Specifying Concurrent System Behavior and Timing Constraints Using OCL and UML

Shane Sendall and Alfred Strohmeier

Swiss Federal Institute of Technology Lausanne (EPFL)
Software Engineering Laboratory
1015 Lausanne EPFL, Switzerland

email: {Shane.Sendall, Alfred.Strohmeier}@epfl.ch

Abstract Despite advances in implementation technologies for distributed systems during the last few years, little attention has been given to distributed systems within software development methodologies. The contribution of this paper is a UML-based approach for specifying concurrent behavior and timing constraints—often inherent characteristics of distributed systems. We propose a novel approach for specifying concurrent behavior of reactive systems in OCL and several constructs for precisely describing timing constraints on UML statemachines.

More precisely, we show how we enriched operation schemas—pre- and postcondition assertions of system operations written in OCL—by extending the current calculus with constructs for asserting synchronization on shared resources. Also, we describe how we use new and existing constructs for UML statemachines to specify timing constraints on the system interface protocol (SIP)—a restricted form of UML protocol statemachine. Finally, we discuss how both the extended system operation and SIP models are complementary.

Keywords Unified Modeling Language (UML), Object Constraint Language (OCL), Pre- and Postcondition, Software System Specification, Concurrent Programming, Timing Constraints.

1 Introduction

Software-intensive systems are becoming an increasingly important and integral part of everyday life, many of which are distributed and constrained by long lists of non-functional requirements. Expectation and reliance on such software is making the software industry reevaluate the importance of software quality. Assuring that the software has a certain level of quality is a direct concern of the developer. Consequent to this, we believe that there is an increasing need for approaches that target the development of distributed systems and that can provide a reasonable level of quality assurance and rigor in development but still remain cost-effective in terms of time and effort.

In this paper, we propose an approach for specifying two common characteristics of distributed systems: concurrency and timing constraints. Even though such characteristics are inherent in more and more systems, largely due to the increasing amount of internet-based software, UML-based development approaches have paid surprisingly little attention to integrating timing constraints into specifications of functional requirements and to providing guidelines for dealing with synchronization dependencies between operations and resources. This is further compounded by UML's limited support, in general, for the specification of timing constraints and for mechanisms to describe synchronization of concurrent activities.

Our proposal extends our previous work: an approach that uses UML [10] and OCL [16] to specify the behavior of reactive systems [12][13]. Our approach has three principal views [11]:

- a model composed of descriptions of the effects caused by operations, which uses pre- and postcondition assertions written in OCL, called operation schemas;

- a model of the allowable temporal ordering of operations, called the system interface protocol (SIP); and
- a model that describes the system state used in the operation schemas, called the analysis class model (ACM).

In this paper, we show how we enriched operation schemas by extending the current calculus with constructs for asserting synchronization on shared resources. Also, we describe how we use new and existing constructs for UML statemachines to specify timing constraints on the system interface protocol (SIP)—a restricted form of UML protocol statemachine. Furthermore, we discuss how both the extended system operation and SIP models are complementary.

Our aim with this approach is to support the specification of distributed systems in software development practice. And therefore provide the designer with guidelines on synchronization dependencies between operations and timing constraints on their execution. We have a number of criteria that we use to evaluate and guide the development of our approach; for a full list and discussion see [13]. In this paper, we concentrate on two of them:

The descriptions of our approach should support the specification of:

- "quantifiable" non-functional requirements, such as performance constraints, in an integrated way with respect to the functional requirements;
- inherent concurrent properties of the system and quality of service properties.

The paper is composed of six sections. Section 2 briefly presents a case study of an auction system, which is used to highlight our approach and proposals for extending it for specifying concurrency and timing constraints. Section 3 describes our proposals for modeling timing constraints in UML protocol statemachines. Section 4 describes our proposals for specifying concurrent system operations with operation schemas. Section 5 presents related work and section 6 concludes the paper.

2 Auctioning System Case Study

For illustrating our proposals and for use as a common example throughout this paper, we describe an auctioning system, adapted from [6] and [7]. It has many similarities to internet auctioning sites, such as, eBay (www.ebay.com), and uBid (www.ubid.com), although it takes a more conservative view on bid validation. The auctioning system allows people to negotiate over the buying and selling of goods in the form of English-style auctions over the world-wide web. All potential users of the system must first enroll with the system; once enrolled they have to log on to the system for each session, where they are able to sell, buy, or browse the auctions currently running. Customers have credit with the system that is used as security on each and every bid. Customers can increase their credit by asking the system to debit a certain amount from their credit card.

A customer that wishes to sell initiates an auction by informing the system of the goods to auction with the minimum bid price and reserve price for the goods, the start date of the auction, and the duration of the auction, e.g., 30 days. The seller has the right to cancel the auction as long as the auction's start date has not been passed, i.e., the auction has not already started.

Customers that wish to follow an auction must first join the auction. Note that it is only possible to join an active auction. Once a customer has joined the auction, he/she may make a bid. A bid is valid if it is higher than the previous high bid augmented by the minimum bid increment (calculated purely on the amount of the previous high bid, e.g., 50 cent increments when bid is between \$1-10, \$1 increment between \$10-50, etc.), and if the bidder has sufficient funds: the customer's credit with the system is at least as high as the sum of all his/her pending bids. Bidders are allowed to place their bids until the auction closes, and place bids across as many auctions as they please. Once an auction closes, the system calculates whether the highest bid meets the reserve price given by the seller (English auction style reserve price), and if so, the system deposits the highest bid price minus the commission taken for the auction service into the credit of the seller (credit internal to system). The auction system is highly concurrent—clients bidding against each other in parallel, and a client placing bids at different auctions and increasing his/her credit in parallel.

The system operations for the auctioning system are derived from the use case descriptions of the system, not supplied in this paper. A partial view of the input and output events exchanged between the system and its actor is shown, in figure 1, by a UML (specification-level) collaboration diagram. It shows only those events that interest us for the purposes of this paper, i.e., eleven different input events: proposeAuction, joinAuction, cancelAuction, increaseCredit, autoWithdraw, cancelAutoWithdraw, placeBid, logOff, logOn, enrolCustomer, and tick. The tick event from the clock triggers the system operations, closeAuction, increaseCredit and logOff. We use a special icon for the clock actor because we want to highlight that the actor and related events are special.

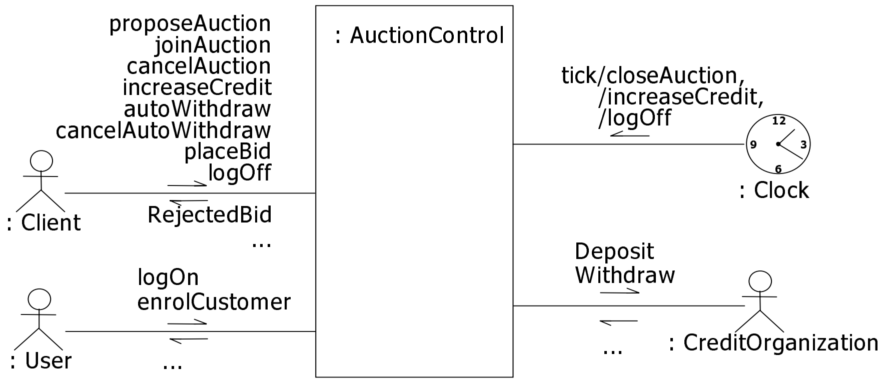


Fig. 1. Collaboration diagram summarizing the interaction between the system and its actors

The analysis class model for the auction control system is shown in figure 2. It shows all the domain concepts and relationships between them, the combination of which provides an abstract model of the state space of the system and defines the system boundary. This model is used as the basis for writing operation schemas, i.e., pre- and postcondition assertions for each system operation. Inside the system there are five classes, Auction, Bid, Goods, Customer, and Credit, and outside four actor classes, Clock, User, Client, and CreditOrganization. The system has six associations: HasHighBid links an auction with the current high bid if it has one, SellsIn links an auction with its respective seller (customer), Makes links a customer to his/her bids, JoinedTo links an auction to all its participants (customers), Has links a customer with his/her credit, and Guarantees links a bid with the credit that ensures its solvency. Finally, an <<id>> stereotyped association means that the system can identify an actor starting from an object belonging to the system. For example, given a Customer, cust, we can find its corresponding client actor with the navigation expression, cust.represents.

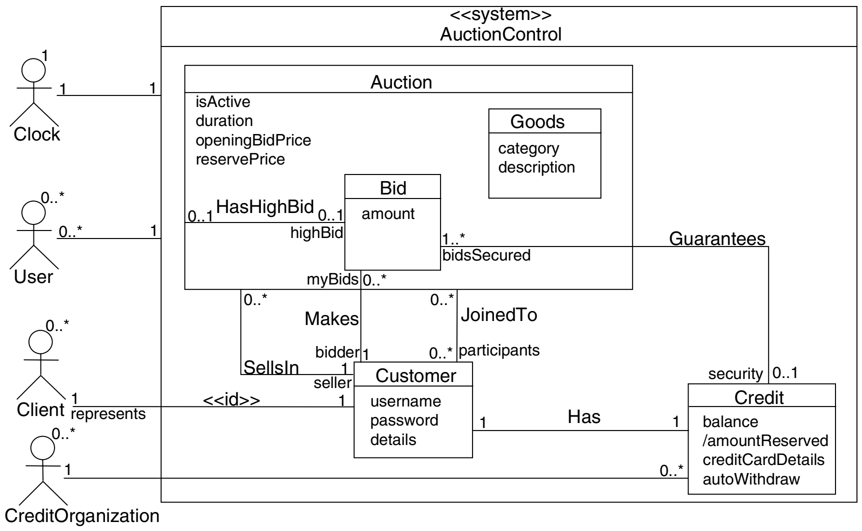


Fig. 2. Analysis class model for the AuctionControl system

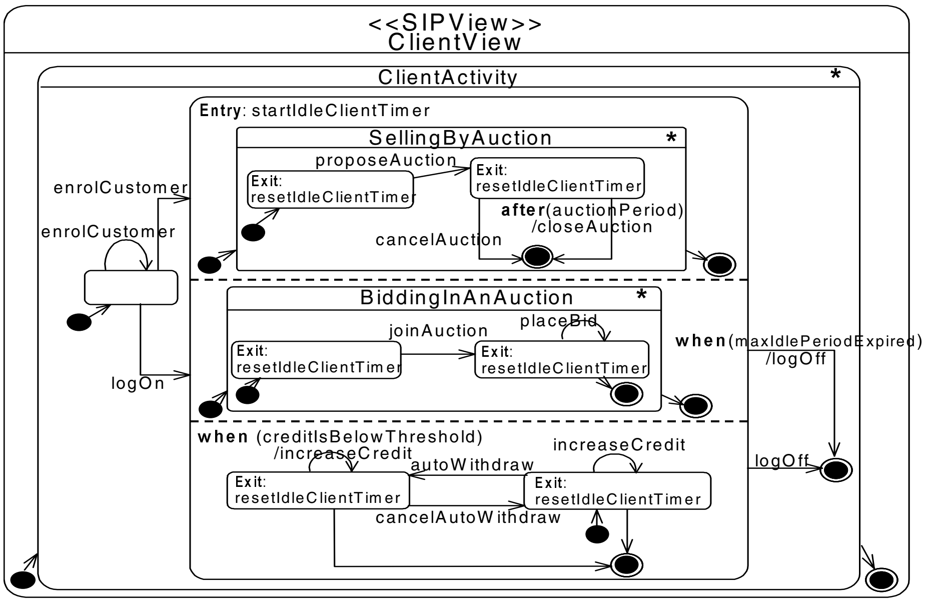
The System Interface Protocol (SIP) defines the temporal ordering of system operations—one aspect of the behavior model of the system. An SIP is described with a UML state diagram. A transition in the SIP is triggered by an input event only if the SIP is in a state to receive it, i.e., there exists an arc with the input event and the guard evaluates to true. If not, the input event that would otherwise trigger an operation is ignored. We use the convention, also mentioned in the UML specification for protocol statemachines [10], that the action (in our case the operation) need not be explicit if it has the same name as the event.

The SIP for sequential and trivial systems can normally be described with a single statemachine. We have made the observation, however, that concurrent systems are better described with multiple views, one view per perspective on the concurrency. Apart from providing a clearer description of the protocol, a multi-view approach also has the advantage that it allows one to focus on and formulate intuitive timing constraints, i.e., one can define views that facilitate the description of certain timing constraints. However, a multi-view SIP requires rules for composing the different views to form the ensemble. Rules for composition, completeness, consistency, etc. are out of the scope of this paper.

We realize the concept of a "view of the SIP" in UML by introducing the stereotype <<SIPView>> to label the corresponding statemachines that describe a view. The SIP is defined by the composition of all the statemachines that are members of the same UML package and that are stereotyped <<SIPView>>.

Two SIP views for the AuctionControl system are shown in figure 3 and figure 4.

Figure 3 shows an <<SIPView>> statemachine that focuses on the concurrency related to clients interacting with the system. The ClientActivity state is an auto-concurrent statemachine, indicated by a multiplicity of many ('*') in the upper right-hand corner. There is a statemachine for each client activity but their number is not predefined, hence the multiplicity many. The main substate of ClientActivity consists of three parallel states. The top-most state, SellingByAuction, is also auto-concurrent with the meaning that the respective client can be selling in many auctions in parallel. The time-triggered termination of the auction is specified by the time event, **after** (auctionPeriod). Similarly, the state in the middle, BiddingInAnAuction, is auto-concurrent with the meaning that the client can be participating (bidding) in many auctions in parallel. Finally, the unnamed, bottom-most state shows that the client may increase his/her credit at any moment. The client has also the option to allow the system to automatically debit his/her credit card once the customer's credit in the system drops below a certain amount. This is realized by the time event, **when** (creditIsBelowThreshold), which triggers the increaseCredit operation. Finally, the customer may log out explicitly or be logged out automatically if he/she has been inactive for a certain period of time, **when** (maxIdlePeriodExpired).



<<SIPView>>
ClientView
ClientActivity
*
Entry: startIdleClientTimer
SellingByAuction
*
proposeAuction
Exit: resetIdleClientTimer
Exit: resetIdleClientTimer
after(auctionPeriod) /closeAuction
cancelAuction
enrolCustomer
enrolCustomer
BiddingInAnAuction
*
joinAuction
placeBid
Exit: resetIdleClientTimer
Exit: resetIdleClientTimer
when(maxIdlePeriodExpired) /logOff
logOn
when (creditIsBelowThreshold) /increaseCredit
increaseCredit
autoWithdraw
Exit: resetIdleClientTimer
Exit: resetIdleClientTimer
cancelAutoWithdraw
logOff

Fig. 3. Client view of the SIP for the AuctionControl system

The client SIP view is complete with respect to the criterion that all (input) events are accounted for, i.e., with respect to figure 1. However, it does not take into account the fact that an auction continues regardless of who logs on and off, and for example an auction may terminate, triggered by **after** (auctionPeriod), without the seller or any other participants being logged in. As a consequence, the event would be ignored because the statemachine is not in a state to trigger a transition. Hence, we need an additional view to model the case that is unaccounted for.

Figure 4 shows an <<SIPView>> statemachine that focuses on the concurrency related to auctions in the system. The AuctionActivity state is auto-concurrent with the meaning that many auctions can take place in parallel, one auction per state. This SIP view covers the full lifecycle of the auction and is thus independent of whether the seller and/or participants are logged on or not.

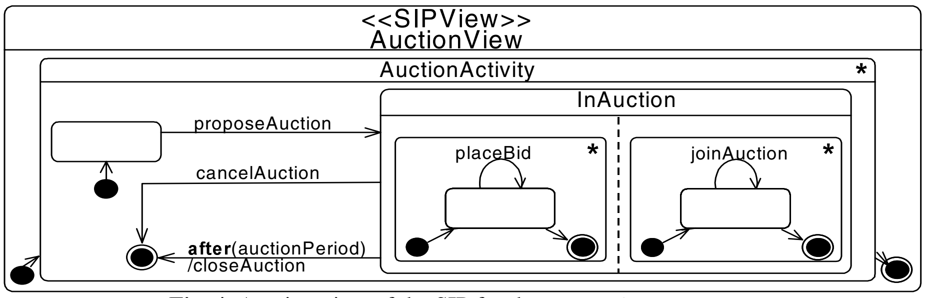


Fig. 4. Auction view of the SIP for the AuctionControl system

We now give a brief overview of the syntax, usage, and semantics of operation schemas. An operation schema declaratively describes the effect of the operation on a conceptual state representation of the system and by events sent to the outside world. It describes the *assumed* initial state by a precondition, and the required change in system state after the execution of the operation by a postcondition, written in UML's OCL formalism [16]. Moreover, we use the same interpretation of assertions as Larch [2]: when the precondition is satisfied, the operation must terminate in a state that satisfies the postcondition. Operation schemas as we define them here specify operations that are assumed to be executed atomically and instantaneously, hence no interference is possible (this assumption is revised in section 4).

Each system operation, proposeAuction, joinAuction, cancelAuction, increaseCredit, autoWithdraw, cancelAutoWithdraw, placeBid, logOff, logOn, enrolCustomer, and closeAuction (highlighted in figure 1) is described by an operation schema. However for reasons of size, we highlight just the placeBid operation schema, shown in figure 5. The placeBid operation schema describes the placeBid system operation. The placeBid system operation occurs as a consequence of a client placing a bid in an auction. The system must decide whether the bid is realistic and solvent; if so, the bid is recorded and the credit of the customer is decremented.

The **Operation** clause provides the signature of the placeBid indicating that the operation has three parameters: the target auction, auct, the bidder, cust, and the amount bid, proposedAmount. The **Description** clause offers a concise natural language description of the operation. The **Use Cases** clause provides cross-references to referring use cases. The **Scope** clause defines all the classes, and associations from the analysis class model defining the name space of placeBid. The last element of the **Scope** clause is an unnamed association that represents the composition association between Bid and Auction: the only unnamed association between the two classes. The **Declares** clause provides two kinds of declarations: naming, and aliases. The first in the list is a variable name, and the last three in the list are aliases, which are used as name substitutes. The **Sends** clause shows that only one type of event may be output by the operation, i.e., the event RejectedBid may be sent to Client actors. The **Pre** clause states that our assumption is that the operation is called while the auction is still active and the customer placing the bid is a participant in the auction.

The **Post** clause states that if the bid is realistic (first **if** block) and solvent (second **if** block) then the system decrements the amount bid from the credit of the bidding customer and records the bid in the system as the current high bid, and if there was a previous bid stored then the customer that held the previous high bid is reimbursed with the amount which he/she bid. Reimbursing a customer once his/her bid is no longer the current high bid allows him/her to immediately reuse the money, e.g., for bidding again or for bidding on other goods. If either the bid was not realistic or solvent then the client was sent an exception indicating that the bid was rejected. An event or exception is specified as sent by placing it in the event queue of the actor instance—**sent** is used as a shorthand for this purpose. For example, the second-to-last line of the **Post** clause states that an event occurrence of type RejectedBid, whose formal parameter reason matches the value #impossibleBid, was placed in the event queue of the client actor, denoted by the navigation expression cust.represents. Further details on the syntax, informal semantics, and usage of operation schemas can be found in [13][15].

```
Operation: AuctionControl::placeBid (auct: Auction, cust: Customer, proposedAmount : Money);
Description: A bid is placed with the system. The bid is accepted if it is >= to the highest bid so
far plus the minimum increment or >= to the min. initial price if it is the first bid in the auction;
Use Cases: buy item under auction;
Scope: Auction; Bid; Customer; Credit; HasHighBid; Makes; Guarantees; JoinedTo; Has;
Declares:
    bid: Bid;
    prevHighBid: Bid Is auct.highBid;
    creditOfPrevHighBid: Credit Is prevHighBid.customer.credit;
    isFirstBid: Boolean Is auct.bid->isEmpty ();
Sends:
    Type: Client::{RejectedBid};
Pre: auct.isActive and auct.participants->includes (cust); -- auction has not finished and cust is a participant
Post:
if (isFirstBid and proposedAmount >= auct.openingBidPrice) or
  (proposedAmount >= prevHighBid.amount + minBidIncrement (prevHighBid.amount)) then
    if cust.credit.balance@pre >= amount then -- if cust has sufficient funds
        cust.credit.balance -= proposedAmount & -- x -= 1 <==> x = x@pre + 1
        bid.oclIsNew (amount => proposedAmount) & -- attr of new bid, amount, has value proposedAmount
        cust.myBids->includes (bid) & -- <==> cust.bid = cust.bid@pre->including (bid)
        auct.bid->includes (bid) &
        bid.security = cust.credit &
        auct.highBid = bid &
        if prevHighBid->notEmpty () then -- if there was a previous bid
            creditOfPrevHighBid.balance += prevHighBid.amount &
            prevHighBid.security->isEmpty ()
        endif
    else
        (cust.represents).sent (RejectedBid ((reason => #insufficientFunds)))
    endif
else
    (cust.represents).sent (RejectedBid ((reason => #impossibleBid)))
endif;
```

Fig. 5. Operation schema for the placeBid system operation

3 Modeling Timing Constraints

Many real-time and reactive systems exhibit behavior that is constrained by time-related factors, such as response time, waiting time, arrival rate, the number of events processed in some interval of time, etc. Specifying timing constraints is therefore an important activity in the development of such systems. Approaches that support the specification of real-time and reactive systems should support the notion of time-based constraints.

In our approach, the best candidate model for specifying timing constraints is the SIP. This is because statemachines already define part of the necessary vocabulary, such as, event dispatching, event triggered actions, time events, and ordering of events. We choose to keep timing constraints exclusive to the SIP, i.e., we disallow timing constraints in operation schemas, because we want to keep a clean separation of concerns between the functional description of operations, i.e., operation schemas, and their temporal ordering, i.e., the SIP.

We now describe the extensions that we have made to UML protocol statemachines for modeling timing constraints. We propose five time-based properties of transitions, summarized in figure 6. The first two denote absolute time, befT and aftT; the second two denote time periods, durT, distT; and the last one denotes a time frequency, freqT. Such variables provide direct access to timing information related to transitions, facilitating the expression of timing constraints.

Property	**Description**
befT	The (absolute) dispatching time for the last event that could trigger the transition (note the transition may or may not be taken; firing depends on whether the guard was true or not).
aftT	The (absolute) time at which the last transition completed (i.e., entered the destination state).
durT	The duration in time of the last transition from the source to the destination state (aka. execution time of the operation). It is equivalent to: aftT - befT, if and only if the guard holds.
distT	The amount of time since the last event was fired in the same activity at the *same* level in the state hierarchy, i.e., its destination was the source state of this transition.
freqT	The frequency with which the transition is taken, i.e., the number of times the transition was taken over a certain sampling period: #transitions/period.

Fig. 6. A summary of the five proposed time-based transition properties in UML statemachines

Some timing constraints are invariant. Consider the example where the system must enforce that an operation opX never takes longer than 5 seconds to execute. This is a constraint on the performance of the system. In OCL, we could formulate this with the following invariant that is attached to the respective transition (the transition that triggers the operation opX):

```
<<invariant>>
self.durT <= 5*Sec
```

The interpretation of this invariant is that every time the transition is fired (i.e., the guard evaluates to true) its duration, the time from source to destination states, is always less than or equal to five seconds. In the case that an event is dispatched but rejected, the durT variable is not affected and thus there is no obligation from the invariant.

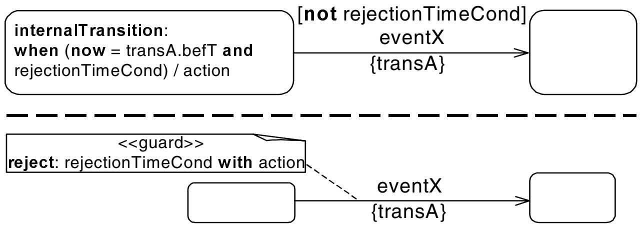
Note that the context is the transition, which is referred to as self. According to OCL, the self keyword is optional, i.e., it may be dropped when the context is clear.

Apart from stating timing invariants, it is often necessary to state the circumstances in which events can not or should not be served by the system. Guards on transitions can be used for that purpose. Using the transition properties summarized in figure 6, we can define time expressions in guards. For example, we could define a transition with a guard that rejects all eventXs that are dispatched less than 3 seconds after the previous event that entered the source state:

```
[distT >= 3*Sec] eventX / action
```

In this case, we used the distT variable, which can be thought of as defining the amount of time spent in the source state of the transition. We could have equally, and perhaps more intuitively, defined distT as a property of states rather than transitions.

It is also useful to be able to relate an action to a timing constraint failure (defined by a guard), e.g., output an exception if a time expression in the guard fails. However, according to the UML specification (pp. 2-170 of [10]), guards on transitions should not use expressions that cause side-effects. In figure 7 (top), we show how we might model such “guard failure” actions in standard UML (plus our proposed time-based properties of transitions), where **now** is a variable denoting the current time. In the same figure (bottom) we show our proposed equivalent shorthand notation. Note that {transA} is a transition label used to identify the transition.



internalTransition:
when (**now** = transA.befT **and** rejectionTimeCond) / action
[**not** rejectionTimeCond]
eventX
{transA}
<<guard>>
reject: rejectionTimeCond **with** action
eventX
{transA}

Fig. 7. Proposed "guard failure" actions in UML statemachines;
Top: Using standard UML notation; *Bottom*: Using our proposed shorthand notation;

The BNF-like grammar of our proposed shorthand notation is as follows:

guard == ["**cond**:" boolean_expression ';'] ("**reject**:" boolean_expression ["**with**" action ';'])*

The "transition guard" shorthand contains two types of clauses: **cond** and **reject**. **cond** indicates the condition that must hold for the transition to be fired; it is equivalent to the original guard definition. We do not, however, allow time expressions to be written in this clause. **reject** defines a time-based boolean expression and an optional action. If the boolean expression evaluates to true then the dispatched event is rejected and the action is executed, if there is one. There may be many **reject** clauses, where the meaning is the disjunction of the boolean expressions of each **reject** clause. The meaning of the proposed guard as a whole is thus defined by the combination of its two parts: the condition of the **cond** clause must be true and the boolean expressions of all the **reject** clauses must be false for the transition to have permission to fire. For example, we attach a guard, which makes use of the "transition guard" shorthand, to the increaseCredit event transition of figure 3. It ensures that all increaseCredit events that are made at a frequency greater than 1 per second are rejected and a SystemOverload exception is output:

<<guard>>
reject: freqT > 1*Hz **with** ^SystemOverload;

We emphasize that all time-related guard expressions are to be placed in the **reject** clause and not in the **cond** clause nor in the operation schemas.

Applying the proposals for time-based constraints made up until now, we present a number of timing constraints on two SIP views of the AuctionControl system: AuctionView and BiddingView, shown in figure 8. Note that to be able to identify transitions easily, we label the ones we reference, e.g., {a}. Also, we attach guards to transitions by UML notes, which use <<guard>> stereotypes. In this way, we can avoid transition clutter, i.e., we avoid to put all the information directly on the transition.

The invariant in the UML note 1 states that the system must guarantee that an auction starts within 5 seconds provided that there are less than or equal to 500 active auctions, and within 10 seconds if the number of active actions is between 500 and 1000 (self refers to the system object). The invariant in the UML note 2 makes references to the two labels, {a} and {b}, which are used to identify the respective transitions. The constraint states that every time an auction activity fires transition a, then b does not occur within 15 minutes of it nor more than 1 year afterwards. Note that in this example we changed the AuctionView SIP view from a definitive auction deadline (i.e., **after** (AuctionPeriod) / closeAuction) to one that depends on the idle period between bids (i.e., **when** (MaxIdleBidPeriodExpired) / closeAuction). These two invariants are performance constraints.

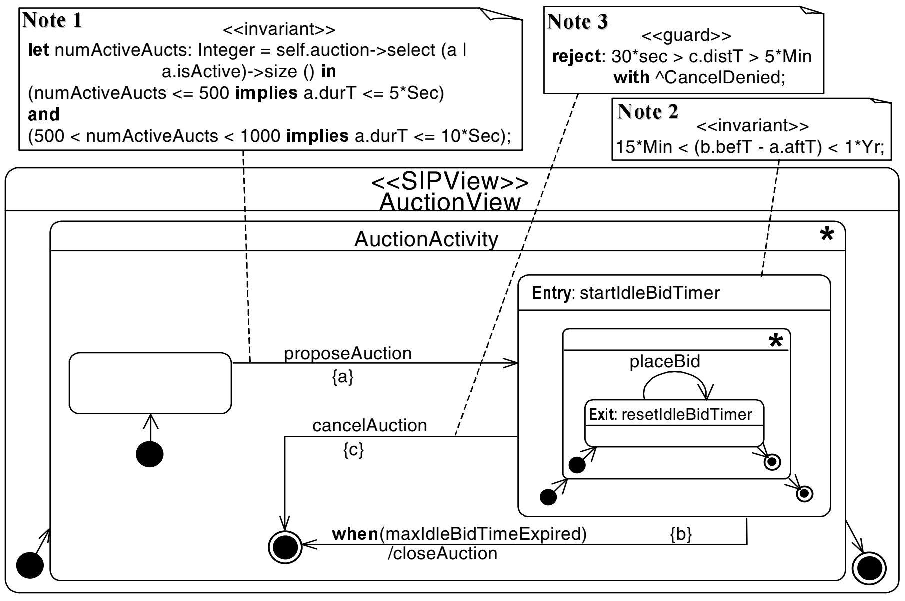

Note 1
<<invariant>>
let numActiveAucts: Integer = self.auction->select (a | a.isActive)->size () in
(numActiveAucts <= 500 implies a.durT <= 5*Sec)
and
(500 < numActiveAucts < 1000 implies a.durT <= 10*Sec);
Note 3
<<guard>>
reject: 30*sec > c.distT > 5*Min
with ^CancelDenied;
Note 2
<<invariant>>
15*Min < (b.befT - a.aftT) < 1*Yr;
<<SIPView>>
AuctionView
AuctionActivity
*
Entry: startIdleBidTimer
proposeAuction
{a}
*
placeBid
Exit: resetIdleBidTimer
cancelAuction
{c}
when(maxIdleBidTimeExpired)
/closeAuction
{b}

Fig. 8. Timing constraints attached to two SIP views

The constraint in the UML note 3 specifies a guard on the cancelAuction event transition with the meaning: any cancelAuction event that is dispatched within 30 seconds of the proposeAuction transition or more than 5 minutes after it is rejected and an exception CancelDenied is output.

When timing invariants and guards are united with time activated transitions as shown in figure 3 and figure 4 (**when** and **after** time events), we have quite some possibilities for defining behavior that is constrained by time-related factors.

4 Modeling Concurrent Operations by Schemas

The online auction system is inherently concurrent at the auction level—clients placing bids in parallel—and at the level of the client's credit—a client placing bids at different auctions and increasing his/her credit in parallel. The different views of the SIP highlight the concurrency between operations. Nevertheless the instantaneous and atomic execution assumption on operations means that operation schemas describe operations that execute in isolation, i.e., there is no interference between operations. Furthermore, if the developer was to design from the placeBid operation schema in figure 5, s/he would be given no help whatsoever as to which resources are possibly shared and what synchronization dependencies on resources there are between these operations.

In this section, we propose extensions to the calculus of operation schemas for the purpose of providing such information to designers, according to the concurrency defined by the SIP. We therefore release the atomicity and instantaneity hypothesis on operations (that was defined in section 2) and allow the specification of fully concurrent operations—operations that are possibly changing the state of the system in parallel.

Absence of interference when updating a shared resource is a safety property of the system. Interference can result in corruption of the resource and erroneous behavior. Exclusive access to a shared resource when updating it is a property that we want all possible solutions to exhibit. We propose to impose it in the model as part of the contract on the software. To highlight this constraint on shared resources in operation schemas, we add a clause to the operation schema format called **Shared**. Resources listed in this clause are constrained to be updated in mutual exclusion by the operation.

When writing operation schemas for concurrent operations, the easiest way to formulate the pre- and postconditions is to write them like we would do if there were no interference by other concurrent operations. Unfortunately, this is not possible when describing changes to a shared variable that can be changed by competing concurrent operations. In particular, the OCL @pre and the implicit "@post" suffixes are almost meaningless for shared variables when describing system state changes in postconditions. Instead, the values of shared variables immediately before and after an update under mutual exclusion are of prime importance. For example, if an operation adds 7 to a shared integer variable val, the effect that one wishes to state is that 7 was added to the value of the variable that was observed immediately before the mutually exclusive update. We, therefore, introduce the suffixes @preAU and @postAU for shared variables, which signify the state of the prefix variable immediately before and after an Atomic Update by the operation, respectively. Thus, we would state the following to signify that 7 was added to the shared variable val:

```
val@postAU = val@preAU + 7
```

Furthermore, when we need to read the value of a shared variable, we need to ensure that we are referring to a consistent value of the variable, i.e., the variable was read outside of any period where the variable was updated. Any consistent value of a shared variable that is taken within the period of the operation's execution is denoted by suffixing the variable name by @rd. The possible suffixes for shared and unshared variables and their meanings are summarized in figure 9.

Schemas can still be structured with *if-then-else* blocks. The branch conditions are evaluated atomically with respect to the blocks, i.e., there is no possibility for racing between the evaluation of the branch conditions and the evaluation of the effects. Also, *if-then-else* blocks are evaluated immediately, i.e., a condition is either true or false, and there is no waiting for the condition to become true.

Also, a situation that can arise in a concurrent environment is when a group of effects relies on a certain condition to stay true during its whole "execution". We model such situations with *rely* blocks (based on the concept of rely conditions, first introduced by Jones [3], see section 5). The *rely* block states a condition that must be true immediately before, immediately after, and during the execution of the body of the block for the body to take effect. If the rely condition does not stay true throughout execution, then the effect of the *fail* part of the rely block is observed to execute instead. The *rely* block imposes neither immediate nor wait semantics on the condition, i.e., an implementation that does a wait until the condition becomes true and then tries to execute the body, or one that fails if the condition is not initially true are both valid refinements. Also, the *rely* block does not stop the implementation from making several attempts (and associated rollbacks) at executing the body of the block. It does however require that if the condition remains true, then the effect described by the body of the block will hold. Also, *rely* blocks should not be nested in *if* blocks due to the immediate evaluation semantics of the *if*. But, *rely* blocks nested in *rely* blocks, and *if* blocks nested in *rely* blocks are possible.

	Shared variables	**Unshared variables**
x@pre	The last possible *consistent* value of x immediately before the start of the operation's execution.	The value of x immediately before the execution of the operation.
x@post	The first possible *consistent* value of x immediately after the termination of the operation's execution.	"@post" is normally implicit (see x).
x@preAU	The value of the variable immediately before the operation's (atomic) update of x.	– (unused)
x@postAU	The value of the variable immediately after the operation's (atomic) update of x.	– (unused)
x@rd	Any *consistent* value of x inside the bounds of the operation execution (but outside of any updates to x).	– (unused)
x	only allowed in eventually functions.	The value of x immediately after the execution of the operation.

Fig. 9. A summary of the possible variable suffixes in postconditions

Figure 10 shows the operation schema (minus a few clauses that stay unchanged) for the placeBid system operation. It applies our proposed extended calculus to the placeBid operation schema of figure 5. The **Operation** and **Pre** clauses have not changed from figure 5. The **Declares** clause is also similar except it uses time expressions in the name substitute. The **Shared** clause defines all associations and variables that are shared. The **Post** clause shows a number of differences from the schema of figure 5. The *rely* block requires that the balance of the bidder's credit stays above or equal to zero and the auction stays active for the body to have effect. If this condition cannot be guaranteed then one or more of the *fail* parts are observed to execute, e.g., one or two RejectedBid exceptions are output to the bidding client. The body of the *rely* block changes the state of the system in the same way as described by the schema in figure 5, except there are shared variables, which are updated in mutual exclusion. It states that if the proposed bid is realistic (because we rely on the fact that it is solvent), then it is recorded and set as the high bid; also the bidder gets the proposed amount debited from his/her credit, and if there was a previous bid, then the corresponding bidder is reimbursed the credit that was previously taken for the bid.

```
Operation: AuctionControl::placeBid (auct: Auction, cust: Customer, proposedAmount : Money);
Declares:bid: Bid;
          prevHighBid: Bid Is auct.highBid@preAU; -- name substitution
          creditOfPrevHighBid: Credit Is prevHighBid.customer.credit;
          isFirstBid: Boolean Is auct.bid@preAU->isEmpty ();
Shared:   Collection(Bid): Makes; Bid: HasHighBid; Collection(Bid): Guarantees;
          Collection(Bid): (Auction, Bid); isActive::Auction; balance::Credit;
Pre: auct.isActive and auct.participants->includes (cust);
Post:
rely cust.credit.balance@rd >= 0 and auct.isActive@rd then
   if (isFirstBid and proposedAmount >= auct.openingBidPrice) or
   (proposedAmount >= prevHighBid.amount + minBidIncrement (prevHighBid.amount)) then
      cust.credit.balance@postAU = cust.credit.balance@preAU - proposedAmount &
      bid.oclIsNew ((amount => proposedAmount)) &
      cust.myBids@postAU->includes (bid) &
      auct.bid@postAU->includes (bid) &
      bid.security@postAU = cust.credit &
      auct.highBid@postAU = bid &
      if prevHighBid->notEmpty () then -- there was a previous bid
         creditOfPrevHighBid.balance@postAU =
            creditOfPrevHighBid.balance@preAU - prevHighBid.amount &
         prevHighBid.security@postAU->isEmpty ()
      endif
   else
      (cust.represents).events-> includes (RejectedBid ((reason => #impossibleBid)))
   endif
fail (cust.credit.balance@rd >= 0) then
   (cust.represents).events-> includes (RejectedBid ((reason => #insufficientFunds)))
fail (auct.isActive@rd) then
   (cust.represents).events-> includes (RejectedBid ((reason => #auctionClosed)))
endre;
```

Fig. 10. Operation schema for placeBid system operation; non-atomic & non-instantaneous version

Both the schemas of figure 5 and figure 10 place an obligation on the system to reimburse the previous high bid. This “undoing” obligation gives bidders the freedom to use the credit on other auctions and not to have it pointlessly locked up on a bid that will not win. One could nevertheless argue that this is overly restrictive and that abstracting at a higher level would make the feature useless and the specification more concise. We can realize such an abstraction by using the temporal logic operator, *eventually* [1]. The *eventually* operator evaluates to true if its corresponding condition becomes true in some future state. We propose to denote this operator in our calculus by the function eventually (...), which takes a boolean expression as parameter.

Using *eventually* we can oblige the system to only debit the bidder’s credit, if and only if the bid wins the auction (i.e., in the future). Highlighting this in the placeBid operation schema, we could integrate the following *if* block into the body of the *rely* block, and remove the effects that debit and credit the current and previous bidders’ balance. We refer to the schema that includes this constraint as version 2, and the original, shown in figure 10, as version 1.

```
if eventually (not auct.active and auct.highBid = bid) then -- if, in the future, the bid wins then
   cust.credit.balance@postAU = cust.credit.balance@preAU - proposedAmount &
   bid.security@postAU = cust.credit
endif
```

The constraint states that if some time in the future the auction is closed and the current bid is in fact the winner, then the bidder’s credit is debited by the bid amount and set as security on the bid. Note that in most cases it should be insured that the *eventually* function can be evaluated after some finite time or when some combination of events occur. In the previous example, for instance, when the auction closes, the *eventually* function can be evaluated.

How an implementation meets the schema be it version 1 or version 2 is another issue that we now discuss. An implementation could, for example, encapsulate the placeBid operation by a transaction and simply abort and rollback once it finds out that it was outbid [7]. This solution would be the most obvious given the schema version 1. However, schema version 2 is less restrictive. For example, a naive implementation could grab the lock on the bidder's credit and hold onto it until the end of the auction. This would mean that all bids on other auctions by the same bidder that come afterwards would be blocked waiting on the end of the auction of the first bid. This could lead to starvation, for instance, the auctions for later bids could finish before the auction of the first bid, etc.

Research is ongoing on integrating a design of a concurrent system into an architecture [5].

5 Related Work

Formal methods have much to say about the specification of concurrency and timing constraints in systems. The approach described by Lano in [8] provided some motivation for this work. It described extensions to Z and VDM for describing concurrency and timing constraints. Z [14] and VDM [4] are both rich formal notations but they suffer from the problem that they are very costly to introduce into software development environments, as is the case with most formal methods, because of their high requirements in mathematical mastery.

A number of temporal logic operators have been proposed over the years to describe concurrent and reactive systems [1]. The advantage of temporal logic operators is that one can obtain an abstract and concise description of a concurrent system. The disadvantage, however, is that the designer is not given much information on how s/he might go about tackling the problem.

Meyer proposes a programming model for concurrent programming called Simple Concurrent Object-Oriented Programming (SCOOP) [9]. SCOOP reinterprets the preconditions of design-by-contract as guards and enforces that all methods are atomically accessed. Therefore, SCOOP assumes a particular object-oriented concurrency framework. In the context of system-level operations, we prefer to take a more fine-grained approach to concurrency control—giving the developer more freedom in designing a solution.

Our motivation for the rely conditions of section 4 comes from the work by Jones [3]. He proposed supplementary clauses to pre- and postconditions called rely and guarantee conditions; they allow one to state under what circumstances the postcondition makes sense in the presence of concurrency. If an operation is invoked in a situation when the precondition is false, or if during the execution of the operation the rely condition becomes false, then the specification does not state what the outcome should be. Otherwise the postcondition will be true at the end of the execution and the guarantee condition will have been maintained throughout. However, it often the case that the enforcement of the rely condition during the whole execution of the operation is a constraint too strong. This is a consequence of stating concurrency constraints at a coarse grain, in a top-down approach. For this reason, we propose a rely condition that can be used at a finer-grain level.

6 Conclusion

The contribution of this paper is a UML-based approach for specifying concurrent behavior and timing constraints. We proposed a novel approach for specifying concurrent behavior of reactive systems based on joint use of operation schemas and UML protocol statemachines (SIP). We also proposed constructs for precisely describing timing constraints in the SIP. The operation schemas and the SIP are complementary: the operation schemas describe the functional responsibilities of the system and the SIP defines the temporal ordering between operations. The extensions to the SIP for specifying timing constraints allow one to formalize behavior that is constrained by time-related factors. The extensions to operation schemas provide guidelines to developers for dealing with synchronization dependencies between operations and resources. Consequently, we believe our approach offers developers better support for designing distributed systems that exhibit concurrent and time-constrained behavior.

References

[1] E. Emerson; *Temporal and Modal Logic*. In J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Amsterdam, 1989, pp. 995-1072.

[2] J. Guttag et al. *The Larch Family of Specification Languages*. IEEE Trans Soft Eng 2(5), Sept. 1985.

[3] C. Jones; *Tentative steps toward a development method for interfering programs*. ACM Transactions on Programming Languages and Systems, 5(4):596-619, 1983.

[4] C. Jones; *Systematic Software Development Using VDM*. Prentice Hall, 1986.

[5] M. Kandé and A. Strohmeier; *Towards a UML Profile for Software Architecture*. UML 2000 — The Unified Modeling Language: Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000, S. Kent and A. Evans (Ed.), LNCS (Lecture Notes in Computer Science), no. 1939, pp. 513-527; Also available as Technical Report (EPFL-DI No 00/332).

[6] J. Kienzle, A. Romanovsky and A. Strohmeier; *Open Multithreaded Transactions: Keeping Threads and Exceptions under Control*. 6th International Workshop on Object-Oriented Real-Time Dependable Systems, Italy, January 2001.

[7] J. Kienzle; *Open Multithreaded Transactions: A Transaction Model for Concurrent Object-Oriented Programming*. Ph.D. Thesis EPFL-DI, no 2393, Swiss Federal Institute of Technology in Lausanne, Software Engineering Lab., 2001.

[8] K. Lano; *Formal Object-Oriented Development*. Springer-Verlag, 1995.

[9] B. Meyer; *Object-Oriented Software Construction*. Second Edition, Prentice Hall, 1997.

[10] OMG Unified Modeling Language Revision Task Force; *OMG Unified Modeling Language Specification*. Version 1.4 draft, February 2001. http://www.celigent.com/omg/umlrtf/

[11] S. Sendall and A. Strohmeier; *UML-based Fusion Analysis*. UML '99 - The Unified Modeling Language: Beyond the Standard, Second International Conference, Fort Collins, CO, USA, October 28-30, 1999, R. France and B. Rumpe (Ed.), LNCS (Lecture Notes in Computer Science), no. 1723, 1999, pp. 278-291, extended version also available as Technical Report (EPFL-DI No 99/319).

[12] S. Sendall and A. Strohmeier; *From Use Cases to System Operation Specifications*. UML 2000 — The Unified Modeling Language: Advancing the Standard, Third International Conference, S. Kent and A. Evans (Ed.), LNCS (Lecture Notes in Computer Science), no. 1939, pp. 1-15; Also available as Technical Report (EPFL-DI No 00/333).

[13] S. Sendall and A. Strohmeier; *Using OCL and UML to Specify System Behavior*. Technical Report (EPFL-DI No 01/359), Swiss Federal Institute of Technology in Lausanne, Software Engineering Lab. 2001; to be published in Lecture Notes in Computer Science, Springer-Verlag.

[14] J. Spivey; *The Z Notation: A Reference Manual*. Prentice Hall, 1989.

[15] A. Strohmeier and S. Sendall; *Operation Schemas and OCL*. Technical Report (EPFL-DI No 01/358), Swiss Federal Institute of Technology in Lausanne, Software Engineering Lab., 2001.

[16] J. Warmer and A. Kleppe; *The Object Constraint Language: Precise Modeling With UML*. Addison-Wesley 1998.