Integrated Design and Process Technology, **IDPT-2003**
Printed in the United States of America, June, 2003
©2003 Society for Design and Process Science

GETTING "WHOLE PICTURE" BEHAVIOR IN A USE CASE MODEL[*]

Frantisek Plasil[1,2], Vladimir Mencl[1]

[1]**Charles University, Faculty of Mathematics and Physics**
Department of Software Engineering
Malostranske namesti 25, 118 00 Prague 1, Czech Republic
{plasil, mencl}@nenya.ms.mff.cuni.cz, http://nenya.ms.mff.cuni.cz,
phone: +420 2 2191 4266, fax: +420 2 2191 4323

[2]**Academy of Sciences of the Czech Republic**
Institute of Computer Science
plasil@cs.cas.cz, http://www.cs.cas.cz

ABSTRACT

Although widely used, traditional use case modeling does not provide explicit means which could be easily used for capturing and testing behavior compliance of the entities involved in a particular use case model. Specifically, a use case model (a set of use cases) related to a system under design provides neither an explicit abstraction to capture the "whole picture" of the behavior of the system, nor to cover the interactions of subsystems and internal actors with the parent system. With the aim to allow for reasoning on the behavior, the paper introduces a simple formal model **Generic UC View** which identifies important abstractions and the relations upon them which target the goal. Among them, the concept of use case expression is the base for the desired reasoning on whether the behavior of an entity (such as an agent, a subsystem or a software component) complies with the composed behavior of its sub-entities, and the behavior on the communication links of two neighboring entities is compliant.

As a proof of the concept, an instance of use case expressions, **pro-case**s is introduced. Based on behavior protocols [20], pro-cases can be checked for compliance via an already existing verifier. As pro-cases' syntax is simple, resembling regular-expressions, there are simple guidelines for transforming a use case written in classical textual form (based on a template) into a pro-case.

1. INTRODUCTION

1.1. USE CASES – BRIEF OVERVIEW

In principle a use case [11, 3, 17] is a description of a set of scenarios specifying how a set S of entities ought to communicate to achieve a certain goal; a communication is viewed as a sequence of events, such as a request or a response exchanged among the entities. Here, an entity can represent a system, a subsystem, an agent or a software component. Frequently, the use case is written from the perspective of one of those entities (*SuD*, system under discussion) – it specifies how SuD executes certain actions while communicating with other entities, *actors*, from S to achieve a specific goal. Basically, a *scenario* is considered a sequence of *actions* to be performed by SuD and the actors which reflects a particular case of their desired communication.

Practitioners, e.g. [3, 14, 21], typically prefer use cases to be specified in plain English, to make them easily comprehendible to "wide audience". Such a use case is inherently informal, even though a predefined template is usually asked to follow. For example, such template can be a form to be filled in to specify in a semi-programing way the desired set of scenarios:

Use Case: #1 Seller submits an offer
Scope: Marketplace
SuD: Marketplace Information System
Level: **Primary Task**
Primary Actor: **Seller**
Supporting Actor: **Trade Commission**
Main success scenario specification:

1. Seller submits information describing an item
2. System validates the description.
3. Seller adjusts/enters price and enters contact and billing information.
4. System validates the seller's contact information.
5. System verifies the seller's history to permit the seller to operate

[*]This work was partially supported by the Grant Agency of the Czech Republic project 201/03/0911 and 102/03/0672

6. System validates the whole offer with the Trade Commission
7. System lists the offer in published offers.
8. System responds with an uniquely identified authorization number.

Extensions:

2a Item not valid
 2a1 Use case aborted
5a Seller's history inappropriate
 5a1 Use case aborted
6a Trade commission rejects the offer
 6a1 Use case aborted

Sub-variations:

2b Price assessment available
 2b1 System provides the seller with a price assessment.

There is a whole variety of ways different authors recommend to write use cases, ranging e.g. from employing preconditions/postconditions in Catalysis [5], Use Case Maps [2], transition systems [23], to abstract state machines with the goal to generate test scenarios [8].

UML [17] includes a use case concept as well. It is, however, primarily focused on use case as an abstraction to capture the existence of a set of interaction scenarios among a set of actors and an SuD; it leaves the way internals of a use case are specified very open (the alternatives explicitly mentioned without any details include plain text, a state machine, activity graph, and specification via preconditions and postconditions). Rather, it concentrates on the relations among use cases (as a use case is a classifier in UML, a relation can be a dependency (extends/includes) or generalization).

In general, the bottom line is that there are many different approaches and hard to compare techniques related to use cases, none of them being strongly recommended nor preferred; an overview is e.g. in [10, 7].

Intuitively, there can be conflicts in use cases specifying two cooperating entities (separate SuDs). Even though there are many approaches to finding conflicts in dynamic and functional requirements, as pointed out in [9], they are frequently based on logic (and typically closely dependent on a particular use case technology) and require highly specialized experts to handle. This is in obvious contrast with practitioners' desire to make a use case easy to read and comprehend as mentioned above.

1.2. Goals and Structure of the Paper

In [20, 19], we developed an agent model, where the agents process sequences (traces) of atomic *events*, and introduced a way to describe (approximate) the agents' behavior via behavior protocols, which was applied on software components in SOFA, in order to specify component behavior and test behavior compliance of components, including neighboring levels of refinement. Based on this experience, we realized that similar compliance checks should be done also for use cases associated with component based (interface-centered [5]) design.

Here, as a variety of different use case techniques might be considered, the key question is what the basic relations among the behaviors of entities in use cases are, provided these relations should allow for capturing the behavior relationships among the proposed components in a hierarchical system. For instance, whether the composed behavior of components (at a particular level of nesting), specified separately for each of the components, corresponds to the behavior specified for the parent component, etc. Another question is what the basic relations upon a set of scenarios are, in order to define some "reasonable" behavior concepts and relations among them (associated with use case entities) which could be easily interpreted in classical formal tools, such as state machine, labeled transition systems, etc. Based on theses objectives, the paper aims at these goals:

(1) Finding a generic view on UC in order to articulate key abstractions allowing for capturing behavior compliance of entities/actors at different levels of their decomposition, resp. refinement, and identifying which relations should be chosen as the basis for such reasoning.

(2) Showing how the abstractions in the proposed generic view correspond to the use case related concepts in UML and how the classical textual, system centric use case specifications can be mapped/interpreted in terms of these abstractions.

(3) Introducing a UC technique which would feature these abstractions and thus provide reasoning, while still simple enough to be easy to apply in practice (emphasis on readability and easy comprehension); for instance, transforming a textual use case to the form the new UC technique would require should be an easy step.

Reflecting this aim, the paper is organized as follows: Sect. 2 targets the goals (1) and (2), by providing "Generic UC View", and its comparison to UML, while Sect. 3 addressed the goal (3) by introducing "Protocol Use Cases". The final two sections 4 and 5 contain an evaluation, discuss practical experience and related work, and articulate a conclusion.

2. Our View on Traditional Use Case Modeling Techniques

2.1. Generic UC View

To provide a basis for reasoning about the key abstraction (and capture their relationship) in the traditional use case modeling [3, 11, 13, 17], we introduce the following generic model (*generic UC view*).

Basic concepts. Assume an entity S is composed of sub-entities A1, ... ,An. By definition S forms *scope* of Ai; the topmost scope is called *system*. An entity Ai communicates through communication links (*connections* for short) with (1) other (*actors*) Aj of the scope S, and potentially (2) with other external actors located in the parent scope, i.e. in the scope of S. In case (1), the communication is observed on the *internal* connections of S, while in case (2) on *external* connection of S. Advantageously, the nesting of entities and their scopes can be expressed as a scope diagram. Here, by convention, the stick-figure symbol (🯅) denotes an entity which is an abstraction of a particular human role, while rectangle denotes an entity which activity is at least partially software driven (typically a SuD), and a line represents a connection. For example, in Fig. 1(a), A1, A2, and A3 are in the scope of $\bar{A}2$. Here, C1 and C3 are external connections of $\bar{A}2$, while C2 and C4 are internal connections of $\bar{A}2$. In a similar vein, C1, C2 and C3 are the external connection of A2. Furthermore, $\bar{A}1$, $\bar{A}2$, and $\bar{A}3$ are in the scope of S. Frequently, not all of the levels of entity nesting are captured on a scope diagram, typically leaving out the targets of the external connections of the outmost scope (Fig. 1(b)).

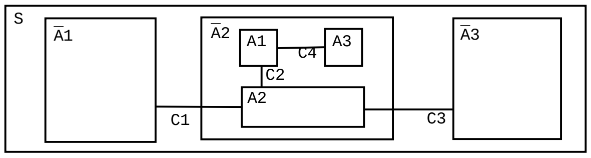

Figure 1 (a)

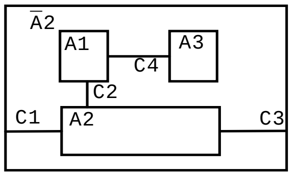

Figure 1 (b)

Figure 2 shows the scope diagram of a sample Marketplace. Within the scope of Martketplace (a business entity), the actors Buyer, Seller, Agency, TradeCommissioner and the Marketplace Information System and their connections are visible. Inside the Marketplace information system are three entities: the Clerk, the Supervisor and the Computer system. The use case demonstrated in Sect 1.1 describes the interaction of the Marketplace information system with its surrounding actors in the scope of Marketplace.

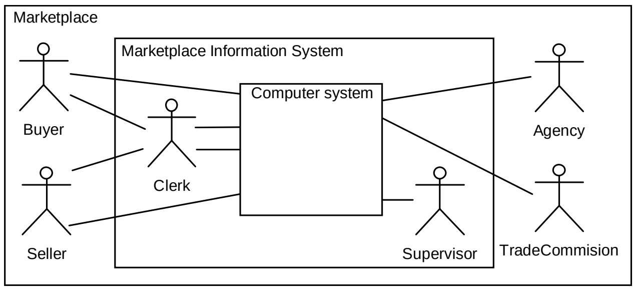

Figure 2

Scenarios. A particular way of communication of an entity A on its connections in a run of system Σ is captured as a *scenario* s ∊ Scenarios. All the scenarios of A in any run of Σ form the *behavior* Com(A) ⊆ Scenarios. On the domain Scenarios we assume the existence of a subscenario relation (partial order).

By convention, Com(A)/ExConn(A) is the behavior of A restricted to its external connections (while Com(A)/InConn(A) is restricted to the internal connections of A); here, Com(A)/Conn denotes restriction of scenarios from Com(A) to the communication observed on the connections from a set of connections Conn. Assuming an entity C is composed of entities A and B, Com(C) is composed of Com(A) and Com(B), written Com(A) $\sqcap_X$ Com(B), where X = InConn(A) ∩ InConn(B), in such a way that

(1) together with Com(A)/InConn(A) ∪ Com(B)/InConn(B) the behavior on the joint connections between A and B becomes Com(C)/InConn(C),

(2) (Com(A)/ExConn(A) ∪ Com(B)/ExConn(B)) –
(Com(A)/ExConn(A) ∩ Com(B)/ ExConn(B))
becomes Com(C)/ExConn(C) (where – stands for set subtraction)

Use case model. Let U^A be the set of *basic uses cases* (behavior specifications) where A is the SuD. A use case UC^A ∊ U^A describes/generates a set of scenarios, so, by convention, we write also Com(UC^A) ⊆ Scenarios. Further, we define a binary relation includes such that UC^A_j includes UC^A_k means that the specification of UC^A_j includes (refers to) the specification of UC^A_k (macro substitution idea, thus no circular dependencies allowed). Also we assume that if a UC^A_j ∊ U^A and UC^A_j includes UC^A_k then also UC^A_k ∊ U^A; moreover we require UC^A_j includes UC^A_k to imply that for any s ∊ Com(UC^A_k) there exists an s’ ∊ Com(UC^A_j) such that s subscenario s’ (*subscenario preservation*).

Notice that even if Com(UC^A_k) = Com(UC^A_j), not necessarily UC^A_k = UC^A_j; thus different specifications can generate the same behavior. Also, in the need to distinguish the elements of U^A we do so by subscripts, writing e.g. UC^A_k (this also reflect that use cases of A are enumerated in a typical use case technology).

A *use case model* of A (denoted UM^A) is a set of *use case expressions*, (also *use cases* for short), where an expression UE^A (syntactically in principle) generates a set of scenarios (by convention we write Com(UE^A) ⊆ Scenarios). A use case expression UE^A is either a basic use case UC^A or is composed by applying operations from a set of operations UEop on their operands – (sub)expressions, assuming some priorities, parenthesis, etc. apply. The semantic of these operations can include sequencing, parallel composition of the scenarios generated by the operands, etc. (as illustrated in Sect. 3.1). The includes relation can be naturally extended to use case expressions.

Whole picture behavior. As a use case UE^A_j ∊ UM^A provides only a partial “j-th” description of A’s behavior (“the whole picture behavior” of A), the *assembled behavior* of UM^A is defined as Com(UM^A) = $\cup_j$ Com(UE^A_j), UE^A_j ∊ UM^A. To emphasize an important special case, we say that UM^A *has a representative* if there is UE^A in UM^A such that Com(UM^A) = Com(UE^A); (also: UE^A is an *representative* of UM^A).

2.1.1. Consistency Issues

Inherent to refinement/synthesis steps in a design of a specified system Σ is the necessity to combine behavior specifications in order to get "the whole picture" behavior specification and capture the behavior specification compliance of several cooperating/nested entities. In particular, the following four issues are closely related to this necessity:

(a) Does the combined behavior as specified by all UM^{Ai} in a scope S comply with the behavior specified for S in UM^{S}.

(b) Does the assembled behavior as specified by a UM^{A} really reflect the desired behavior of A (as this is hard to address directly, we will consider equivalence checking − decidability whether two use case models UM^{A} and $'UM^{A}$ specify the same behavior, i.e. $Com(UM^{A}) = Com('UM^{A})$).

(c) Is the desired communication between Ai and Aj via their connection(s) in S really reflected in the behavior as specified (separately) by UM^{Ai} and UM^{Aj}.

(d) If there is no representative of UM^{A} and the behaviors of use cases in UM^{A} overlap, is there a way to find/construct a representative in order to get a "whole picture behavior" directly from the specifications, without the need to generate scenarios.

In general, addressing (a) and (b) requires defining a *behavior compliance* as a binary relation upon the behavior of entities. Intuitively, Com(A) compliant with Com(B) if B can be replaced in Σ by A by taking over all its external connections in such a way that "B behaves on A's place as it were A". The issue (c) can be addressed by finding a binary relation consent upon behavior of entities: Intuitively, Com(A) consent Com(B) if there is no inconsistency resp."erroneous scenario" in the behavior of A and B on their joint connections. To define these relations (and ⊓) precisely, a specific interpretation of Scenarios and the relations/operations available for it have to be known. Note that the relations are (intentionally) defined only for Com(A) and Com(B) and not A and B, as in a concrete UC view, behavior of A will be approximated by $Com(UM^{A})$. Sect. 3 will define such an approximation for the pro-case model; for more information, please refer to [20].

Assuming the existence of compliant with and consent, the issues (a) - (c) can be rephrased as

(a) $Com(UM^{A1}) \sqcap Com(UM^{A2}) \sqcap .... \sqcap Com(UM^{An})$ compliant with $Com(UM^{S})$, and

(b) $Com(UM^{A})$ compliant with $Com('UM^{A})$ and $Com('UM^{A})$ compliant with $Com(UM^{A})$

(c) $Com(A_i)$ consent $Com(A_j)$

Obviously a big advantage can be taken of extending the definitions of compliant with, consent and ⊓ to UM^{A} (to make them applicable not only on Com(A), i.e. the behavior itself but also on the behavior specification). Then, assuming a set UEop is "reasonably" defined, the consistency issues (a) - (d) can be addressed by reasoning on uses case expression (for an example see Sect. 3.2).

2.2. Case Study: Textual Scenario Specifications

There is a variety of use case techniques/technologies based on textual specification of UC^{A} (ranging from unstructured narratives to semi-structured scripts, see overviews in [10, 7]). In this section we analyze the textual use case specification described in [3], which is based on a structured template (Sect 1.1). We choose it because of its broad recognition [4].

Basic concepts. The central concept, actor has an important special case: stakeholder, having a goal. Scope is introduced as the entity A for which a UC^{A} is written (A is SuD). Communication links (connections) are introduced indirectly by identifying the cooperating actors in each UC^{A}.

Scenarios. A scenario is "sequence of steps showing how actions and interactions unfold" (thus sequence of interacting actions in principle); however, the domain Scenarios is not explicitly defined. The subscenario relation is not introduced explicitly, being considered only indirectly, as a consequence of the includes relation (below).

Use cases and relations. A goal *k* of a stakeholder H with respect to an entity A (considered as SuD and also scope) is identified as the reason for writing a use case UC^{A}_{k}. To emphasize that UC^{A}_{k} is focused on a goal of A, the stakeholder H is also called (ultimate) primary actor in UC^{A}_{k}. The description of communication of A with external entities is "white-box" based, i.e. all external actors of A are visible, regardless of how deep is A nested in the entity hierarchy forming Σ. On the contrary, it is strongly discouraged to specify any communication of A with internal entities in any UC^{A}_{k} (black box view recommended). No UM^{A} is introduced, but there is the concept of Σ (the computer system to be designed).

As illustrated in Sect.1.1, the structure of the textual descriptions is defined as template, providing generic guidelines as to how to specify the generated scenarios by means of main success scenario, its extensions, and variants, plus which fields should be included in the header of a use case (Stakeholder, Scope = SuD, etc.).

Addressing includes in principle, a sub use case relation is defined with the semantics UC^{A}_{j} "calls" UC^{A}_{k} where acyclic property is silently assumed and indirectly supported by introducing specific levels of nested use case calls (cloud, sea-level, underwater). Thus, the subscenario preservation is silently guarantied on the domain assumed for UC^{A}_{j}. Here, any top-most use case is also a *summary use case* – this roughly corresponds to the choice of use cases to form a UM^{A}. Moreover, inspired by UML, the extends relation is defined, based on the idea of ex post defined extension points in a use case: UC^{A}_{j} extends UC^{A}_{k} means that UC^{A}_{j} explicitly states which parts of UC^{A}_{k} are the extension points (could be described by a set of context rewriting rules). Generalization (again UML inspired) is not explicitly considered in the recommended template (using generalization is discouraged).

For UC^A_j, $Com(UC^A_j)$ is explicitly defined as the set of scenarios generated by UC^A_j. Com(A) is considered the collection of scenarios where A is involved as SuD (generated by a UC^A_j) – considering just summary use cases to be involved in behavior assembling is silently assumed.

Whole picture behavior. The idea of selecting use cases from U^A to form UM^A is reflected in the concept summary use case. In principle, the summary use case idea implies nesting and sequencing of use cases from U^A (via includes or extends) in the form described by a higher level use case from UM^A. On the other hand, the precondition and postconditions of a UC^A_i and UC^A_j from UM^A allow for partial ordering of the behaviors $Com(UC^A_i)$ and $Com(UC^A_j)$, including a parallel composition; here, preconditions can be interpreted as unary operations from UEop for which only basic use cases are allowed as operands. As the semantical spectrum of behavior composition via preconditions can be really broad, the original paper on use cases [12] assumes that no concurrency is modeled in a use case.

Addressing consistency issues. No compliant with, consent, nor $\sqcap$ are defined, therefore none of the key consistency issues is addressed (except for reasoning "by hand" on an intuitive basis).

2.3. Comparing Generic UC View and UML

Basic concepts. The UML use case package (a subpackage of the Behavior Elements package) is a very generic framework for specifying use cases. UML specifies the actor and use case as the central concepts. These corresponds to our entity A and UC^A concepts. The idea of communication links (connections) in UCview is reflected in UML very indirectly: the information about connections among the entities in a subsystem can be obtained only by a systematic walk-through of the use cases (for these entities) and recording their associations with actors – an use case of an entity (SuD) A contains (as an association) the info with which external entities (actors) A communicates. As to nesting of entities, a UML actor can model (not be) a system, subsystem, or class; in a hierarchical system, it is very hard to imagine capturing the communication among entities (their behavior) at such different abstraction levels under the circumstances that the info on their communication is so hard to get.

Scenarios. A scenario is a *use case instance* (no Scenarios introduced). As to the way of how a UC^A is actually specified, UML is very generic (p.2-142): *A use case can be described in plain text, using operations and methods together with attributes, in activity graphs, by a state machine, or by other behavior description techniques, such as preconditions and postconditions.* Even though UML considers nested entities, it does not bring an explicit scope concept.

Use cases and relations. UML is specific on the relations defined for U^A. It defines the include, extend, and generalize relations, but in a very generic way: As to include, it requires nothing more than *a use case contains the behavior defined in another use case*. We interpret this in such a way that include corresponds to our includes (and also that subscenario preservation is required by the UML include).

The extend relation and generalize relation are defined for use cases UC^A_i and UC^A_j, but in a very general way. While generalization allows for systematic modifications of scenarios (one could imagine employing rewriting rules for this purpose), extends allows only for insertion of several subscenarios into an existing scenario at its predefined extension points. But elsewhere it is argued that these relations are not well defined, e.g. [3, 7, 6]. In addition, there is the concept of superordinate and subordinate use case (below).

Whole picture behavior. There is no concept similar to $Com(UM^A)$, so it is not clear where "the generation of scenarios really starts". Using the terminology from Sect. 2.1, consider UC^A_i and UC^A_j from a U^A and assume UC^A_i include UC^A_j and $UC^A_i \in UM^A$; it is not clear whether $UC^A_j \in UM^A$, i.e. whether UC^A_j contributes also to $Com(UM^A)$ or produces just subscenarios employed in $Com(UC^A_i)$.

Consistency issues. The issues (a), (b) are addressed in only in a very general way by asking (p. 2-145): *Thus, if use cases are used for the specification part of the system package, these use cases are equivalent to those in the use-case model of the system; that is, they express the same behavior but possibly slightly differently structured.* and *Furthermore, if several models are used for modeling the realization of a system (for example, an analysis model and a design model), the set of use cases of all system packages and the use cases of the use-case model must be equivalent.*

In addition (a) is also, again in a very general way, addressed by the subordinate/superordinate relations upon UC_j^{Ai} and UC^S where A_i are in the scope S by asking all UC_j^{Ai} subordinate to UC^S to "cooperate" in a way described by a collaboration diagram. No compliant with, consent, nor $\sqcap$ are defined, therefore none of the key consistency issues is addressed. Also, no use case expressions (and thus no operations UEop) are considered.

3. Protocols Use Cases – Pro-cases

In Sect. 2, we saw what the benefits of the relations compliant with, consent, and the operations for creating use case expressions could be. This section presents a concrete instance of our Generic UC View – Protocol use cases (*pro-cases*) as a proof of the concept having all these features.

3.1. Basic Idea

In [20], behavior of an entity (called agent in [20]), is modeled as the set of traces – finite sequences of atomic events, capturing the communication on the entity's connections (both internal and external). A regular expression-like notation is used to approximate the actual behavior of entities by regular languages. These concepts are applied to a hierarchical component model [18, 22]. As the behavior is formally defined and there are powerful operations upon the protocols and behavior (languages), decidable relations and operations exist ($\sqcap$, compliant with, consent [1]) which allow to decide on compatibility of two components (their specifications). We show, that (and how) the behavior protocol concept fits into the generic UC view (a key idea is that a "use case" corresponds to a "protocol").

Basic concepts. An entity exchanges atomic events with other entities on its external connections. In case of an entity S composed of entities A_1, A_2, connections among A_1, A_2 are internal connections of S, events on these internal connections are internal events of S. Other external connections of A_1 and A_2 are external connections of S. S is the scope of both A_1 and A_2.

Scenarios. A scenario (called a *trace* in behavior protocols) is a finite sequence of atomic events. The events are denoted by event tokens from a domain ACT in [20]. For our purpose, we assume ACT* corresponds to the Scenarios domain. The subscenario relation is thus defined as a substring (therefore decidable). An event is modeled as an event token *a* either emitted *(!a),* absorbed *(?a)* or internally processed (τa). Com(A) $\subseteq$ ACT* denotes the behavior of entity A (a language upon ACT). (In [20], L(A) is used for the language; to be consistent with the generic UC view, we use Com(A) here).

The following example suggests (in a simplified form), how the scenario generated by a use case could be mapped to a trace of a behavior protocol.

<?sic.submitItem, τValidateItem, ?sic.submitPrice, τValidateSeller, τVerifySellerHistory, !tradecom.validate, τListOffer, !sellernotify.putAuthNr>

The trace corresponds to the main success scenario specification of the use case used in Sect. 1.1. Actions performed internally by SuD are represented as internal actions (τ), actions taken by an actor towards SuD are captured as events absorbed (by SuD, *?* used), actions taken by SuD towards an actor are captured as being emitted (*!*). The event token domain ACT contains names composed of a connection name (e.g., *sic*) and event name (*submitItem*).

Use cases and relations. A *pro-case* with an entity A as SuD (notation PE^A) is a behavior protocol $Prot^A$ approximating the behavior of A by bounding the behavior of A (see below).

Syntactically, a protocol $Prot^A$ is an expression generating a set of traces over ACT*, denoted Com($Prot^A$). The behavior protocol notation stems from the regular expressions notation. In addition to the basic regular expression operators (given in priority order): * (repetition), ; (sequencing), + (alternative), there is also parallel operator | (A|B means parallel execution of A and B) and || (A||B stands for A + B + A|B). Parallel execution is an arbitrary interleaving of the atomic events captured in a pair of traces generated by A and B. The composed operators (taking a set X $\subseteq$ ACT as a third operand) are:

composition A $\sqcap_X$ B (events from X emitted by A (*!a*) and absorbed by B (*?a*) or vice versa are replaced with internal events τa in the resulting trace; except for this explicit synchronization, the traces are arbitrarily interleaved);

adjustment A $|_X|$ B (similar to composition, but exact match (instead of *?*/*!* correspondence) of events from X is required) and

consent A $\triangledown_X$ B (similar to composition, but erroneous traces are included where the interaction of A and B might result into an error, e.g., *Bad Activity*, when A tries to emit *a* (via *!a*), but B is not ready to absorb the event – issue *?a*).

For the full definition, please refer to [20, 1]. It is important to note that even with the additional operators, the language generated by a protocol remains regular. Furthermore, the notation uses abstractions of specific sequences of atomic action (e.g., procedure call), which yield much higher "expressive power" with respect to readability of the protocols.

In particular, procedure calls are modeled as pairs of events; an operation *a* is modeled as a *request* ($a\uparrow$) and a *response* ($a\downarrow$). In the notation, *?a* (resp. *!a*) can be used as shortcut for $?a\uparrow$, $!a\downarrow$ (resp. $!a\uparrow$, $?a\downarrow$). Moreover, *?a*{*Prot*} (resp. *!a*{*Prot*}) can be used for $?a\uparrow$ *Prot* $!a\downarrow$ (resp. $!a\uparrow$ *Prot* $?a\downarrow$). This way, it can be expressed that a sequence of events occurs during a procedure call, while the higher level abstraction of the procedure call is still preserved. This significantly improves the readability of the notation. The events $a\uparrow$ and $a\downarrow$ are atomic, while the shortcut abstractions are non-atomic, embracing method call-like pairs of events.

As an example, we show a behavior protocol fragment corresponding to actions 1 and 2 of the sample use case used in the introduction (the extensions and variations pertaining to these lines are considered). The events corresponding to the main success scenario specification are printed in **bold**.

?sic.submitItem { τValidateItem ; (**Null** + τPriceAssessmentAvailable ; !sellernotify.putPriceAssessment + τInvalidItem) **}**

Here, the *?sic.submitItem{}* shortcut is used to reflect that the action step 2 is performed within (as a part of) the action step 1. The extensions and variations attached to the action step 2 are captured as alternatives (+). In the main success scenario specification, no additional processing is done after performing action 2 (Null). The condition of the extension (resp. variation) is expressed as an internal action (*τPriceAssessmentAvailable* and *τInvalidItem*); this represents the internal choice to be performed by the entity Marketplace Information System.

The language generated by this protocol fragment contains three traces:

1. *<?sic.submitItem↑, τValidateItem, !sic.submitItem↓ >*
2. *<?sic.submitItem↑, τValidateItem,*
   τPriceAssessmentAvailable,
   !sellernotify.putPriceAssessment↑,
   ?sellernotify.putPriceAssessment↓,
   !sic.submitItem↓ >
3. *<?sic.submitItem↑, τValidateItem, τInvalidItem,*
   !sic.submitItem↓ >

For a protocol Prot^A, $\text{Com}(\text{Prot}^A)$ denotes the set of traces generated by Prot^A. As $\text{Com}(\text{Prot}^A)$ is a regular language; but Com(A) is not in general, Prot^A only approximates behavior of A. In [20], the approximation is based on the *bounded behavior* relation, A is bounded by Prot^A, if Com(A) is compliant with $\text{Com}(\text{Prot}^A)$. As to the definition of compliance, roughly, Com(A) is compliant with $\text{Com}(\text{Prot}^A)$ on set $S \subseteq ACT$ (S divided into inputs S_{prov} and outputs S_{req}) if Com(A) can respond to any sequence of inputs dictated by $\text{Com}(\text{Prot}^A)$ and for such inputs, creates only outputs anticipated by $\text{Com}(\text{Prot}^A)$. Formally, via the adjustment operator: (i) $\text{Com}(\text{Prot}^A)/S_{prov} \subseteq \text{Com}(A)/S_{prov}$ and (ii) $\text{Com}(\text{Prot}^A)/S_{prov} |_{S_{prov}}| \text{Com}(A)/S \subseteq \text{Com}(\text{Prot}^A)/S$.

The includes relation is defined as an inclusion of behavior protocol specifications (similar to macro substitution, naturally acyclic); the subscenario preservation property is implied from the definition of pro-cases.

Whole picture behavior. Typically, only a single pro-case is used (as the representative of UM^A); such a pro-case is called the *frame pro-case* (inspired by *frame protocol* in SOFA [20, 18]). The basic and parallel operators used in the behavior protocols notation can be advantageously employed as the operations for use case expressions (the UEop set); thus assembling the behavior via use case expressions is natural here.

Addressing consistency issues. Even though the variety of the behavior protocol operators provides strong expressive power, the behavior (language) generated is a regular language. This significant advantage allows for comparing behavior described by behavior protocols, as, e.g., inclusion of regular languages is decidable. The composition operation $\sqcap_X$ conforms to the generic view of composition ($\sqcap$). The relations compliant with and consent are defined and are decidable, thus the consistency issues (a), (b) and (c) are addressed here.

3.2. Deriving Pro-cases from Textual Use Cases

In this section, we show a way to transform a textual use case model UM^A into a pro-case model PM^A (to emphasize the difference between the Generic UC view and an its instance Pro-cases, we write PC^A, PM^A, PE^A, and P^A instead of UC^A, UM^A, $\text{UE}^{A,}$ and U^A).

We illustrate the proposed transformation on the Marketplace textual use case model. Based on its scope diagram (Fig. 2), we identified the corresponding future software components and their interfaces (Fig. 3).

The basic idea is that the textual use case models of the entity Marketplace Information System (*M*) and of the nested entities Clerk (*CL*), Computer System (*CS*) and Supervisor (*SU*) will be transformed into pro-case models PM^M, PM^{CL}, PM^{CS} and PM^{SU} where, advantageously, the assembled behavior is expressed via the operations from PEop as a single protocol – a representative pro-case). Using the composition operation ($\sqcap_X$), the behavior of the nested entities (*CL*, *CS*, and *SU*) will be composed together and verified for consistency with the assembled behavior of the enclosing entity (*M*). For this purpose, a protocol verifier tool will be employed to evaluate the compliant with relation.

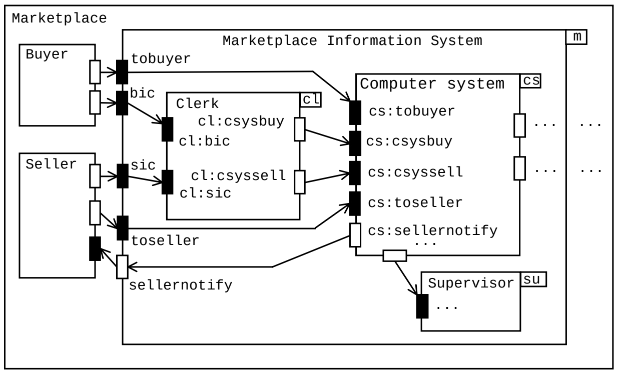

Figure 3

The stages of the whole transformation are:

(i) **Intermediate form**. Based on the textual description of each action (in an action step), we select an event token (roughly: future method) to represent the corresponding action, Event token can contain composed names to reflect connection names among entities and the action they represent. for example *?sic.submitPrice* is an action token meaning accepting event *submitPrice* on the connection *sic*.

Assuming the textual descriptions use a restricted (simple) form of English (e.g., the "Subject-Verb-Direct object(s)-Prepositions-Indirect object(s)" form (SVDPI) [7]), an action description has typically the form of a request issued by one entity and addressed to another one. We suggest using verb-noun phrases for the action symbols; the connection and the information whether the event is emitted or absorbed is typically obvious from the action step. For the use case "Seller submits an offer" (Sect 1.1), we get the following intermediate form.

Main success scenario specification:

1. ?sic.submitItem
2. τValidateItem
3. ?sic.submitPrice
4. τValidateSeller
5. τVerifySellerHistory
6. !tradecom.validate
7. τListOffer
8. !sellernotify.putAuthNr

Extensions:

```
2a τInvalidItem
    2a1 Null   //Abort
5a  τVerifyFailed
    5a1 Null   //Abort
6a τTradeComValidateFailed
     6a1 Null  //Abort
Sub-variations:
2b  τPriceAssessmentAvailable
    2b1 !sellernotify.putPriceAssessment
```

(ii) **Transforming the intermediate form into a pro-case**. Applying the sequencing operator **;** to action steps is the natural way to transform the main success scenario of an intermediate form into a protocol; however, it is important to identify whether an action actually spans the execution of several consecutive action step(s) – in such a case the *a{Prot}* shortcut is to be applied.

Extensions and variations are transformed in a similar way and inserted as alternatives (using +) at their respective positions; their condition is represented as an internal event (τ). Applying these guidelines to our example yields the following pro-case (walk-through of the main success scenario shown in **bold**):

?sic.submitItem { τValidateItem ; (NULL +
    τPriceAssessmentAvailable ;
    !sellernotify.putPriceAssessment + τInvalidItem) **}** ;
(**?sic.submitPrice { τValidateSeller** ;
        τVerifySellerHistory ; (**!tradecom.validate** ; (
        τListOffer ; **!sellernotify.putAuthNr** +
        τTradeComValidateFailed) + τVerifyFailed)
    } + τInvalidItem)

(iii) **Assembling behavior**. Based on the set P^A, i.e. several protocols similar to the one above, the pro-case model PM^A is constructed, by forming use case expressions via operations from PEop.

As an example, consider again UM^M. In addition to what we have presented so far, the whole U^M is to be considered (available in [15]); at this point, listing the names and numbers of the use cases in U^M will do: *#1 Seller submits an offer, #2 Buyer searches for an offer, #3 Buyer buys a selected item, #4 Seller cancels an offer, #5 Seller checks on the status of an offer, #6 Seller updates an offer, #7 Buyer makes a purchase*. In [15], it is easy to see that use case #7 includes use cases #2 and #3. Resulting from that, UM^M contains all these use cases except for #2 and #3, which generate only subscenarios of #7.

At this stage, we assume a PC^M_k has been constructed in the set P^M for each use case #*k* in U^M (we will refer to a pro-case by the number of the original use case as subscript). In our example, the use cases #1, #4, #5, #6 describe communication of the entity *M* with the same actor, therefore we do not assume any parallelism here and use the + (alternative) operator to assemble these use cases into a sub-expression (PE^M_1); this actually corresponds to joining their behavior (languages) with ∪ (union). As use case #7 has a different primary actor (the one initiating the generated scenarios), we regard this as source of a possible parallelism and we assemble this use case with PE^M_1 via the || (parallel-or) operator (which permits alternative execution, besides interleaved traces). To allow for arbitrary iteration, we suffix each of the operands of || with *.

Thus, we represent the behavior of *M* with a single use case expression

$$PE^M_R = (PC^M_1 + PC^M_4 + PC^M_5 + PC^M_6)\ \mathbf{*}\ \|\ (PC^M_7)\ \mathbf{*}$$

and then construct $PM^M = \{ PE^M_R \}$; obviously, PE^M_R is the frame pro-case of *M*.

(iv) *Compliance test*. Having transformed the textual use case models into pro-case models which have a representative, we apply the composition operator $\sqcap_X$ to acquire the composed behavior of the internals of *M*. With the protocol verifier tool [22], we check that the composed behavior of entities *CL*, *CS* and *SU* is compliant with Com(PM^M). This means we check $PE^{CL}_R \sqcap_{X1} PE^{CS}_R \sqcap_{X2} PE^{SU}_R$ compliant with PE^M_R; here X1 resp. X2 is the set of event tokens (roughly action names) on the connections between *CL* and *CS*, resp. *CS* and *SU*.

4. Evaluation and Related Work

Practical experience: (1) As a proof of the concepts, we (re)designed a part of a project featuring nested entities (Fig. 3); Marketplace Information System (*M*) yielded 7 use cases, while the internal three entities 2, 8 and 2 use cases. About 20 min were required to transform a textual use case (written using a unified template [3]) into a pro-case. To assemble the behavior into frame pro-cases, the time requirement was about 10 min for each of the four components. An important side-effect of this process had been realizing that one of the pro-cases of *M* can be added to its assembled behavior via parallel composition, which in principle means that we have added 8th (very complex, describing all potential trace interleavings) use case to the original behavior description of *M*. Compliance of the composition of the three internal frame pro-cases with the frame pro-case of *M* was done by the SOFA Protocol Verifier [22] (written in Java), finding three inconsistencies in the original specification.

(2) In a major group assignment, students taking their first software design course were instructed to transform their use cases into pro-cases. It took them only about 15 minutes to get the behavior protocol idea. Our observation had been that students had no difficulty creating pro-cases and actually very much preferred them over System Sequence Diagrams [14]. Furthermore, as the requirements were specified more precisely, the projects progressed more smoothly compared to a previous run of the course where classical data-driven design was used.

Related work: Most of the recent work in the use case field pertains to uses cases in UML. In [23], it is emphasized that the use case concept in UML has not reached the point where a tool could support the use of use cases without making major decisions concerning how to interpret the standard. It basically argues that use cases should be formalizable in a soundly-based, tool-supportable way, in order to relate use cases to the design of a respective system. One of the ways of formalizing use cases in UML is to formally interpret them as sequences of actions. Here, to avoid possible limitations imposed by choosing a concrete process algebra, a labeled transition system (LTS) is recommended; a high-level formalism is intentionally avoided. It is highlighted that UML state machine should not be used to realize an LTS, due to its extra power with respect to LTS and the correspondingly complex semantics. Compared to this, our approach in Generic UC View is to capture the generic properties of use cases (and the behavior they describe) in such a way that a variety of concrete instances (Concrete UC Views) can be defined later on, including a LTS (which our pro-cases in principle are), each of them featuring these generic properties.

In [24], Message Sequence Charts (MSC) are translated into an LTS which can by analyzed by model checking for deadlock, safety, and liveness properties. The concept of High-level MSCs (hMSCs) composed of basic MSCs (bMSCs) resembles our concept of use case expressions, however, only a single level of composition is allowed, moreover, the set of operations is weaker than in our approach (e.g., parallel composition, compliance and consent are not considered).

In [8], a use case is written in an abstract state machine language; compliance of use cases is not considered, as the focus is on generating tests for a particular use case. The authors of [25] performed a study of readability of formal requirements, evaluating how the presence of certain features in a notation influences readability of the notation. It is emphasized that readability is crucial for acceptance of formal methods by the industrial community; we believe that our pro-cases meet this requirement well. Cockburn [3] touches the problem of hierarchy of systems (SuD), but does not consider compliance among different levels. The recommended design methodologies in [14, 5] consider system sequence diagrams as a way to identify the interface of the system under design. Since in a sequence diagram, only a single scenario is captured, there would we be a significant number of diagrams to balance a single frame pro-case. An interesting approach is taken in [9] where a use case is represented as an activity diagram and for each action in it, a graph transformation rule is defined upon collaborations of the objects involved in the action. Conflicts/dependencies between actions in different use cases (UC^A_i and UC^A_k in our terminology) can be identified.

As Sect. 2.3 was devoted to comparison of UML use cases and our Generic UC View, we limit ourselves here to a comment to Sect. 2.11.4.2 of [17]: It requires an activity in a use case to be initiated by a message of an actor – via pro-cases, it can be easily shown that behavior composition $Com(UM^A) \sqcap Com(UM^B)$ of cooperating A and B would always yield an empty behavior if this UML assumption were really applied!

Discussion: While assembling the "whole picture behavior", additional variants (especially parallel combinations) of behavior can be identified as desirable. Naturally, capturing such aspects of the requirements is important for the design in general, but very hard to express in textual use cases (for instance, the idea of summary use cases as the means of behavior assembly is very limiting). On the contrary it can easily done if use case expressions with an appropriate set of operations UEop are available. When adopting such new variants, use case expressions also simplify the necessary corrections back in lower level use cases. Here, evaluation of compliance is inevitable.

Favoring readability, behavior protocols lack direct support for conditions, neither consider the data transferred (e.g., as method parameters). The key argument of the experimental study [25] is that readability is the most important property of a specification to be accepted by industrial community. Though textual use cases have a high level of readability, it is very hard to obtain the "whole picture" behavior in a textual use case and to reason on compliance (very important as the risk of ambiguities and inconsistencies is inherent to any textual specification). Wherever a textual specification is highly desirable for communication at whatever level of the project management, our recommendation it to create both textual specification and pro-cases which are easy to get and allow for reasoning on the compliance issues (a) - (d) articulated in Sect. 2.1.1.

5. SUMMARY AND FUTURE WORK

Meeting the goal (1) articulated in Sect.1.2, we presented Generic UC View on use case modeling. The message of this model is as follows: (i) For very basic modeling define a domain Scenarios. If an includes relation on use cases is desirable then define also a subscenario relation upon Scenarios as it helps define semantics on <u>includes</u> clearly. (ii) It has to be stated without any ambiguity how the scenarios of a use case UC^A contribute to scenarios in Com(A). Therefore the Generic UC View distinguishes the set U^A of all use cases of A and a use case model UM^A of A as the set of use cases where "the scenario generations start". (iv) The whole picture behavior of A (Com(A)) can be infinite, it is desirable to "see" it as a single use case – representative. This can be advantageously done when use case expressions are defined. (v) To reason on specification consistency of cooperating entities, in both nested an "sibling" position, Generic UC View articulates three relations/operations to be defined.

Addressing the goal (2) we compared the UML use case package. Here the bottom line is that UML does not clearly address (ii) above which in principle means that it is not clear what the whole picture behavior of an entity A really should be if there are more use cases written for A. Being very generic, UML does not consider reasoning on use case specifications.

As a proof of the concepts, protocol use cases (pro-cases), based on behavior protocols [20], have been introduced (addressing goal(3)), which support all the features (i) - (v) above. Moreover, as they are based on in principal regular languages, all the relations and operations introduces in Generic UC View are decidable. To summarize the benefits from creating a pro-case model: (I) With use case expressions, behavior can be assembled to form a single pro-case as the "whole picture" of an entity's behavior, in a readable and comprehendible notation; (II) Parallelism can be captured, creating a more precise specification; (III) Using the decidable compliant with relation, the consistency issues (a), (b), (c) can be addressed; (IV) Pro-cases can be helpful in early stage of design – showing the interactions of an entity, they can be useful when assigning responsibilities to classes, identifying operations, behavior (structure) of methods; thus, pro-cases can serve as a powerful replacement of system sequence diagrams [14].

In use cases, failure handling is a important issue. In the current state, pro-cases allow for expressing extensions as alternatives, with the condition captured as an internal event. In the case of nested failures, the readability of the pro-case can be negatively influenced. Our future work aims at enhancing behavior protocols with exceptions, to allow for a more elegant way to handle error conditions. Our intention is to preserve the current readability and lightweight nature of the notation; the key point is to avoid turning the notation into process equations. Moreover, erroneous traces are being investigated [1] to capture inconsistencies in composition.

Furthermore, we consider investigating the opportunity to employ a natural language parser to provide a level of automation while creating a pro-case model based on a textual use case model.

REFERENCES

[1] Adamek, J., Plasil, F.: Behavior Protocols Capturing Errors and Updates, Accepted for publication in proceedings of the Second International Workshop on Unanticipated Software Evolution, ETAPS, Warsaw, 2003

[2] Amyot, D., Mussbacher, G.: On the Extension of UML with Use Case Maps Concepts. UML 2000, York, UK, October 2-6, 2000, in Proceedings LNCS 1939, Springer 2000

[3] Cockburn, A.: Writing Effective Use Cases, Addison-Wesley Pub Co, ISBN: 0201702258, 1st edition, January 2000

[4] Cockburn, A.: Harnessing Convection Currents of Information, Invited Talk, OOPSLA 2001, October 14-18 2001, Tampa Convention Center, Tampa Bay, FL

[5] D'Souza, D. Components with Catalysis, www.catalysis.org, 2001

[6] Genova, G., Llorens, J., Quintana, V.: Digging into Use Case Relationships, UML 2002, Dresden, Germany, 2002

[7] Graham, I.: Object-Oriented Methods: Principles and Practice, Addison-Wesley Pub Co, ISBN: 020161913X, 3rd edition December 2000

[8] Grieskamp, W., Lepper, M., Schulte, W., Tillmann, N.: Testable Use Cases in the Abstract State Machine Language, APAQS'01, December 10 - 11, 2001, Hong Kong

[9] Hausmann, J. H., Hecke, R., Taentzer, G.: Detection of Conflicting Functional Requirements in a Use Case-Driven Approach, ICSE 2002, Orlando, Florida, USA, May 19-25, 2002

[10] Hurlbut R. R.: A Survey of Approaches For Describing and Formalizing Use Cases, Expertech, Ltd., Document: XPT-TR-97-03

[11] Jacobson, I., Christerson, M.: A Growing Consensus on Use Cases, JOOP 8(1): 15-19 (1995)

[12] Jacobson, I.: Formalizing Use-Case Modeling., JOOP 8(3): 10-14 (1995)

[13] Jacobson, I., Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley Pub Co; ISBN: 0201544350; 1st edition (June 30, 1992)

[14] Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, Prentice Hall PTR, ISBN: 0130925691, 2nd ed, 2001

[15] Plasil, F., Mencl, V.: Use Cases: Assembling "Whole Picture Behavior", TR 02/11, Dept. of Computer Science, University of New Hampshire, Durham, 2002

[16] Nierstrasz, O., Arévalo, G., Ducasse, S., Wuyts, R., Black, A., Müller, P., Zeidler, C., Genssler, T., Born, R. van den: A Component Model for Field Devices. IFIP/ACM Conference on Component Deployment, Berlin, Germany, June 2002

[17] OMG: Unified Modeling Language (UML), version 1.4, formal/2001-09-67, http://www.omg.org/uml/

[18] Plasil, F., Balek, D., Janecek, R.: SOFA/DCUP Architecture for Component Trading and Dynamic Updating, Proceedings of the ICCDS '98, Annapolis, IEEE Computer Soc., 1998

[19] Plasil, F., Visnovsky, S., Besta, M.: Bounding Behavior via Protocols, Proceedings of TOOLS USA '99, Santa Barbara, CA, Aug. 1999.

[20] Plasil F., Visnovsky, S.: Behavior Protocols for Software Components. Transactions on Software Engineering, IEEE, vol 28, no 11, Nov 2002

[21] Schneider, G., Winters, J. P.: Applying Use Cases: A Practical Guide, Addison-Wesley Pub Co, ISBN: 0201708531, 2nd edition, March 2001

[22] SOFA Behavior Protocol Verifier, the SOFA project, http://nenya.ms.mff.cuni.cz/projects/sofa/tools/

[23] Stevens, P.: On Use Cases and Their Relationships in the Unified Modelling Language in Proceedings, FASE 2001 (Part of ETAPS 2001), Genova, Italy April 2-6, 2001, Springer LNCS 2029, ISBN 3-540-41863-6

[24] Uchitel, S., Kramer, J.: A Workbench for Synthesising Behaviour Models from Scenarios, ICSE 2001, 12-19 May 2001, Toronto, Ontario, Canada

[25] Zimmerman, M. K., Lundqvist, K., Leveson, N.: Investigating the Readability of State-Based Formal Requirements Specification Languages, ICSE 2002, Orlando, Florida, USA, May 19-25, 2002

Appendix A: Frame pro-case expanded

To illustrate the semantics of use case expressions, we provide here a fully fledged frame pro-case acquired by assembling the pro-cases of system *M* using the use case expression PE^{M}_{R} as shown in Sect 3.2.

(
?sic.submitItem { τValidateItem ; (NULL +
τPriceAssessmentAvailable **;**
!sellernotify.putPriceAssessment +
τInvalidItem) **} ;**
(**?sic.submitPrice { τValidateSeller ;**
τVerifySellerHistory ;
(**!tradecom.validate ;** (
τListOffer ;
!sellernotify.putAuthNr +
τTradeComValidateFailed) +
τVerifyFailed)
} + τInvalidItem)
+
?toseller.locateOffer ; ?toseller.requestCancelOffer
{ !sellernotify.getAck ; τValidateAck ;
(τInvalidAck **;** !sr.provideAck **;** tValidateAck) * **;**
(**τRemoveOffer** + τInvalidAck)
}
+
?toseller.locateOffer ; ?toseller.requestStatus
{ !sellernotify.getAck ; τValidateAck ;
(τInvalidAck **;** !sr.provideAck **;** tValidateAck) * **;**
(**τGetOfferStatus** + τInvalidAck)
}
+
?toseller.locateOffer ; ?toseller.updateOffer
{ !sellernotify.getAck ; τValidateAck ;
(τInvalidAck **;** !sr.provideAck **;** tValidateAck) * **;**
(**τUpdateOffer** + τInvalidAck)
}
)*****
||
(
?tobuyer.search ; ?tobuyer.narrowSearch* **;**
?tobuyer.requestDetails ; (**?bic.initBuyOffer {**
τValidateOffer + τOfferInvalid **} ;**
(**?bic.doBuyOffer { !agency.validate ;**
τPerformSale ;
!sellernotif.shipItem ;
τTransferPayment } +
τOfferInvalid) + NULL)
)*****