Managing Design-time Uncertainty in OCL Expressions

Robert Clarisó*, Lola Burgueño*, and Jordi Cabot†,*
*Universitat Oberta de Catalunya, Spain
†ICREA, Spain

ABSTRACT During the software design phase software models are created. These models must be *eventually* complete and correct. But achieving this state is challenging, and even more in early phases of development, where there are still plenty of unknown details about the system-to-be. This design-time uncertainty especially hinders the definition of those model elements that require a high level of precision: the system's business rules.
Nevertheless, waiting for the requirements to be clear to build a complete model is not an efficient option, and often not even a feasible one as this could delay too much the whole project. Instead, uncertain models (*i.e.*, models where only some details are present or are present up to a certain degree of certainty) can be used to advance other aspects of the development process (*e.g.*, building a partial prototype that can help us explore architectural decisions) while, in parallel, we keep refining the models with the stakeholders.
This paper proposes an extension to the OCL language to be able to specify uncertain OCL constraints and shows how we could operate with them to perform preliminary quality checks on the partial business rules. Moreover, we propose a prototype tool implementing this approach and discuss potential extensions.

KEYWORDS design, uncertainty, OCL.

1. Introduction

Designing software systems is a complex process that requires domain knowledge and the participation of all relevant stakeholders. Errors in a software design such as missing or incorrect information may cause faults at later stages of the development process, when they are more expensive to fix (Boehm & Basili 2001; Endres & Rombach 2003). Thus, there is a strong incentive to ensure the quality of the artifacts built in the software design phase.

On the other hand, design is an early stage of the software development process. Therefore, designers may not yet have a clear understanding about the domain or the business rules of the system being developed. Furthermore, different stakeholders may be unable to reach an agreement regarding the expected behavior in specific scenarios. Finally, it may be desirable to postpone some design decisions until later stages of the design process, when more information is available. In all of these situations, it is useful to capture the uncertainty during the design process instead of just waiting for all these aspects to be clear.

So far, design uncertainty has limited to core structural modeling concepts, such as the existence of a particular attribute or association. Nevertheless, an aspect where design uncertainty is particularly significant, and that has not been addressed yet, is the definition of the *integrity constraints* of a software system. Integrity constraints may describe precise details about system behavior or the validity and consistency of the information base. These details may be unknown or undecided at this stage of the development, so it is desirable to define and manage uncertain integrity constraints.

In this paper, we target the definition of integrity constraints in UML diagrams using the Object Constraint Language (OCL). OCL constraints take the form of class invariants that must hold true for all instances of that class or pre/post-conditions of operations. We propose an extension of the OCL notation that supports the definition of uncertain integrity constraints and

JOT reference format:
Robert Clarisó, Lola Burgueño, and Jordi Cabot. *Managing Design-time Uncertainty in OCL Expressions.* Journal of Object Technology. Vol. 21, No. 4, 2022. Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2022.21.4.a8

their analysis using existing off-the-shelf tools for the verification and validation of OCL. We also present a prototype tool implementing the proposed method. Finally, we illustrate the operation of this method with an example.

Paper organization: The remainder of the paper is structured as follows. Our method is presented in Sections 2 and 3, which considers two complementary activities: the definition of OCL constraints that incorporate uncertainty (Section 2) and the analysis of OCL constraints that incorporate uncertainty, together with an implementation of the method (Section 3). Then, Section 4 describes an example to illustrate the operation of this method in practice. Section 5 outlines some of the challenges, limitations and alternatives of the proposed notation. Section 6 describes related work on modeling design-time uncertainty and distinguishes it from other types of uncertainty modeling. Finally, Section 7 concludes and discusses future work.

2. Design-Time Uncertainty in OCL

When designing software, initially, the designer's knowledge of the domain and business rules may be imperfect, incomplete, underspecified etc. Due to this uncertainty some integrity constraints may be too vaguely defined to allow a precise formalization. Nevertheless, in many cases, it is beneficial to include this partial information about integrity constraints in the model and refine it in later iterations.

To this end, we extend the syntax of OCL by introducing a notation for denoting uncertainty. This notation indicates that the value of a particular subexpression within a constraint is uncertain: it could take several potential alternative values, and currently it is not possible to decide which alternative is the correct one.

Uncertain invariants It is possible for a designer to be uncertain about whether a certain constraint should exist or not in the system being developed. This concept will be denoted with the following notation:

```
[? constraint ]
```

This allows the designer to include relevant knowledge in the model without committing to enforcing this particular constraint.

Example 1 *A designer ignores whether a company's policy forbids employees from being clients of the company, but thinks that this is a reasonable assumption. This potential constraint can be included in the model while noting the uncertainty about this requirement:*

```
[? context Employee inv EmployeeNotClient:
  not Client.allInstances()→
    exists(c | c.id = self.id) ]
```

Uncertain expressions and operations The generic pattern for describing an uncertain OCL expression is the following:

```
[ alternative1, ... , alternativeN ]
```

indicating that this OCL expression (operation name) may take one of the alternative values (operations) denoted between brackets. All the alternative values should either be expressions computing a value of the same type or operation names. At some point during the design process, the uncertainty in an OCL constraint will be removed by selecting one of the alternatives as the value for this OCL subexpression.

Example 2 *An e-commerce site sells individual products and bundles of several products. The site will not sell bundles if all the products inside the bundle are unavailable, but it has not decided yet what to do if only some products are unavailable. This uncertainty can be encoded in two different ways:*

```
context Bundle inv BundleAvailability1:
  self.available =
    self.products→
      [ exists, forAll ](p | p.available)

context Bundle inv BundleAvailability2:
  self.available =
    [ self.products→exists(p | p.available),
      self.products→forAll(p | p.available) ]
```

Integer expressions For the sake of conciseness, when describing uncertain integer expressions it is possible to specify ranges of values using the syntax `lower-bound..upper-bound`, where the lower and upper bounds are defined as integer constants. One or more intervals can be specified as follows.

```
[ min1..max1, min2..max2, ...,  minN..maxN ]
```

Example 3 *Let us consider an invariant stating that, for legal reasons, the users of a system should be adults. At this moment, we might not know in which country the system will be deployed. Hence, we may not know the exact legal threshold for adulthood. However, we know it will be between 18 and 21. As a result, we may write the following uncertain invariant:*

```
context User inv Adult:
  self.age >= [ 18..21 ]
```

Labels It is possible to assign a label to one or more of the alternative values. Labels can be given to none, some, or all the alternatives, and labels must be distinct. The syntax for assigning labels is the following:

```
[ *label1* alt1, ... , *labelN* altN ]
```

Labels can be reused in other uncertain expressions within the same model. The semantics of this reuse is the following: selecting an alternative with label `L` in a given uncertain OCL expression also selects the alternatives with the same label `L` in other uncertain OCL expressions. That is, several uncertain constraints may be related among them, and making a particular choice in one of them removes the uncertainty from other choices.

Example 4 *Let us consider two uncertain invariants regarding a system that could be deployed in the European Union (EU) or the United States of America (USA). First, we consider the age threshold for adulthood and whether the user needs to consent to data collection according to the European General Data Protection Regulation (GDPR) or not. Some possible uncertain constraints are as follows:*

```
context User inv Adult:
  self.age >= [ *EU* 18, *USA* 21 ]
context User::gdprRequired(): Boolean
  post consent:
    result = [ *EU* true, *USA* false ]
```

Note that a *conflict* appears when an uncertain expression includes labels for each alternative and previous choices would require two different alternatives to be selected. In this situation, one or more of the previous choices needs to be reconsidered.

Example 5 *Let us consider the following two uncertain invariants about an e-mail system:*

```
context Message inv MaxLength:
  self.length <= [ *small*  128,
                   *medium* 256,
                   *large*  512]
context Folder inv MaxCapacity:
  [ *low* self.maxCapacity < 50,
    *high* self.maxCapacity >= 100 ]
context Folder inv CompressFolders:
  self.compressed = [ *low* true,
                      *small* false ]
```

Several scenarios are considered for the message length (small, medium or large messages) and the capacity of the server (low, high). Folders can be compressed, and according to our labels compression should be activated if the server has low capacity and deactivated if the message length is small. However, a problem arises in a scenario where labels "small" and "low" are both selected: a conflict appears and we do not know what was the intended meaning for the invariant `Compression`*.*

Unknowns In some cases, designers may have no information about the value of a particular subexpression in an OCL constraint. The designers can denote this fact by using the keyword `unknown`. This keyword acts as a place-holder to denote the part of the constraint that is not certain at the moment.

Example 6 *Users of a library system who have been sanctioned (e.g., because they have damaged books or returned them late) may be forbidden from borrowing books for some time. During the early stages of the design of a library's information system, the designer may have an incomplete picture of the situations that trigger this penalty. Hence, this situation can be described as follows:*

```
context User::borrow(b: Book)
  pre: self.pendingBooks.size() < self.quota
       and self.lateBooks->isEmpty()
       and self.damagedBooks = 0
       and unknown
  post: self.pendingBooks.includes(b)
```

Documentation The syntax presented in this paper focuses on the *definition* of uncertain constraints. However, an aspect which is as important as these definitions is the documentation of the decisions behind them. Designers need to record why certain characteristics of the integrity constraints are considered uncertain and which are not, which alternatives were considered, which were discarded and the reasons why this happened. For instance, when defining labels, it is important to document the vocabulary of labels used in a model and their meaning. This information will be useful to debug potential issues with conflicting labels.

Defining a notation and tool support for documenting these design decisions is considered out of the scope of this paper.

3. Analysis of Uncertain OCL Constraints

Several existing tools provide facilities to analyze, test, validate or verify OCL constraints included in UML models (Brucker & Wolff 2008; Cabot et al. 2014; Dania & Clavel 2016; Kuhlmann et al. 2011; Rull et al. 2015; Soeken et al. 2010; Soltana et al. 2020; Wu 2017). Nevertheless, such tools do not support uncertain OCL expressions. In the following, we describe how we can leverage existing tools in order to analyze uncertain OCL models. In this way, it may be possible to detect problems such as unsatisfiable OCL constraints or inconsistencies between OCL constraints early on, i.e., even when some of these constraints are still uncertain.

We present two alternative methods offering a different trade-off regarding the precision and cost of the analysis.

Exact method This methods enables the generation of all the potential *concrete* variants for uncertain OCL constraints. To do this, starting from the innermost uncertain OCL subexpression, we generate a copy of the OCL constraint that takes each alternative as its value. Each copy is a concrete OCL expression with no uncertainty that can be checked using existing tools with no adaptations required. Constraints including *unknown*s will be excluded from this concretization[1].

A challenge of the exact method in practice is its computational complexity. Given that a model may have n uncertain expressions (each with 2 or more alternatives), there is a risk of potential combinatorial explosion as all combinations need to be generated (at least 2^n variants or more).

Example 7 *Let us revisit the uncertain age requirement from Example 3. Using the exact method, we would generate the four concrete invariants corresponding to each potential alternative:*

```
context User inv Adult_A: self.age >= 18
context User inv Adult_B: self.age >= 19
context User inv Adult_C: self.age >= 20
context User inv Adult_D: self.age >= 21
```

[1] A more detailed analysis could consider taking into account the known part of the OCL expression.

If this system included another uncertain OCL invariant with 3 alternatives, then the system would have $4 \times 3 = 12$ concrete realizations: all combinations of concrete realizations of each uncertain invariant.

Labels can be used to prune those variants where labeling is not preserved, *i.e.*, an alternative labeled `L` is selected in an uncertain OCL expression but in another uncertain OCL expression the alternative labeled `L` is discarded.

Example 8 *Let us revisit our labeled uncertain constraints from Example 4. Even though there are four combinations of alternative values, if we take the labels into account, only two concrete realizations are possible:*

```
context User inv Adult_EU:
  self.age >= 18
context User::gdprRequired(): Boolean
  post consent_EU: result = true

context User inv Adult_USA:
  self.age >= 21
context User::gdprRequired(): Boolean
  post consent_USA: result = false
```

Again, each of these concrete realizations can be checked using existing UML/OCL analysis tools.

Approximate method In order to improve the efficiency of the analysis of uncertain OCL constraints, it is possible to consider an approximate method. This method aims to reduce the number of variants being generated by aggregating the set of potential alternative values of all alternatives. As a trade-off, this approach may produce *false negatives*: it may detect some errors, but it will be unable to detect all of them.

The criteria that we will use for this aggregation is the following. Given an uncertain OCL expression with alternatives `[ alt1, ..., altN ]`:

- If the uncertain OCL expression is of boolean type, and it appears inside boolean operators (and, or, xor, not, implies) or a universal or existential quantifier, we will only generate two concretizations:

  ```
  (alt1 and alt2 and ... and altN)
  (alt1 or alt2 or ... or altN)
  ```

  One concretization represents a system that is *more restrictive* (an under-approximation) while the other represents a system that is *more relaxed* (an over-approximation). An analysis of the OCL constraint is required to identify which of the two concretizations is the over/under-approximation.
- If the uncertain OCL expression is of numeric type and is used within a relational operator ($>, <, \leq, \geq$) or a `min`/`max`/`floor`/`round`/`abs` operation, we will generate two concretizations:

  ```
  alt1.min(alt2.min(.....altN-1.min(altN)))
  alt1.max(alt2.max(.....altN-1.max(altN)))
  ```

  That is, we will only consider the lower and upper bound of the interval. In particular, if the numeric alternatives are defined as `lower..upper`, we will directly use `lower` and `upper` as the two concretizations.

Example 9 *Let us revisit our uncertain age requirement from Example 3. Considering our approximate method of realization, only two concrete realizations would be generated:*

```
context User inv Adult_MIN: self.age >= 18
context User inv Adult_MAX: self.age >= 21
```

These two realizations allow us to test the boundary cases of the age attribute. Given that in this scenario we are using a relational operator, the extreme values may be sufficient to identify relevant properties of the constraint. For example, we may discover that an age requirement of 18 conflicts with other constraints within the system. On the other hand, it may be possible for problems to arise in one of the intermediate values (i.e., 19 and 20) that are not being tested explicitly. Thus, while this approach can help us to identify errors efficiently, it is not sufficient to ensure the lack of errors.

Implementation We have implemented a tool[2] to support designers aiming to use our uncertainty extension of OCL. The tool implements the exact method, with a simple command-line interface receiving an uncertain OCL specification as input and providing as output its set of potential concretizations. Then, these concretizations can be used as input to other tools supporting regular OCL.

The tool is implemented in Java, using JFlex and CUP for creating the uncertain OCL parser. It supports all features of the OCL extension used in the examples presented in this paper, such as the `unknown` keyword, uncertain invariants, alternative expressions, integer ranges and labels. Uncertain expressions can be nested. Moreover, the tool automatically detects and removes concretizations with conflicting labels. Nevertheless, the tool still does not support the approximate method.

4. Example

In the following, we discuss an example to illustrate how to model design uncertainty in OCL constraints. To this end, we will consider a class diagram modeling a fragment of the information system of a newly created bank, which includes several uncertain integrity constraints.

Figure 1 depicts the UML class diagram on which the integrity constraints are defined. Clients may have several types of banking products: accounts, credit cards, loans (such as mortgages) or personal insurance. Each product requires the payment of a monthly fee. For the sake of simplicity, all currency amounts used in this diagram (such as balances or fees) use Real values rather than defining a custom Currency datatype.

Clients who do a lot of business with the bank (*e.g.*, paying a large amount of fees) may be awarded a VIP status, which offers several benefits. On the other hand, clients with a low balance and/or large loans may be considered a credit risk and may be forbidden from performing certain operations.

[2] The tool can be found at https://github.com/SOM-Research/uncertain-ocl.

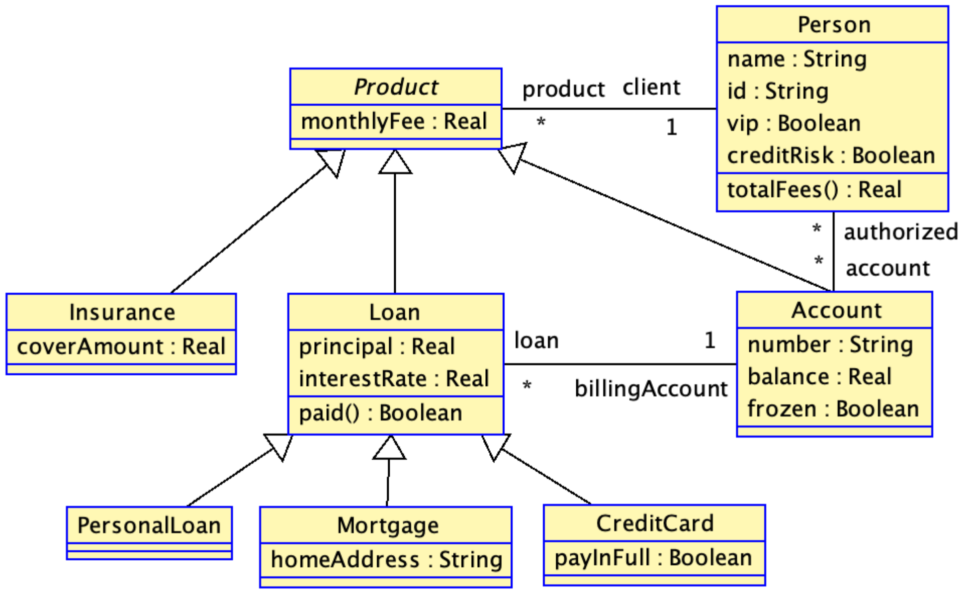

Figure 1 UML class diagram describing the information system of an online bank.

Bank accounts are special products that may be related to several clients: one is identified as the owner while the rest are considered authorized to withdraw money. Owners can decide to freeze an account, forbidding any money from being withdrawn. Accounts can also be set as the billing accounts for a particular loan, as long as the loan belongs to the owner of the account or an authorized person.

The following OCL invariants summarize some integrity constraints of this information system: fees, rates and debts should not be negative amounts; and accounts associated to loans should be billable.

```
context Product inv PositiveFees:
  self.monthlyFee > 0.0
context Loan inv NonNegativeDebts:
  self.principal >= 0.0
context Loan inv NonNegativeRates:
  self.interestRate >= 0.0
context Account inv OwnerNotInAuthorized:
  self.authorized→excludes(self.client)
context Loan inv AccountBillable:
  self.client = self.billingAccount.client or
  self.billingAccount.authorized→
     includes(self.client)
```

Nevertheless, the bank is still defining the business rules for managing some aspects of these products. For example, the following topics are still open:

– The number of unpaid mortgages that a client can have at a given point in time.

```
context Loan::paid(): Boolean
body: self.principal = 0.0

context Mortgage inv MultipleMortgages:
  if (not self.paid()) then
    Mortgage.allInstances()→select(m |
      m <> self and (not m.paid()) and
      m.client = self.client)→size() <=
    [0..2]
  else
    true
  endif
```

– What criteria will be used to identify VIP clients, *e.g.*, a threshold in the amount of monthly fees, a total balance among all the owned accounts, . . .

```
-- Total fees payed monthly by a client
context Person::totalFees(): Real
  body: self.product→collect(p:Product|
    p.monthlyFee)→sum()

-- Total balance of a client
-- Should loans be included?
context Person::totalBalance(): Real
  body: self.product→collect(p: Product |
    if (p.oclIsTypeOf(Account)) then
      p.oclAsType(Account).balance
    else [ 0.0,
      if (p.oclIsKindOf(Loan)) then
        p.oclAsType(Loan).principal
      else
          0.0
      endif ]
    endif)

context Person inv VIPClient:
  self.vip =
    [ ( self.totalFees() >= 500.0 ),
      ( self.totalBalance() >= 1000.0 ) ]
```

– What fees will be paid for accounts that are overdrawn.

All accounts are charged a base maintenance fee of 20 euros. At design time, we do not know whether overdrawn accounts will be charged an *overdraft fee*. Moreover, in case there is an overdraft fee, we have not decided yet how it will be computed. Two alternative ways of doing it are: as a a flat amount on top of the base fee (30 euros more, *i.e.* 50 euros in total) or as a percentage of the overdraft amount *instead* of the base fee (*e.g.*, 5%).

```
context Account inv NegBalanceFees:
  [ if self.balance <= 0.0 then
    self.monthlyFees =
      [ 50.0, -(self.balance*0.05) ]
  else
    self.monthlyFees = 20.0
  endif,
  self.monthlyFees = 20.0 ]
```

– What is the maximum interest rate in a loan, as setting a very high rate may be perceived as predatory behavior and generate bad publicity.

```
context Loan inv MaxInterestRate:
  self.interestRate <= unknown
```

– Whether the owner of an account can freeze it if a loan is billing that account.

```
[? context Loan inv NoFrozenAccount:
  (not self.paid()) implies
  (not self.billingAccount.frozen) ]
```

The definition of these uncertain constraints induces the following number of alternative concretizations:

Invariant	Alternatives	#
MultipleMortgages	Max 1-3 mortgages per client	3
VIPClient	Minimum fee or min balance	4
	Balance includes loans or not	
NegBalanceFees	20, 50 or 5% of balance	3
MaxInterestRate	Threshold for max rate	?
NoFrozenAccount	Can/cannot freeze billing acc.	2

That is, in this example there are $3 \times 4 \times 3 \times 2 = 72$ potential concretizations. Notice that invariant `MaxInterestRate` cannot be analyzed until more information is known about the maximum interest rate allowed. Nevertheless, being able to include this uncertain invariant reminds the designer and other stakeholders that it will be necessary to select a maximum rate at some point of the design process.

These concretizations can be used together with model finders in order to compute sample instances or identify potential inconsistencies among some of the uncertain constraints. For example, imagine that we want to test what happens in our system if we have an account with a zero balance by adding the following invariant:

```
context Account inv ZeroBalance:
  Account.allInstances()→
    exists(a: Account | a.balance = 0.0)
```

By using a model finder, we discover that 24 of the proposed concretizations have a consistency problem: in this scenario there are no valid instances that satisfy all integrity constraints simultaneously. The problem is caused by the interaction of the integrity constraint `PositiveFees` and one of the concretizations of `NegBalanceFees`, the one that charges a fee of 5% of the missing balance.

```
context Account inv NegBalanceFees_3:
  if self.balance <= 0.0
    self.monthlyFees = -(self.balance*0.05)
  else
    self.monthlyFees = 20.0
  endif
```

Notice that the fee is also computed as 5% of the balance when the balance is zero. In this situation, the fee is 5% of zero (that is, zero), so a client can avoid paying fees simply by having a balance of zero. This contradicts invariant `PositiveFees` requiring all fees to be greater than zero. The design can be modified to fix this problem in several ways: accepting zero fees in some products, adding a base flat fee to the 5% of balance when a client has negative balance or apply a regular fee in the case when the balance is zero (by rewriting the condition `self.balance <= 0.0` to `self.balance < 0.0` in invariant `NegBalanceFees`).

5. Discussion

Our proposal is a first step towards a complete support for design-time OCL uncertainty concerns. This section discusses a few possible extensions to our proposal.

5.1. Support for Real and String data types

A list of concrete Real values and String literals could be directly expressed as separate alternatives using the regular syntax for uncertain expressions.

However, when needing to express a set of Real options, our convenient *range* notation for Integer expressions cannot be directly used as there is an infinite number of potential real value options in a range. An option to overcome this would be to extend our syntax by indicating the max number of decimal digits to be used when unfolding the range, or the increase to apply to the lower bound iteratively until reaching the upper-bound.

Ranges for String values could be expressed as a (finite) regular expression. All Strings accepted by the regular expression would be part of the range.

5.2. Manipulating uncertain OCL expressions

We have seen how we can analyze and validate uncertain OCL expressions. We may want to manipulate them as part of a model-to-model or a model-to-text transformation, too. For example, when we have an uncertain OCL precondition in a

UML class method that should be translated into an equivalent *if* check for the corresponding Java class method automatically derived from the UML one. Same as we do not want to wait for all uncertainties to be clarified before analyzing the system, we may also want to be able to start creating and refining lower level models as soon as possible.

If the transformation does not use as input any uncertain OCL expression the target model is generated as usual. Otherwise, the target model must be created such that its set of potential concrete expressions is the same as if we had applied the transformation on every single input concrete variant.

Existing works covering transformation techniques for partial models (see the related work in Section 6) could be reused here, as OCL expressions can be regarded as instances of the OCL metamodel.

5.3. Not all uncertainties need to be equally uncertain

Our notation enables designers to indicate the several potential alternative values of a particular uncertain expression but all alternatives are regarded as equally probable.

We could go one step further and add a *confidence* degree to each alternative value. Then, this confidence level could be used to favour some specific scenarios in the analysis phase. In particular, this confidence could allow us to define a *default* or *preferred* alternative, which can be used to perform a quick analysis in situations where a recomputation of all concretizations is not desiarable.

5.4. Collapsing uncertain alternatives

The most challenging aspect of managing uncertain OCL expressions is the need to take into account all their possible concretizations every time we need to operate on them.

The *approximate method* from Section 3 is a good strategy to reduce the combinatorial explosion problem but we still need to deal with large expressions combining all the possible alternatives. There are several alternatives to deal with this combinatorial explosion, such as using the confidence value mentioned in the previous subsection to select the most likely alternative for a quicker analysis.

Another option would be, for every uncertain expression, to reduce the alternatives to the single most constraining one. On the one hand, this is one of the possible worst-case scenarios for the model. On the other hand, it enables the removal of the uncertainty in order to analyze the model using standard OCL tools while avoiding any type of combinatorial explosion issue. One can think of this approach like the most *extreme* version of the approximate method where we do not only reduce the number of different constraints to consider but also the size (in terms of number of subexpressions) of each constraint, speeding up their preliminary evaluation.

Another possibility is, instead of choosing the most constraining one, to do the opposite and take the less constraining one to simulate the best-case scenario.

Note that deciding the most/least constraining alternative depends on the specific operator/s appearing in the subexpression and (recursively) the parent elements of such subexpression.

6. Related Work

There is extensive work dealing with different types of uncertainty in software modeling and design such as measurement uncertainty, belief uncertainty, behaviour uncertainty and design uncertainty, as well as recent secondary studies such as the survey by Troya et al. (Troya et al. 2021), in which they report on the state of the art on representing uncertainty in software models. In this section, we focus on existing works on uncertainty in software and systems which use a model-based approach paying special attention to those works focusing on design uncertainty.

6.1. Modeling Design Uncertainty using Partial Models

Partial models are models that help represent the uncertain alternative scenarios of a system early in the design phase, which can be later refined when more information is available. The partial models contain *partialities* that are a set of notations applied to the models to represent various uncertainties (Salay et al. 2012; Salay & Chechik 2015). For instance, the *May partiality* represents the uncertainty about the presence of a model element in a model, and the *Abs partiality* represents uncertainty about the number of model elements in a model. Using partialities, it is also possible to define formal conditions for uncertainty to reduce refinement between not only individual models but within model transformations (Salay et al. 2015), too.

Partial models have been applied to solve problems of different nature. Related to our work, in (Salay et al. 2013), Salay et al. used partial models for representing uncertainties in software requirements. They use an approach to reduce uncertainty in requirements by modeling the requirements with a particular type of partiality called *MAVO* using the i* language (comprising of actors, goals, tasks and resource). Further Horkoff et al. (Horkoff et al. 2014) use the i* language to capture goal models. The i* model is converted to FOL and then to a partiality for analysis (Salay et al. 2013).

Partial models are also applied in specific areas, such as software product lines. Famelis et al. (Famelis et al. 2017) discuss the presence of uncertainty in the domain of software product line engineering and proposes different ways to resolve it.

Finally, in (Famelis & Chechik 2019), Famelis et al. propose a methodological approach for managing uncertainty using partial models that characterize the tasks needed to manage the lifecycle of uncertainty-related design decisions.

Techniques based on partial models provide an extensive insight on how uncertainty can be captured using models, in particular, state machines and timed automata. However, these models do not support the definition of integrity constraints using dedicated languages such as OCL to make them more meaningful and precise. Thus, the approach presented in this paper complements some of these other works based on partial models, as we do not capture design uncertainty in models but in the integrity constraints that enrich those models.

6.2. Design space exploration in Uncertain Model Transformations

In (Eramo et al. 2015), Eramo et al. deal with the inconsistencies due to the synchronization process in bidirectional transformations as a process of reducing uncertainty. They proposed a metamodel-independent approach to represent a solution space by means of a model with uncertainty. Later, they extended their work to translate the unknown uncertainty at design-time into known uncertainty at run-time by generating multiple choices. They achieve this by means of using delta lenses (Diskin et al. 2016). While these two works are tailored for bidirectional transformations and the particular problem caused by their non-bijective nature, our approach does not focus on this particular case. Nevertheless, our approach can be used to capture design uncertainty in model transformations written in languages with support to OCL such as ATL and QVTo. In such as case, our proposed strategies will generate alternative transformations, each one considering a possible scenario.

In (Laghouaouta & Laforcade 2020), the authors use the partial patterns to express various possibilities of transformation scenarios. Partial patterns are based on partial models (Salay et al. 2012). This approach uses a configuration model that can be seen as the specification of the uncertain transformation. As mentioned above, when comparing our approach to those based on partial models, while these focus on the transformation itself without considering integrity constraints, our work focuses on the latter.

Finally, Burgueño et al. (Burgueño et al. 2018) propose an approach to deal with measurement uncertainty in model transformations. For this, they use the concept of *confidence*, which is the degree of belief that one has on the occurrence of each model element. While this approach focuses on uncertainty at the *instance* level and assumes that the design of the model is certain, our current proposal focuses on *design-time* uncertainty.

6.3. Capturing Instance-Level Uncertainty in MBE

Vallecillo et al. (Burgueño, Mayerhofer, et al. 2019; Bertoa et al. 2020) present an approach and methodology to capture measurement uncertainty in instances of UML/OCL models. They have extended the basic UML/OCL datatypes with new datatypes such as UReal, UInteger and UString and have defined an algebra of operations for these new types.

Zhang et al. propose U-Model (Zhang et al. 2016), a detailed conceptual model to capture uncertainties in cyber-physical systems (CPS). The uncertainties are defined as lack of knowledge on inputs, states, outputs and events. Using U-Model uncertainties can be identified and captured at all levels: application, infrastructure, and integration. In (Zhang et al. 2019) the authors extend U-Model for testing purposes.

There are works that capture uncertainty of epistemic nature, this is, due to limited data, evidence or knowledge. Martin-Rodilla et al. (Martín-Rodilla & Gonzalez-Perez 2019) propose the ConML language to annotate the elements of instance models with information representing the user's confidence in the truthfulness of that element. Burgueño et al. (Burgueño, Clarisó, et al. 2019; Burgueño et al. 2022) propose an UML profile and its operationalization to express the degree of belief uncertainty that an agent has about the elements of an instance model using probabilities and subjective logic.

Unlike other previous works, Giese et al. (Giese et al. 2011) propose the use of runtime models as a means to cope with uncertainty during development-time and runtime. This approach is suitable for different types of models such as architectural and behavioral models.

All the previous works differ from our approach in the fact that they deal with uncertainty at an instance level while our approach considers uncertainty at the design level. Furthermore, even in those works which allow the user to define OCL constraints, these constraints are assumed to be correct and are not questioned. On the contrary, our proposal considers the uncertainty that a modeler may find when defining OCL constraints.

7. Conclusions

We have proposed a mechanism to extend OCL in order to support uncertainty in the definition of integrity constraints. The proposed approach allows designers to introduce uncertainty within OCL expressions. Moreover, it is possible to leverage existing tools for the analysis of OCL constraints in the analysis of this uncertain constraints. This facilitates detecting faults even in the presence of uncertain constraints.

Our current tool supports an exact method for managing uncertainty: generating all concrete variants of uncertain constraints. A limitation of this approach is the (potential) combinatorial explosion in the number of variants. As future work, we plan to address the discussion points presented in Section 5 and extend our toolset to support the approximate method (see Section 3), in order to reduce the impact of the combinatorial explosion in large specifications. We also plan to study the interplay between uncertainty about integrity constraints and uncertainty about modeling concepts, particularly in behavioral models.

Acknowledgments

The authors would like to thank Antonio Vallecillo for many years of fruitful mentoring, interesting collaborations and memorable experiences. Through his positive attitude, energy and helping nature, he has had a very important impact growing the MDE community in Spain and internationally. It has been both an honor and a pleasure working with him and we hope to continue doing so in the future.

This work is partially supported by the Spanish Agencia Estatal de Investigación through the project “LOw-COde development of Smart Software” (LOCOSS, PID2020-114615RB-I00/AEI/10.13039/501100011033).

References

Bertoa, M. F., Burgueño, L., Moreno, N., & Vallecillo, A. (2020). Incorporating measurement uncertainty into OCL/UML primitive datatypes. *Softw. Syst. Model.*, *19*(5), 1163–1189. doi: 10.1007/s10270-019-00741-0

Boehm, B., & Basili, V. R. (2001). Defect reduction top 10 list. *Computer*, *34*(1), 135–137.

Brucker, A. D., & Wolff, B. (2008). HOL-OCL: a formal proof environment for UML/OCL. In *International conference on fundamental approaches to software engineering (fase)* (Vol. 4961, pp. 97–100). doi: 10.1007/978-3-540-78743-3_8

Burgueño, L., Bertoa, M. F., Moreno, N., & Vallecillo, A. (2018). Expressing confidence in models and in model transformation elements. In *Proc. of the 21th ACM/IEEE international conference on model driven engineering languages and systems (MODELS'18)* (pp. 57–66). ACM. doi: 10.1145/3239372.3239394

Burgueño, L., Clarisó, R., Cabot, J., Gérard, S., & Vallecillo, A. (2019). Belief uncertainty in software models. In M. Chechik, D. Strüber, & D. Varró (Eds.), *Proc. of the 11th international workshop on modelling in software engineerings (MiSE@ICSE'19)* (pp. 19–26). ACM. doi: 10.1109/MiSE.2019.00011

Burgueño, L., Mayerhofer, T., Wimmer, M., & Vallecillo, A. (2019). Specifying quantities in software models. *Inf. Softw. Technol.*, *113*, 82–97. doi: 10.1016/j.infsof.2019.05.006

Burgueño, L., Muñoz, P., Clarisó, R., Cabot, J., Gérard, S., & Vallecillo, A. (2022). Dealing with belief uncertainty in software models. *Transactions on Software Engineering and Methodology*. (submitted for publication)

Cabot, J., Clarisó, R., & Riera, D. (2014). On the verification of UML/OCL class diagrams using constraint programming. *Journal of Systems and Software*, *93*, 1–23. doi: 10.1016/j.jss.2014.03.023

Dania, C., & Clavel, M. (2016). OCL2MSFOL: a mapping to many-sorted first-order logic for efficiently checking the satisfiability of OCL constraints. In *Acm/ieee 19th international conference on model driven engineering languages and systems (models)* (pp. 65–75).

Diskin, Z., Eramo, R., Pierantonio, A., & Czarnecki, K. (2016). Incorporating uncertainty into bidirectional model transformations and their delta-lens formalization. In *Proc. of the 5th international workshop on bidirectional transformations (bx'16)* (Vol. 1571, pp. 15–31).

Endres, A., & Rombach, H. D. (2003). *A handbook of software and systems engineering: Empirical observations, laws, and theories*. Pearson Education.

Eramo, R., Pierantonio, A., & Rosa, G. (2015). Managing uncertainty in bidirectional model transformations. In *Proc. of the 2015 ACM SIGPLAN international conference on software language engineering (SLE'2015)* (pp. 49–58). ACM.

Famelis, M., & Chechik, M. (2019). Managing design-time uncertainty. *Softw. Syst. Model.*, *18*(2), 1249–1284. doi: 10.1007/s10270-017-0594-9

Famelis, M., Rubin, J., Czarnecki, K., Salay, R., & Chechik, M. (2017). Software product lines with design choices: Reasoning about variability and design uncertainty. In *Proc. of the 20th ACM/IEEE international conference on model driven engineering languages and systems (MODELS'17)* (pp. 93–100). IEEE Computer Society. doi: 10.1109/MODELS.2017.3

Giese, H., Bencomo, N., Pasquale, L., Ramirez, A. J., Inverardi, P., Wätzoldt, S., & Clarke, S. (2011). Living with uncertainty in the age of runtime models. In *Models@run.time - foundations, applications, and roadmaps [dagstuhl seminar 11481, november 27 - december 2, 2011]* (Vol. 8378, pp. 47–100). Springer. doi: 10.1007/978-3-319-08915-7_3

Horkoff, J., Salay, R., Chechik, M., & Sandro, A. D. (2014). Supporting early decision-making in the presence of uncertainty. In T. Gorschek & R. R. Lutz (Eds.), *Proc. of the 22nd international conference on requirements engineering (RE'14)* (pp. 33–42). IEEE Computer Society. doi: 10.1109/RE.2014.6912245

Kuhlmann, M., Hamann, L., & Gogolla, M. (2011). Extensive validation of OCL models by integrating SAT solving into USE. In *49th international conference on objects, models, components, patterns (tools)* (Vol. 6705, pp. 290–306). doi: 10.1007/978-3-642-21952-8_21

Laghouaouta, Y., & Laforcade, P. (2020). Dealing with uncertainty in model transformations. In C. Hung, T. Cerný, D. Shin, & A. Bechini (Eds.), *Proc. of the 35th ACM/SIGAPP symposium on applied computing (SAC'20)* (pp. 1595–1603). ACM. doi: 10.1145/3341105.3373971

Martín-Rodilla, P., & Gonzalez-Perez, C. (2019). Conceptualization and non-relational implementation of ontological and epistemic vagueness of information in digital humanities. *Informatics*, *6*(2), 20. doi: 10.3390/informatics6020020

Rull, G., Farré, C., Queralt, A., Teniente, E., & Urpí, T. (2015). AuRUS: explaining the validation of UML/OCL conceptual schemas. *Software and Systems Modeling*, *14*(2), 953–980. doi: 10.1007/s10270-013-0350-8

Salay, R., & Chechik, M. (2015). A generalized formal framework for partial modeling. In A. Egyed & I. Schaefer (Eds.), *Proc. of the 18th conference on fundamental approaches to software engineering (FASE'15)* (Vol. 9033, pp. 133–148). Springer. doi: 10.1007/978-3-662-46675-9_9

Salay, R., Chechik, M., Famelis, M., & Gorzny, J. (2015). A methodology for verifying refinements of partial models. *J. Object Technol.*, *14*(3), 3:1–31. doi: 10.5381/jot.2015.14.3.a3

Salay, R., Chechik, M., Horkoff, J., & Sandro, A. D. (2013). Managing requirements uncertainty with partial models. *Requir. Eng.*, *18*(2), 107–128. doi: 10.1007/s00766-013-0170-y

Salay, R., Famelis, M., & Chechik, M. (2012). Language independent refinement using partial modeling. In *Proc. of the 15th conference on fundamental approaches to software engineering (FASE)* (Vol. 7212, pp. 224–239). Springer. doi: 10.1007/978-3-642-28872-2_16

Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., & Drechsler, R. (2010). Verifying UML/OCL models using boolean satisfiability. In *Design, automation and test in europe (date)* (pp. 1341–1344). IEEE Computer Society. doi: 10.1109/DATE.2010.5457017

Soltana, G., Sabetzadeh, M., & Briand, L. C. (2020). Practical constraint solving for generating system test data. *ACM Trans. Softw. Eng. Methodol.*, *29*(2), 11:1–11:48. doi: 10.1145/3381032

Troya, J., Moreno, N., Bertoa, M. F., & Vallecillo, A. (2021). Uncertainty representation in software models: a survey. *Softw. Syst. Model.*, *20*(4), 1183–1213. doi: 10.1007/s10270

-020-00842-1

Wu, H. (2017). MaxUSE: A tool for finding achievable constraints and conflicts for inconsistent UML class diagrams. In *International conference on integrated formal methods (ifm)* (Vol. 10510, pp. 348–356). doi: 10.1007/978-3-319-66845-1_23

Zhang, M., Ali, S., Yue, T., Norgren, R., & Okariz, O. (2019). Uncertainty-wise cyber-physical system test modeling. *Softw. Syst. Model.*, *18*(2), 1379–1418. doi: 10.1007/s10270-017-0609-6

Zhang, M., Selic, B., Ali, S., Yue, T., Okariz, O., & Norgren, R. (2016). Understanding uncertainty in cyber-physical systems: A conceptual model. In A. Wasowski & H. Lönn (Eds.), *Proc. of the 12th european conference on modelling foundations and applications (ECMFA'16)* (Vol. 9764, pp. 247–264). Springer. doi: 10.1007/978-3-319-42061-5_16

About the authors

Robert Clarisó received his BSc (2000) and PhD (2005) in Computer Science from UPC-Barcelona Tech. Since 2005, he is an associate professor at the Faculty of Computer Science, Multimedia and Telecommunications of the Universitat Oberta de Catalunya (UOC). He is also a member of the SOM Research Lab within the Internet Interdisciplinary Institute (IN3-UOC). His research interests include formal methods, model-driven engineering and tools for e-learning You can contact the author at rclariso@uoc.edu or visit https://robertclariso.github.io.

Lola Burgueño is a postdoctoral researcher and lecturer at the IN3 at the Open University of Catalonia. Her research interests focus on the application of artificial intelligence techniques to improve model-based software development processes, uncertainty management during the software design phase, model-based software testing, and the design of algorithms and tools for improving the performance of model transformations. For more information, please visit https://som-research.uoc.edu/lburgueno/.

Jordi Cabot is an ICREA Research Professor at Internet Interdisciplinary Institute (UOC) where he leads the Software and Systems Modeling (SOM) Research Lab. His research interests include software and systems modeling, analysis of open source / open data communities, pragmatic formal verification and the role AI can play in software engineering (and vice versa). For more information, please visit https://jordicabot.com.