On Use Cases and Their Relationships in the Unified Modelling Language

Perdita Stevens[*]

Division of Informatics
University of Edinburgh

Abstract. In the Unified Modelling Language, use cases are provided as a way of summarising the requirements on a system. They are defined informally as specifications of sets of sequences of actions, and several relationships between use cases are informally defined. Dependable, tool-supported software development necessitates precise definitions of all these concepts but the topic has so far received little attention in the literature, beyond what is present in Catalysis. This paper explores how these notions can be formalised whilst staying as close as possible to the UML standard, makes some suggestions and raises further questions.

1 Introduction

The Unified Modelling Language has been widely adopted as a standard language for modelling the design of (software) systems. Nevertheless, certain aspects of UML are not yet defined precisely. This paper is concerned with one such aspect: use cases and their relationships. The relationship between a use case and its constituent actions, and especially the implications of this for the relationships between use cases, are a particularly frequent source of confusion; we have not reached the point where a tool could support the use of use cases without making major decisions concerning how to interpret the UML standard.

In this paper we discuss how use cases and their relationships may be formalised. We aim to use the minimum of formal machinery possible; so, for example, we model use cases as plain labelled transition systems, rather than as processes in some particular process algebra. [1] A major aim of this work is to be faithful to the UML standard wherever possible: UML is a hard-fought compromise and it seems sensible to try to formalise what is currently intended rather than, or at least before, suggesting improvements to that intention. There are, however, several points at which our formalisation attempt forces us to conclude that the UML standard needs amendment.

[*] Email: Perdita.Stevens@dcs.ed.ac.uk. Fax: +44 131 667 7209

[1] Given that process algebras are typically explained and given semantics using labelled transition systems, working directly with the LTSs is indeed "more basic", requiring strictly less machinery: it gives us a well-defined notion of what it would mean for an alternative translation into a process algebra to be consistent with our decisions, without requiring us to enter into the process algebra wars.

The paper is structured as follows. The remainder of this section includes a note on standards and terminology, and a brief discussion of related work. In Sect. 2, we discuss the way UML sees use cases as made up of constituent actions. We model use cases and raise questions concerning their relationship with the system that provides them. In Sect. 3, we go on to discuss the "dependency" relationships between use cases, covering ≪include≫ in detail. In Sect. 4 we consider the more controversial relationship, generalisation between use cases; here we fail to find an adequately simple formalisation of UML's intentions, and indeed expose some problems with even the informal description. Finally in Sect. 5 we conclude and discuss future work.

1.1 Note on standards and terminology

This paper is based on UML1.3, the current standard at the time of writing. It is hoped that this work may feed into the ongoing construction of UML2.0, a major aim of which is to increase the precision of UML. The reader is assumed to be familiar with UML and should, in particular, be aware that UML's treatment of use cases changed significantly between versions 1.1 and 1.3, in response to criticisms of UML1.1's treatment: many UML books, however, still refer to UML1.1. We will refer to [10] as evidence of the intentions of some of the authors of UML, but it is important to note that the definition of UML is the OMG standard [7], not what is contained in any UML book. We cite material from [7] by page; thus, [7] (3-90) indicates page 90 of Sect. 3 of the UML standard, that is, of the Notation Guide.

Catalysis is described principally in [5]. Readers need to be aware that the notation used there, though UML-like, is not standard UML; and also that some technical terms have different meanings in UML and in Catalysis. Unless otherwise stated, this paper uses the UML notions.

1.2 Related work

Perhaps surprisingly, given the frequency of mailing list questions and informal discussions of use cases and their relationships, there seems to be no previous work that formalises use cases in a way comparable to the present. In practice, many use case experts such as Alistair Cockburn (see e.g. [4]) advocate an informal approach to use cases, in which use cases are simply considered as tasks. This is arguably the right approach for a practitioner to take; but even so, see Sect. 5 for Cockburn's comment on deficiencies in current understanding of generalisation of use cases. The closest comparator to the present work is Gunnar Övergaard's recent PhD thesis [9], which reached the author too late to be considered in detail here. At first sight, a major difference is that Övergaard's formalisation does not seem to consider fully the implications of the dynamic choices that are made by the parties to a use case. The thesis supercedes an earlier paper [8], which formalised some aspects of use cases in UML1.1 and was influential on the development of UML1.3, considered here. The authors of the development method Catalysis [5], have also been more concerned with preciseness than many of those involved in UML: we will naturally turn to Catalysis for ideas.

There has been more work that formalises UML activity diagrams. We mention particularly Bolton and Davies' work defining activity diagrams as CSP processes [3]. In the present work we are interested, as stated, in formalising use cases using the minimal machinery possible, so we refrain from introducing any particular process algebra for describing use cases; however, it would be interesting in future to combine the two approaches, especially with the aim of formalising the interactions between different use cases in an overall system workflow. Note that in the present work we do not consider interactions or interference between use case instances, either of the same or different use cases, at all.

Theoretical work which is relevant to particular sections of this work is discussed in those sections.

2 What is a use case and what is it made of?

A use case diagram gives a high-level view of the requirements of a system: it shows the *actors* of a system – that is, the roles played by humans or external systems that interact with it – and the tasks, or use cases, which the instances of the system and actors collaborate to perform. Not all actors need be involved in all use cases; an association between an actor and a use case shows the potential for communication between instances of the use case and the actor. Use cases may in fact be used to describe the requirements on subsystems, components etc. (that is, on any Classifier), so we follow the UML standard and use the term "entity" to describe the system, subsystem, component etc. whose requirements are documented by a use case.

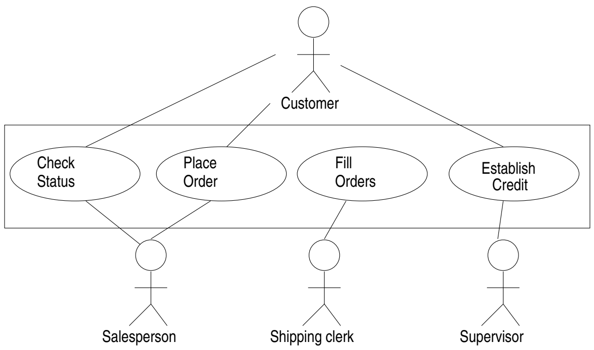

Fig. 1. Basic use case diagram, adapted from [7] (3-90)

The most crucial passages on use cases in the standard are the following:

> The use case construct is used to define the behavior of a system or other semantic entity without revealing the entity's internal structure. Each use

case specifies a sequence of actions, including variants, that the entity can perform, interacting with actors of the entity. In the metamodel UseCase is a subclass of Classifier, specifying the sequences of actions performed by an instance of the UseCase. The actions include changes of the state and communications with the environment of the UseCase.

[7] (2-120)

Each use case specifies a service the entity provides to its users, i.e. a specific way of using the entity. The service, which is initiated by a user, is a complete sequence. This implies that after its performance the entity will in general be in a state in which the sequence can be initiated again. A use case describes the interactions between the users and the entity as well as the responses performed by the entity, as these responses are perceived from the outside of the entity. A use case also includes possible variants of this sequence, e.g. alternative sequences, exceptional behavior, error handling etc.

[7] (2-124)

Like most design notations, use cases are used in two different ways: (a) by someone wanting to *design* the entity, to describe what a correct design must do; or (b) by someone wanting to *use* the entity as a black box – for example, in the design of another entity – to describe the services offered and their correct use. The main difference between the needs of these users will be the nature of the actions involved: the designer will probably begin with abstract, conceptual actions which may never have an exact counterpart in any implementation (even the direction of a communication may not have been decided when the first use cases are described [5]) whereas the user will require a description in which the actions do correspond directly to communications with the entity. The distinction will not be important for our purposes here, however, where action can be taken in either sense.

It is also said ([7] 2-120,2-125) that use cases may be described in a variety of ways – for example, text, state charts, pre- and post-conditions – which may shed some light on what a use case is intended to be, as we shall discuss. None of these descriptions is privileged as a definition, however.

The first thing to note is that a use case has a dual existence, like other Classifiers (classes, components, etc). On one hand, it may often usefully be identified with the set of its instances. That is, just as it is sometimes useful to identify a class with the set of objects in that class, it is sometimes useful to identify a use case with the set of instances of the use case. An instance of UseCase is one particular (maximal) sequence of actions from the many which comprise the use case. More precisely a UseCaseInstance is

the performance of a sequence of actions specified in a use case. [...] An explicitly described UseCaseInstance is called a scenario.

(2-120) One may think of a UseCaseInstance as a single-branch labelled transition system, and of a scenario as the sequence of labels on the branch; the difference will not be important for our purposes.

On the other hand, just as classes may often be more usefully regarded as "things" in their own right – collections of attribute and operation specifications and implementations – use cases too may be studied as Classifiers, neglecting their instances.

We will consider the two aspects in more detail and propose a synthesis.

2.1 Use cases as classifiers

In the UML metamodel, UseCase is a specialisation of Classifier. This implies, among other things, that use cases can have structural and behavioural features, such as attributes and operations.

Structural features The attributes of a use case are the means by which it is recorded how the state of a use case changes as a sequence of actions is carried out. In practice, it is usual to regard the entity and the actors as having certain conceptual attributes – conceptual in that there is no intention that a given attribute has a direct counterpart in any implementation – and then the state space of one of the use cases of the entity should induce an equivalence relation on the set of (joint) conceptual states of the entity and the actor instances. Notice, here, the subtlety introduced by the lack of any constraint in UML on the multiplicity of associations between actors and use cases: the number of instances of a given actor (potentially) communicating with a given instance of a use case might even be unbounded! In practice it is hard to imagine an example where more than one *instance of* a given actor communicates with a given *instance of* a use case (there are none in [7] or [10] for example). If there are two distinct such actor instances then almost by definition they play different roles in the use case and should therefore be modelled as instances of different actors. We suggest that this should be enforced in UML by constraining the actor end of any association between a use case and an actor to have multiplicity either "0,1" or "1", and will assume this constraint.

Behavioural features In the UML metamodel an operation is an example of a behavioural feature. We omit to explain the details here: suffice it to say that the behavioural features of a use case will specify those actions that the entity can send and receive which are relevant to the use case. We will work simply with unanalysed actions; as with states, different choices of high-level syntax are possible to reflect the structure of actions as, for example, operation invocations with particular parameter values. Notice that there is in particular no general expectation that an action should have a reply, though a high level use case specification language that translated down into our basic formalisation might impose such restrictions when appropriate.

2.2 Use cases as specifications of sequences of actions

A use case "specifies a sequence of actions with variants". Taken at face value, this suggests that what is needed is a definitive way to decide, given a use case and a sequence of actions, whether the sequence of actions matches the specification given by the use case.

However, we must take more care over “with variants”. In practice, use cases are normally described in English or pseudocode and include statements like “if [something happens within a calculation] the system ...” and “the user enters a datum and the system acts accordingly: ...”. That is, the points at which the variations may happen (the branch points) are specified.

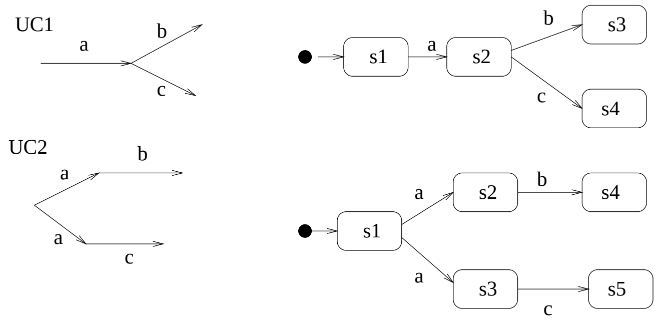

Fig. 2. Location of branch points

To illustrate the implications of this, let us represent the choices as branch points in a labelled transition system, or equivalently as states with more than one output transition in a statechart. (We illustrate both this one time to show that it is possible to use existing UML notions: however, we do not recommend the use of UML statecharts for this purpose, because their extra power relative to LTSs is not required and because their correspondingly complex semantics makes misunderstandings more likely.) The top line of Fig. 2 specifies the same set of sequences of actions – $\{ab, ac\}$ – as the bottom line. However, we argue that they should not be regarded as interchangeable. Suppose that each of $UC1$ and $UC2$ are representations of a use case offered by a system, S, and that b and c represent “input” actions chosen by (an instance of) actor A (for the sake of argument, and to make it clear that it is not reasonable to expect A to “react intelligently”, let A be an external system with which S must communicate). Then if what S provides is $UC1$, A does not have to commit to entering b or c until after the action a has happened; the choice between b and c may be made, for example, after some computation within the actor that involves third parties outside the scope of a model of S and does not begin until after a has occurred (and this might be essential: a could be an “output” action which carries information that A uses to choose the third party). However, an implementation of A that behaves this way may fail if it tries to carry out $UC2$: by the time the action a has happened, a commitment may have been made to carry out b, and if A decides that c is what is intended there will be deadlock.

In many cases, of course, such considerations will not apply: it will often be enough to know the set of maximal traces of the use case, without specifying the branching structure. This makes it understandable that the UML standard is written in terms of sets of sequences of actions. However, when describing a use case in text, one has to go out of one's way to avoid specifying where branch points are; it is standard practice that use case descriptions do include this information, because textual descriptions which do so are more natural. Moreover, a use case described using a state chart necessarily has this information, and discarding it would require deliberate identification of structurally different state charts. Therefore it seems likely that our next proposal for a modification to the UML standard may be acceptable: we suggest that "a use case specifies a *process, which determines* a set of sequences of actions".

The set of sequences of actions so determined is the set of maximal traces of the process. We are using the term "process" independently of any particular process algebra; we could use "labelled transition system", "automaton" (but *not* "finite state automaton") or "(restricted) state chart" instead.

2.3 Formalisation and discussion

Definition 1. *A use case $u = (S, L, \rightarrow, S_0)$ consists of:*

- *a (possibly infinite) set S of states*
- *a (possibly infinite) set L of labels*
- *a transition relation $\rightarrow \subset S \times L \times S$; as usual we write $s \xrightarrow{l} t$ rather than $(s, l, t) \in \rightarrow$*
- *a (possibly infinite) set of initial states $S_0 \subseteq S$*

We will use the usual variants on the notation, writing for example $s \rightarrow t$ when there is some l such that $s \xrightarrow{l} t$, and $s \not\rightarrow$, or equivalently $s \in \mathit{final}(u)$, when there are no l, t such that $s \xrightarrow{l} t$. We write $l_1 \dots l_n \in \mathit{sequences}(u)$ iff there exist $s_0, \dots s_n$ such that $s_0 \in S_0$ and $s_0 \xrightarrow{l_1} s_1 \dots \xrightarrow{l_n} s_n \not\rightarrow$.

For technical reasons we will insist that $S_0 \cap \mathit{final}(u) = \emptyset$, that is, that every start state has some transition.

Specifying a use case We deliberately do not concern ourselves with how the components of a use case are given, because there are many choices that will be appropriate in different circumstances. The choice, more than our underlying LTS structure, will determine what computation it is possible for a tool to carry out concerning a use case and with what complexity. For example, if it is possible to abstract away from data to the extent that the sets S and L are finite, the whole range of model-checking techniques becomes available. More usually, these sets will be infinite (or extremely large, depending for example on whether one models with real integers or machine integers) but structured according to the structural and behavioural features of the use case, and ultimately of the entity and the actors.

Nevertheless, any way of giving the components should be expected to satisfy a sanity condition that the state of the use case "contains no more information than" the states of the participants in the use case. Precisely:

Definition 2. *Let $u = (S, L, \rightarrow, S_0)$ be a use case for an entity with statespace E in which the entity communicates with actors with statespaces $A_1 \dots A_n$.*[2] *A* quotient map *of u is a surjective function*

$$h : E \times A_1 \times \dots A_n \rightarrow S.$$

A specification formalism should include such a quotient map for each use case. Most simply, if the statespace of the use case is constrained to be written in terms of a subset of the attributes of the entity and actors involved, the obvious induced quotient map might be assumed.

Why maximal traces? We have chosen to restrict attention to finite sequences of actions, where no further action is possible, because there is an assumption running through the UML standard that the sequences of actions that comprise a use case are finite. This does not imply that the process needs to be a tree, or even acyclic. We could restrict attention to acyclic processes, which would obviously facilitate reasoning about total correctness. However to impose such a restriction in general would seem unreasonable: there is no obvious objection to reaching the same state by two different routes, or even several times. Considering only maximal traces, in this way, does limit the usefulness of use cases for describing continuing behaviour; but this is already a recognised weakness of use cases. This formalisation might provide a basis for remedying the weakness, but this is beyond our scope here.

Specifying branching structure As we have argued, branching is a normal feature of use case descriptions. It is normal, however, for the conditions under which the branches are taken to be underspecified at this stage; the precise conditions may not be known, or may depend on data of the system and/or the actors which the specifier does not choose to model. There is a style of specification (and Catalysis' recommendations, for example, are close to it) in which the specifier would add "conceptual" attributes to system and actors and write conditions in terms of these; but it is far from universal. For simplicity, therefore, we will model choices as simple non-determinism in an LTS.

2.4 Use cases as (pre- and) post-conditions

Pre- and post-conditions (hereinafter PPCs) are suggested in the UML standard as one way of describing a use case, and in the Catalysis method they are the

[2] Notice the use of our restriction that there is at most one instance of a given actor communicating with a given use case: this enables us to confuse actor instances with actors with impunity, and we shall continue to do so

main such method. Evidently they are not sufficient for recording intended contracts between the system and its actors which are more complex than a simple relation between the states before and after an instance of the use case occurs – they provide no way to talk about the nature of the interaction between the system and its actors. Thus two implementations of a use case could in general satisfy the same PPC, but not be substitutable for one another from the point of view of an actor, if, for example, one implementation expected two inputs from the actor where the other expected three. (In Catalysis use cases are identified with (joint and/or abstract) actions; use cases are decomposed into sequences (sic) of smaller use cases/actions. This is an interesting approach; however, in UML use cases and actions are quite distinct concepts, so we will not consider it further here.)

PPCs are, however, a very convenient way to specify use cases which are simple, and/or which should be specified at a very high level. How does this view interact with the others?

Following Catalysis, we consider PPCs written in terms of conceptual attributes of the system *and the actors*: recall that this is the same information which determines our proposed states of use cases via quotient maps.

Definition 3. *Let $u = (S, L, \rightarrow, S_0)$ be a use case for an entity E with actors $A_1 \dots A_n$, and let $h : E \times A_1 \times \dots A_n \rightarrow S$ be its quotient map.*[3]

Let a PPC be given as a relation $PP \subseteq (E \times A_1 \times \dots A_n) \times (E \times A_1 \times \dots A_n)$, where if p fails the pre-condition $(p, q) \notin PP$ for any q; we write p fails pre(PP). Then

1. *PP and u are* compatible *if PP respects h, in the sense that whenever $h(p) = h(p')$ and $h(q) = h(q')$ we have $(p, q) \in PP \Leftrightarrow (p', q') \in PP$.*
2. *u* satisfies *PP iff both*
   (a) *PP and u are compatible, and*
   (b) *whenever $s_0 \in S_0$ and $s_0 \xrightarrow{l_1} s_1 \dots \xrightarrow{l_n} s_n \not\rightarrow$ and $h(p) = s_0$ and $h(q) = s_n$, we have either $(p, q) \in PP$ or p fails pre(PP).*

Intuitively, to say that a use case and a PPC are compatible is to say that the PPC uses no more state than is recorded in the use case. If this is not the case, then the question of whether the use case satisfies the PPC is nonsensical.

Notice that in the simplest case, where the state of a use case is determined by giving values for a subset of the attributes of the entity and its actors, a PPC written in terms of that same subset of attributes will automatically be compatible with the use case, as expected.

We shall later need the following:

Definition 4. *Let $u = (S, L, \rightarrow, S_0)$ be a use case for an entity E with actors $A_1 \dots A_n$, and let $h : E \times A_1 \times \dots A_n \rightarrow S$ be its quotient map. The* induced pre- and post-condition *of u, written $PP(u)$, is*

$$\{(p, q) \in (E \times A_1 \times \dots A_n) \times (E \times A_1 \times \dots A_n) : \\ h(p) = s_0 \in S_0 \text{ and } \exists s_0 \rightarrow s_1 \dots \rightarrow s_n \not\rightarrow \text{ such that } h(q) = s_n\}$$

[3] by a slight abuse of notation we use E both for the entity and its state space, etc.

Lemma 1. *Any use case satisfies its induced pre- and post-condition.* □

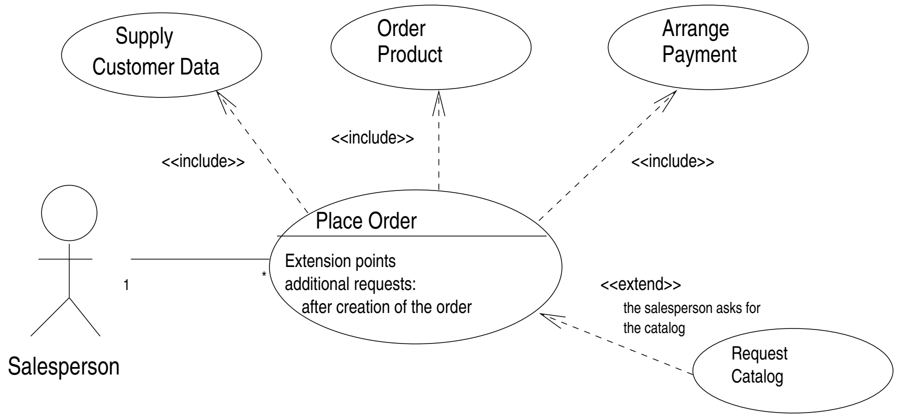

Fig. 3. ≪include≫ and ≪extend≫ dependencies between use cases: copied from [7] (3-93)

3 ≪include≫ dependency between use cases

The intention of the ≪include≫ relationship is to show subtasks as separate use cases; for example, in order to show that certain subtasks are shared by ≪include≫ing them in several use cases, or to demonstrate the use of a pre-existing component to carry out a subtask. The most relevant passage is:

> An include relationship between two use cases means that the behavior defined in the target use case is included at one location in the sequence of behavior performed by an instance of the base use case. When a use-case instance reaches the location where the behavior of an another use case is to be included, it performs all the behavior described by the included use case and then continues according to its original use case. This means that although there may be several paths through the included use case due to e.g. conditional statements, all of them must end in such a way that the use-case instance can continue according to the original use case. One use case may be included in several other use cases and one use case may include several other use cases. The included use case may not be dependent on the base use case. In that sense the included use case represents encapsulated behavior which may easily be reused in several use cases. Moreover, the base use case may only be dependent on the results of performing the included behavior and not on structure, like Attributes and Associations, of the included use case.

([7] 2-126: a subset of this information also occurs on 2-120)

We must take care over the interpretation of “the included use case may not be dependent on the base use case”. In [10] it is stated that the included use case has access to the attributes of its base, which apparently contradicts the lack of dependence; but indeed, it is not clear in what sense an included use case can *have* a result if it is not allowed to affect the state of its parent! We interpret the lack of dependence as meaning simply that the included use case makes sense, that is, is a valid use case, in its own right. Now as an included use case can define its own attributes as well as having access to those of its parent, we conclude that its quotient map should induce a possibly finer equivalence on the state of the entity and actors than its parent did.

Definition 5. *A use case $v = (S', L', \to', S'_0)$ with quotient map h_v is* suitable for inclusion in *$u = (S, L, \to, S_0)$ with quotient map h_u if*

1. *h_v is finer than h_u; that is, $h_v(p) = h_v(q) \Rightarrow h_u(p) = h_u(q)$;*
2. *$|h_v(h_u^{-1}(s)) \cap S'_0| \leq 1$ for all $s \in S$.*

We write h_{uv} for the unique surjective map $S' \to S$ such that $h_{uv} \circ h_v = h_u$

In terms of attributes, condition 2 amounts to saying that if the included use case defines a new attribute, then given values for the attributes already present in the base use case there must be a single (“default”) value for the new attribute. This avoids the need to invent data for the extra attribute(s). If there are no new attributes, that is, if $h_u = h_v$, the condition is vacuous.

We interpret the instruction that the base use case must depend *only* on the result of the included use case as meaning that the base use case must be describable in terms which allow the substitution of any use case with the same induced PPC as the intended one. We restrict the circumstances under which inclusion is allowed, for convenience and to reflect the intention that ≪include≫ (unlike ≪extend≫) is used for unconditional, once-only inclusion of behaviour. Thus we define:

Definition 6. *A use case $u = (S, L, \to, S_0)$ with quotient map h_u includes use case $v = (S', L', \to', S'_0)$ with quotient map h_v via Source $\xrightarrow{I}$ Target if*

1. *v is suitable for inclusion in u;*
2. *$I \in L \setminus L'$;*
3. *Source, Target, S_0 are pairwise disjoint subsets of S;*
4. *$h_u(p) \xrightarrow{I} h_u(q)$ is a transition in u exactly when all of: $(p, q) \in PP(v)$, $h_u(p) \in$ Source, $h_u(q) \in$ Target hold;*
5. *any element of sequences(u) includes at most one occurrence of I;*
6. *the only transitions out of Source or into Target are those labelled I.*

Such a use case depends only on the result of a suitable v, not on its details. To get a fully described use case including v itself we replace the source-target gap by the actual behaviour of v, identifying source states with the appropriate start states of v and target states with the appropriate final states; precisely:

Definition 7. *If u includes v via Source $\xrightarrow{I}$ Target then the composite use case* $include(u, v, I) = (S'', L'', \rightarrow'', S''_0)$ *where*

1. $S'' = S \setminus Source \setminus Target \dot{\cup} S'$
2. $L'' = L \setminus \{I\} \dot{\cup} L'$
3. *For s and t in S'', $s \rightarrow'' t$ iff one of the following holds:*
   - $s \rightarrow t$ *in u (note that by insisting s, t are in S'' we automatically discard transitions to Source or from Target)*
   - $s \rightarrow s' \in Source$ *in u and* $t \in h_v(h_u^{-1}(s')) \cap S'_0$ *(that is, t is a start state of v corresponding to a source in u)*
   - $t' \in Target$ *and $t' \rightarrow t$ in u and* $t' = h_{uv}(s)$ *(that is, s is a final state of v corresponding to a target in u)*
   - $s \rightarrow' t$ *in v*
4. $S''_0 = S_0$

The easy result that relates this process view to the trace inclusion view found in the standard is:

Lemma 2. *If* $l_1 \ldots l_n \in sequences(include(u, v, I))$*, then either*

- $l_1 \ldots l_n \in sequences(u)$*; or*
- *for some* $1 < m \leq p \leq n$ *we have* $l_1 \ldots l_m I l_p \ldots l_n \in sequences(u)$ *and* $l_{m+1} \ldots l_{p-1} \in sequences(v)$.

□

We make no formal restriction on when the first case can occur, but to stay within the spirit of the ≪include≫ relation this should indicate that something exceptional happened "before" the "inclusion point" of v. Because use cases are allowed to include exceptional or erroneous cases, in which the use case instance might be "aborted" before v becomes relevant, it does not seem sensible to take literally the view that every trace of the composite use case should include a trace of v. One could make this artificially the case by writing v to include a trivial "not applicable" trace to be included at the end of any trace of u on which v is never truly reached, but this seems pointless.

The ≪extend≫ relationship between use cases is similar; it is more complex but raises no substantially new issues. For reasons of space we omit a detailed treatment.

4 Generalisation of use cases

This is the most controversial of UML's relationships between use cases. (Cockburn writes, for example "In general, the problem with the *generalizes* relation is that the professional community has not yet reached an understanding of what it means to subtype and specialize behavior..." [4] p241.) The Notation Guide contains no example of generalisation of use cases, and says only the "A generalization from use case A to use case B indicates that A is a specialization of B" (3-92). The example pictured above is taken from [10] and is typical of the way the relationship is used: the idea is that the child use case should be a more fully described version of its parent. The Semantics Guide appears more helpful:

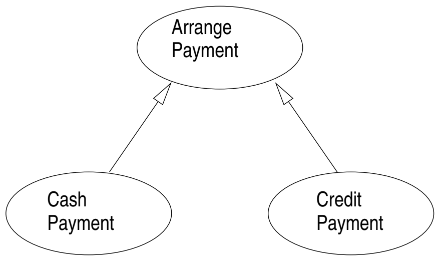

Fig. 4. Generalisation of use cases, from [10]

> Generalization between UseCases means that the child is a more specific form of the parent. The child inherits all Features and Associations of the parent, and may add new Features and Associations.

(2-120)

> A generalization relationship between use cases implies that the child use case contains all the attributes, sequences of behavior, and extension points defined in the parent use case, and participate [sic] in all relationships of the parent use case. The child use case may also define new behavior sequences, as well as add additional behavior into and specialize existing behavior of the inherited ones. One use case may have several parent use cases and one use case may be a parent to several other use cases.

(2-126)

Trying to combine this with the property which any UML Generalization is supposed to have, namely, that the child GeneralizableElement may be safely substituted for the parent, demonstrates that this description is at least incomplete. To see this, suppose A is an external system, behaving as an actor expecting to interact with an entity E according to a use case u. Presumably A includes code to react correctly to all actions which it may receive from E. Scenarios in the use case u contain a mixture of actions sent from A to E, those sent from E to A, and internal actions which may for example change the state of E. If we replace u by a specialising use case v which is allowed to contain arbitrary new scenarios, there is no guarantee that A will be able to react correctly, or at all, to actions which it may receive as part of the new scenario. That is, v will not automatically be substitutable for u. Let us examine what it should mean for v to be a specialised version of u.

Essentially, the wording of the current standard does not distinguish between two fundamentally different possible relations between u and v, each of which arguably reflects the view that more design decisions have been reflected in the description of v than of u:

1. v is more deterministic than u in what it says about the entity's actions

2. v replaces descriptions of high-level actions by lower-level more detailed descriptions.

Each of these points requires further analysis. (2) is essentially what we did with ≪include≫, though a more flexible treatment would be useful. The main thing to notice about (1) is that it only makes sense if we know which actions are chosen by the entity and which by an actor. So far, we have not needed to make such a distinction, and indeed we pointed out that it is sometimes desirable to describe use cases before such decisions have been taken. From the point of view of an actor for whom v should be substitutable for u, it would be quite reasonable for v to include fewer transitions chosen by the entity, or indeed by other actors, but it would be unreasonable to include fewer transitions chosen by A. Again, A may have been coded to assume that it is permitted to take a certain transition, and might break if v forbade this. From a theoretical point of view the most obviously appealing formalism to use to reflect this question of where choices are made would be the alternating transition systems of [2]; but as remarked before this conflicts with our desire to use minimal machinery – especially as we would need to do at least a little adaptation to deal with the presence of more than the two choice-makers considered there. Perhaps the reactive module work of [1], might help. For now we adapt an approach due to Jackson in [6], and say that *from the point of view of a given actor A*, the actions that make up a use case may be classified into inputs (actions initiated by A affecting E), outputs (actions initiated by E affecting A) and all others (including internal actions by E and communications between E and actors other than A: recall that UML does not permit direct communication between actors, so this list is exhaustive). To what extent does an actor care about more than its own input/output? It presumably does care about the PPC of the use case; after all, this is what specifies what the use case actually achieves. For simplicity in these early explorations, let us assume that u and v have identical label-sets and a bijection f from the start-states of v to the start-states of u.

From the point of view of an actor A who sees the actions of a use case u as FromMe $\dot\cup$ ToMe $\dot\cup$ Other, write $\stackrel{l}{\Rightarrow}$ for $\stackrel{l}{\rightarrow}$ preceded or followed by arbitrarily many Other transitions. Then a candidate definition is:

Definition 8. *$v = (S', L, \rightarrow', S'_0)$ specialises $u = (S, L, \rightarrow, S_0)$, iff the following all hold:*

- *v satisfies $PP(u)$*
- *there is some relation $\preceq \subseteq S' \times S$ such that $s_v \preceq s_u$ implies*
  - *whenever $i \in$ FromMe and $s_u \stackrel{i}{\Rightarrow} s'_u$ is a transition in u, then there is a transition $s_v \stackrel{i}{\Rightarrow}{}' s'_v$ in v such that $s'_v \preceq s'_u$;*
  - *whenever $o \in$ ToMe and $s_v \stackrel{o}{\Rightarrow}{}' s'_v$ is a transition in v, then there is a transition $s_u \stackrel{o}{\Rightarrow} s'_u$ in u such that $s'_v \preceq s'_u$;*
  - *$s'_0 \preceq f(s'_0)$ for all $s'_0 \in S'_0$.*

A point that follows on from this is that if u involves several actors, which use cases v are specialisations of u depends on from which actor's point of view we see the relationship. What is the relation that results from insisting that v be a specialisation from the point of view of all (beneficiary) actors? What difference does it make if actors are assumed capable of cooperation? Can these relations be put in a testing framework? Given appropriate high-level syntax, can they be computed? We have abandoned the total correctness aspects of [6] (because the assumptions seem too strong for our context); can we recover something useful when we want it? For these or some similar definitions, do ≪include≫ and generalisation interact cleanly, for example, is there some sense in which you can be permitted to include a more specialised version of what you expected, and is the resulting composite use case a more specialised version of what you would have had with the more general included use case? This may be too much to hope.

5 Conclusions and further work

It is possible, and often pragmatically advisable, to use UML use cases in a completely informal way, to describe the tasks to be carried out with the help of the system. Sometimes, however, we would like to have the option of relating use cases to the design of the system that implements them – or to that of another system that uses them – in a soundly-based, tool-supportable way. Moreover, regardless of whether users of UML choose to describe their use cases in full detail, it seems reasonably to expect that the UML standard's informal explanations of what would be the case if one did, would be formalisable. In this paper we have begun an attempt to provide, using the minimum of formal machinery, such a formalisation. It is surprising (at least to the author), given the modest scope of the present work, how much already has to be done. Directions of future work might include, besides the answering of questions left open here:

High level formalisms for describing use cases We have deliberately avoided choosing a high-level formalism for defining use cases, preferring to work at the basic level of labelled transition systems or processes. These can for example be defined using any process algebra or any kind of automata that has a labelled transition system semantics; alternatively, they could be defined using (a suitable formalisation of) statecharts. We suggest that it is probably impractical to standardise on one such language within UML, at least at this point; were a formalisation of statecharts to be agreed in UML2.0, this would be an obvious candidate, but others might be more appropriate in different circumstances. It seems more appropriate to use the minimal notion of labelled transition system in the UML standard.

Interference We should consider interactions and interference between use case instances, both of the same use case and of different use cases. Pace [7]'s statement "This implies that after its performance the entity will in general be in a state in which the sequence can be initiated again.", ordering dependencies between use cases often exist and are sometimes modelled with activity diagrams; how can this be formalised? What is the relationship between the set of all use cases and the entity itself? [7] (2-124) says "The complete set of use cases specifies all different ways to use the entity, i.e. all behaviour of the entity is expressed by its use cases."; in what formal sense can this be true?

Acknowledgements

I am grateful to the British Engineering and Physical Sciences Research Council for funding in the form of an Advanced Research Fellowship, and also to the referees for constructive comments.

References

[1] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15:7–48, 1999.

[2] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, , and Moshe Y. Vardi. Alternating refinement relations. In *Proceedings of the Ninth International Conference on Concurrency Theory (CONCUR 1998)*, number 1466 in LNCS, pages 163–178. Springer-Verlag, 1998.

[3] C. Bolton and J. Davies. Activity graphs and processes. In *Proceedings of IFM 2000*, 2000.

[4] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley, 2001.

[5] Desmond D'Souza and Alan Cameron Wills. *Catalysis: Objects, Frameworks and Components in UML*. Addison-Wesley, 1998.

[6] Paul B. Jackson. Total correctness refinement for sequential reactive systems. In *proceedings of TPHOLs 2000. (13th International Conference on Theorem Proving in Higher Order Logics)*, number 1869 in LNCS, pages 320–337. Springer-Verlag, August 2000.

[7] OMG. *Unified Modeling Language Specification version 1.3*, June 1999. OMG document 99-06-08 available from www.omg.org.

[8] G. Övergaard and K. Palmkvist. A Formal Approach to Use cases and their Relationships. In P.-A. Muller and J. Bézivin, editors, *Proceedings of ≪UML≫'98: Beyond the Notation*, pages 309–317. Ecole Supérieure des Sciences Appliquées pour l'Ingénieur – Mulhouse, Université de Haut-Alsace, France, June 1998.

[9] Gunnar Övergaard. *Formal Specification of Object-Oriented Modelling Concepts*. PhD thesis, Department of Teleinformatics, Royal Institute of Technology, Stockholm, Sweden, 2000.

[10] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1998.