one of which has an even higher priority than the rule-based inconsistency. Another interesting result is that one rule-based inconsistency becomes the lowest priority due to the doubt assigned on the occurrence of S14 in the code.

The final list of inconsistencies found by our algorithm in Setting 1 is presented below. They are sorted by their priority according to our approach.

Prio.	**Type**	**Detected inconsistency (Setting 1)**		
!	Unc.	?S11@SysML	$\Longleftrightarrow$	!S11@CPP
_	Rule	S13@SysML	$\not\Longleftrightarrow$	S13@CPP
∼	Unc.	∼S12@SysML	$\Longleftrightarrow$	S12@CPP
∼	Unc.	e22@SysML	$\Longleftrightarrow$	∼e22@CPP
?	Unc.	S23@SysML	$\Longleftrightarrow$	?S23@CPP
??	Rule	S14@SysML	$\not\Longleftrightarrow$	??S14@CPP

We distinguish between "Rule" and "Uncovered" (Unc.) inconsistency types, where the former refers to a violation of one of the consistency rules defined for the system and the latter is due to a difference in the degrees of doubt (or certainty) of the related elements.

In Setting 2, the algorithm finds six uncovered inconsistencies, on top of the five rule-based inconsistencies that we originally had. They are shown below.

Prio.	**Type**	**Detected inconsistency (Setting 2)**		
!	Unc.	!C1@PM-RD	$\Longrightarrow$	??C1@PM
!	Unc.	!C1@PM-RD	$\Longrightarrow$	?C1@RA-RD
_	Rule	C1_C9@PM-RD	$\not\Longrightarrow$	C1_C9@RA-RD
_	Rule	C0@PM	$\not\Longrightarrow$	C0@RefArch
_	Rule	C8@R.A.-RD	$\not\Longrightarrow$	C8@RefArch
?	Unc.	C6@PM-RD	$\Longrightarrow$	?C6@RA-RD
?	Unc.	C2@PM-RD	$\Longrightarrow$	?C2@PM
?	Unc.	C2@PM-RD	$\Longrightarrow$	?C2@RA-RD
?	Unc.	??C1@PM	$\Longrightarrow$	?C1@RefArch
??	Rule	??C9@PM-RD	$\not\Longrightarrow$	C9@PM
??	Rule	??C9@PM-RD	$\not\Longrightarrow$	C9@RA-RD

We can see how annotated doubts can be used for inconsistency prioritization as well as for uncovering previously hidden inconsistencies. In addition, the annotated doubts may be used as a guide for further refinement of the models, as they indicate places that may require attention.

VI. Discussion

A. Proposed methodology of working with annotating doubts

This section illustrates the process we envisage to annotate model elements with doubts and to detect and prioritize inconsistencies between model elements. The goal is to help engineers resolve inconsistencies as quickly as possible but tolerating those for which there is insufficient information until it is obtained. Our approach is primarily intended in a model-driven agile development environment that works in sprints.

Annotating doubts shall be possible at any time. Once created, they will be propagated and evaluated at the end of each sprint. After the propagation phase, inconsistencies are identified and ranked. The prioritized list of inconsistencies provides a guideline on the order to follow to address the resolution (or tolerance) of both inconsistencies and doubts.

The inconsistencies with the highest priority should be checked first, to ensure that the doubts are removed or the inconsistency is resolved. Normally, doubts are removed when the reasons that justified them are investigated and resolved.

It is important to clear as many doubts as possible after each sprint, especially the easiest to resolve ("∼" and "?"), in order to keep the overall design uncertainty under control. Note that most of the design uncertainty in models is epistemic in nature, meaning that it can be mitigated by providing more information. Finally, lower-priority inconsistencies usually resolve themselves over the course of model refinement, or as doubts disappear.

B. Feedback from industry partners

We also presented our proposal to the engineers of the two companies that work exactly with the settings we studied, asking them for feedback and suggestions.

Feedback from company A: For company A, prioritization was of primary importance for its estimated business value. It is often difficult to motivate some of the tasks that need to be done, especially when they are cumbersome or costly. This is critical for some quality-related aspects, such as consistency checking. However, when you have prioritized inconsistencies and, in addition, know how often a particular component is reused in different products, it may be easier to motivate the business value of working to resolve inconsistencies.

The "lightweightness" of the proposed annotations was also appreciated, and compared to other coarse-grained estimations they use in the company, such as using "T-shirt sizes" (S, M, L, XL) for estimating the effort required to implement pieces of functionality. They also saw a potential for annotated doubts to be used as requirements for degrees of consistency. For example, a modeler might annotate an element with a certainty (!) and the rationale: "I want this part to definitely be consistent". The uncovered inconsistencies become then very relevant when discovering, e.g., a corresponding code element for which a software engineer has expressed a doubt such as "I do not care much if this is consistent or not."

Lastly, they were interested in considering the hierarchical propagation of uncertainties. For example, annotating certainty in a component, for them also implied certainty in all its contents. So far, we have considered instead that the hierarchy does not guarantee such propagation, but we expect this to largely depend on the specificities of the industrial setting.

Feedback from company B: Engineers at company B were not so concerned about prioritizing many inconsistencies, because their number was expected to be low. Essentially, the engineers rely on "copy-and-modify" for the creation of a new product from the reference architecture. Consequently, not many inconsistencies are expected. However, they were very interested in scenarios where annotations would make consistency checks clearly more valuable. For instance, in a situation in which a component is deprecated and needs to be updated, the architects found our proposal useful as a tool to guide the evolution. Moreover, they found certainties ("!") very valuable for suppressing inconsistency warnings. For example, to assert that in a certain product, they want to continue using version n of a particular component, even if the reference architecture prescribes version $n + 3$. Being able to smoothly tolerate exceptions to inconsistency rules is a common requirement in industrial environments [17].

These initial reactions to our proposal are encouraging, and we have started to consider further empirical evaluations in different industrial settings to understand better how our proposal can be used. For example, we would like to explore whether other, more expressive notations for expressing subjective opinions are acceptable to engineers. Furthermore, work is needed to identify the types of uncertainty that are most relevant to in diverse industrial settings to improve the applicability of our proposal. We think of this as the natural way to continue our work.

C. Limitations and generalizability

This approach considers ranking of inconsistencies only on the basis of the annotated uncertainty. In the considered cases, this is the only information available and therefore the only one possible to use. In general, the prioritization of inconsistencies to be resolved can have many more inputs, such as the expected impact of the inconsistency, or the relative importance of adherence to a particular consistency rule. Thus, in general, inconsistency prioritization is a matter of optimizing many variables, although in this paper we limited ourselves to only one such factor.

We have aimed for industrial model-based development settings in which models may be of a rather informal nature. We have not evaluated the generalizability of the approach beyond the presented case studies. However, we think that,in general, providing lightweight means to manage uncertainty and inconsistency can be helpful in settings similar to those shown here. Future research is needed to evaluate how the methodology presented in Section VI-A generalizes.

D. Related work

The work presented in this paper relates to several research areas. First, many works deal with consistency management; relevant recent secondary studies include [15], [26]. By its nature, dealing with inconsistencies is not a problem that can have a single, general, solution, but rather it depends on many factors from particular industrial settings. Thus, there is a continued relevance for new consistency management approaches that address the reality of industrial settings [17]. In this work, we address the requirements of two concrete industrial settings and explore how they improve their consistency management by making uncertainty explicit.

Uncertainty management is the second area of research addressed in this paper. Previous work has implicitly hinted at combining uncertainty and inconsistency, by noticing that uncertainty is equally inevitable as inconsistency, and by taking steps towards instead managing uncertainty [27]. Another approach is to live with design-time uncertainty by generating all options at run-time, as done in JTL [28]. In our studied settings, we aim to make the uncertainty already explicit at design time. Another set of works, e.g., [10], [22] deals with the explicit representation of belief uncertainty in models; see [9] for a survey on uncertainty representation in software models. However, none of these works use uncertainty to address inconsistency management as we do here.

Another set of related works, e.g., [29]–[31], combines uncertainty and consistency, although they focus on uncertainty in the definition of traceability links between related elements. In our case, we have focused on uncertainty on the elements themselves and on their properties, assuming simple consistency rules and thus simple traceability links. As future work we could investigate whether it makes sense in our case studies to incorporate belief uncertainty in traceability links, and how to combine it with our current approach.

Detected inconsistencies, to be manually resolved, can be the result of static analysis over multiple development artifacts. Within the field of static analysis, various approaches have been proposed to prioritizing warnings, e.g., based on the change history [32], and machine learning [33]. Our proposal prioritizes inconsistencies based only on the uncertainty associated with the elements involved and does not consider the type of violation. Finally, to the best of our knowledge, this is the first work that attempts to uncover "hidden" inconsistencies, using uncertainty information to do so.

VII. Conclusion

Inconsistency and uncertainty are both integral and inevitable aspects of the model-based development of software-intensive systems. During the development process, they are both expected to decrease until ideally zero for a final product. In this paper, we have proposed a means to explicitly annotate models (be they strict or rather informal) with design-time uncertainties in a lightweight way. To allow so, we propose a single degree of doubt that covers values for belief, disbelief, and uncertainty in subjective logic. We then show how to use the explicit doubts as an input to various subjective logic operations to obtain prioritized inconsistencies, as well as to uncover previously hidden potential inconsistencies between development artifacts. The approach is demonstrated in two examples from industrial settings. Results from these settings give an indication of the potential usefulness of our proposal.

There are several aspects that we plan to address as part of our future work. First, we want to evaluate the usability, applicability, scalability and usefulness of the proposal with more case studies in different industrial settings. Exploring alternative, more expressive ways of representing doubts is also on our agenda. Finally, we also want to get more feedback from practitioners about our proposal, including other use cases where explicit expression of doubts can bring more value to industrial processes, beyond the prioritization of inconsistencies. The impact of our proposal depends primarily on the value it brings to the industry.

Acknowledgment: The authors thank the anonymous reviewers for their helpful comments. This work was partially funded by the Swedish Software Center https://www.software-center.se and by the Spanish research projects PID2021-125527NB-I00 and TED2021-130523B-I00.

References

[1] R. Balzer, "Tolerating inconsistency," in *Proc. if ICSE'91*. IEEE Computer Society, 1991, pp. 158–165.

[2] JCGM 100:2008, *Evaluation of measurement data—Guide to the expression of uncertainty in measurement (GUM)*, Joint Com. for Guides in Metrology, 2008. [Online]. Available: http://www.bipm.org/utils/common/documents/jcgm/JCGM_100_2008_E.pdf

[3] M. Famelis and M. Chechik, "Managing design-time uncertainty," *Softw. Syst. Model.*, vol. 18, no. 2, pp. 1249–1284, 2019.

[4] P. Stevens, "Maintaining consistency in networks of models: bidirectional transformations in the large," *Software and Systems Modeling*, vol. 19, no. 1, pp. 39–65, 2020.

[5] A. D. Kiureghian and O. Ditlevsen, "Aleatory or epistemic? does it matter?" *Structural Safety*, vol. 31, no. 2, pp. 105–112, 2009.

[6] W. L. Oberkampf, S. M. DeLand, B. M. Rutherford, K. V. Diegert, and K. F. Alvin, "Error and uncertainty in modeling and simulation," *Reliability Engineering & System Safety*, vol. 75, no. 3, pp. 333–357, 2002.

[7] A. Jøsang, *Subjective Logic - A Formalism for Reasoning Under Uncertainty*. Springer, 2016.

[8] M. Zhang, B. Selic, S. Ali, T. Yue, O. Okariz, and R. Norgren, "Understanding uncertainty in cyber-physical systems: A conceptual model," in *ECMFA'16*, ser. LNCS, vol. 9764. Springer, 2016, pp. 247–264.

[9] J. Troya, N. Moreno, M. F. Bertoa, and A. Vallecillo, "Uncertainty representation in software models: A survey," *Software and Systems Modeling*, vol. 20, no. 4, pp. 1183–1213, 2021.

[10] L. Burgueño, P. Muñoz, R. Clarisó, J. Cabot, S. Gérard, and A. Vallecillo, "Dealing with belief uncertainty in domain models," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 2, pp. 1–34, 2023.

[11] B. de Finetti, *Theory of Probability: A critical introductory treatment*. John Wiley & Sons, 2017.

[12] S. J. Russell and P. Norvig, *Artificial Intelligence, A Modern Approach*, 3rd ed. Prentice Hall, 2010.

[13] G. Shafer, *A Mathematical Theory of Evidence*. Princeton University Press, 1976.

[14] ISO/IEC/IEEE, "ISO/IEC/IEEE 42010:2011(E) Systems and software engineering – Architecture description," ISO, Tech. Rep., Dec 2011.

[15] W. Torres, M. G. J. van den Brand, and A. Serebrenik, "A systematic literature review of cross-domain model consistency checking by model management tools," *Softw. Syst. Model.*, vol. 20, no. 3, pp. 897–916, 2021.

[16] A. Reder and A. Egyed, "Incremental consistency checking for complex design rules and larger model changes," in *Model Driven Engineering Languages and Systems: 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30–October 5, 2012. Proceedings 15*. Springer, 2012, pp. 202–218.

[17] R. Jongeling, F. Ciccozzi, J. Carlson, and A. Cicchetti, "Consistency management in industrial continuous model-based development settings: a reality check," *Softw. Syst. Model.*, vol. 21, no. 4, pp. 1511–1530, 2022.

[18] R. Jongeling, F. Ciccozzi, A. Cicchetti, and J. Carlson, "From informal architecture diagrams to flexible blended models," in *ECSA'22*, ser. LNCS, vol. 13444. Springer, 2022, pp. 143–158.

[19] R. Jongeling, J. Fredriksson, J. Carlson, F. Ciccozzi, and A. Cicchetti, "Structural consistency between a system model and its implementation: a design science study in industry," *J. Object Technol.*, vol. 21, no. 3, pp. 3:1–16, 2022.

[20] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," in *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 2013, pp. 25–34.

[21] F. J. Navarrete and A. Vallecillo, "Introducing subjective knowledge graphs," in *Proc. of EDOC'21*. IEEE, 2021, pp. 61–70.

[22] P. Martín-Rodilla and C. Gonzalez-Perez, "Conceptualization and non-relational implementation of ontological and epistemic vagueness of information in digital humanities," *Informatics*, vol. 6, no. 2, p. 20, 2019.

[23] J. R. Romero, J. I. Jaen, and A. Vallecillo, "Realizing correspondences in multi-viewpoint specifications," in *Proc. of EDOC'09*. IEEE Computer Society, 2009, pp. 163–172.

[24] R. Jongeling and A. Vallecillo, "Uncertainty-aware consistency checking in industrial settings – replication package," https://doi.org/10.5281/zenodo.8109635, 2023.

[25] I. Dejanović, R. Vaderna, G. Milosavljević, and v. Vuković, "TextX: A Python tool for Domain-Specific Languages implementation," *Knowledge-Based Systems*, vol. 115, pp. 1–4, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950705116304178

[26] A. Cicchetti, F. Ciccozzi, and A. Pierantonio, "Multi-view approaches for software and system modelling: a systematic literature review," *Software & Systems Modeling*, pp. 1–27, 2019.

[27] S. Bernardi, M. Famelis, J.-M. Jézéquel, R. Mirandola, D. P. Palacin, F. A. Polack, and C. Trubiani, "Living with uncertainty in model-based development," *Composing Model-Based Analysis Tools*, pp. 159–185, 2021.

[28] R. Eramo, A. Pierantonio, and G. Rosa, "Managing uncertainty in bidirectional model transformations," in *SLE'15*. ACM, 2015, pp. 49–58.

[29] R. Kretschmer, D. E. Khelladi, R. E. Lopez-Herrejon, and A. Egyed, "Consistent change propagation within models," *Softw. Syst. Model.*, vol. 20, no. 2, pp. 539–555, 2021.

[30] C. Trubiani, A. Ghabi, and A. Egyed, "Exploiting traceability uncertainty between software architectural models and extra-functional results," *J. Syst. Softw.*, vol. 125, pp. 15–34, 2017.

[31] A. Ghabi and A. Egyed, "Exploiting traceability uncertainty among artifacts and code," *J. Syst. Softw.*, vol. 108, pp. 178–192, 2015.

[32] S. Kim and M. D. Ernst, "Which warnings should i fix first?" in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007, pp. 45–54.

[33] G. Liang, L. Wu, Q. Wu, Q. Wang, T. Xie, and H. Mei, "Automatic construction of an effective training set for prioritizing static analysis warnings," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 93–102.