Specifying Precise Use Cases

Jon Whittle

Dept of Information & Software Engineering
George Mason University
4400 University Drive
Fairfax, VA 22030
jwhittle@ise.gmu.edu

Abstract. Despite attempts to formalize the semantics of use cases, they remain an informal notation. The informality of use cases is both a blessing and a curse. Whilst it admits an easy learning curve and enables communication between software stakeholders, it is also a barrier to the application of automated methods for test case generation, validation or simulation. This paper presents a precise way of specifying use cases based on a three-level modeling paradigm strongly influenced by UML. The formal syntax and semantics of *use case charts* are given, along with an example that illustrates how they can be used in practice.

1 Introduction

Since their introduction, use cases have become a method of choice for elaborating software requirements. A use case—defined by Cockburn as a description of "the system's behavior under various conditions as the system responds to a request from one of the stakeholders" ([Coc00])—is typically represented as a combination of a UML use case diagram [BRJ05] and loosely structured text in one of many suggested template formats. The templates show the main sequence of steps that define a use case as well as some additional sequences that may capture exceptions, alternatives or extensions. The templates are usually related at a more abstract level using a UML use case diagram ([OMG]) in which use cases are given graphically by ellipses and the actors that trigger those use cases are shown using standardized icons. Use cases are almost exclusively defined in an informal way—use case diagrams have no commonly agreed semantics and the semantics of the text templates is deliberately left unspecified in UML because there are no restrictions on what kind of text can be given.

The informality of use cases makes them very easy to use but is a barrier to the application of automated analysis methods such as test case generation, simulation, validation etc. Usually, little attention is paid to how different use cases interact—whether, for example, they can execute sequentially or concurrently, whether there are inconsistencies, or whether they are complete.

Many attempts have been made to introduce rigor into use case descriptions, ranging from structural restrictions on the text that can be used in templates (e.g., [Smi04,Wil04]) to the development of a formal semantics for aspects of use case diagrams (e.g., [Ste01,OP99]). Approaches based on formalizing the text in templates usually define a restricted grammar for a subset of natural language and may also enforce that words in the text come from a dictionary. Approaches for defining a formal semantics for use cases focus on poorly specified constructs in UML use case diagrams, such as the UML ⟨⟨include⟩⟩ and ⟨⟨extend⟩⟩ relationships [Ste01] or the generalization of use cases [Iso04]. This paper takes a different approach. It gives an alternative, precisely defined, graphical language for use cases. It does not attempt to formalize UML's notion of a use case.

UML2.0 ([OMG]) introduces interaction overview diagrams, a notation based on activity diagrams, for specifying relationships between interaction diagrams (e.g., sequence diagrams). Interaction overview diagrams (IODs) can be used to more precisely describe use cases as a set of interaction diagrams connected by activity diagram relationships, e.g., concurrency. IODs are based on high-level message sequence charts (hMSCs) [IT96], a well-established notation for specifying interactions originally developed for the telecommunications domain. Whilst IODs provide much needed expressiveness for relating interaction scenarios, their semantics is still somewhat unclear since neither activity nor interaction diagrams have a formal semantics. In addition, IODs model only a single use case at a time and do not specify relationships *between* use cases. Nevertheless, IODs are an important step in precise use case modeling and form the basis for the *use case charts* presented in this paper.

In this paper, *use case charts*, a 3-level notation based on extended UML activity diagrams, is proposed as a way of specifying use cases in detail. The main application of use case charts to date has been to simulate use cases but use case charts are also precise enough for test generation and automated validation.

The idea behind use case charts is illustrated in Figure 1. For the purposes of this paper, a use case is considered to be a set of scenarios, where a scenario is an expected or actual execution trace of a system. The functionality of a system can be given as a set of use cases—that is, a set of sets of scenarios.

A use case chart specifies the scenarios for a system's use cases as a 3-level description: level-1 is the *use case chart*, an extended UML activity diagram in which the nodes are use cases; level-2 is a set of *scenario charts*, or extended activity diagrams where the nodes are scenarios; level-3 is a set of UML2.0 ([OMG]) interaction diagrams. Each level-1 use case node is defined by a level-2 scenario chart (i.e., a set of connected scenario nodes). Each level-2 scenario node is defined by a UML2.0 interaction diagram. In Figure 1, 7 use cases are connected in a level-1 use case chart that starts with an initial use case and then forks into 4 "threads". Each of these 7 use cases is defined by a level-2 scenario chart. In Figure 1, the scenario chart for the use case at the source of the dashed arrow is shown. In this scenario chart, there are three scenario nodes. Each node is defined by a UML2.0 interaction diagram.

Semantically, control flow of the entire use case chart starts with the initial node of the use case chart (level-1). Flow then passes between use case nodes along the edges of the level-1 activity diagram. When flow reaches a use case chart node at level-1, level-2 scenario chart defining this node is executed, with

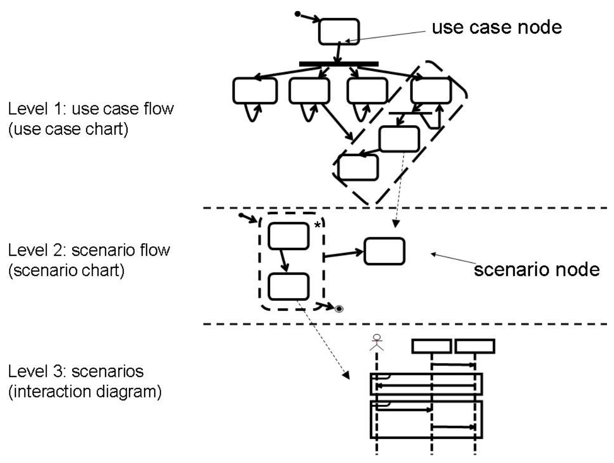

Fig. 1. Use Case Charts.

flow starting from the scenario chart's initial node. Flow exits a scenario node when a final node is reached. Scenario charts may have two types of final nodes—a final success node represents successful completion of the scenario chart and a final failure node represents completion but with failure. Flow only continues beyond the current use case node if a final success node is reached in the use case's defining scenario chart. The semantics of each scenario chart is similar to that for high-level message sequence charts (hMSCs) [IT96]. Each scenario chart node is defined by a UML2.0 interaction diagram. Hence, when flow passes into a scenario chart node, the defining interaction diagram is executed. When the interaction diagram completes, flow returns to the level-2 scenario chart, exits the scenario node at that level and continues with the next scenario node.

The intention is to reuse as much of the notation of UML2.0 as possible. This makes it easy for practitioners to learn the language. The activity diagrams used are a restriction of UML2.0 with some additional features. Although use case charts rely on the notation of UML activity diagrams, the semantics is quite different. UML2.0 activity diagrams are a general purpose modeling language for workflow modeling and business process modeling. Their (informal) semantics is petri-net based [OMG]. In contrast, the formal semantics for use case charts is a denotational, trace-based semantics.

2 Example of Use Case Charts

Figures 2, 3 and 4 give an example of how use case charts can be used to precisely describe use cases. The system under development is an automated train shuttle service in which autonomous shuttles transport passengers between stations [Sof05]. When a passenger requires transport, a central broker asks all active shuttles for bids on the transport order. The shuttle with the lowest bid wins. A complete set of requirements for this application is given in [Sof05]. Figure 2 shows a use case chart that includes use cases for initialization of the system, maintenance and repair of shuttles, and transportation (split into multiple use cases). Each use case node in Figure 2 is defined by a level-2 scenario chart—Figure 3 is the scenario chart for Carry Out Order. Figure 4 is a level-3 interaction diagram for the scenario chart defining Make A Bid.

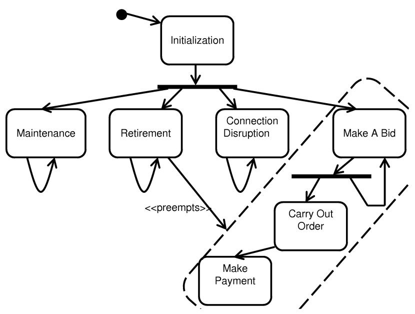

Fig. 2. Shuttle System Use Case Chart.

Figure 2 shows that the shuttle system first goes through an Initialization use case. After that, four use cases execute in parallel. If the Make A Bid use case is successful, it can be followed by Carry Out Order or another bidding process (executed in parallel). The Retirement use case represents the case when the shuttles are shut down. It preempts any activity associated to Make A Bid. This is represented by a stereotyped preemption relationship that applies to a region. A region is a set of nodes enclosed in a dashed box.

Figure 3 is a description of what happens in the Carry Out Order use case. Transportation of passengers takes place and the broker is informed of success. The asterisk in the region represents the fact that the region may execute in parallel with itself any numbers of times, i.e., there may be multiple concurrent transports. The requirements of the problem state that during transport, shuttles may not move to intermediate stations except to pick up or drop off passengers. This is captured by introducing a negative scenario node with a stereotyped negation arrow. Note that scenario charts must have at least one final success or final failure node. A final success node represents the fact that execution of the use case has successfully completed and is depicted graphically as in Figure 3. A final failure node says that the use case completes but that execution should not continue beyond the use case. This is given graphically using the final flow node of activity diagram notation, i.e., a circle with a cross through it[1]. As an example, suppose that the passenger transport cannot be completed for some reason. This could be captured by introducing a scenario node capturing the

[1] Note that this is not the standard UML2.0 interpretation for the final flow node.

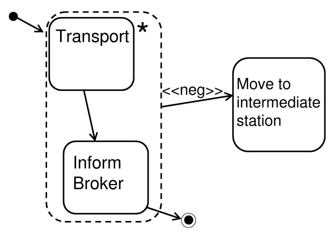

Fig. 3. Shuttle System Scenario Chart for Carry Out Order.

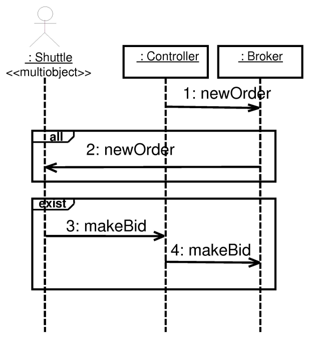

Fig. 4. Shuttle System Interaction Diagram for a scenario in Make A Bid.

failure and then an arrow to a final failure node. In this case, when the final failure node is reached, the Make Payment use case in Figure 2 will not execute, i.e., payment will not be paid for an unsuccessful transport.

Each scenario node in Figure 3 is described by a UML2.0 interaction diagram. Figure 4 shows an interaction diagram that is part of the Make A Bid use case. This particular example is shown to illustrate extensions that use case charts introduce to UML2.0 interaction diagrams, namely, multiobjects and universal/existential messages. We introduce two new interaction operators, **exist** and **all**. We also introduce a stereotype ⟨⟨multiobject⟩⟩ which denotes that an interaction applies to multiple instances of a classifier. In the figure, Shuttle is stereotyped as a multiobject which means that multiple shuttles may participate in the interaction. There are two interaction fragments. In the first, the Broker sends messages to *all* shuttles. In the second, there must be *at least one* makeBid message to Controller followed by at least one makeBid message to Broker.

The activity diagrams used in use case charts and scenario charts are a restricted version of UML2.0 activity diagrams but with some additional relationships between nodes. They are restricted in that they do not include object flow, swimlanes, signals etc. They do include additional notations, however. The abstract syntax is defined in Section 3. The concrete syntax reuses as much of the activity diagram notation as possible. Informally, the allowed arrow types between nodes (either in use case or scenario charts) are given as follows, where, for each arrow, X and Y are either both scenario nodes or both use case nodes:

1. X continues from Y (i.e., the usual activity diagram arrow)
2. X and Y are alternatives (the usual alternative defined by a condition)
3. X and Y run in parallel (the usual activity diagram fork and join)
4. X preempts Y—i.e., X interrupts Y and control does not return to Y once X is complete, shown by the stereotype ⟨⟨preempts⟩⟩ from X to Y.

5. X suspends Y—i.e., X interrupts Y and control returns to Y once X is complete, shown by the stereotype ⟨⟨suspends⟩⟩ from X to Y.
6. X is negative—i.e., the scenarios defined by X should never happen. This is shown by an arrow stereotyped with ⟨⟨neg⟩⟩ to X and where the source of the arrow is the region over which the scope of the negation applies.
7. X may have multiple copies—i.e., X can run in parallel with itself any number of times. This is shown by an asterisk attached to node X.

In addition, use case charts and scenario charts may have regions (graphically shown by dashed boxes) that scope nodes together. Arrows of type (4), (5), (8) may have a region as the target of the arrow. Arrows of type (7) may have a region as the source of the arrow. All other arrows do not link regions.

Arrow types (4), (5), (6) and (8) are not part of UML2.0 activity diagrams (although there is a similar notation to (4) and (5) for interruption). Activity diagrams do have a notion of region for defining an interruptible set of nodes. Regions in use case charts are a general-purpose scoping mechanism not restricted to defining interrupts. In addition to the arrow and region extensions, there are minor extensions to interaction diagrams.

3 Use Case Chart Syntax

The abstract syntax for interaction diagrams is not given as it is assumed to be the same as in UML2.0 except for the multiobject, universal/existential message extensions. The concrete syntax for use case and scenario charts has already been described and will not be addressed further.

3.1 Abstract Syntax for Scenario Charts (Level-2)

The abstract syntax of a scenario chart is given first. The abstract syntax for use case charts is almost the same since both are based on activity diagrams.

Definition 1. *A* scenario chart $(S, R_S, E_S, s_0, S_F, S_{F'}, L_S, f_S, m_S, L_E)$ *is a graph where S is a set of scenario nodes, $R_S \subseteq \mathcal{P}(S)$ is a set of regions, $E_S \subseteq (\mathcal{P}(S \cup R_S) \times \mathcal{P}(S \cup R_S) \times L_E)$ is a set of edges with labels from L_E, $s_0 \in S$ is the unique initial node, $S_F \subset S$ is a set of final success nodes, $S_{F'} \subset S$ is a set of final failure nodes, L_S is a set of scenario labels, $f_S : S \to L_S$ is a total, injective function mapping each scenario node to a label and $m_s : S \cup R_S \to \{+, -\}$ is a total function marking whether or not each scenario or region can have multiple concurrent executions. The labels in L_S are references to an interaction diagram. L_E is defined to be the set $\{normal, neg, preempts, suspends\}$. L_S is the set of words from some alphabet Σ.*

This definition describes a graph where edges may have multiple source nodes and multiple target nodes. This subsumes the notion of fork and join from activity diagrams which can be taken care of by allowing edges to have multiple source nodes and/or multiple target nodes. Multiple source nodes lead in the use case chart graphical notation to a join and multiple target nodes lead to a fork. An edge with both multiple sources and multiple targets is equivalent to a join followed by a fork. Regions are a scoping mechanism used to group nodes. As stated previously, the intuition behind final success and final failure nodes is that a final success node denotes successful completion of the scenario chart; a final failure node denotes that the scenario chart completes but unsuccessfully. Definition 1 omits the notion of conditions on edges, for the sake of clarity, but it is enough to say that guards could be placed on arrows leaving a node.

3.2 Abstract Syntax for Use Case Charts (Level-1)

The abstract syntax for a use case chart is almost identical except that a use case chart has only one type of final node (for success) and each use case node maps to a scenario chart not an interaction diagram. Only one type of final node is required for use case charts because there is no notion of success or failure—either a use case chart completes or it does not.

Definition 2. *A* use case chart $(U, R_U, E_U, u_0, U_F, L_U, f_U, m_U, L_E)$ *is a graph where U is a set of nodes, $R_U \subseteq \mathcal{P}(U)$ is a set of regions, $E_U \subseteq (\mathcal{P}(U \cup R_U) \times \mathcal{P}(U \cup R_U) \times L_E)$ is a set of edges, $u_0 \in U$ is the unique initial node, $U_F \subset U$ is a set of final nodes, L_U is a set of scenario chart labels, $f_U : U \rightarrow L_U$ is a total, injective function mapping each use case node to a scenario chart label and $m_U : U \cup R_U \rightarrow \{+, -\}$ is a total function marking whether each use case or region can have multiple concurrent executions. The labels in L_U are references to a scenario chart. L_E is as given in Definition 1.*

4 Use Case Chart Semantics

A trace is a sequence of events where an event may be the sending of a message, $!x$, or the receipt of a message, $?x$.

Definition 3. *The semantics of a 3-level use case chart, U, is a pair of trace sets, (P_U, N_U), where P_U is the set of positive traces for U and N_U is the set of negative traces for U.*

Positive traces are traces that are possible in any implementation of the use case chart. Negative traces may never occur in a valid implementation of the use case chart. An implementation satisfies a use case chart if every positive trace is a possible execution path and if no negative trace is a possible execution path.

First, the semantics of (a restriction of) UML2.0 interaction diagrams is given, followed by the semantics for scenario charts, and finally, use case charts.

4.1 Semantics of UML2.0 Interaction Diagrams (Level-3)

The semantics for UML2.0 interaction diagrams follows the one given by Haugen & Stølen [HHRS05], extended to include **all** and **exist** fragments.

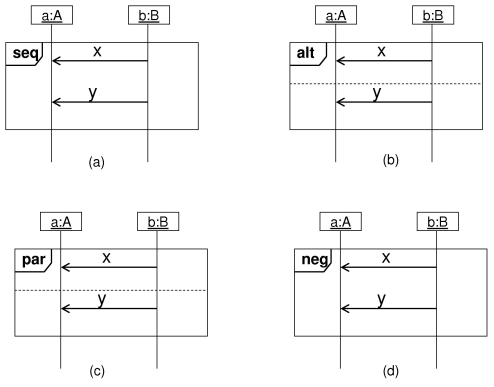

Fig. 5. UML2.0 Interaction Fragments.

A message, x, in a UML2.0 interaction has two events—a send event, $!x$, and a receive event, $?x$. In any valid event trace, the send event must come before the receive event. In UML2.0, as shown in Figure 5, messages can be composed using interaction fragments, where a fragment has an interaction operator and a number of interaction operands. For example, Figure 5(b) shows an alternative fragment with two operands; 5(c) shows a parallel fragment with two operands; and 5(d) shows a negative fragment with a single operand. The default operator in UML2.0 is the sequential operator, **seq** (Figure 5(a)), which represents weak sequencing. Any messages not explicitly contained within a fragment are by default assumed to be contained within a **seq** fragment.

A message is a triple (s, tr, re) of a signal s, a transmitter instance, tr, and a receiver instance, re. Each transmitter instance has a type, $tr : Tr$. Similarly, $re : Re$. Let M denote the set of all messages and L the set of all lifelines. An event is a pair of kind and message: $(k, m) \in \{!, ?\} \times M$. Let E denote the set of all events. A trace is a sequence of events. Let $tr(e)$ denote the transmitter for event e and $re(e)$ denote its receiver. Let H be the set of valid event traces, that is, event traces such that for any message x, the send event, $!x$, comes before the receive event, $?x$. Define the following operators on traces. $h_1 \frown h_2$ is trace concatenation. $h_1|_B$ is the trace h_1 restricted to events in the event set B — i.e., all events not in B are removed.

The semantics for the four fragments in Figure 5, as well as for universal/existential messages, is summarized in Figure 6. e is an event and d_i is an interaction diagram, for all i. The semantics of a single event is a single positive trace. Interaction operators are represented textually using the keywords **neg**, **alt**, **par** and **seq**. For example, **neg** d represents an interaction diagram d that is negated by a negative interaction fragment.

$$[\![e]\!] = (e, \emptyset)$$

$$[\![\textbf{neg } d]\!] = (\emptyset, p \cup n)$$

$$[\![d_1 \textbf{ alt } d_2]\!] = (p_1 \cup p_2, n_1 \cup n_2)$$

$$[\![d_1 \textbf{ par } d_2]\!] = (p_1 \| p_2, (n_1 \| p_2) \cup (n_1 \| n_2) \cup (p_1 \| n_2))$$

$$[\![d_1 \textbf{ seq } d_2]\!] = (p_1 \succeq p_2, (n_1 \succeq p_2) \cup (n_1 \succeq n_2) \cup (p_1 \succeq n_2))$$

$$[\![\textbf{all } d]\!] = (\textbf{all } p, \textbf{all } n)$$

$$[\![\textbf{exist } d]\!] = (\textbf{exist } p, \textbf{exist } n)$$

$$\text{where } (p, n) = [\![d]\!], (p_1, n_1) = [\![d_1]\!] \text{ and } (p_2, n_2) = [\![d_2]\!]$$

Fig. 6. UML2.0 Interaction Diagram Semantics.

For **alt**, the set of positive traces is the union of the set of positive traces from each operand. The set of negative traces is the union of the set of negative traces from each operand. The **neg** operator simply negates all traces—its set of negative traces is the union of the positive and negative traces of its operand. This captures the fact that the negation of a negative trace remains negative.

par is defined by interleaving traces from each of its operands. In Figure 6, $\|$ denotes interleaving and is formally defined below. **par**'s positive traces are the interleavings of positive traces from both operands. Its negative traces are the interleavings of negative traces from both operands, or a positive trace from one operand with the negative trace from the other operand. Interleaving is defined as follows for trace sets s_1, s_2 (adapted from [HHRS05]):

$$s_1 \| s_2 = \{h \in H \mid \exists o \in \{1, 2\}^\infty \cdot \pi_2((o, h)|_{\{1\} \times E}) \in s_1 \ \wedge \ \pi_2((o, h)|_{\{2\} \times E}) \in s_2\}$$

The infinite sequence o is an oracle to resolve non-determinism in the interleaving. π_2 is a projection operator returning the second element in a pair. Any trace in the set $s_1 \| s_2$ is an interleaving of events from a trace in s_1 with events from a trace in s_2.

seq fragments are defined in UML2.0 to have a weak sequencing semantics ([OMG]): the ordering of events within each operand is maintained; events on different lifelines from different operands may come in any order; events on the same lifeline from different operands are ordered such that an event from the first operand comes before an event from the second operand. Any **seq** fragment joins traces from each of its operands in a way that satisfies these three constraints. Informally, the positive traces for **seq** are all possible ways of joining a positive trace from the first operand and a positive trace from the second operand. The negative traces for **seq** are those derived from joining a positive trace from the first operand with a negative trace from the second, or a negative trace from the first with either a positive or negative trace from the second.

The definition in Figure 6 relies on a definition of $\succeq$, weak sequencing for trace sets (adapted from [HHRS05]), which captures formally the three constraints stated above. $ev(l)$ is the set of events that take place on lifeline l.

$$s_1 \succeq s_2 = \{h \in s_1 \| s_2 \mid \exists h_1 \in s_1, h_2 \in s_2 \cdot \forall l \in L \cdot h|_{ev(l)} = h_1|_{ev(l)} \frown h_2|_{ev(l)}\}$$

The semantics for the multiobject extensions are now given. Consider first the interaction operator **all** applied to a single positive event trace, **all** $e_1, e_2, \ldots$. The resulting positive traces are all those that can be derived by replacing each e_i by its *image under* **all**. If t_i is a receive event where the receiving instance is a multiobject, then the image under **all** is the trace $e_{i_1}, e_{i_2}, \ldots$ where each e_{i_j} is the same event as e_i but with a different receiver, namely, instance j. The corresponding send event is also replaced by a set of send events, one for each instance j. The same logic applies if e_i is a send event where the sending instance is a multiobject. In this case, e_i is replaced by a set of send events, one for each instance of the multiobject, and the corresponding receive events for the new send events are added.

For an event e, define $e \frown_{(I,re)}$, where I is a set of type instances, as a concatenation of copies of e where each element of the concatenation has the receiver of e replaced by an element of I. Similarly, $e \frown_{(I,tr)}$ is a concatenation of copies of e where each element of the concatenation has the transmitter replaced by an element of I. Furthermore, $\bigfrown_{e_i \in h}$ defines repeated concatenation indexed over the events e_i of an event trace h. If $tr(e) : Tr$ and $re(e) : Re$, then let $inst_{tr}(e)$ denote the set of all instances of Tr (including $tr(e)$ itself). Similarly, $inst_{re}(e)$ is the set of all instances of Re (including $re(e)$).

Now define **all** e as follows:

$$\textbf{all}\ e = \begin{cases} e \frown_{(inst_{tr}(e),tr)} & \text{if } tr(e) \text{ is a multiobject} \\ e \frown_{(inst_{re}(e),re)} & \text{if } re(e) \text{ is a multiobject} \\ \bigfrown_{e_i \in e \frown_{(inst_{tr}(e),tr)}} e_i \frown_{(inst_{re}(e),re)} & \text{if both } tr(e) \text{ and} \\ & \quad re(e) \text{ are multiobjects} \\ e & \text{otherwise} \end{cases}$$

The intent of this definition is to effect the replacement of events by multiple events, one for each instance, as described above. In the case that an event has a multiobject receiver and a multiobject transmitter, the definition describes a "nested" replacement, in which the replacement is first done for the transmitter and then the result is processed with receiver replacement.

For a trace h, $h[e'/e]$ is defined as the trace h with all occurrences of event e replaced by e'. Multiple replacements are separated by commas and applied sequentially. Now define **all** h for an event trace $h = e_1, e_2, \ldots$ as follows:

$$\textbf{all}\ h = h\ [(\textbf{all}\ e_1)/e_1, (\textbf{all}\ e_2)/e_2, \ldots]$$

The definition extends naturally to a set of traces, s:

$$\textbf{all}\ s = \{h \in H \mid h = (\textbf{all}\ h_1) \wedge h_1 \in s\}$$

The definition of the semantics of **all** applied to an interaction diagram, as given in Figure 6, is now clear. The case for **exists** is similar and is not presented here, for lack of space.

This concludes the definition of the trace-based semantics for UML2.0 interaction diagrams. UML2.0 contains other constructs not considered here.

4.2 Semantics of Scenario Charts (Level-2)

The semantics is extended to scenario charts in the natural way—the semantics is also given as a pair of a set of positive traces and a set of negative traces.

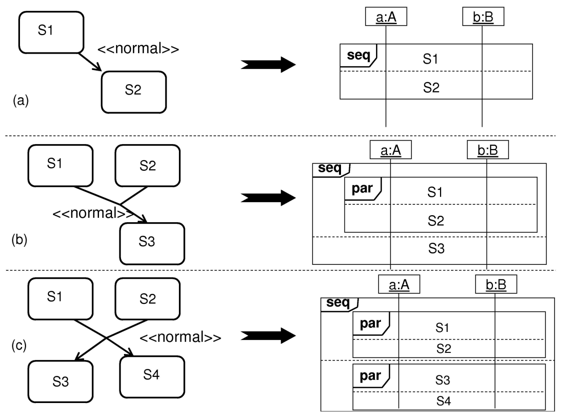

Fig. 7. Flattening Scenario Charts.

Edges of type *normal* in scenario charts can be given a semantics by "flattening" the edge—i.e., create a new interaction diagram that takes the interaction diagrams represented by the source and target of the edge and connects them using an interaction fragment with a particular interaction operator. See Figure 7. Normal edges with only one source and target scenario node can be flattened using the **seq** interaction operator for sequential composition. This captures the weak sequential semantics of one-to-one *normal* edges. Many-to-many *normal* edges are flattened using the **par** interaction operator. This is because the semantics of a one-to-many edge is defined to be a forking and that of a many-to-one edge is defined to be a joining of "threads". Hence, a many-to-many edge can be replaced by a fork and join in the usual activity diagram notation. Since *normal* edges can be eliminated in this way, their semantics is not explicitly given here but the semantics is assumed to be that of the equivalent "flattened" interaction diagram. This leaves only edges of type *neg*, *preempts* and *suspends*.

In what follows, c_1 **preempts** c_2 informally means that scenario node c_1 preempts scenario node c_2. c_1 **suspends** c_2 means that c_1 suspends c_2 and c_1 **negative during** c_2 means that c_1 can never happen during the execution of c_2. $c_1 \rightarrow c_2$ denotes a normal edge between scenario nodes. Edges can also be between sets of scenario nodes. c_1* denotes that multiple occurrences of c_1 can occur in parallel. The semantics for preemption, suspension and negation are given only for one-to-one edges, but can be extended to many-to-many edges. Figure 8 summarizes the semantics. In this figure, c_1, c_2 are scenario nodes defined by interaction diagrams d_1 and d_2, respectively. C_1 and C_2 are sets of scenario nodes defined by sets of interaction diagrams D_1 and D_2 where there is a bijective mapping from C_i to D_i. **par** X, for a set of interaction diagrams $X = \{x_1, x_2, \ldots\}$, is shorthand for x_1 **par** x_2 **par** .... $size(X)$ returns the number of elements in X. If $size(X) = 1$, X' refers to its only element. $prefix(h)$ denotes the set of prefixes of event trace h.

$$[\![C_1 \rightarrow C_2]\!] \begin{cases} D'_1 \ \mathbf{seq}\ D'_2 & \text{if } size(D_1) = 1 \wedge size(D_2) = 1 \\ (\mathbf{par}\ D_1)\ \mathbf{seq}\ D'_2 & \text{if } size(D_1) > 1 \wedge size(D_2) = 1 \\ D'_1\ \mathbf{seq}\ (\mathbf{par}\ D_2) & \text{if } size(D_1) > 1 \wedge size(D_2) = 1 \\ (\mathbf{par}\ D_1)\ \mathbf{seq}\ (\mathbf{par}\ D_2) & \text{if } size(D_1) > 1 \wedge size(D_2) > 1 \end{cases}$$

$$[\![c_1 \ \mathbf{preempts}\ c_2]\!] = (\{h \in H \mid \exists h_1 \in p_1, h_2 \in H, h' \in p_2 \cdot h = h_2 \frown h_1 \wedge h_2 \in prefix(h')\}, n_2)$$

$$[\![c_1 \ \mathbf{suspends}\ c_2]\!] = (\{h \in H \mid \exists h_1 \in p_1, h_2 \in p_2, h_{2a}, h_{2b} \in H \cdot h = h_{2a} \frown h_1 \frown h_{2b} \wedge h_2 = h_{2a} \frown h_{2b}\}, n_2)$$

$$[\![c_1 \ \mathbf{negative\ during}\ c_2]\!] = (p_2, n_2 \cup p_1 \| p_2)$$

$$[\![c_1 *]\!] = d_1 \ \mathbf{par}\ d_1\ \mathbf{par} \ldots$$

where c_1, c_2 are defined by interaction diagrams d_1, d_2 respectively and $(p_1, n_1) = [\![d_1]\!]$, $(p_2, n_2) = [\![d_2]\!]$

Fig. 8. Semantics for Edges in Scenario Charts.

For preemption, a positive trace for (c_1 **preempts** c_2) is any trace made up of a prefix of a positive trace of c_2 concatenated with a positive trace of c_1. Note that a preempting scenario cannot have negative traces. Furthermore, (c_1 **preempts** c_2) does not introduce any new negative traces because preempting traces have no effect on the original negative traces. The case for suspension is similar except that control returns to the suspended scenario once the suspending scenario is complete.

In the case of negation, the positive traces of (c_1 **negative during** c_2) are simply the positive traces of c_2. Negative traces, however, can be any trace that is an interleaving of a positive trace of c_2 with a positive trace of c_1. This, in effect, defines a monitor for traces of c_1—if a positive c_1 trace occurs at any point, even with events interleaved from c_2, then this defines a negative trace. Note that c_1 cannot have negative traces.

The semantics for multiple concurrent executions (the asterisk notation) is given by interleaving and hence can be described in terms of flattening using **par** operators. The number of **par** operators is unbounded since there can be any number of executions of the node.

Regions are sets of connected nodes and so, their semantics is a pair of trace sets. Hence, their semantics is not given explicitly here but follows the same rules as in Figure 8. The semantics for final success and final failure nodes are given in the following subsection.

Figure 8 defines the semantics for single edges. This is extended to an entire scenario chart as follows. A path through a scenario chart is a (possibly infinite) sequence of scenario nodes $s_0, s_1, s_2 \ldots$ where s_0 is the unique initial node. If the path is finite, it must be ended by either a final success or final failure node. A path is maximal if it is not a proper prefix of any other path. The set of positive traces of a scenario chart is the set of traces that follow a maximal path through the chart. Similarly, for the set of negative traces.

4.3 Semantics of Use Case Charts (Level-1)

The semantics for use case charts is essentially the same as for scenario charts because both a scenario and a use case are given meaning as a pair of trace sets. For use case charts, however, the meaning of a *normal* edge is given by strong not weak sequential composition. Operationally, this means that before execution can continue along an edge to the next use case, *all* participants in the interaction must complete (where completion is defined below). In contrast, in scenario charts, some participants may complete and continue to the next node while others remain in the current node. Strong composition is chosen to define use case charts because nodes represent use cases. Use cases are considered modular functional units in which the entire unit must complete before control goes elsewhere. Strong composition enforces the modularity. Semantically, strong composition of traces is defined to be concatenation.

A use case chart node completes if and only if its defining scenario chart reaches a final success or final failure node. If the scenario chart reaches a final success node, control continues to the next use case node. If the scenario chart reaches a final failure node, the use case "thread" terminates. Semantically, each trace in a scenario chart is either infinite, ends with a final success node (a success trace) or a final failure node (a failure trace). Suppose a use case chart has two nodes, u_1 and u_2, connected by a single edge from u_1 to u_2. Then the positive trace set of the use case chart is the union of three trace sets: the positive infinite traces of u_1, the set of traces formed by concatenating positive success traces from u_1 with positive traces from u_2, and the set of positive failure traces from u_1. The first of these three trace sets captures the fact that infinite traces of u_1 never reach u_2. The second of the trace sets captures strong composition and the final trace set corresponds to the case when traces in u_1 end at a final failure node. This is captured formally in Figure 9.

For any use case node, u_i, let $s(u_i)$, $f(u_i)$ denote the set of positive traces of u_i that end in a final success and final failure node, respectively, and let $inf(u_i)$

denote the infinite set of positive traces. Then the set of positive traces of u_i is the disjoint union of $s(u_i)$, $f(u_i)$ and $inf(u_i)$. In Figure 9, the definition of concatenation is extended to sets of traces, in the natural way, as follows:

$$s_1 \frown s_2 = \{h \in H \mid \exists h_1 \in s_1, h_2 \in s_2 \cdot h = h_1 \frown h_2\}$$

where s_1, s_2 are trace sets. For negative traces, the final success and final failure nodes have no effect; negative traces are composed using strong composition. Finally, only normal edges are affected by final success and final failure nodes, i.e., preemption, suspension and negation edges retain the same semantics.

$$[\![c_1 \to c_2]\!] = (inf(c_1) \cup f(c_1) \cup (s(c_1) \frown p_1), (n_1 \frown p_2) \cup (n_1 \frown n_2) \cup (p_1 \frown n_2))$$

Fig. 9. Semantics for Normal Edges in Use Case Charts.

5 Related Work

At first glance, use case charts look quite similar to hMSCs [IT96] and UML2.0 [OMG] IODs. However, in IODs, there are only two levels of hierarchy — activity diagrams connect references to interaction diagrams but use cases are not handled. In hMSCs, nodes can be references to other hMSCs so there is an unlimited number of levels. However, there is no semantic difference between nodes at different levels—references to hMSCs are just syntactic sugar and can be flattened to references to basic MSCs—so there are in effect only two levels, interactions and references to interactions. Use cases again are not handled.

Use case charts contain relationships that do not exist in UML or hMSCs, as noted in Section 2. Finally, there is a formal semantic model for use case charts. There is no official formal semantics for UML2.0 IODs. Although one can infer a semantics for UML2.0 activity diagrams (or at least part of them) because the UML specification [OMG] bases the semantics on petri-nets, the semantic assumptions of generic UML activity diagrams do not carry over to IODs because a number of restrictions and modifications are made to the activity diagrams used in IODs. The semantics given here for use case charts is declarative. It is also possible to define an operational semantics based on petri-nets but this is outside the scope of this paper.

Activity diagrams and hMSCs can, of course, be used in a variety of ways to support use-case based development. Some authors (e.g., [MB02]), for example, suggest the use of activity diagrams to connect use cases. Others (e.g., [MZ99]) suggest to define each use case by an hMSC. The former approach does not consider how to use activity diagrams to define each use case. The latter only connects use cases using a standard UML use case diagram. Use case charts essentially combine these two approaches in that activity diagrams are used both to relate use cases and to define those use cases. As such, the contribution of this paper is more in the formal semantics than the syntax.

6 Conclusion

This paper presented a precise notation for specifying use cases. The notation is based on UML and is defined on three levels: use cases, scenarios and interactions. A formal syntax and semantics of the notation is presented.

Use case charts are precisely and unambiguously defined, and can therefore be executed. A project is currently underway to implement a simulator for use case charts that is compliant with the semantics defined in this paper. This will enable users to immediately execute their use cases and validate the use case specification. Clearly, use case charts require a degree of rigor and effort above and beyond what is normal for use case definition. The author feels, therefore, that the notation is most beneficial when applied to the specification of systems with stringent functional requirements, e.g., systems that are highly distributed, concurrent and/or safety-critical. Use case charts have been applied on a number of industrial case studies, most notably a transaction-based weather data system that is part of a NASA air traffic control application.

References

[BRJ05] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide, 2nd Edition*. Addison-Wesley Professional, 2005.

[Coc00] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[HHRS05] Øystein Haugen, Knut Eilif Husa, Ragnhild Kobro Runde, and Ketil Stølen. Stairs: Towards formal design with sequence diagrams. *Journal of Software and System Modeling*, 2005. To Appear.

[Iso04] Sadahiro Isoda. On uml2.0s abandonment of the actors-call-use-cases conjecture. *Journal of Object Technology*, 4(6), 2004.

[IT96] ITU-TS. Recommendation z.120. Technical report, 1996.

[MB02] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley, Boston, USA, 2002.

[MZ99] Nikolai Mansurov and D. Zhukov. Automatic synthesis of sdl models in use case methodology. In *SDL Forum*, pages 225–240, 1999.

[OMG05] OMG. Unified Modeling Language 2.0 specification, 2005. http://www.omg.org.

[OP99] Gunnar Overgaard and Karin Palmkvist. A formal approach to use cases and their relationships. In *First International Workshop on The Unified Modeling Language UML'98*, pages 406–418. Springer-Verlag, 1999.

[Smi04] Michal Smialek. Accommodating informality with necessary precision in use case scenarios. *Journal of Object Technology*, 4(6), 2004.

[Sof05] Software Engineering Group, University of Paderborn. Shuttle system case study, 2005. http://www.cs.uni-paderborn.de/cs/ag-schaefer/CaseStudies/ShuttleSystem/.

[Ste01] Perdita Stevens. On use cases and their relationships in the unified modelling language. In *FASE '01: Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, pages 140–155, London, UK, 2001. Springer-Verlag.

[Wil04] Clay Williams. Towards engineered, useful use cases. *Journal of Object Technology*, 4(6), 2004.