A State Transition Model for the HL7 Technical Committee on Orders and Results

Gunther Schadow
Regenstrief Institute for Health Care, Indianapolis

Id: order-states.tex,v 1.8 1998/09/29 21:01:31 schadow Exp

Abstract

In this document I propose a state-transition model for the HL7 technical committee on orders and results. Since the San Francisco Meeting in 1997 I took part in this exciting discussion about this model. However, we did never have enough time in the meetings to dive deep into the model and to finish a draft that would be near complete. Our discussions where rather on a very high level. We therefore agreed to put out a "strawdog" proposal that would be more complete to speed up the process and focus the discussion on important details.

I divided this document into three main parts: First I summarize the scope of the state transition model, because a detailed state-transition model can only be worked out for a specific class of the information model. Second I give a summary on state-transition modeling and explain the features and "language" that I use. Third, and finally, I present the model for the class *service intent or order* and its adjactent "relationship" class.

Contents

List of Figures

1 What we are talking about

State transition modeling in the UML pertains to a specific class. Every class should have its own state-transition model. In the previous meetings of the order/result working group we did not decide to which class our model pertains. This was due to the confusion we had about the meaning of the class *patient service order* in contrast to the class *patient service event*. So we made our model, knowing that it pertained to an order, but also to the ordered service, to the service in process up to its fulfillment.

There has always been this tension in the term "order" as used in clinical information systems in general and HL7 in particular. We had "order entry systems" that were far more than just a program accepting user-*entry*, filling out an (electronic) order form, and be done with it. Order entry systems are rather sophisticated pieces of software that help doctors investigate the patient chart and make decisions of all kinds, some of them ordered to nurses or other departments (e.g., lab, pharmacy), some of them reflecting their plans for the patient, executed by themselves or their collegues. The term "order" was always used in a much broader sense then just ordering a service to someone else. The "order" thus became a well understood metaphor for our committee, covering all kinds of clinical information and processes.

Indeed, in the beginning, the order/results committee was the only one in HL7 to contribute clinical contents. This was true up to HL7 version 2.2. In this first period of HL7, this committee did a good job. It invented innovative concepts for the HL7 standard that proved to be very up to date, even 10 years later! These concepts include:

1. The ORC segment was an abstract concept for the order, independent from its special content, in turn known as the *order detail*. There are order detail segments (and groups) for test orders (OBR), dietary orders (ODS, ODT), supply orders (RQD), and medication/treatment (RXO). This structure was a piece of object oriented design long before this became a buzzword: The ORC by itself represents the abstract order class. The combinations of ORC with one order detail OBR, ODS-ODT, RQD or RXO represents concrete classes as specializations of the abstract order class. This is a perfect gen/spec relationship, inheritance hierarchy, or what other words we now have for this key concept of object-oriented design.

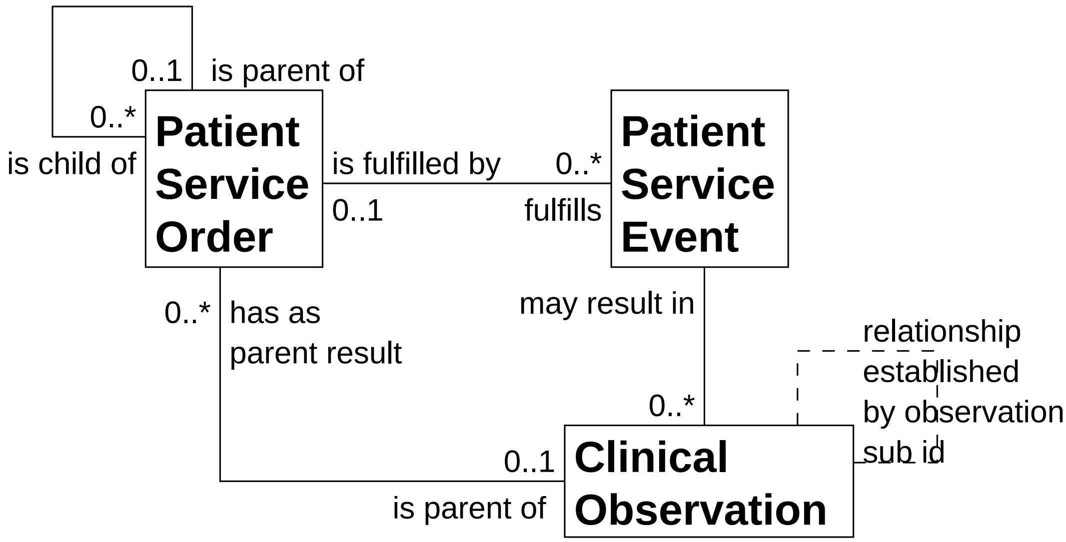

Figure 1: Class diagram showing the order/results classes in their RIM 0.84 form.

2. The order was designed to be the subject of a whole conversation rather than just a single message exchange. The order status code along with the order control code have implemented a state-transition model, where order status codes were the states and order control codes where the transitions. This is nothing else than a *life-cycle*, another key concept of the object oriented method.

3. The order, of course, needed an identity. Anything that has a life cycle needs an identity. The identity remains stable whatever attributes of the object might change over time. But just how this identity was expressed was the innovative step: with a *pair* of identifiers, one for each of the two communicating parties. The order/results authors knew better than suggesting just another "universal" identifier like everyone else is quick to suggest today—a suggestion that is very hard to be implemented in practice!

4. The parties communicating about the order could take on one of a pair of complementary *roles*: placer and filler. Thus independent from their absolute identity, they would know each other as the *placer* or the *filler* of a particular order.

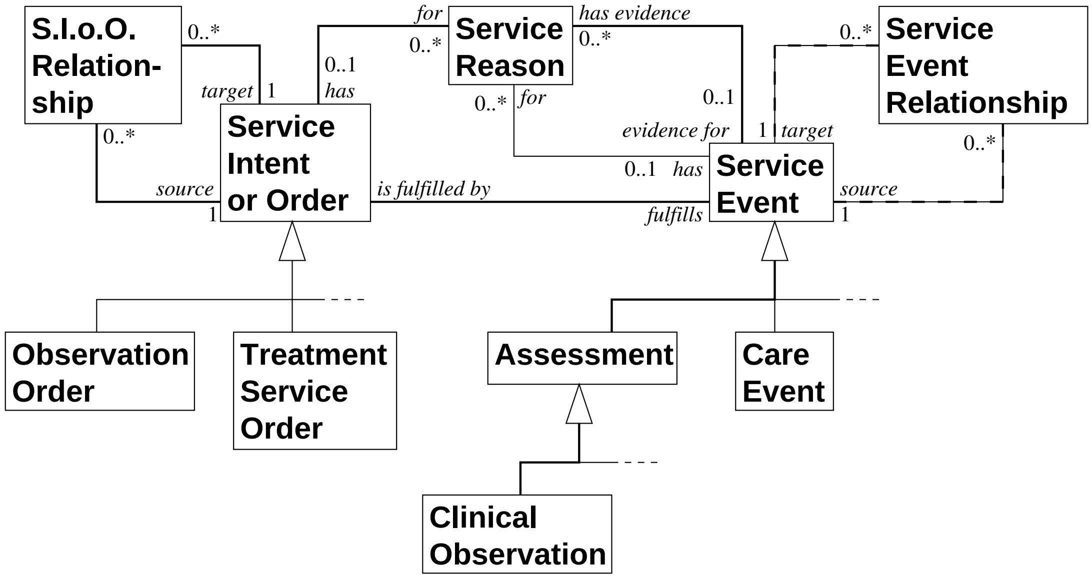

Figure 2: Class diagram showing the order/results classes in the recently updated RIM.

5. The hierarchical organization of order instances through the parent/child relationship.

6. The OBX segment as a general name-value-pair of any observation was with HL7 from its version 1 (under a different name, though) long before this way of expressing information became a market hype.

These innovative features of the order/results chapters made it into the RIM and recently have attracted interest from other technical committees. The Universal Service Action Model Proposal (USAMP) that was issued jointly by Order/Results and Patient Care had a big impact on the RIM. Basically all clinical areas of the RIM do now use the overall strcuture created by Order/Results. This was a very important development, because the whole RIM is now much better structured.

The impact that Order/Results had on the RIM puts a responsibility on us to care for other technical committees as well. The class *patient service order* was renamed to class *service intent or order* to make clear that by an "order" we really mean much more: any clinical service from its initial intention or plan, up to its fulfillment involving one to many provider parties. Since this was always true for our committee, there is really no fundamental change here. The name had to be changed though, to make the scope of this class clear to other committees as well.

However, there are other changes, too many to account for all of them in this overview. Figures 1 and 2 show a comparison of the RIM 0.84 with the RIM as it came out of the Boston harmonization meeting.[1]

Most important for our state-transition modeling is that the difference between the class *service intent or order* and the class *service event* is now much clearer. The class *service intent or order* covers the service along its way from planning and ordering, over all steps of execution (including scheduling), to its fulfillment. The involved parties may discuss and change the instances of the class *service intent or order* up to the last minute. Conversely, the class *service event* is static, and holds only fulfilled (or otherwise terminated) services and their results. There is nothing to discuss or change about objects of the class *service event* because those do only relflect what has been done, after it is done. Thus it is clear that our state transition model is about the life-cycle of instances of the class *service intent or order*.

A second noteworthy change is that the class *service intent or order relationship* takes on all the structuring of services into subservices including the parent/child-relationship and reflex orders.

The different types of relationships are accounted for in the attribute named "relationship type code." But the relationship class can do much more: it can specify the timing and repeating of each subservice depending on the service it is included in. The role names "source" and "target" seem to be a bit ambiguous. The confusion can be resolved by thinking of the relationship class as an arrow, originating in the subservice (source) and aiming in its target, the super-service, i.e. the service that includes the subservice.

For the parent/child type of relationship the arrow point "up" the hierarchy of inclusion. Thus an arrow pointing from A to B can be read as "A is used in B". The direction of the arrow needs to be defined for each relationship type. It should always point upwards in hierarchies and towards the purpose of a service.

The metaphor of the arrow also helps to understand the multiplicities: every arrow has exactly two ends, thus it links exactly two service intents, one source,

[1]Unfortunately the people in Boston achieved a little bit less than expected, we will have to follow up on this at the upcoming San Diego meeting. The basic ideas, however, are set.

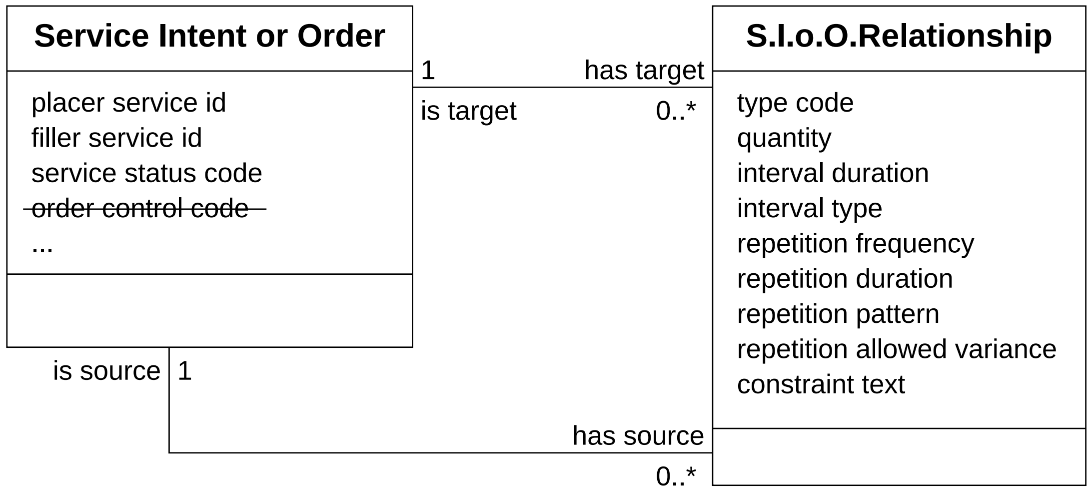

Figure 3: Class diagram showing the subjects of our state-transition-modeling

one target. But there may be zero to many such arrows linking services together.

The same relationship construct now occurs three times: in the master file area, the service intent or order, and at the service event. When a complex master service is ordered a whole bunch of instances of the class *service intent or order* and their relationships is created. Those have the same structure as the ordered master service. This has the advantage that the ordering (or “intending”) healthcare practitioner can take a master service as a default and modify it as necessary to fit the given situation. Of course, if no changes are required and the compound master service is understood by placer and filler, a single service intent is sufficient.

The class *service event relationship* maintains the composition struture of the services for the record. Repeated services are unrolled, because what is interesting is mainly what services actually occured at what time. Knowing the number of times a service *should* have been performed is of secondary relevance.

2 State-Transition-Modeling

The proposed state-transition-model is shown in figure 7 (p. 12). It is admittedly much more complex than the 5-states or 3-states diagrams we were discussing about in the last three meetings. This is because I tried to no longer delay the

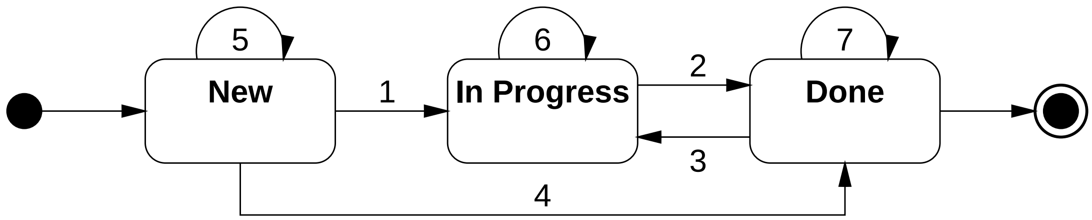

Figure 4: State-transition-model as discussed in the Baltimore meeting.

questions of what happens inside those high level states.

Before I discuss the details of this model I want to make sure we all have a common understanding about state transition modeling. Our models and the examples from the MDF were only using a very limited set of the features defined by the UML's state-transition-model notation,[2] although we all understand that there is something going on "inside" the states New, In Progress, and Done as described in the minutes of the Baltimore meeting and reproduced in figure 4.

Figure 4 shows the basic features of state transition modeling. States are rounded boxes and transitions are arrows. Transitions (normally) link two states. Their semantics is that if an object is in state New it may at some point in time go into state In Progress. But it is undefined in the given model at what time the state is changed. In UML there are "events" associated to transitions. When an event occurs, the associated transition is triggered, it "fires."

The figure also shows one fundamental arrangement of states and transitions: a sequence. Although there are branches that allow a transition not only from New over In Progress to Done but also directly from New to Done, and although there is a step back from Done to In Progress any object for which this state-transition-model holds will be in only one state at a time. Therefore, looking back on the life-cycle history, there will be one enumerable sequence of states the object went through, so that we can call this arrangement *sequential*.

Figure 5, in contrast, allows the object to be in more than one state at the same time. The two states New and In Progress are now enclosed by a state Not Done. This means that whenever the object is in one of those two substates it is also in the enclosing super-state. When a transition fires that heads directly to a substate (e.g., 3), the substate and the super-state are both entered at the

[2]See the *UML Notation Guide* available under http://www.rational.com/uml/documentation.html.

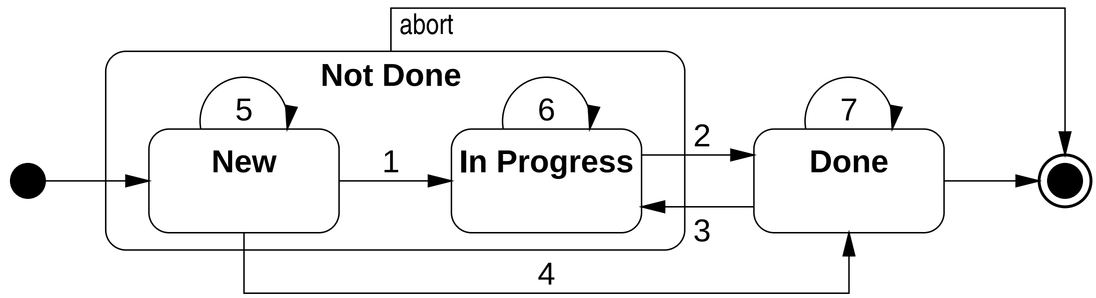

Figure 5: State-transition-model with nested states and preemption.

same time. Likewise does a transition that links from an inner state to an outer state (2 and 4) cause both the inner state and its super-state to be left at the same time.

But this example is not that different from figure 4 above: we can still write down the course of an object through its life-cycle as a sequence of states. And the "normal" course of the object would be the same as in figure 4. So far discussed, the only difference is that we subsume the states New and In Progress as Not Done.

The abort transition in the example shows the ability to leave all substates in Not Done and head to the final pseudo-state preemptively. Regardless of which of those two substate the object is in it will be left as the abort transition fires.

The life-cycle of objects complying to figure 6 is no longer a sequence of states: the state Held can be entered and exited independently from the other two states within their common super-state Not Done. It is a *concurrent* state. In UML, concurrent states are depicted by "tiling" of a common enclosing state, i.e., by dividing the super state using dashed lines into concurrent regions. The transitions of the concurrent regions within the "tiled" super-state can occur independently. Transitions that cross the dashed line may synchronize the otherwise concurrent processes. Without synchronization, there is no notion of "same time", "before" or "after" between the concurrent regions.

However independent transitions within different concurrent regions may fire, the regions are not completely independent. All concurrent regions within the same super-state are co-dependent on that super-state. When the object in figure 6 leaves the state Not Done either through its normal transition (2) or through the preemptive transition abort all internal states are left at the same time. Thus,

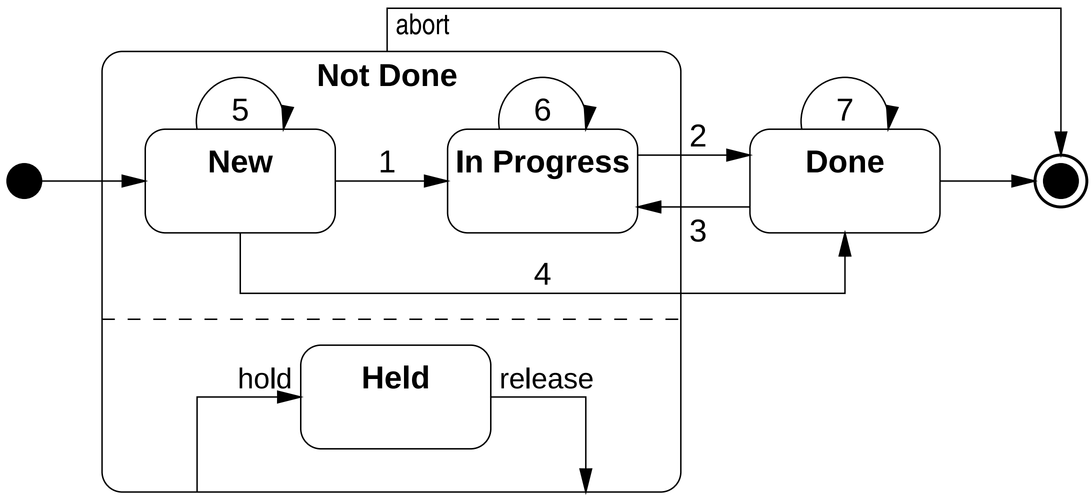

Figure 6: State-transition-model with parallel states

if the object happened to be in state Held when the abort transition fires, the state Held is left as well as all the other inner states of Not Done.

Such concurrent inner state machines can have initial and final pseudo-states (shown through black bullets) as well. This means that whenever the enclosing state is entered the initial transition of all concurrent state-machines fires. However, from that point on the concurrent state machines run in parallel without any implicit synchronization.

Note also the special configuration of the state Held and its adjactent transitions hold and release with respect to the state Not Done. The transitions link the super-state with one of its sub-states. The semantics of this configuration is implicitely defined by the general rules of our state-transition modeling: (1) The state Held can be entered only if the object is in state Not Done before and transition hold fires. But (2) the super-state Not Done is not left because the new state Hold is one of its substates.

3 Service Intent or Order

Figure 7 shows the state-transition diagram of the general class *service intent or order*. In this model, all of the features of state-transition modeling discussed above are applied. This diagram seems different from the example state-

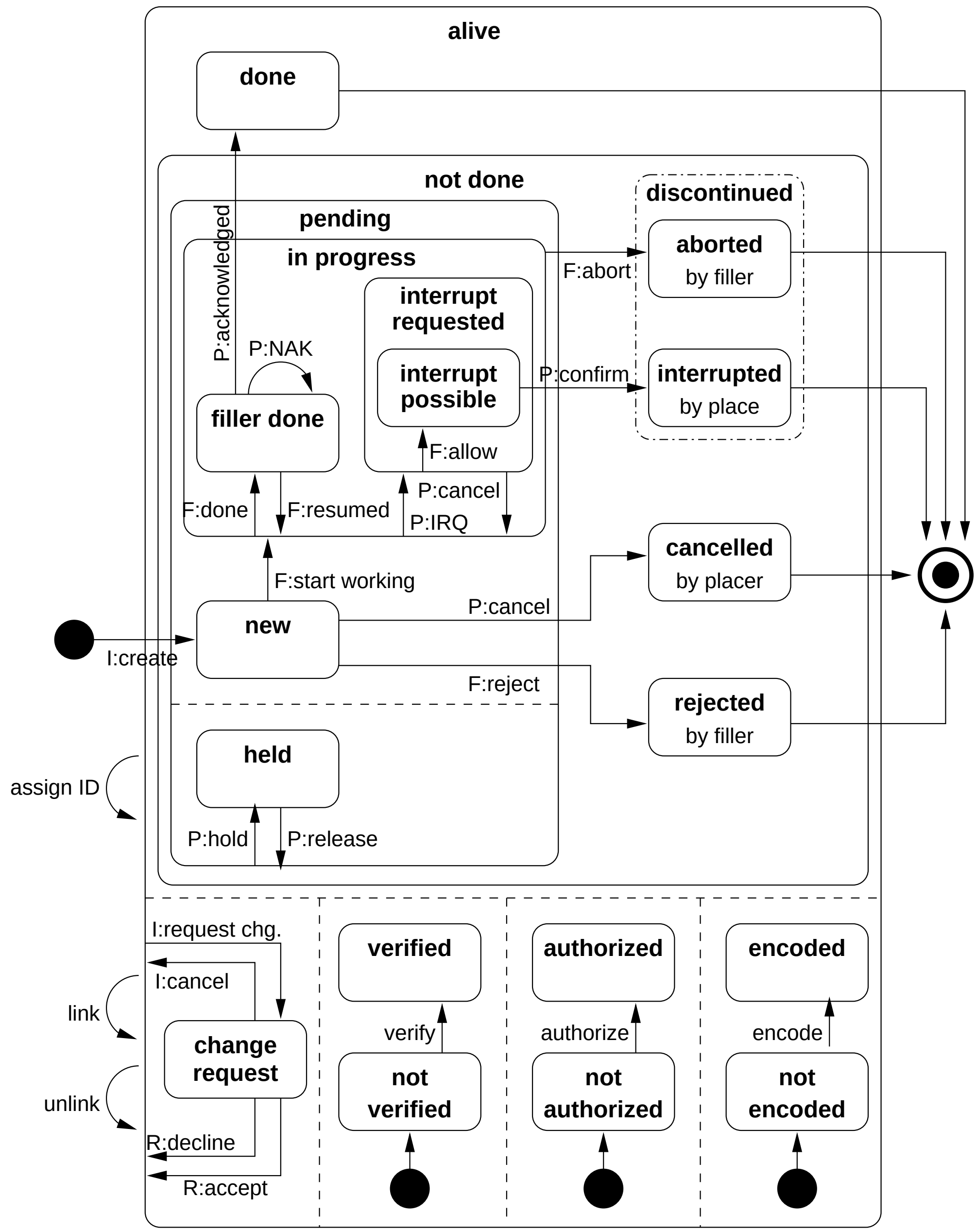

Figure 7: Proposed state-transition-model of the class *service intent or order*

transition diagrams given in the UML Notation Guide or the MDF. But these differences are due to the very nature of our subject, which is not just an example, not just a piece of program design, but an abstract and very dynamic subject of real life, the service intent or order.

Obviously there are many nested states, almost like the layers of an onion. In the following discussion we will slowly peel the onion.

3.1 Peeling the Onion

3.1.1 Alive vs. Non-Existent

Compared with the usual UML examples, this outer layer seems quite unusual: there is only one outer state alive aside from the initial and final pseudo-states. Thus a very rough sketch of the model would just list the initial pseudo-state, the state alive and the final pseudo-state arranged in a sequence. That an object is either "alive" or "non-existent" seems trivial and not worth to be shown in the model, would it not be for two reasons:

1. There are transactions to objects of this class that cause them to change in ways not shown in the model.

2. There are concurrent regions of the main state-transition model of this class, which, in the UML, requires a common enclosing super-state.

Notably the assignment of identifiers (placer/filler order number) by the transition assign ID and the linking of this service to other services by the transitions link and unlink cause the object to change significantly. These transactions are worth noting in the model, although their effect is out of the focus of this state model.

What *is* in the focus of this model has been developed in the figures 4 through 6 in the previous section. When ignoring all the detail, the basic structure developed in the previous examples can be recognized. You can see (through the layers of onion skin) the three states, new, in progress, and done with new and in progress having a common super-state not done. Of course, the main path of transitions between those states is the same as in the examples before. The concurrent state held somewhere in the state not done has already been introduced too.

3.1.2 Done vs. Not-Done

After peeling off the state alive we find the state not done. This state is in opposition only to the state done. A service can be completed or not completed. A service will reach its state done only if it was not discontinued or otherwise stopped before. If an unfinished service is discontinued, it doesn't reach its state done but remains in state not done. I will discuss the modes of stopping a service in state not done further below. Let's peel off this layer and look at the next one.

3.1.3 Pending vs. Non-Pending

A service is "pending" if, negatively spoken, it is neither completed nor canceled nor discontinued in any way. Positively, it is pending if it is new and waiting to be worked on, or if it is currently being worked on. While a service is pending it can be temporarily put into state held, i.e., interrupted for a while, with the option to release it and let the work on it continue. Thus, the ability to hold a service is in addition to any other states that the pending service has and does not affect these other states.

To put a service on hold that is done, cancelled or discontinued, obviously makes no sense. Hence the state pending is due to the state held to be applicable both to new services and such that are already in progress.

3.1.4 Doing the Work: In Process

If we peel off this last skin, we come to the core of our model. Interestingly, the distinction between two states like new and in progress is suggested already by HL7 version 2. There, we had the two quadruples of order control codes "cancel order" (CA), "order canceled as requested" (CR), "unable to cancel" (UC) and "order canceled" (unsolicited, OC) versus "discontinue" (DC), "discontinued as requested" (DR), "unable to discontinue" (DU), and "order discontinued" (unsolicited, OD). The existence of two different ways of ending an ordered service prematurely suggests two different major states the order can be in.

It seems like the order can be only discontinued if it is already "continuing", i.e. in progress. On the other hand, an order in progress cannot just be cancelled. There may have been considerable resources spent on the service already (e.g., work or costly reagents like antibodies, etc.) thus you cannot just cancel such a service and forget about it, especially you cannot expect that you won't get billed for the filler's expenses on behalf of your cancelled service request. However, in the discussion of the working group we found that it is neither clear what causes an order to enter the un-cancel-able state, nor is it clear what consequences the discontinuation has (e.g., whether you get billed or not). Cases can be made that an order may be considered cancel-able but still may generate a bill (e.g., an expert already worked on it and made recommendations).

We should simplify these issues as much as possible. I believe that the criterion of work and expenses that have been spent on a service is quite solid to justify two different states. Also I do not remember any other competitive criterion having been suggested in the working group discussions. With this premise, it is obvious that only the filler can decide at what point in time it will transit into the state in progress. This transition should be notified to an (interested) placer though a message.

If the placer is informed about the time-out for his ability to cancel the order by a message from the filler, the question arises, what happens if the placer just sent off a cancel request the second before the transition notification arrives? In some circumstances it is useful to have some time in which the placer is granted the right to cancel an order. For instance, customer protection laws in many countries grant those rights. It highly depends on the kind of service whether such a right is applicable or not. In laboratory services such a right certainly makes not much sense because we are happy if our orders are processed ASAP and not delayed until some cancelation period has expired. Trading partner agreements or conformance claims must be in place to specify how long such a period to cancel may be for each orderable service. During this period, the order is nominally in state new.

Of course, an order can in many situations be rejected by the filler. For example, if the filler does not provide some service ordered by the placer. Other reasons for rejection might be that the order failed some verification or authorization. See the section 3.4 below on these issues.

3.2 Exceptional Termination

3.2.1 Terminate New Services: Cancel vs. Reject

Cancelling and rejection out of the state new are simple. The filler transits to the state rejected and notifies the placer about this transition. Conversely, the placer can transit to state cancelled and notify the filler about this. No discussion is possible. The choice of either side to do the transition is definitive. After cancelling or rejecting, the order or service intent is no longer in state pending. This means, no further transition is defined other than being purged by transition to the final pseudo-state. The model also shows that a cancelled of rejected order is left in state not done.

3.2.2 Terminate Services in Progress: Abort vs. Interrupt

Terminating an order (or service intent) in progress is not so simple. The rationale here is the same as for distinguishing cancel versus discontinue: resources have been spent. But it's not just the bill that defines the difference. A service might be in a critical condition, where interruption would be fatal. For example, an ongoing abdominal surgery will have to be continued as the performing surgeon (in the role of the filler) decides. A discontinuation would leave the patient with the usual risks post abdominal surgery without the benefit of an improvement. Of course, the operation may be ended prematurely (e.g., if an inoperable tumor has been found), but not without result (e.g., anus praeternaturalis), and not because of some request by the placer. The important point is that terminating an ongoing service can involve a complex decision or discussion between placer and filler. The details are so versatile, they cannot be regulated in a general state-transition model for all services.

The model outlines two general mechanisms to handle these situations. Both end in a state discontinued. I have shown this state with a dash-dotted line to indicate that this is the discontinuation of HL7 version 2, where both placer and filler could initiate a discontinuation. However, for the new model I gave different names to each of the two modes of terminating a pending service prematurely: (filler) abort and (placer) interrupt.[3]

The filler can abort the service in progress upon notice to the placer. This right is granted to the filler because it is the filler, who is ultimately responsible for his actions. There may be rules to prevent the filler-abort transition for certain services and conditions, but generally the filler can deny further work on the service, which is reflected by the abort mode of termination.

[3]I borrowed this terminology from the ANSI X3.28 protocol, which I had to implement a couple of years ago. ANSI X3.28 has an abort initiated by the current sender, and an interrupt, initiated by the current receiver. Although ANSI X3.28 is about lower level communication, the design pattern is applicable to high level services as well.

The placer, on the other hand, can only *request* an interruption of the service. Upon notification of an interrupt request (IRQ) by the placer, both placer and filler enter the state interrupt requested. This state is still within the state in progress, which means that the service continues. The filler decides at what point the procedure allows interruption. Note that the same is true for the hold request: the nature of the service, and ultimately the filler, determines at what point the ongoing procedure can be interrupted.

Once an interrupt request is issued, and once the process reached a point where it can be stopped, the filler transits into state interrupt possible and notifies the placer about this so that he can follow into the same state. At this point the placer can decide whether he wants to confirm the termination of the service or if he wants to cancel the interrupt and let the service continue. If the placer reconfirms the interrupt, the service or order will the be left undone in non-pending state interrupted until purged.

This interrupt protocol implements a kind of a two-phase commit. It is designed to provide the important functionality, yet be minimalistic with regards to states and transitions. Note that the "discussion" between placer and filler can be refined *ad infinitum* by inventing new substates and cancel transitions from partial interrupts. However, it is not desireable to blow up the state-transition diagram with states and transitions. In a similar sense it is not desireable to have an inverse for every transition. There must be definitive decisions, points of no return: a discontinued service is discontinued forever, a cancelled service is cancelled forever, and a rejected service is rejected forever.

The two-phase commit protocol for interruption and completion assures that a definitive decision is not made "light heartedly" but agreed upon by both placer and filler.

3.3 Normal Termination

The transition to the state done involves a similar interaction between filler and placer. The filler can indicate that he considers his service completed and transit to the state filler done. However, the placer need not agree that this is truly so. Usually, the filler will hand a piece of evidence to the placer, to show that the service is really completed (e.g., results of a lab test, the goods of a purchase) and usually the placer will accept this evidence. When the placer acknowledges the assertion that the service is done, both placer and filler will transit to the state done.

If the placer does not agree, it will respond with a negative acknowledgement (NAK). The placer cannot force the filler to continue working on the service, but the filler might find the disagreement of the placer justified. The filler might also realize before the placer that the service is in fact not yet completed. In both cases the filler will exit the state filler done and notify the placer.

This is another kind of commit protocol. We cannot just reuse the interrupt design pattern, because the roles are different. It is the filler, who did the work, and who now claims his work to be completed. The placer, however, should explicitly commit the transition to done by his acknowledgement. When the placer sends his acknowledgement he also accepts the obligations that the completed service presents to himself: the bill. I think that HL7 communication should be designed to be legally binding and definitive. Such commit protocols, along with digital signatures, can be made court-proof, i.e. can have the same legal dignity as signed contracts have in the paper world.

The question arises, what happens if the placer responds with a negative acknowledgement to the declaration of completion by the filler. In theory, the service can be in the state filler done for an indefinite time, until other instances (e.g., humans using telehones) have clarified the issue. As a last resort, the filler can choose to abort the service. However, this leaves the service in the state not done which may have different consequences for billing, quality control etc.

In the model discussed at the meetings and shown in figure 4, there was a transition leading from the state done back to the state in progress and even to new (which was called "renew"). I did not admit these transitions into the model for the reasons discussed above: The model should be concise, i.e., complete but minimal. The model should also show a definitive flow of the life-cycle towards completion or other termination of the service. I am in favor of definitive decisions for service objects, "points of no return." One important argument for irreversible transitions is the ease of implementation: programs must be able to terminate somehow, and the more loops are in a program, the more there is a risk for non-termination.

To specify a model for things of the real world certainly is a challenge because reality is always slightly more difficult than anticipated. Of course we can resign upfront and construct use cases for missing transitions between any two arbitrary states. However, a model that links any two states by a pair of transitions, heading into either direction, is of little power. It does not sort out anything, it does not tell how things should work. It does not understand reality, but resignates before the real world's complexity. A good model does not only describe the real world, it also defines it. It does not only show what *does* happens, it also tells what *should* happen. It should tell the normal from the exceptional.

For a similar reason, I did not admit the transition 4 of figure 4 that led directly from the state new to the state done. A service that never goes through a state in progress, i.e. that is never really worked on, is not a real service. It may be that the real service (e.g., Appendectomy) has already been performed before. This, however, does not shortcut the new service towards the state done but should lead to a rejection of this additional service request (e.g., hopefully before the laparotomy).

This shows that there is some healthy "loose coupling" between what is ordered and intended in terms of HL7, and what goes on in reality. If an HL7 service is completed but we want it to continue, why not issue an additional service request? This additional request would be just a copy of the first one, with updated placer order number, time stamps, etc. Why do we need a "renew" transition for this? Even if we define the "renew" transition from the state done to new, we have to answer the following question: At what point is the filler allowed to purge the service intent or order object? When is it really dead? When can the resources (i.e., storage) be cleared? When can we finally forget about an old service?

We also discussed about if there is anything in the description of the service that would determine in general when the service is completed. What needs to happen for the service to proceed to the state done? It seemed like a considerable group of working group attendees held that a "stop date/time" attribute of a service would be the ultimate determinant for completion of a service. Consequently this group of people would argue that changing the stop date time (even after the original stop date time) would restore the order into the state in progress.

I do not agree to this position. While it is true that the end of some services may be defined in terms of time, this is (1) not possible for all services and (2) there are other alternatives in most cases. E.g. in surgery or in supply orders (purchase order) we do not care so much about when a service is completed but we do care that it should be completed in full. Hence the criteria for completion of those servces are not defined in terms of time. In radiotherapy we want to achieve a cummulative dose of radiation on the tumor, this is our goal, whether we apply single doses or, as usual, repeated fractionated doses. Again, a "stop date/time" is not the primary goal that defines completion of the service.

In pharmacy it is similar, although examples for the "stop date/time" argument are often taken from pharmacy orders. When we prescribe "Amoxicillin 250 mg three times a day for 10 days," the goal is to achieve a sufficiently high concentration of the antibiotics in the infectious focus until the infection is cleared. Time does play a role here, but it is secondary. The goal in treatment is the clearance of the infection not the consumption of 30 tablets nor the passage of 10 days. As a conclusion, quantity timing (or service intent or order relationship) may specify a goal where time may be one factor. But still, completion of a service needs to be declared by the filler, and should be acknowledged by the placer, to make sure that a possible disagreement is detected early in the process.

3.4 Verification, Authorization and Endoding

The order/results working group has previously discussed the various verifications, encodings and countersignatures that an order may undergo. It seemed that these states are with no direct connection to the other life-cycle states as shown in figure 7 at the three concurrent regions in the lower right corner of the state alive. This may not be an obvious insight, it may even be disputable, so the rationale is given here in some more detail.

First of all, we can distinguish two differnent uses of these special states: (1) verification and (2) authorization.

3.4.1 Verification

Verification (or validation) of the plausibility or applicability of the contents of the ordered service. Such a validation would include testing whether the ordered service is at all to be found in the service catalog of the filler, whether sufficient data was provided by the placer, and whether both administrative and medical constraints on the given service are met. Such constraints may include required admission status, required pre-test results, or a diagnosis in the list of required indications. Failure of the order to meet these requirement may lead to rejection or further discussions. The mechanisms that govern these exceptional conditions are not shown in the model.

3.4.2 Authorization

While verification can be performed completely automatic, authorization is always a conscious act of responsible human beings. By authorization, a human takes on some responsibility for the service. For instance, the attending doctor may have to countersign orders issued by residents, interns, or students. A representative of the filler organization may have to countersign the request to take over responsibilities for its fulfilment or for the complications it may cause to the patient.

Some comparable states in existing business-processes may combine both aspects, verification and authorization. Indeed, authorization would not make sense without the authorizing individual to also verify the service. Such is the "encoding" of medication orders an act of verification *and* authorization. I explicitly showed the state `encoded` in the model because of its outstanding role in HL7's current pharmacy/treatment process model.

There are many possible kinds of such authorizations and verifications which are generally hard to standardize. At most some general patterns can be preformulated by HL7, but much of the detail is up to the communicating institutions and their policies, or laws and regulations that apply for them. The class *service intent or order*, as an interdisciplinary and international class, cannot specify much more than to indicate the existence of authorization states and a general support to mainpulate these states. It is therefore important not to link these states too tightly into the process model of the service, notably the general possibility of fulfillment of a service should not depend on any of these states.

We can always find examples where an unverified, non-authorized, and unencoded service intent or order must be brought to completion without the information system's rules preventing or delaying this. Life threatening situations will always require such exceptions to otherwise justified rules. An intern on emergency room night duty will order a range of tests and will apply a range of medications without further explicit authorization required in routine cases during the daytime.

The conclusion of this discussion is that (1) there are too many different possible states of verification and authorization than can be reasonably modeled explicitely for a class that shall be applicable by many disciplines and under different legal circumstances. (2) There may always be important exceptions to whatever states of verification or authorization are considered required for routine cases. For these reasons the respective states were decoupled from the main part of the model.

3.5 Arbitrary Change

A service intent or order can not only be changed in ways that are relevant to its life-cycle. We need a mechanism by which almost any attribute and association of an order or service intent can be modified by both placer and filler.

The change that either side can make to an order can be trivial or essential. Again we cannot determine in general which attribute changes are trivial and which are essential for all kinds of orders and service intents. We *can* make a distinction, though, between life-cycle relevant changes (e.g., change to the traditional "order status code") and those that do not influence the main state machine. But these other changes may or may not be regarded essential for a given service in a given state.

We agreed on that life-cycle relevant changes are to be made using the proper message that communicates the exact transition that the change implies. I would even argue that we make the "order status code" an untouchable attribute: it can only be read, but noone may ever change it directly. If in a given message the "order status code" is different from the state as expected by the receiver (after updating the receiver's state machine according to the message), the receiver must reject such a message as an application level error. In no case may an order status code as received overwrite the actual order status code.

Also I propose that we make all state transitions explicit. A change in any other attribute may not implicitly update the state. If the state must be updated on behalf of some arbitrary information change, this will only be possible by one allowable transition to fire. Finally I argue that all transitions should be reported to the other party. Only on explicit request by the other party may a transition fire without notification.

The question whether changes are not allowed, essential or trivial is a matter of conformance statements or trading partner agreements. Of course we want HL7 to be interoperable and not depend on arms-length negotiations anymore. Therefore, we need a protocol where changes can be negotiated dynamically and automatically.

An initiator of a change, which may be either the placer or the filler, will send a request to the other party (the responder). The responder can accept the change as proposed or it can decline it. Also the initiator of the change can

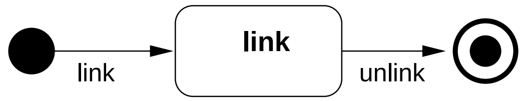

Figure 8: State-transition-model of the class *service intent or order relationship*

cancel the change at any time before the responder has made its decision.

If a change is declined the responder should indicate why this is so and probably suggest a change by himself that would come as close as possible to the original change request. This change protocol may need to be further elaborated.

Note that the filler should not generally have more right to change a service request than the placer. Thus, a simple message "order changed unsolicited" from filler to placer should not generally be allowed, at least the placer must have a chance to cancel or interrupt the service if he cannot agree to the change that the filler suggested. An important question also is, who will cover the filler's expenses in case the service was discontinued because of a change by the filler that was not acceptable for the placer.

It is also important to note that a change will almost certainly affect the validation and authorization status of the order: a prior validated or authorized service may fall back into non-validated or non-authorized state. This is subject to the details of the service and the policies of the institutions using these messages.

4 Service Intent or Order Relationships

The class *service intent or order relationship* allows to construct higher-level (target) services from lower level (source) services. The state-transition model of the general service intent or order relationship is as simple as shown in figure 8: a connection between two classes can be made and released, i.e. two services can be linked and unlinked.

The state-transition diagram of the relationship is not very interesting. Conversely it is interesting to discuss the consequences that source-services have on the life-cycle of target-services and what the effects of life-cycle changes in target services are on source services. An ongoing service may be thought of as a *process*, quite in the same sense as an operating system process. There are four fundamental ways of how processes can relate to each other (see also figure 9):

1. sequential
2. parallel, without join
3. parallel, with awaited join
4. parallel, with enforced join

The spawning off one process (child) from another process (parent) is also called a *fork*. If the parent process hibernates until the child process finishes we have a sequential execution of the processes (1). If the parent process continues to execute, or spawns off other child processes, we have processes that execute in parallel (2–4). These processes can continue to run in parallel regardless if either of them terminates (2) or they can be ultimately re-synchronized, or "joined" (3, 4). If the processes are joined we have two options: either the process who finished earlier waits for the one who is still working (3) or the early process terminates the late process prematurely (4).

An attribute of the relationship class will tell whether a source service is executed in one of these 4 modes. Note that the "workflow management" school uses a slightly different terminology. They speak of an "AND-split" where they mean a parallel execution of tasks and an "OR-split" where they mean simple branching, i.e. a selection of one task among other alternatives. On the other side they speak of an "OR-join" and an "AND-join." Decisions on the terminology should not be guided by adherence to any school of process modeling, but to the most general and concise alternative. I am neither a workflow expert nor am I looking with awe to the workflow idea. I certainly want the relationship class to be at least as expressive as workflow management, yet with a second goal towards minimality and simplicity.

In the working group discussions since the San Francisco meeting we have frequently mentioned the open issue of parent-child service relationship and its influence on the life cycle of the parent service. When is a parent service completed? One reasonable answer is: at the time all of its child-services are completed. However, a sub-service might not have an independent end-condition, but should last as long as another service (e.g., its parent) which it supports. In any case, the prefered rule depends on the details of the service.

Having defined the basic building blocks of inter-process relationships as in the enumeration above, we can model every kind of situation. Thus, a sequential

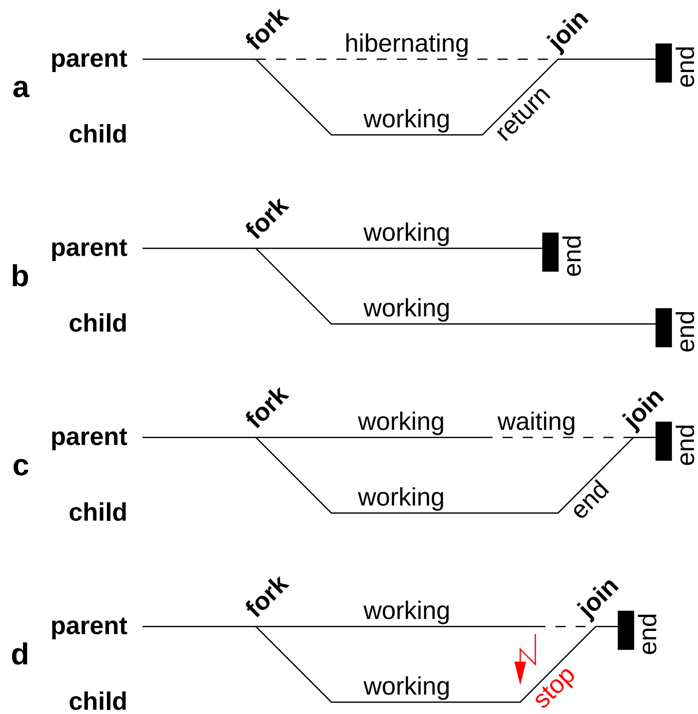

Figure 9: The four fundamental dynamic relationships between processes: (a) sequential, (b) parallel without join, (c) parallel with awaited join, and (d) parallel with enforced join.

service plan arranges the child-services sequentially and will terminate on termination of the last child-service in the sequence. Child-services of lab orders will almost certainly be executed in parallel, e.g., in an electrolytes-status it does not matter whether Sodium is measured before Potassium, vice versa, or both at the same time. We will have an awaited join if the parent service will be complete only when all of its children are completed. A supporting service will have an enforced join because it exists only for the sake of its parent service. When the parent is finished, the child will be terminated as well.