Taming uncertainty with MDE: an historical perspective

Jean-Marc Jézéquel

Taming Uncertainty with MDE: an Historical Perspective

Jean-Marc Jézéquel

Univ Rennes, IUF, Inria, CNRS, IRISA

Abstract

Uncertainty in Informatics can stem from various sources, whether ontological (inherent unpredictability, such as aleatory factors) or epistemic (due to insufficient knowledge). Effectively handling uncertainty, encompassing both ontological and epistemic aspects, to create predictable systems is a key objective for a significant portion of the software engineering community, particularly within the Model-Driven Engineering (MDE) realm. Numerous techniques have been proposed over the years, leading to evolving trends in model-based software development paradigms. This paper revisits the history of MDE, aiming to pinpoint the primary aspects of uncertainty that these paradigms aimed to tackle upon their introduction. Our claim is that MDE progressively came after more and more aspects of uncertainty, up to the point that it could now help fully embrace it.

1 Introduction

Uncertainty is an inherent and multifaceted challenge in software development, stemming from various sources that span from the problem domain to the execution platform. At the heart of software engineering, uncertainties arise from the fuzziness of the problem domain, where requirements, encompassing business or legal rules and human behavior expectations, often present themselves as incomplete and subject to evolution over time. Assumptions about the world with which the software interacts are, more often than not, coarse, implicit, and insufficiently inclusive, often neglecting to consider all potential corner cases.

Beyond these issues with the problem domain, we define *Platform Uncertainty* as the uncertainty that exists at the platform level, either at development time or operation time. At operation time, ontological (inherent) uncertainties are due to hardware faults and network delays, adding an additional layer of unpredictability to software operation. At development time, epistemic uncertainties arise from factors such as misunderstanding of programming language semantics, compiler option effects or library behavior, thus introducing unexpected challenges to the development process. Moreover, the ever-looming threat of cyber-attacks and malicious tampering further amplifies the spectrum of uncertainties that software engineers must grapple with throughout the software development lifecycle.

We call *Uncertainty Management* the endeavor of dealing with the complexity due to uncertainty in software development and operation. It usually requires identifying and isolating (or at least reducing the coupling with) the parts that are subject to uncertainties. The concept of Separation of Concerns then becomes pivotal to explicitly manage uncertainties, and facilitate effective decision-making at various stages of the software lifecycle —from requirement and design time to compile time, load time, JIT time, and even

runtime [36]. But Separation of Concerns is not a silver bullet: modern techniques helping implement it (e.g., Aspect Oriented Programming (AOP), Spring framework) bring in complex frameworks and fragile supply chains that inject back additional ontological layers of uncertainty.

In this article we claim that uncertainty management has long been an unformulated key issue in software development, and certainly one of the main underlying drivers of Model Driven Engineering (MDE). We support this claim by looking back at the history of trends in MDE to see what particular problem of uncertainty management these trends actually wanted implicitly or explicitly to address. We show that MDE progressively came after more and more aspects of uncertainty, up to the point of standardizing Uncertainty Modeling[1] so that MDE could now help fully embrace it.

Beyond software development, managing uncertainty is also an important issue when testing software systems under uncertainties or testing software systems with inherent uncertainties. We chose however to leave it out of the scope of this article, because it would deserve a full article on its own.

The rest of this paper is organized as follows. Section 2 provides some background and discusses related work on the topic of sources of uncertainty in Software Engineering. Then we discuss 4 generations of approaches progressively addressing uncertainty as an ever more central concern:

- Handling a specific uncertainty: From CASE tools to Model Driven Architecture
- Managing uncertainty at model level through Separation of Concerns with Models, Aspects and Features
- Considering Uncertainty at the Language Level
- Towards Fully Embracing Uncertainty

We conclude by summarizing the ongoing challenges of handling uncertainty in and about models.

[1] See *e.g.,* the Precise Semantics for Uncertainty Modeling (PSUM) standard, recently published at OMG `https://www.omg.org/spec/PSUM`.

2 Sources of Uncertainties: Background and Related Work

Uncertainty in Software Engineering is a widely studied subject [68, 53, 30]. Jousselme et al. [40] examine numerous hierarchies and ontologies, offering definitions from a situational analysis viewpoint, including concepts like ignorance (as a state of mind) and uncertainty (stemming from observational limitations). Similarly, Padulo and Guenov [64] conduct a review of prior research, distilling a summary that frames the design challenge as twofold: uncertainty *about* the problem and uncertainty *within* the problem.

A useful categorization of various uncertainty types is presented by Esfahani and Malek [18]. They articulate it using two dimensions:

- *Reducible* versus *Irreducible*, and
- *Aleatory* versus *Epistemic*.

The authors assert that *"aleatory and epistemic represent the essence of uncertainty, while irreducible and reducible represent the managerial aspect of uncertainty."* They elaborate that the differentiation between epistemic and aleatory uncertainty *"is motivated by the location of the uncertainty — in the decision-maker or in the physical system."*

Irreducible uncertainty pertains to situations in which uncertainty persists despite having complete information. Such occurrences are inherently unknowable. In contrast, reducible uncertainty denotes instances where additional knowledge can be acquired, ultimately eliminating all uncertainty.

Aleatory uncertainty has an ontological nature caused by factors such as noise in input data, memory layout, network delays, the internal state of the processor, ambient temperature, and even the age of the processor [14]. On the other hand, epistemic causes manifest in the form of misunderstandings of user needs, unpredictable behavior of APIs, variable behavior among functionally similar libraries, or subtle differences in programming language semantics, both within and across programming languages e.g., x/0 has undefined behavior of in C, while -3%2 evaluates to -1 in Java but to 1 in Python which might confuse the programmer.

Expanding beyond the confines of computers and software, the broader field of engineering endeavors to navigate both complexity and uncertainty, aspiring to create predictable systems within the bounds of engineering [83]. In disciplines like mechanical and civil engineering, systems are purposedly designed to function as intended under expected real-world conditions. However, uncertainties arise when the system encounters situations beyond the expected "operational envelope", leading to failures in structures or machinery under unforeseen circumstances such as earthquakes or metal fatigue.

Engineers conventionally address these uncertainties by attempting to minimize them and subsequently, in non-safety-critical areas, often choose to ignore them [5]. Recognizing existing uncertainties, however, can significantly reduce their impact and enhance confidence in a given model [44]. Within the modeling community, researchers are progressively directing their focus towards identifying and modeling uncertainty, acknowledging that not all uncertainties can be entirely traced back to their origin, eliminated, or fully explained [17, 26, 73]. This shift in perspective underscores the importance of understanding, managing, and explicitly modeling uncertainties in Model Driven Engineering [79, 8].

3 Taming Epistemic Uncertainty: From CASE Tools to MDA

3.1 CASE Tools

Computer Assisted Software Engineering (CASE) tools emerged in the 1980s, incorporating functionalities like consistency checking, validation, and code generation, particularly in sectors such as telecommunications. Notably, the European telecom industry adopted Formal Description Techniques, including SDL (Specification and Description Language), Estelle, and Lotos, which were based on extended state machines or process algebra. These languages provided well-defined syntax and semantics, facilitating simulation and early attempts at industrial-scale model-checking.

CASE tools empowered engineers to translate specifications into instant implementations through code generation, offering a promising prospect of high-level, confidently validated programming for distributed computers. The approach successfully addressed the *epistemic uncertainty* of protocol specification (you are not yet sure of what should be the best protocol to solve a problem) by separating it from the accidental complexity of implementation, thus streamlining adaptation to new requirements [37].

Yet, there were two significant drawbacks. Firstly, the abstract and mathematical nature of Formal Description Techniques posed training challenges for telecom engineers, with *e.g.,* Lotos considered intricate for C programmers. Secondly, the proprietary nature of code generators became a critical issue. While some instances demonstrated success aligned with engineering needs, the black-box nature hindered adaptability to diverse constraints, such as speed, code compactness, memory usage, and interfacing with legacy software.

In response, engineers sometimes attempted post-processing steps after code generation to address spe-

cific constraints. This was not without risk [1]. For example, a large telecom company was using an SDL code generator and had the issue of a state machine transition that took too long to execute. The consequence was that some real time constraints could not be met. So the engineers broke the transition into two blocks, yielding the control to the scheduler in the middle. That worked for meeting the real time constraints. However there was a catch that was only seen much later after a catastrophic crash of the system. By breaking the transition, that in SDL semantics was atomic, the engineers involuntarily introduced new behavior with a different interleaving of event processing that in some rare cases led to this crash.

Ultimately, the industry abandoned the one-size-fits-all black box code generation approach. While these tools addressed some specific aspect of the epistemic uncertainty, their lack of flexibility out weighted the benefits, making them unable to deal with more diverse sources of epistemic uncertainty as discussed in the next sections.

3.2 The Time of Model Driven Architecture (MDA)

In [58] the OMG Architecture Board described its Model Driven Architecture (MDA) vision to support interoperability with specifications that address integration through the entire systems life cycle: from business modeling to system design, to component construction, to assembly, integration, deployment, management, and evolution. MDA had the primary goal to separate the fundamental logic behind a specification from the specifics of the particular middleware that implements it. This would allow rapid development and delivery of new interoperability specifications that use new deployment technologies but are based on proven, tested business models. In the OMG vision, organizations would use MDA to meet the integration challenges posed by new platforms, while preserving their investments in existing business logic based on existing platforms. The term platform was used here to refer to technological and engineering details that are irrelevant to the fundamental functionality of a software component.

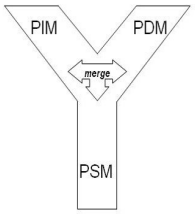

Figure 1: Y-development

Once again, the kind of uncertainty addressed here is *epistemic uncertainty*, more precisely requirement and design uncertainty due to the lack of knowledge on which exact platform the software should be running on.

In its most idealized form, as depicted in Figure 1, the Model-Driven Architecture (MDA) envisions the amalgamation of a Platform-Independent Model (PIM) with a Platform-Dependent Model (PDM) to seamlessly generate a Platform-Specific Model (PSM). However, the practical implementation of MDA was more complex than this idealized concept, possibly contributing to its limited adoption.

Firstly, MDA predominantly operates as a forward engineering approach, transforming models into implementation artifacts like code, database schemas, software configuration scripts, and test cases through a fully or partially automated generation process. When uncertainty is not adequately captured in the source models, manual modification of the generated code becomes necessary to fix the issue. This manual intervention quickly becomes burdensome for maintenance, even with vendor-provided strategies to mitigate the challenges of synchronizing manual modifications with the generated code.

Secondly, in specific business domains involving fault-tolerant, distributed real-time computations, the engineering know-how is not just about the business domain (the PIM), but also about the design expertise essential for making these systems operational in real-world scenarios (the transformation from PIM to PSM). In certain instances, this transformation was more intricate than the PIM itself. Although significant knowledge was embedded in such transformations, the challenge was that it occurred at an inappropriate level of abstraction due to the limited support for a proper separation of concerns in transformation languages like QVT or ATL. One potential solution would have been to implement the Y-shaped approach outlined in Figure 1, incorporating an explicit PDM and a simpler model transformation facilitating a model composition between the PIM and the PDM to derive the PDM. Regrettably, no one successfully modeled a non-trivial platform to serve this purpose.

There is indeed so much uncertainty coming with any given platform, both epistemic and aleatory (see Section 2), that trying to capture it all in a formal way is a daunting task. However *using* this platform in a specific way only requires to know about the specific issues of interest to a given application, which is several orders of magnitude simpler because the programmer knows the context and thus can manually resolve all the epistemic uncertainty, and assess and most often neglect any remaining aleatory uncertainty.

The bottom line is that MDA (defined here as a specific technology implementing the OMG standards) was neither necessary nor sufficient to handle the epistemic uncertainty of business modeling.

4 Managing Uncertainty with Separation of Concerns

As discussed in the previous section, MDA had the goal to isolate the fundamental logic behind a specification from the uncertainty about which particular middleware would be used to implements it. This is indeed one important epistemic uncertainty concern to isolate from the core functionality of an application. But that is not the only one. In any complex application, there are *many other epistemic uncertainty concerns* stemming from aspects such as fault tolerance, concurrency, distribution, data persistency, energy consumption, safety, security, user experience, and many more.

Managing these various dimensions of epistemic uncertainty can only be done if these concerns are somehow reified. This is made possible with the time honored idea of separation of concerns, and its modern implementation with the notion of Models and Aspects, as well as variability modeling, all connected through MDE, as outlined below.

4.1 Separation of Concerns

The term *separation of concerns* has probably been coined by Dijkstra as far back as 1974 [16]:

> *Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects. We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. In another mood we may ask*

ourselves whether, and if so: why, the program is desirable. But nothing is gained —on the contrary!— by tackling these various aspects simultaneously. It is what I sometimes have called **"the separation of concerns"***, which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by "focusing one's attention upon some aspect": it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.*

4.2 Models and Aspects

As Dijkstra pointed out, separation of concerns naturally leads to the idea of aspects that can be analyzed separately, which is precisely what we are trying to achieve with modeling. My preferred definition of a model is indeed [38]:

A model is the abstraction of an aspect of reality for handling a given concern.

Note that the Aspect Oriented Programming community had a much narrower definition of an aspect as being only the modularization of a cross-cutting concern [22]. If we indeed have an already existing "main" decomposition paradigm (such as object orientation), there are many classes of concerns for which a clear allocation into modules is not possible (hence the name "cross-cutting"). Examples include both allocating responsibility for providing certain kinds of functionality (such as logging) in a cohesive, loosely coupled fashion, as well as handling many non-functional requirements that are inherently cross-cutting, *e.g.,* security, resource management, etc. But purely outside the programming world [70], there was

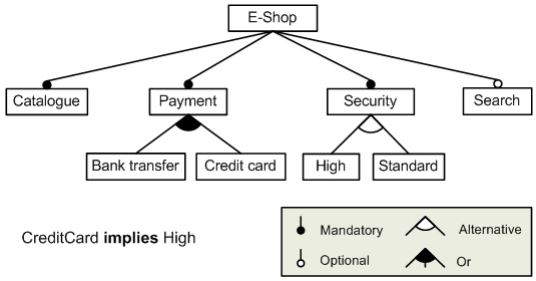

Figure 2: A Feature Model Example

an acceptance for a wider definition where an aspect is a concern that can be modularized. In that sense an aspect becomes quite close to the concept of a feature, as in Feature Oriented Programming [2].

4.3 From Software Product Lines to Self-Adaptive Systems

Once we have dealt with epistemic uncertainties by identifying and modularizing our concerns as model level aspects [89], we still need to manage which variant of which concern goes into which product. This led to the development of the Software Product Line concept [67], and in particular to the pervasive feature diagram notation, as illustrated in Figure 2, to describe and explicitly manage variability[2]. SPL can thus be seen as a technique to control epistemic uncertainty: the engineer knows at design time that something could be either A or B, but doesn't know yet what is the final decision, because in some context (*e.g.,* for some clients) it could be A and for others it could be B.

Modeling is the activity of separating concerns in the problem domain, an activity often called *analysis*. If solutions to these concerns could be described as aspects, the design process would then be characterized as a weaving of these aspects into a detailed design

[2]Taken from https://commons.wikimedia.org/w/index.php?curid=25197577

model [31]. This is actually what software designers have been actually doing forever. Most often however, the various aspects were not *explicit*, or when there were, it was in the form of informal descriptions. So the task of the designer was to do the weaving in her head more or less at once, and then produce the resulting detailed design as a big tangled program. While it could work pretty well for small problems, it is a major headache for bigger ones. Of course, modern techniques such as object-orientation help keep some form of separation of concern in the design, which is already extremely helpful. This separation of concern may even still exist at runtime, with the notion of components as deployment units, or more recently SOA or micro-services. Still some aspects are difficult to encapsulate with any such decomposition paradigms (aka the "tyranny of the dominant decomposition" [76]).

Note that the real challenge here is not on how to design the system to take a particular aspect into account: there is a huge design know-how in industry for that, often captured in the form of design patterns. Taking into account more than one aspect as the same time is a little bit more tricky, but many large scale successful projects in industry are there to show us that engineers do ultimately manage to sort it out.

The real challenge in a product-line context is to deal with uncertainty in such a way that the engineer is able to change her mind on which version of which variant of any particular aspect she wants in the system. And she wants to do it cheaply, quickly and safely at different stages of the software lifecycle — from requirement and design time to compile time, load time, JIT time, and even runtime in the case of Self-Adaptive Systems.

In self-adaptive systems change activities are indeed shifted from development time to runtime, and the responsibility for these activities is shifted from software engineers or system administrators to the system itself. Still a good way to conceptualize Self-Adaptive Systems is to see them as Dynamic Software Product Lines [32, 65] for safely handling various sources of uncertainties at runtime [83].

4.4 Model Driven Engineering

In this context, MDE can be seen as a methodology for mechanizing the process experienced designers follow by hand [90]. The idea is that when a new product has to be derived from the product-line, we can automatically replay the design process, just changing a few things here and there [59].

MDE technologies, such as executable metamodeling [61], support development of models that capture software functionality and properties at different levels of abstraction and from different perspectives, as well as rigorous analysis of models, and transformation of models into software artifacts that serve specific development purposes.

Cleanly separating epistemic uncertainty concerns of a system is not always completely straightforward, but its difficulty is more or less proportional to the inherent complexity of the problem at hand. But once concerns have been separated into aspects, even simple ones as in the AOP view (i.e. oblivious and cross-cutting), one still needs to compose them.

While weaving a single aspect is pretty straightforward, weaving a second one at the same join point is indeed another story. When a second aspect has to be woven, the initial join point might not any longer exist because it could have been modified by the first aspect advice. If we want to allow aspect weaving on a pair-wise basis, we must then define the join point matching mechanism in a way that considers these composability issues. However, with this new way of specifying join points, two new problems arise (see [38] for a detailed explanation with the example of sequence diagrams):

- It is in general difficult (or even statically undecidable [12, 42]) to identify join points when

the patterns we are looking for are based on the properties of the computational flow.

- The composition of the advice with the detected part cannot any longer be just a replacement of the detected part by the advice: we also have to define more sophisticated compositions operators.

More generally, the problem is that nobody ever came up with composition operators with good mathematical properties such as commutativity and associativity. That makes the tool support for these approaches extremely difficult and costly to build.

It probably means that there is no hope for a fully general-purpose, meta-model independent, model-level aspect weaver. Still, it should be possible to develop aspect weaving software components handling several aspects of aspect weaving, from general-purpose model-level pattern matching [69] to automated support for composing models written in a particular language (through a definition of model composition behavior in the metamodel defining the language [24]), to specializable model composers [33]. These aspect weaving components could then be customized and combined to build domain specific or even project specific aspect weavers.

A very good example of that has been recently popularized under the names of low-code and no-code approaches [10]. No-code development platforms generally offer prebuilt templates where all cross-cutting concerns have been solved for once, or at least with a finite set of possible choices. Then a business person without any programming skill can build applications by merely configuring these prebuilt templates. It works very well as long as one is staying within the predefined envelop of what has been pre-designed, *i.e.,* that the epistemic uncertainty is reduced to a finite set of pre-defined choices, as in product-lines.

Low-code platforms follow the same logic, but provide a little bit more flexibility by allowing developers to specify both data models and business rules with high level DSLs, such as ER/class diagrams or graphical tables on one side and BPMN, statecharts, or graphical action languages on the other side. That's indeed a great way to isolate business-level epistemic uncertainty from the complexity of the deployed software: the business person can easily change her mind and immediately and without effort get the resulting software.

5 Considering Aleatory Uncertainty

5.1 Modeling Aleatory Uncertainty

The previous sections described how MDE approaches progressively took into account more and more aspects of *epistemic uncertainty*. But until relatively recently, very few approaches have been developed to systematically deal with the other kind of uncertainty, i.e.; *aleatory uncertainty*. Contrary to epistemic uncertainty which is uncertainty *about* the model, aleatory uncertainty can be seen as uncertainty *within* the model. We thus need ways of representing and managing it inside a modeling language with the notion of *uncertain attributes*. Depending on how uncertainty is described, we can identify three main categories of uncertain attributes: (i) *probabilistic*, *e.g.,* the probability density functions of the uncertain attributes are assumed to be known [56, 80]; (ii) *possibilistic*, e.g., uncertain attributes are described with fuzzy boundaries [3, 34]; (iii) *interval analysis*, *i.e.,* the limits of variation are investigated to deduce best/worst cases [20, 11].

5.1.1 Intervals

For example, Fujimoto [25, 52] use time intervals to deal with the concepts of Approximate Time and

Approximate Time Event Ordering in the context of Discrete Event System Specification (DEVS) [86]. In their proposal, two events are considered concurrent if the intervals representing their timestamps have a non-empty intersection. Saadawi and Wainer also explored replacing time datatype in models by intervals in their RTA-DEVS formalism [72]. Furthermore, two new extensions to DEVS, called UA-DEVS and IA-DEVS, provide methods to specify uncertainty in the state, input, and output variables in addition to the time variable [82]. The former defines a formal specification of models including uncertainty specifications as intervals. The latter enables the simulation of models based on computational constraints (time, memory, etc.). This separation of concerns allows the domain expert to define the model once, and then simulate it with different constraints without redefining the model.

There are two major limitations of approaches based on intervals for specifying the possible values of uncertain variables. On the one hand, they are very coarse-grained, which results in very conservative (also called *cautious*) simulations [85]. On the other hand, specifying and operating with intervals require a significant effort by the modeler since there is no direct support for making computations with them, such as arithmetic operations or comparisons, which are really burdensome and error-prone tasks.

5.1.2 Probabilities and Possibilities

Going beyond intervals requires to treat uncertain attributes as implicit random variables, leading to stochastic models. These models can then be simulated, using *e.g.,* Monte Carlo methods [28, 88] or other statistical methods for the calculation of confidence intervals for the mean of a simulation output [13]. They typically obtain more accurate results than those proposals that use interval arithmetic. However, both the complexity of their calculations and their computational costs might hinder their applicability.

The explicit representation of uncertainty in models indeed remains a challenge [79]. In particular, there are very few libraries for programming or modeling languages that support measurement uncertainty, i.e., the representation and operation of uncertain datatypes. Even those that support the propagation of uncertainty (e.g., [45, 46, 4, 84]) are quite complex to use and do not support the comparison between uncertain numbers. This is a general problem that can be observed in most uncertainty modeling proposals: they only deal with uncertain reals. However, in the physical world, all other primitive data types also have uncertain values. In particular, logical variables representing decisions or comparisons between quantities rarely have crisp true or false values. Instead, extensions to the Boolean logic enable dealing with this type of uncertainty, including probability theory [21, 15], possibility theory (based on fuzzy logic [91, 71]), plausibility (a measure in the Dempster-Shafer theory of evidence [75]), and uncertainty theory [50]. These approaches assign different probabilities to propositions, rather than truth values, and probability formulas replace truth tables.

Leveraging these ideas, reference [39] has recently proposed to include the sources of uncertainty in the system models as first-class entities using such *explicit* random variables, in order to model and simulate systems more faithfully. That includes not only the use of random variables to represent and operate with uncertain values, but also to represent *decisions* based on their comparisons. For implementing their approach, they have used the Java library of datatypes extended with measurement uncertainty defined in [6], which is also implemented in the tool USE to support uncertain numbers in UML and OCL [63]. This greatly simplifies the management of uncertain numbers in both Java programs and UML/OCL models.

5.2 Integrating Uncertainty Management in Software Languages

If we want to have a wider adoption of uncertainty aware software, providing libraries that are arguably rather difficult to use is probably not enough: uncertainty management needs to become a first class citizen of both modeling and programming languages.

There are several practical ways of implementing the support for uncertainty management in Software Languages.

- By providing a library, with *e.g.,* `uncertainties` for Python or [6] for Java. This last library supports `Float/Double` values with uncertainty that are represented in terms of `Float/Double` values, which are composed of pairs (x,u), also noted as $x \pm u$, where x is the value, and u represents its uncertainty as the standard deviation of its possible variations, according to the GUM international standard [35]. Likewise, the library provides a `UBoolean` value B, which is a pair $B = (b,c)$ in which c is a floating point number in the interval $[0,1]$ representing the confidence we assign to b. Comparison operators between uncertain variables return `UBoolean` values. For example, if $x = 1.0 \pm 0.3$ and $y = 1.5 \pm 0.25$, then $x < y$ = `Boolean(true,0.893)`, meaning that $x < y$ with a confidence of 0.893 [6]. Projection operation `confidence()` applied to an uncertain Boolean returns a probability, *i.e.,* the confidence assigned to that Boolean.

- By extending an existing language to support uncertain data types as in [63]. This extension extends the basic UML and OCL primitive datatypes (Real, Boolean, Integer,...) with uncertainty by defining super-types for them, as well as the set of operations defined on the values of these types. Thus, `Real` values with uncertainty are represented in terms of `UReal` values, with the underlying implementation provided by the Java library discussed above. Likewise, a `Boolean` value b is lifted to an `UBoolean` value B as above, with direct support for the projection operation `confidence()` returning a probability. Uncertain values then become first-class entities of models, and can be managed and operated in a natural way by the underlying type system. Propagation of uncertainty through operations is transparently taken care of by the type system, and comparisons are lifted to `UBoolean` values when required. In a similar way, the Object Management Group initiative on uncertainty modeling brought together a range of industrial and academic experts to design a metamodel of uncertainty in cyberphysical systems [87] that aims to capture the (un)certainty of the modeler, by expressing beliefs about information.

- By providing an external DSL dedicated to uncertainty management. For example, BPMN (Business Process Model and Notation) can be associated to a DSL called BPSim to simulate business processes with stochastic lead times.

Conceptually speaking, these different ways to provide support for uncertainty in models are actually different faces of the same dice. Indeed a library, especially if it uses a fluent API, can be seen as a low overhead implementation of a DSL's concepts. Similarly, an extension to an existing language can be seen as an internal DSL, that could even be automatically obtained from a composition of both languages (or meta-models) [60].

5.3 Example: Combining SPL and Probabilities

Recently [23] proposed to combine the modeling of both epistemic and aleatory uncertainties to help find optimal solutions while implementing Program Development Plans (PDPs) at Airbus. That boils down to combine the idea of SPL with explicit random variables modeling of lead times in BPMN.

A PDP is indeed a description of a highly integrated development thread from program preparation to handover to series. It highlights the key deliverables, decisions and the maturity levels to be achieved at each step of a program. It helps to identify the contribution of each stakeholder in a long and complex ecosystem. It inherits a long history of aircraft project management experience and incorporates a large number of lessons learned (e.g. A380, A350, NEO...).

During the early phases of a program, the uncertainty on the future program assumptions is still high (e.g. mission, range, make or buy policy, number of aero load stress loops), and consequently the associated PDP is not stable yet, but can already be modeled as a Product Line, *i.e.,* a set of BPMN models expliciting the foreseen variability as a consequence of this epistemic uncertainty. These BPMN models are traditionally used for end-to-end business process stochastic simulation with a BPSim extension to deal with aleatory uncertainty on *e.g.,* lead times or resources, which are typically modeled with random variables with known distributions.

By automatically exploring the BPMN Product Line variability space with clever search-based algorithms, and then running the corresponding stochastic simulation, it made it possible to identify optimal configurations with associated confidence intervals [23]. Combining the explicit modeling of both epistemic and aleatory uncertainty in this way then allows for informed and grounded decision making: turning the PDP design into such a multi-criteria optimization problem makes it possible to present upper management with a Pareto's front of possible choices, each associated with confidence intervals.

6 Towards Embracing Uncertainty

6.1 Uncertainty is everywhere

As in other sciences such as physics, uncertainty is starting to be widely acknowledged as consubstantial of computer science. Uncertainty is so omnipresent in modern processor micro-architectures that some work have even tried to leverage it to generate truly random numbers. For instance HAVEGE (HArdware Volatile Entropy Gathering and Expansion) [74] uses the hardware clock cycle counter of the processor to gather part of the entropy/uncertainty introduced by operating system interrupts in the internal states of the processor. Since the internal state of HAVEGE includes thousands of internal volatile hardware states, it is impossible in practice to reproduce the generated sequences. Two executions of the very same program on the very same processor would then produce different results with an extremely high probability.

For most applications however, this would be considered more as a *bug* than as a *feature*. For instance, reference [55] complains that:

> *Our results and experience with this work suggest that the default assumption should be that Earth System Model are not replicable under changes in the HPC environment, until proven otherwise.*

In the same way, reference [57] gives an excellent account of the enormous problems they faced when just trying to reproduce/replicate experiments in computational fluid dynamics. Since Navier-Stokes equations are nonlinear and can exhibit chaotic behavior under certain conditions, even a slight variation

anywhere in the hardware/software stack could have important consequences:

> *Computational science and engineering makes ubiquitous use of linear algebra libraries like PETSc, Hypre, Trilinos and many others. Rarely do we consider that using different libraries might produce different results. But that is the case. Sparse iterative solvers use various definitions of the tolerance criterion to exit the iterations, for example. The very definition of residual could be different. This means that even when we set the same value of the tolerance, different libraries may declare convergence differently! This poses a challenge to reproducibility, even if the application is not sensitive to algebraic error. The situation is aggravated by parallel execution. Global operations on distributed vectors and matrices are subject to rounding errors that can accumulate to introduce uncertainty in the results.*

A further dimension of the problem lies into the fact that these causes of uncertainty may not always be orthogonal. It has been shown in [49, 48] that several layers can interact in non monotonous ways, *e.g.,* between compile-time and run-time options. In [47] we have coined the term *Deep Software Variability* to refer to the interaction of all external layers modifying the behavior or non-functional properties of a software. Deep software variability challenges practitioners and researchers: the combinatorial explosion of the epistemic and aleatory variability causes complicates the understanding, and thus the design, the configuration, the maintenance, the debug, and the test of software systems [41].

6.2 Why it does matter

In [27], authors show the lack of reproducibility of neuroimaging analyses across operating systems, due to floating point arithmetic issues. Since the IEEE 754 standard for floating point computations does not require correct rounding of mathematical functions, users will get different results with different libraries, or even with different versions of the same library.

Beyond these floating-points issues, neuroimaging studies are characterized by a very large variation space. To build their analyses, practitioners must indeed choose among several possible methods, software versions, algorithms, threshold parameters, etc. For many years, those choices have been considered as “implementation details" but evidence is growing that they can lead to different and sometimes contradictory results. For instance, the same dataset of functional Magnetic Resonance Imaging (fMRI) results was independently analyzed by 70 teams, testing 9 ex-ante hypotheses [19]. Significant variations appeared in reported results, with substantial effects on scientific conclusions, thus jeopardizing the confidence one could have in these studies. That might ultimately lead to wrong medical diagnosis, hence compromising people health, which is a growing concern in this domain [43].

Reference [66] investigates the effect of something as trivial as random seed selection on the accuracy of computer vision when using popular deep learning architectures. It turns out that it is surprisingly easy to find outlier seeds that perform much better or much worse than the average. This means that the prediction given by a neural network for important things such as *Is this a terrorist or a woman with a baby?* might heavily depends on the arbitrary choice of a seed in the PyTorch library.

Even with simple things such as integer-only computations, different results can be obtained from search-based algorithms because of time limits in the

search for the best solution [51]. So even differences in non functional properties such as computation time can lead to differences in results [78]. At the level of a single component, it seldom is a big issue. However modern software intensive systems are made of thousands of components, and small differences propagate in widely non-linear ways that may ultimately trigger systemic catastrophes. This is one of the reasons why the aerospace industries still insist on securing full deterministic control on each of their components, at the expense of an exponentially growing cost/performance ratio [62]. Considering that kind of issues, Moshe Vardi recently wrote [81] about software that *the hope for 'mathematical certainty' was idealized and not fully realistic*.

We should now raise our head out of the sand and face it, i.e. embrace uncertainty in informatics.

6.3 Embracing Uncertainty with the help of MDE

Despite its shortcomings outlined in Section 4.4, we claim that MDE is helpful and probably even necessary to embrace uncertainty in informatics, both epistemic and aleatory. A lot of efforts have recently be made to deal with uncertainty in domain models [8]. There is a real epiphany in the software engineering community on the fact that there is no hope of completely removing all sources of uncertainty from a computation, and thus that we have to assume that the result of a computation is *stochastic*.

We must then deal with that with statistical methods to build confidence envelopes of computations along several dimensions such as *e.g.,* accuracy, time, or power consumption. Our vision is that there is a need to systematically identify and explicit points of variability that may give rise to uncertainty issues (*e.g.,* language, libraries, compiler, virtual machine, OS, processor, environment variables, etc.). Capturing and modeling variability at different layers would make it possible to use MDE to reason about the configuration space and build confidence envelopes of computations.

To give a very simple example, if we realize that in a C program, $a+(b+c)$ could also have been written as $(a+b)+c$ or even $(c+b)+a$, and that the program is compiled with the option -O2 but it could also be -O3, then we could model these variability points using the formalism of feature models, as illustrated in Figure 3

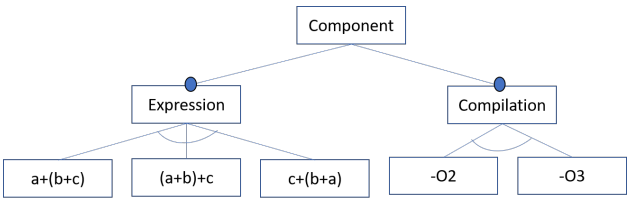

Figure 3: Simple Feature Model

Then we could resort to statistical reasoning *à la* Monte Carlo to obtain an *uncertainty profile* for software components, along several axes such as accuracy, response time, power consumption or reliability. Monte Carlo methods are a set of approaches that rely on repeated random sampling to obtain numerical results, such as an estimate of the probability distributions that we need here. To extrapolate from observed executions, applied statistics provides a rich set of tools and methodologies that can be adopted for our purposes.

The result of the random sampling would produce *resolved* models, which are the result of choosing a specific configuration among all the offered possibilities. For example, from the feature model of Figure 3 we could select $a+(b+c)$ and $-O3$, and automatically obtain a particular variant of our component, while a choice of $(a+b)+c)$ and $-O2$ would yield another one. Of course, other variation points, such as the chosen programming language, the operating system, the processor, etc. can well be specified and

then systematically explored in a similar way. Then running these variants would provide a set of results, as illustrated in Figure 4. From this sample of execu-

`pos.c:`	`pos.c:`
... x=a+(b+c);	... x=(a+b)+c;
`$ gcc -O2 pos.c`	`$ gcc -O3 pos.c`
`$ pos`	`$ pos`
$\rightarrow x = 0.999$	$\rightarrow x = 1.0001$
$\rightarrow time = 1.01$	$\rightarrow time = 0.99$

Figure 4: Compiling and running 2 variants

tions, we can come with an envelope for the `pos.c` component:

$$x = 1 \pm 0.001, time = 1 \pm 0.01$$

Of course, turning this simple idea into reality is subject to many open challenges as described below.

6.4 Open Challenges

The **first challenge** is to *extract latent deep variability* in a given application domain, from the processor layer up to the application and its input data through all intermediate layers. This is challenging due to both the range of expertise that is needed and the scale of the resulting variability, expected to be many orders of magnitude larger than anything existing up to now.

The **second challenge** is to *model and manage deep variability* in such a way that it is amenable to *automatic processing* using MDE. Indeed statistical sampling requires many different variants to be automatically and efficiently built across the many different formalisms used along the deep variability stack — from processors, infrastructure and OSs configurations, compiler and library configurations, up to applicative configurations and input data. This automated build of variants from a variability model, known as *realization* in the Software Product Line (SPL) community only exists in limited contexts, most often with no more than one or two formalisms. We need to invent generic ways to implement realization across many different formalisms so that it could scale up to deep variability needs. Greal, a MDE approach described in [23], could be a first step into that direction.

The **third challenge** is to *estimate a component's uncertainty profile*, that is the probability distribution of its behavior along several axes such as accuracy, response time, power consumption, using uniform statistical sampling in a cost efficient way.

Statistical methods heavily depend on their ability to sample the search space (the *satisfiable* space of a feature model in our case) in such a way that samples are independent and identically distributed. Developing such uniform sampling methods remains a challenge because on the one hand there are numerous constraints among options (*e.g.,* some options are mutually exclusive) and on the other hand the number of options (hence variables) is many orders of magnitude beyond the capabilities of state-of-the-art SAT samplers ($\approx 10^{10.000}$ vs. $\approx 10^{300}$). Large scale uniform sampling of a non uniform (because highly-constrained) configuration space is an extremely active research field that has recently made huge breakthroughs: Heradio et al. [29] reported on BDDSampler, a new tool vastly surpassing previous efforts at doing a uniform sampling of highly configurable systems in a scalable way. Still, BDDSampler (as well as most of its competitors) rely on Binary Decision Diagrams (BDD) technology. Synthesizing the BDD encoding of a variability model is sometimes unattainable because the variable ordering chosen to build a BDD dramatically impacts its size, and finding the optimal ordering is an NP-complete problem. For instance, synthesizing the BDD of the Linux kernel is still an open problem [77]. In practice, there is thus a need to find operational trade-offs that are close to

uniformity with a controlled uncertainty, using for instance the notion of *Feature Importance*, *i.e.,* the fact that variability factors can be ranked w.r.t. the impact they have on the performance of a component [54]. Such importance can be automatically learned or can come from experts' knowledge that would typically prioritize variability of interest.

Another problem is measuring properties of interest (speed, energy, accuracy) without perturbing too much the computation [9]. Last but not least, testing the behavior of software components under multiple different assumptions and configurations has a cost, both in terms of time and energy. Ideally, we should run the same computation hundreds of times to get an idea of its uncertainty profile. When a component is already costly to run, that is highly unrealistic. All of this asks for new statistical methods with built-in support for tuning their accuracy vs. cost ratio.

If these challenges can be overcome, we could foresee the possibility of safe statistical reasoning about uncertain components. It could have a high impact on (1) the way mission critical software systems would be developed *e.g.,* aligning software engineering practices with system engineering ones; and on (2) providing confidence intervals for predictions coming out of scientific models in geosciences, health, etc.: *e.g.,* no more misleading medical diagnosis due to deep variability.

7 Conclusion

In this paper we claimed that uncertainty management has long been an unformulated key issue in software development, and certainly one of the main underlying drivers of MDE. We supported this claim by looking back at the history of trends in MDE to see what particular problem of uncertainty management these trends actually wanted implicitly or explicitly to address. We have shown that MDE progressively came after more and more aspects of uncertainty, from epistemic to aleatory. While uncertainties may be mitigated as more detailed information becomes available, early exploration of performance can inform developers and product owners about the consequences of different resolutions of uncertainty.

Explicit recognition and representation of uncertainty are crucial for effective management, requiring training and procedures within the software development process. This explicit representation of uncertainty in modeling is essential for ensuring that vital information about uncertainty is not confined solely to the minds of engineers. Ongoing research aims to improve the representation of uncertainty in modeling through proposed models and language extensions, *e.g.,* in UML [7, 87].

However, an inherent contradiction arises when representing uncertainty in models, as models themselves are uncertain representations of reality or developer intentions. Striking a balance between representing uncertainty in models and acknowledging the inherent uncertainties within the models poses a trade-off, emphasizing the need for further research.

Despite efforts to study and mitigate uncertainties in specific cases, handling uncertainties systematically remains an open challenge. A methodology supporting the identification, modeling, and mitigation of uncertainties could greatly assist engineers in addressing various forms and occurrences of uncertainty. Some uncertainties, particularly those related to unknowable features or eventual system context and usage, may resist mitigation by design methodologies. In these cases, it becomes crucial to qualify or quantify the importance and potential effects of uncertainty and to document its existence.

References

[1] M. Andreu, M. Haziza, C. Jard, and J.-M. Jézéquel. Analyzing a space-protocol: from specification, simulation to experimentation. In *Proc. of the Fifth International Conference on Formal Description Techniques*, Perros-Guirrec, France, Oct. 1992.

[2] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming. In R. Glück and M. Lowry, editors, *Generative Programming and Component Engineering*, pages 125–140, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[3] D. Arcelli, V. Cortellessa, and C. Trubiani. Performance-based software model refactoring in fuzzy contexts. In *Proceedings of International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 149–164, 2015.

[4] M. Baudin, A. Dutfoy, B. Iooss, and A.-L. Popelin. *OpenTURNS: An Industrial Software for Uncertainty Quantification in Simulation*, pages 1–38. Springer, 2016. `https://openturns.github.io/`.

[5] S. Bernardi, M. Famelis, J.-M. Jézéquel, R. Mirandola, D. P. Palacin, F. Polack, and C. Trubiani. *Living with Uncertainty in Model-Based Development*, pages 159–185. Springer International Publishing, July 2021.

[6] M. F. Bertoa, L. Burgueño, N. Moreno, and A. Vallecillo. Incorporating measurement uncertainty into OCL/UML primitive datatypes. *Softw. Syst. Model.*, 19(5):1163–1189, 2020.

[7] M. F. Bertoa, N. Moreno, G. Barquero, L. Burgueño, J. Troya, and A. Vallecillo. Expressing measurement uncertainty in OCL/UML datatypes. In A. Pierantonio and S. Trujillo, editors, *Modelling Foundations and Applications*, pages 46–62. Springer, 2018.

[8] 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 (TOSEM), To appear.*, 2022.

[9] M. Böhme. Statistical reasoning about programs. In *NIER Track of ICSE*, May 2022.

[10] J. Cabot. Positioning of the low-code movement within the field of model-driven engineering. pages 1–3, 10 2020.

[11] V. Cardellini, T. G. Grbac, M. Nardelli, N. Tanković, and H.-L. Truong. Qos-based elasticity for service chains in distributed edge cloud environments. In *Autonomous Control for a Reliable Internet of Services*, pages 182–211. Springer, Cham, 2018.

[12] W. Cazzola, J.-M. Jézéquel, and A. Rashid. Semantic join point models: Motivations, notions and requirements. In *SPLAT 2006 (Software Engineering Properties of Languages and Aspect Technologies)*, Mar. 2006.

[13] R. C. H. Cheng and W. Holland. Calculation of confidence intervals for simulation output. *ACM Trans. Model. Comput. Simul.*, 14(4):344—-362, Oct. 2004.

[14] L. Cheung, L. Golubchik, N. Medvidovic, and G. Sukhatme. Identifying and addressing uncertainty in architecture-level software reliability modeling. In *2007 IEEE International Parallel*

and Distributed Processing Symposium, pages 1–6, Mar. 2007.

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

[16] E. W. Dijkstra. *On the Role of Scientific Thought*, pages 60–66. Springer New York, New York, NY, 1982.

[17] N. Esfahani, E. Kouroshfar, and S. Malek. Taming uncertainty in self-adaptive software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE'11, pages 234–244. ACM, 2011.

[18] N. Esfahani and S. Malek. *Uncertainty in Self-Adaptive Software Systems*, pages 214–238. Springer, 2013.

[19] R. B.-N. et al. Variability in the analysis of a single neuroimaging dataset by many teams. *Nature*, 582(7810):84–88, June 2020.

[20] L. Etxeberria, C. Trubiani, V. Cortellessa, and G. Sagardui. Performance-based selection of software and hardware features under parameter uncertainty. In *Proceedings of the International Conference on Quality of Software Architectures (QoSA)*, pages 23–32, 2014.

[21] W. Feller. *An Introduction to Probability Theory and Its Applications*. Wiley, 2008.

[22] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA 2000*, Oct. 2000.

[23] D. Foures, M. Acher, O. Barais, B. Combemale, J.-M. Jézéquel, and J. Kienzle. Experience in Specializing a Generic Realization Language for SPL Engineering at Airbus. In *MODELS 2023 - 26th International Conference on Model-Driven Engineering Languages and Systems*, pages 1–12, Västerås, Sweden, Oct. 2023. ACM and IEEE, IEEE.

[24] R. France, F. Fleurey, R. Reddy, B. Baudry, and S. Ghosh. Providing support for model composition in metamodels. In *EDOC'07 (Entreprise Distributed Object Computing Conference)*, Annapolis, MD, USA, 2007.

[25] R. Fujimoto. Exploiting temporal uncertainty in parallel and distributed simulations. In *Proc. of PADS'99*, pages 46–53. IEEE Computer Society, 1999.

[26] C. Ghezzi, L. S. Pinto, P. Spoletini, and G. Tamburrelli. Managing non-functional uncertainty via model-driven adaptivity. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 33–42, Piscataway, NJ, USA, 2013. IEEE Press.

[27] T. Glatard, L. Lewis, R. Ferreira Da Silva, R. Adalat, N. Beck, C. Lepage, P. Rioux, M.-E. Rousseau, T. Sherif, E. Deelman, N. Khalili-Mahani, and A. Evans. Reproducibility of neuroimaging analyses across operating systems. *Frontiers in Neuroinformatics*, pages 1–14, 2015.

[28] M. B. Gordy and S. Juneja. Nested simulation in portfolio risk measurement. *Management Science*, 56:1833—-1848, Aug. 2010.

[29] R. Heradio, D. Fernández-Amorós, J. A. Galindo, D. Benavides, and D. S. Batory. Uniform and scalable sampling of highly configurable systems. *Empir. Softw. Eng.*, 27(2):44, 2022.

[30] S. M. Hezavehi, D. Weyns, P. Avgeriou, R. Calinescu, R. Mirandola, and D. Perez-Palacin. Uncertainty in self-adaptive systems: A research community perspective. *ACM Trans. Auton. Adapt. Syst.*, 15(4), dec 2021.

[31] W. M. Ho, J.-M. Jézéquel, F. Pennaneac'h, and N. Plouzeau. A toolkit for weaving aspect oriented UML designs. In *Proceedings of 1st ACM International Conference on Aspect Oriented Software Development, AOSD 2002*, Enschede, The Netherlands, Apr. 2002.

[32] P. Istoan, G. Nain, G. Perrouin, and J.-M. Jézéquel. Dynamic Software Product Lines for Service-Based Systems. In *9th IEEE International Conference on Computer and Information Technology*, Xiamen, CHINA, China, 2009.

[33] A. Jackson, O. Barais, J.-M. Jézéquel, and S. Clarke. Toward a generic and extensible merge operator. In *Models and Aspects workshop, at ECOOP 2006*, Nantes, France, July 2006.

[34] P. Jamshidi, A. Sharifloo, C. Pahl, H. Arabnejad, A. Metzger, and G. Estrada. Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures. In *Proceedings of the International Conference on Quality of Software Architectures (QoSA)*, pages 70–79, 2016.

[35] JCGM 100:2008. *Evaluation of measurement data—Guide to the expression of uncertainty in measurement (GUM)*. Joint Com. for Guides in Metrology, 2008.

[36] J. Jézéquel. Reifying configuration management for object-oriented software. In K. Torii, K. Futatsugi, and R. A. Kemmerer, editors, *Forging New Links, Proceedings of the 1998 International Conference on Software Engineering, ICSE 98, Kyoto, Japan, April 19-25, 1998*, pages 240–249. IEEE Computer Society, 1998.

[37] J.-M. Jézéquel. Experience in validating protocol integration using Estelle. In *Proc. of the Third International Conference on Formal Description Techniques, Madrid, Spain*, November 1990.

[38] J.-M. Jézéquel. Model Driven Design and Aspect Weaving. *Software and Systems Modeling*, 7(2):209–218, 2008.

[39] J.-M. Jézéquel and A. Vallecillo. Uncertainty-aware Simulation of Adaptive Systems. *ACM Transactions on Modeling and Computer Simulation*, pages 1–18, Mar. 2023.

[40] A.-L. Jousselme, P. Maupin, and E. Bossé. Uncertainty in a situation analysis perspective. In *6th International Conference of Information Fusion*, pages 1207 – 1214. IEEE, 2003.

[41] J. Kienzle, B. Combemale, G. Mussbacher, O. Alam, F. Bordeleau, L. Burgueño, G. Engels, J. Galasso, J.-M. Jézéquel, B. Kemme, S. Mosser, H. Sahraoui, M. Schiedermeier, and E. Syriani. Global Decision Making Over Deep Variability in Feedback-Driven Software Development. In *ASE 2022 - 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–6, Rochester, MI, United States, Oct. 2022. IEEE.

[42] J. Klein, L. Hélouet, and J.-M. Jézéquel. Semantic-based weaving of scenarios. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06)*, Bonn, Germany, Mar. 2006. ACM.

[43] B. Knowles, A. Smith-Renner, F. Poursabzi-Sangdeh, D. Lu, and H. Alabi. Uncertainty

in current and future health wearables. *Communications of the ACM*, 61(12):62–67, Nov. 2018.

[44] T. Kobayashi, R. Salay, I. Hasuo, K. Czarnecki, F. Ishikawa, and S. Katsumata. Robustifying controller specifications of cyber-physical systems against perceptual uncertainty. In *Proc. of NASA Formal Methods 2021*, volume 12673 of *LNCS*, pages 198–213. Springer, 2021.

[45] E. O. Lebigot. Uncertainties package. `https://pythonhosted.org/uncertainties/`, 2016. Accessed: May 30, 2022.

[46] A. Lee. SOERP uncertainties package. `https://pypi.org/project/soerp/`, 2013. Accessed: May 30, 2022.

[47] L. Lesoil, M. Acher, A. Blouin, and J.-M. Jézéquel. Deep Software Variability: Towards Handling Cross-Layer Configuration. In *VaMoS 2021 - 15th International Working Conference on Variability Modelling of Software-Intensive Systems*, pages 1–8, Krems / Virtual, Austria, Feb. 2021.

[48] L. Lesoil, M. Acher, A. Blouin, and J.-M. Jézéquel. Beware of the Interactions of Variability Layers When Reasoning about Evolution of MongoDB. In *ICPE 2022 - 13th ACM/SPEC International Conference on Performance Engineering*, pages 1–5, Beijing, China, Apr. 2022.

[49] L. Lesoil, M. Acher, X. Tërnava, A. Blouin, and J.-M. Jézéquel. The Interplay of Compile-time and Run-time Options for Performance Prediction. In *SPLC 2021 - 25th ACM International Systems and Software Product Line Conference - Volume A*, pages 1–12, Leicester, United Kingdom, Sept. 2021. ACM.

[50] B. Liu. *Uncertainty Theory*. Springer, 5 edition, 2018.

[51] A. Lodi and A. Tramontani. Performance variability in mixed-integer programming. In *INFORMS TutORials in Operations Research*, 2014. https://doi.org/10.1287/educ.2013.0112.

[52] M. L. Loper and R. M. Fujimoto. Pre-sampling as an approach for exploiting temporal uncertainty. In *Proc. of PADS'00*, pages 157–164. IEEE Computer Society, 2000.

[53] S. Mahdavi-Hezavehi, P. Avgeriou, and D. Weyns. *A Classification Framework of Uncertainty in Architecture-Based Self-Adaptive Systems With Multiple Quality Requirements*, chapter 3, pages 45–77. Morgan Kaufmann, Boston, 2017.

[54] H. Martin, M. Acher, J. A. Pereira, L. Lesoil, J.-M. Jézéquel, and D. E. Khelladi. Transfer Learning Across Variants and Versions: The Case of Linux Kernel Size. *IEEE Transactions on Software Engineering*, 48(11):4274–4290, Nov. 2022.

[55] F. Massonnet, M. Ménégoz, M. Acosta, X. Yepes-Arbós, E. Exarchou, and F. J. Doblas-Reyes. Replicability of the EC-Earth3 Earth system model under a change in computing environment. *Geoscientific Model Development*, 13(3):1165–1178, Mar. 2020.

[56] I. Meedeniya, I. Moser, A. Aleti, and L. Grunske. Architecture-based reliability evaluation under uncertainty. In *Proceedings of the International Conference on Component-Based Software Engineering and Software Architecture (CompArch)*, pages 85–94, 2011.

[57] O. Mesnard and L. A. Barba. Reproducible and replicable computational fluid dynamics: It's

harder than you think. *Computing in Science & Engineering*, 19(4):44–55, 2017.

[58] J. Miller and J. Mukerji. Model driven architecture. https://www.omg.org/cgi-bin/doc?ormsc/2001-07-01, 7 2001.

[59] B. Morin, O. Barais, and J.-M. Jézéquel. Weaving Aspect Configurations for Managing System Variability. In *2nd International Workshop on Variability Modelling of Software-intensive Systems*, Essen, Germany, Germany, 2008.

[60] B. Morin, G. Perrouin, P. Lahire, O. Barais, G. Vanwormhoudt, and J. Jézéquel. Weaving variability into domain metamodels. In *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009, Denver, CO, USA, October 4-9, 2009. Proceedings*, volume 5795 of *Lecture Notes in Computer Science*, pages 690–705. Springer, 2009.

[61] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of MODELS/UML'2005*, Montego Bay, Jamaica, Oct. 2005.

[62] J. Nowotsch and M. Paulitsch. Leveraging multi-core computing architectures in avionics. In *2012 Ninth European Dependable Computing Conference*, pages 132–143, 2012.

[63] V. Ortiz, L. Burgueño, A. Vallecillo, and M. Gogolla. Native support for UML and OCL primitive datatypes enriched with uncertainty in USE. In *Proc. of OCL@MODELS'19*, volume 2513 of *CEUR Workshop Proceedings*, pages 59–66. CEUR-WS.org, 2019.

[64] M. Padulo and M. D. Guenov. A methodological perspective on computational engineering design under uncertainty. In *European Congress on Computational Methods in Applied Sciences and Engineering*, pages 7509 – 7528. T.U. Wien, 2012.

[65] G. Perrouin, F. Chauvel, J. Deantoni, and J.-M. Jézéquel. Modeling the Variability Space of Self-Adaptive Applications. In S. Thiel and K. Pohl, editors, *2nd Dynamic Software Product Lines Workshop (SPLC 2008, Volume 2)*, pages 15–22, Limerick, Ireland, Ireland, 2008. IEEE Computer Society.

[66] D. Picard. torch.manual_seed(3407) is all you need: On the influence of random seeds in deep learning architectures for computer vision. Arxiv, 2021.

[67] K. Pohl, G. Böckle, and F. van der Linden, editors. *Software Product Line Engineering*. Springer, 2006.

[68] A. J. Ramirez, A. C. Jensen, and B. H. C. Cheng. A taxonomy of uncertainty for dynamically adaptive systems. In *7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2012*, 2012.

[69] R. Ramos, O. Barais, and J.-M. Jézéquel. Matching model-snippets. In *In Proceedings of ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS 07)*, Nashville, TN, USA, October 2007.

[70] A. Rashid, A. Moreira, and J. Araújo. Modularisation and composition of aspectual requirements. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, AOSD*, pages 11–20. ACM, 2003.

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

[72] H. Saadawi and G. A. Wainer. Rational time-advance DEVS (RTA-DEVS). In *Proc. of SpringSim'10*, pages 143:1–143:8. SCS/ACM, Apr. 2010.

[73] H. Samin, N. Bencomo, and P. Sawyer. Decision-making under uncertainty: be aware of your priorities. *Softw. Syst. Model.*, 2022.

[74] A. Seznec and N. Sendrier. Havege: A user-level software heuristic for generating empirically strong random numbers. *ACM Trans. Model. Comput. Simul.*, 13(4):334–346, oct 2003.

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

[76] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*, pages 107–119, 1999.

[77] T. Thüm. A bdd for linux? the knowledge compilation challenge for variability. In *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*, SPLC '20, New York, NY, USA, 2020. Association for Computing Machinery.

[78] L. Traini, D. Di Pompeo, M. Tucci, B. Lin, S. Scalabrino, G. Bavota, M. Lanza, R. Oliveto, and V. Cortellessa. How software refactoring impacts execution time. *ACM Trans. Softw. Eng. Methodol.*, 31(2), dec 2021.

[79] J. Troya, N. Moreno, M. F. Bertoa, and A. Vallecillo. Uncertainty representation in software models: a survey. *Softw. Syst. Model.*, 20(4):1183–1213, 2021.

[80] C. Trubiani, I. Meedeniya, V. Cortellessa, A. Aleti, and L. Grunske. Model-based performance analysis of software architectures under uncertainty. In *Proceedings of the International Conference on Quality of Software Architectures (QoSA)*, pages 69–78, 2013.

[81] M. Y. Vardi. Program verification: Vision and reality. *Commun. ACM*, 64(7):5, jun 2021.

[82] D. Vicino, G. A. Wainer, and O. Dalle. Uncertainty on discrete-event system simulation. *ACM Trans. Model. Comput. Simul.*, 32(1):2:1–2:27, 2022.

[83] D. Weyns, N. Bencomo, R. Calinescu, J. Cámara, C. Ghezzi, V. Grassi, L. Grunske, P. Inverardi, J.-M. Jézéquel, S. Malek, R. Mirandola, M. Mori, and G. Tamburrelli. *Perpetual Assurances for Self-Adaptive Systems*. 2017.

[84] Wikipedia. List of uncertainty propagation software. `https://en.wikipedia.org/wiki/List_of_uncertainty_propagation_software`, Accessed: May 30, 2022.

[85] B. Wittenmark. Stochastic adaptive control methods: a survey. *International Journal of Control*, 21(5):705–730, 1975.

[86] B. P. Zeigler, A. Muzy, and E. Kofman. *Theory of modeling and design: Discrete Event and Iterative System Computational Foundations*. Academic Press, 3 edition, 2018.

[87] M. Zhang, S. Ali, T. Yue, R. Norgren, and O. Okariz. Uncertainty-wise cyber-physical system test modeling. *Software & Systems Modeling*, 18(2):1379–1418, 2019.

[88] H. Zhu, T. Liu, and E. Zhou. Risk quantification in stochastic simulation under input uncertainty. *ACM Trans. Model. Comput. Simul.*, 30(1):1:1–1:24, Feb. 2020.

[89] T. Ziadi, L. Hélouët, and J.-M. Jézéquel. Towards a UML profile for software product lines. In *Proceedings of the Fifth International Workshop on Product Familly Engineering (PFE-5)*, RENNES, France, 2003.

[90] T. Ziadi and J.-M. Jézéquel. *Product Line Engineering with the UML: Deriving Products*, pages 557–586. Number ISBN: 978-3-540-33252-7. Springer Verlag, 2006.

[91] H.-J. Zimmermann. *Fuzzy Set Theory – and Its Applications*. Springer Science+Business Media, 2001.