From use cases to test cases via meta model-based reasoning

Position paper: work in progress

Stefan Gruner

Abstract In use cases considered harmful, Simons has analyzed the logical weaknesses of the UML use case notation and has recommended to “fix the faulty notion of dependency” (Simons: Use cases considered harmful, 29th Conference on Techn. of OO Lang. and Syst., pp 194–203, 1999). The project sketched in this position paper is inspired by Simons’ critique. The main contribution of this paper is a detailed meta model of possible relations between use cases. Later in the project this meta model is then to be formalized in a natural deduction calculus which shall be implemented in the P络LOG. As a result of such formalization a use case specification can be queried for inconsistencies as well as for test cases which must be observable after a software system is implemented based on such a use case specification. Software tool support for this method is also under development.

Keywords Use cases - Test cases - Meta model - P络LOG

1 Motivation and overview

The UML is notorious not only for its commercial popularity but also for its vagueness and ambiguity. For this reason various sub-languages of the UML have already been subject to the application of precision enhancing techniques; for example there is the well known OCL in support of UML’s structural notations (class and object diagrams), whereas a considerable number of papers deals with formal representations of UML’s state transition diagrams in more precise notations such as B [15].

Another few papers, however (see section “Related work” below), deal with the precision of UML’s use case (UC) diagrams, in spite of popular voices announcing UC diagrams as the premier language of the UML, upon which everything else depends [7]. If UC modeling is really as relevant as it is often announced to be, then great care must be taken about the precise meaning of a UC specification before any misunderstandings can propagate themselves as errors and defects in the software code being derived from it. For example, what would it mean for the processes of a “live” subject system if an actor could trigger a UC which is designed as a mandatory inclusion of another UC? Figure 1 depicts such a questionable scenario, simply for the sake of stimulating the reader’s problem awareness.

In this context it is interesting to note how the authors of [7], arguing explicitly from a commercial position, praise exactly that kind of above-mentioned ambiguity and vagueness which the scientifically minded software engineer is determined to stamp out. Therefore we1 will not take [7] too seriously their affinity to vagueness and ambiguity is concerned, but we take them seriously as far as their emphasis of the UML-UC notation as the starting point of user-centred requirements engineering in the early phases of a development project is concerned.

In contrast to the authors of [12], who only make a small subset of the UC notation accessible to FDR model checking, we aim at a theory for the full UC notation that allows not only for checking the internal logical consistency of a UC specification, but shall also enable us to—eventually—generate high-level test cases directly from a consistent UC specification.

The main contribution of this paper is a rich meta model of possible relations which could be established between UC or, more precisely, their process instances. This set of possible relation types exceeds by far the few relation types which are defined upon UC by the UML. As soon as this meta model of relations is established, meta relations (i.e. axioms and rules) can be formulated which enable the search for inconsistencies within a given UC specification as well as the search for implementation consequences arising from the given specification; these consequences should then be empirically testable after the system is implemented.

The definition and formalization of all those consistency rules, however, is ongoing project work and, therefore, not in the scope of this concept proposal paper any more.

1.1 Method

We distinguish various UC relations about which we want to reason. These relations are classified into the following categories:

Diagrammatic relations are those ones which are depicted by various types of lines between UC and actors in the standard UML diagrams [7]. We can also call these relations explicit because of the “visual” appearance in the UC diagrams.

Modal relations are those ones (between UC and/or actors) which are newly introduced in our approach, for the sake of enriching the information which is transported by a UC diagram between the stakeholders of a software development project. In our theory we introduce basically two new modal categories of UC relations:

Temporal: to reason about “before,” “during,” and “after” in several variations, and

Causal: to reason about “enable,” “trigger,” preconditions, etc.2

whereby it should be pointed that a particular temporal order does not necessarily imply a causal one. We also call these relations implicit, because they cannot be “seen” in the classical UML UC depictions.3

Note that, in principle, a UC could also be somehow related to itself in structural recursion, though this is rarely seen in industrial UC specifications. Such self-references to the UC specification level would have to be adequately interpreted in terms of their actual instances during the lifetime of the subject system. For example, a UC which itself could model a recursive system in which a parent process generates child processes of its own kind. Or for example, if there is mutual exclusion relation of a UC from itself on the level of specification (UC diagram), then we would have actually modeled a “singleton pattern” to such an extent that no two process instances of that UC can be alive in the subject system at the same moment in time.

Consequently we must distinguish clearly between UC as descriptions and their process instances, in analogy to the distinction between classes and their object instances in OOP.

For the sake of well-behaved software systems derived from such UC descriptions we stipulate:

- The possible infinity of a system stems from the infinite number of UC instances living sequentially or simultaneously over the time, whereas each individual UC instance is finite and must terminate after a non-infinite amount of time.

- The generation of new UC instances by already existing UC instances, which might possibly result in an infinite system, may be of recursive nature and are depicted in the UC diagram (as description level) by a looping trigger-line from a UC symbol to itself.4

- For each actor role symbol (“stick-man”), we assume exactly one singleton instance at any time.

In the logic description of such an enhanced UC meta model, which is needed for consistency checks and property deduction on UC specifications, the basic properties will be described by terms whereas the UC relations will be expressed by predicates (see below for more explanations).

1 Future work could also be dedicated to hierarchical compositions of UC, which are also not precisely defined in the standard UML, that describe how an "outer" UC can be composed of "inner" UC that are not visible from the outer perspective. This feature, which could be graphically depicted by smaller UC "bubbles" being completely contained within a bigger surrounding UC "bubble", similar to inner classes in OOP, would be in support of hierarchic system modeling strategies which make use of abstraction and composition.

2 In other words: any sloppy speaking of "an infinite UC" (or something like that) means that terminating instances of such UC can be created again and again, and a "looping" UC would be one in which some instance u would give birth to a successor instance u+1 before terminating itself.

3 Springer
To be able to relate those categories to UC and actors, we must define a meta model that classifies the properties (attributes) of them, such as "begin," "end," or other intrinsic states of them. In the meta model we regard actors as special cases of UC for the sake of theoretical uniformity. Therefore we regard the various relationships between UC as extrinsic properties, not as their intrinsic states.

The main part of our work will be the establishment of meta relations as consistency axioms and rules about the primary UC relations. Our approach is "two-dimensional" in the sense that it establishes laws (or meta relations) for (at most) pairs of relations; in other words, conclusions are drawn from maximally two premises on top of the conclusion line. Formally this is the usual rule structure in "natural" deduction calculi, and materially (w.r.t. a UC diagram) this corresponds to a locality principle in which only small sub-graphs of a UC specification graph are under scrutiny at the same time. Thus, we will be dealing with set of consistency axioms and rules of the following forms (schemes):

\[
\begin{align*}
U & \times U \rightarrow U & \text{UC relations} \\
\rightarrow & \rightarrow & \text{consequences} \\
\end{align*}
\]

whereby \( \rightarrow \) are various possible relations in which two UC (or, more precisely, their respective instances) can be found, and the consequence is another description of UC relationships or properties that must hold for the sake of the consistency of the specification. This will make automated reasoning about an entire UC specification possible. Those abstract rule schemes can then be made concrete by instantiation, yielding rules which can be implemented in the PROLOG in a straightforward manner.

For example if we know that \( a \) is a sub UC of \( c \) (via the inheritance relation) and \( d \) has any property \( R \) (which could even be a relation to yet another UC \( a \) then the automated reasoning engine must be able to conclude that \( a \) is in possession of property \( R \), too, formalized as:

\[
\begin{align*}
& a \rightarrow c \\
& c \rightarrow d \\
& d \rightarrow a \\
\end{align*}
\]

Another example of a consistency rule of this scheme: if an event \( e \) is happening before another event \( e' \), and \( e' \) happens before \( e' \), then \( e \) also happens before \( e' \).

Finally we had to make a decision about how to represent the "ontology" of our model; though we could have chosen some notation developed in the field of Description Logics (DL) which have attracted considerable attention in the "ontology" community, we have chosen to express our model directly in the well known executable logic specification language PROLOG, not at least because of nowadays available PROLOG/JAVA interfaces, which should allow for a comparatively easy integration of the executable PROLOG UC model into a mostly JAVA-implemented prototype for the demonstration of the feasibility of our idea.

1.2 Use cases and their instances

Commercial literature on UC modeling, for example [7], does not clearly distinguish between a UC description (represented by an oval "bubble" in a UML UC graph) and its actual runtime instances which are, in fact, processes. This difference is in analogy to the relationship between classes and objects in OOP. In a "live" software system, more than once process instances could possibly exist to any one UC "bubble," either simultaneously at the same time, or sequentially at different points of time, or even in a combination of both. Consequently, if \( a \) and \( a' \) is a relationship defined in a UC specification, we will have to reason especially about the consequences of \( a \) as far as the process instances of \( a \) and \( a' \) are concerned. For this reason our model will also make use of the usual existential quantifiers (on UC instances), as it is further described in the remainder of this paper.

2 Use case meta model

As the detection and formalisation of valid consistency rules on UC specifications, as mentioned above, is still ongoing work in its early stages, the main contribution of this proposal paper is the definition of a meta model which shows (and structures) the "pool" of all possible relations between (or properties of) UC. These relations can be used to translate the materially empty rule schema (introduced in the previous section), in order to obtain the concrete applicable consistency rules on which the operations of the planned UC reasoning engine are based.

To reason formally about a UC model, a number of attributes must be introduced to a UC in the fashion of an "ontology." Whilst some of those attributes will be static in nature (for example, some UC could be a mandatory or optional sub-UC to some other UC, which is always the case), other UC properties are of dynamic nature in the sense that their value can change during the lifetime of a UC instance. For example, temporal reasoning about one UC instance happening before or during after another UC instance does only make sense if there exists a changeable state attribute which reflects the "lifecycle" of the time frame of the UC instance. In the following, we present our meta model at two conceptual levels:

- A class diagram shows what types of entities (e.g. normal UC or actors) we have and which kinds of attributes they may have.
- Whereas some attributes represent unary relations (e.g. internal values such as instance birth time) other attributes represent binary relations between two entities.
- Whereas some attributes in the meta model represent standard relations between UC according to the UML (which we also called the diagrammatic ones), other relational attributes represent the new contributions of our meta model; these are also called the modal ones.
- Then, on a conceptually finer level, possible values of the dynamic (time related) attributes are defined in terms of a simple finite state machine.
- The different states of a UC evolving over time allow for finer temporal modelling; e.g. we cannot only say: "this UC happens before that UC" but we could also say more precisely, for example: "this UC terminates after that UC started," or something like this.

As mentioned above, all UC instances are regarded as "mortal" and finite in time, and the possibly infinity of a subject system would result from the unlimited creation of such finite instances—"unlimited" either in the number of instances simultaneusly existing at any particular point of time, or unlimited as far as the life span of the entire system is concerned (possibly only a finite number of instances existing at any point in time); this is probably the practically relevant case.

Anyway, this reduction of infinity to finite instance components allows us to model any UC instance as a simple state machine as shown below. Thereby, the internal states of such a UC machine are the above-mentioned basic (unary) properties that underly the modal reasoning about the various relationships amongst each other. Moreover, our notion of a "successfully terminated" UC instance—in contrast to an "aborted" one—may include the generation of data which might be used as input by other UC instances. According to the learned practice of software engineering, our model deliberately abstracts away from such detail at this early stage of requirements engineering.

As far as the actors are concerned, we follow the usual UML convention according to which the actors represent the external world, from the system's perspective. Therefore we do not assume anything about actors except their unlimited existence and ability to act; consequently we do not attribute any internal states to them.

UC counting would be even n-ary.

Fig. 2 Meta model: use cases, actors, relations, and states

2.1 Top layer of the meta model

Figure 2 shows the top layer of our UC meta model. The central concept is, indeed, the relation, and not the UC itself. The picture of Fig. 2 is explained as follows:

- "Relations" have entities that they "bind" together, as well as—possibly—some quantification (existential or universal) as far as the instance processes to the participating entities are concerned. Sub-classes of relations will be shown and explained later in this paper; fit for the possible sub-classes of instance-quantification where applicable.
- "Entities" have time attributes representing their "birth" and "death" of UC instances. These time attributes allow for reasoning about what is called "before" or "after" relationships, etc.
- "Actors" are entities which are represented by the well known "stick-man" in the pictorial UC diagrams. Belonging to the external world outside the system bound, we cannot attribute any system properties to them. They are assumed to be always available, thus for any actor instance we assume actor.birth = oo, and actor.death = oo.
- "Use cases" are the system-internal entities which are represented by the well-known "bubbles" in a UC specification. The lifetimes of their process instances are limited, thus we have 0 < (p.death - p.birth) < oo for every UC instance p. Moreover, every UC instance can go through a sequence of states during its lifetime (as far as explained below); therefore a process state class is associated with the UC class in our meta model.6

2.2 Inner structure of use cases

A UC is more than a "bubble" in a UC diagram; it has an inner structure which, according to the industrial literature [7], comprises the following attributes: name, model

---

6 Not depicted in the UC concept of Fig. 2 are the other internal UC and actors that are usually found in a UC specification, such as pre-condition, post-condition, etc. [7]—some of them are modeled explicit.

---
Successful termination will eventually occur, unless the process is halted either by internal failure or by external abortion (triggered by an actor from outside the system boundaries). As explained above, no UC instance can thus "live" forever, though the subject system as a whole could well "live" forever by giving birth to process instances in arbitrary numbers. Of course, transition a from E to R in Fig. 3 could also be induced by another UC via an <<inclusion>> or <<extend>> relationship—this should be obvious to any reader with some experience in UC modelling and does not need any further mentioning.

2.3 Classical instance-quantified relation types

In our meta model, all relationships between UC "bubbles" in a UC specification are binary and directed. As a UC "bubble" is only a representation of its extension (set of instances) the question arises how a relation \( r \) \( u ' \) between UC \( u \) and \( u ' \) should be interpreted in their extensions \( e(u) \) and \( e(u') \).

This needs to be further specified by the designers of the subject system.

For this purpose, every relation \( R \) in the meta model is associated with two quantifiers: one for the domain side of the relation, and one for the range side of the relation; thus the following is true:

\[
\forall Q, Q' \in \mathbb{D} = \{V, I, B, \mathbb{D}\}
\]

\[
\exists Q, Q' \in \mathbb{D} = \{V, I, B, \mathbb{D}\}
\]

where \( Q, Q' \in \mathbb{D} \) are the set of the four classical syllogistic quantifiers "one" (\( I \)), "some" (\( B \)), "none" (\( V \)), and "all" (\( V \)).

Example: Given two UC \( u \) and \( u' \), their extensions \( e(u) \) and \( e(u') \) and a UC relation \( R \) relating \( u \) and \( u' \), then the process instance relation

\[
e(u) \circ R \circ e(u')
\]

would be interpreted as:

\[
\exists p \in e(u) \land (\forall p' \in e(u') : p R p')
\]

whereby \( p \) and \( p' \) are runtime instances (processes) of UC within a "living" subject system. The reader can easily imagine that many useful relation types can be stipulated in this way, including injection, bijection, surjection, the complete relation "all-to-all," etc.

In this context we conjecture that UC specifications can be made more precise and testable by quantifying UC relationships in the form of above. Figure 4 depicts the corresponding part of the meta model: note the self-association of the superclass which denotes the binary pairing of those four classical extension quantifiers.

7 This would include the production of data, which is, however, not explicitly modeled by our high-level model. Moreover, we would assume the fulfillment of any post-conditions, which our theory does not model explicidy either, only in this successful termination state.
For example, if an actor a may or may not (optionally) trigger some UC u via a trigger relation T—thus: a \( \triangleright u \rightarrow T \) and then T could get in some form existentially quantified (rather than all-quantified) in order to express this possibility of the action.

2.8 Negation

Except of the B quantifier in \( B \) negation is generally expressed implicitly, namely by omission, in our model. For example, if there is no trigger relation between a particular actor a and a particular UC u, then we may conclude that a must not trigger a under any circumstances. For the PROLOG-based reasoning “behind” such a UC specification we would thus work with PROLOG’s well known negation-by-failure semantics.

3 Ongoing work in this project

After having outlined an elaborate meta model of the UC language in the previous section, we must now state what we want to do with it in the next phases of our project.

3.1 Meta relations

Meta relations are consistency axioms and consistency rules about the relations which are defined in the meta model of the previous section. With all those many relations available the combinatorial possibility for such consistency rules are large, and it will take time and effort to discover the relevant and useful ones before they can be formalised and implemented. The difficulty of this rule finding exercise stems from two sources:

- We want to reason about testable UC instances (i.e. processes at runtime) rather than the abstract UC “bubbles” which only represent those instances at the highest possible level of abstraction.
- The relationships about the UC instances (processes) are quantified (universally or existentially) on either side of the binary relationship, which multiplies the number of potential rule candidates to be examined for validity.

Once a valid consistency rule has been found, its implementation in PROLOG (or any other deduction language for that matter, such as OPS5) should be a rather straightforward exercise: Once the rules are implemented, it shall be possible to:

- Detect logical flaws within a UC specification before any software development takes place and
- To query the PROLOG model with regard to properties of UC instances which must hold after the software development has taken place. Then we could ask questions such as: “is it true that all instances of UC u must terminate, as the last instance of UC u’ can be born?”. In other words, we shall be able to generate test cases directly from a UC specification.

3.2 Graphical user interface of a prototype

To make the UC specification system more user-friendly for the industrial practitioner, its logic engine should be “hidden” behind a graphical user interface. The idea is that the user should be able to “attach” the additional specifications (as defined by our meta model) to a graphical UC specification which consists mainly of the typical “stick-men” and “bubble” diagrams. Such an enhanced UC specification must then be translated into textual form (in an XML-like formalism similar to [2, 11]) such that it becomes amenable to (text-based) automated reasoning.

From the XML representation of a logically enriched UC specification, we could then derive the facts on which the PROLOG engine can start its work. The technical (not so much scientific) challenges in this scenario are thus:

- To extract PROLOG facts from a mainly graphical, logically enhanced UC specification.
- To couple a Java-based UC-Editor with an underlying PROLOG interpreter, and
- To propagate graphical (“visual”) information back into the graphical UC specification editor after the PROLOG reasoning process has discovered any inconsistency in a UC specification; for example to highlight a logically impossible specification element in red colour, or something like this.

3.3 Nested use cases

Our meta model does not contain an n-ary nesting relation on UC which would allow for drawing smaller UC “bubbles” within a larger “bubble” of a higher-order UC. Higher-order (or nested) UC are explicitly discouraged by [7], but nevertheless we think that they might be useful for the purpose of top-down system modeling at different levels of abstraction. Future work would have to expand the meta model as well as the set of consistency rules into this direction; thereby the logic rules for nested UC would probably have the character of refinement rules.

3.4 Related work and literature studies

Though we have reason to believe that our approach is quite original, we are aware that we are not operating in an un-explored field. In our yet ongoing literature studies we have found a number of interesting papers which are pointing into the direction of which our project is going. For example, the application of modal and deontic logic in computer science and software engineering is studied by [3, 5, 9, 13]. Another ontology (meta model) approach to UC reasoning can be found in [4]. Approaches to giving process semantics to UC specifications can be found in [1, 6, 8].

4 Summary

This position paper (category: work in progress) outlined a future project towards making UC specifications—previously considered harmful [14]—more useful and less ambiguous. This shall be achieved by a rich arsenal of UC relations, which exceeds by far the small set of UC relation types defined by the UML. A more or less fully elaborated meta model of such relations has been provided in this paper as its main contribution. As soon as the consistency rules (meta relations) on these rules are discovered and formalised, logic reasoning about UC specification will be possible. The objective of such reasoning is twofold, namely to detect logical flaws within a UC specification itself (before system implementation), and to query a UC specification for test cases which must hold empirically (after system implementation).

Acknowledgments

I would like to thank my student helpers Erez Jivan and Pierre-Nantes Kaeze for their efforts with the implementation of the prototype, as well as for their help with the literature search. Erez Jivan is also helping with the discovery of valid consistency rules over the domains of UC relations as outlined in the meta model. Formal Methods Europe (FME) and the University of Pavia are supporting this project financially. The P-UCML (precise UML) community has provided valuable hints about literature and related work. Last but not least thanks to the anonymous reviewers of the UML+PM’08 Workshop for their critique; it has been taken into account for this published version of our paper.

References

3. den Haan N (1995) Investigations into the application of deontic logic. JNCS 897
   In: Proceedings of 13th Australian DB conference, pp 157–166