*Journal of Universal Computer Science, vol. 11, no. 5 (2005), 687-702*
*submitted: 30/11/04, accepted: 31/1/05, appeared: 28/5/05 © J.UCS*

# On Atomicity and Software Development

**Jörg Kienzle**
School of Computer Science, McGill University, Montreal, Canada
Joerg.Kienzle@mcgill.ca

**Abstract:** This paper shows how the concept of atomicity can ease the development of concurrent software. It illustrates by means of a case study how atomicity is used to reduce the complexity of concurrency by presenting simplified models or views of the system at certain stages of the development cycle. As the development process goes on, the atomic views from the early stages are refined – broken up into smaller pieces – to slowly introduce concurrency back into the system. Finally, at the design stage, low-level concepts that provide atomicity, such as transaction or monitors, are used to ensure consistent concurrent updating of the application state.
**Key Words:** Atomicity, Software Development, Concurrency, UML, OCL, Transactions, Monitors
**Category:** D.2, D.1.3, D.1.5

## 1 Introduction

The concept of *atomicity*, from the Greek word *atomos – indivisible*, has many meanings in the context of computer science. It has been used in the hardware community to designate indivisible processor instructions. Implementors of higher-level constructs, such as semaphores, rely on atomic machine instructions to build efficient synchronization primitives to be used in concurrent programming.

In the database world, *transactions* group a set of operations together and execute them with the so-called *ACID properties* – *A* standing for atomicity. From the perspective of the caller of a transaction, the execution of the transaction appears to move the system from its initial state directly to the result state, without any observable intermediate state. If, however, the transaction can not be completed for some reason, for instance, because of the failure of some component, it appears as though the system had never left the initial state. Atomicity guarantees that either all operations of a transaction are execute successfully, or none is. Combined with structured exception handling, atomicity can be used to confine erroneous information, and hence facilitates the provision of fault tolerance [Romanovsky, 1999].

Many researchers rely on the concept of atomicity in developing structuring approaches for system design. In this context, the execution of atomic units is indivisible, and hence they provide an elegant way to encapsulate state and behavior. No intermediate execution results can be seen from the outside. This facilitates reasoning about the system, system understanding, verification and development. For instance [Best, 1996, Kurki-Suonio and Mikkonen, 1998] show that concurrent object-oriented systems are easier to understand and to analyze if their execution is built out of atomic units encapsulating several objects and method calls. The design of process-oriented systems can also benefit from using atomic actions (see section 5 for more details).

This paper shows how atomicity can simplify the development of concurrent software throughout all stages of software development. The main focus of the approach is to avoid interference of system operations. The ideas are presented in the context of an object-oriented software development method called Fondue, and illustrated by means of a case study application – an on-line auction system. The paper is structured as follows. Section 2 introduces object-oriented software development in general, the Fondue method in particular, and the auction system case study. Section 3 shows how atomicity can be used by a developer to present simplified views of the system when specifying its behavior during analysis, and how these models can be refined to gradually introduce concurrency. Section 4 presents how low-level concepts such as transactions and monitors are used to structure concurrent execution at the design level, and finally section 6 draws some conclusions.

## 2 Object-Oriented Software Development

Object-orientation is a way of thinking about problems. It is an approach to viewing the world and building software in terms of objects. Object-orientation is built upon well established principles, namely *abstraction*, *information hiding*, *modularity* and *classification*. These principles are achieved using the notion of *objects* and *classes*.

Software development methods are well-defined processes that lead a development team from the requirements elicitation process, over analysis, architecture, design to implementation. Object-oriented ideas can be applied throughout all these phases of software development [Meyer, 1997]. Several object-oriented software development methods have been developed. Popular methods include OMT [Rumbaugh et al., 1991], Booch [Booch, 1994] and the newer unified process [Jacobson et al., 1999].

### 2.1 The Fondue Method

To illustrate how the use of atomicity can simplify the development of concurrent software, this paper presents a case study that uses the object-oriented software development method Fondue , developed at the Swiss Federal Institute of Technology, Lausanne (EPFL) [Sendall and Strohmeier, 1999]. Fondue uses a consistent approach to cover all development phases, from requirements elicitation on to analysis, design and implementation. Fondue has its origins in the well-known Fusion method [Coleman et al., 1994]; it adopts its process, but uses the UML notations. In addition to Fusion, Use Cases are proposed for requirements elicitation and are taken into account during the analysis phase [Sendall and Strohmeier, 2000]. The Fondue method not only provides an internal view of the class model and the behavior of individual classes, but it includes modeling of system-wide functionality and a step-by-step process that leads the development team from an initial requirements document through to the implementation of an object-oriented software system. Fondue defines a number of deliverables: a Domain Model and a Use Case Model during requirements elicitation, an Environment Model, a Concept Model, a Protocol Model and an Operation Model during analysis, a Design Class Model, an Inheritance Model, an Interaction Model and a Dependency Model during design, and an Implementation Class Model during implementation. For more details on Fondue, the interested reader is directed to [Sendall, 2002].

Some of these models, in particular the Environment Model, the Operation Model and the Interaction Model, make use of atomicity to reduce the system complexity. The main advantage of this is that atomicity drastically reduces the system complexity. Abstracting away from complex interactions, the atomic models present a simpler view of the system, which allows the developer to focus on the essential system functionality. As the development process goes on, the atomic views presented in one model are refined, or broken up into smaller pieces, to slowly introduce concurrency back into the system.

### 2.2 The Auction System Case Study

The auction system case study used in this paper, adapted from [Kienzle, 2003], allows members to negotiate over the buying and selling of goods using English style auctions. The application is similar to Internet auction sites such as *eBay* (www.ebay.com). The main difference is that members debit their credit card to deposit money into an account controlled by the auction system itself. That way, the auction system can guarantee that bidders always have enough money to pay for their bids.

From a concurrency point of view, developing an auction system is non trivial. It is a highly dynamic system, featuring competitive and collaborative concurrency. The concurrency stems from the fact that a customer can participate in multiple auctions simultaneously, and that the system must be able to serve multiple customers at a given time.

## 3 Atomicity in Analysis

### 3.1 The System as an Indivisible Black-Box

Fondue is designed for developing reactive systems. Every change to the system state is caused by the execution of a system operation. A system operation is triggered by an event that is generated with the reception of an input message sent by some actor or by an internally generated time event.

One of the first steps when specifying the behavior of a system using Fondue is to establish the *Environment Model*. At this stage, the system is viewed as a black-box, as an indivisible, atomic component. Details of the system are hidden – the focus is on defining the system boundaries, and showing how the system interacts with its environment. The environment is represented by a set of actors, which are autonomous entities external to the system. For example, humans that interact with the system under development are represented by actors, but also sensors or other computerized systems.

The environment model of the auction system case study is shown in Fig. 1. There are two external actors: the *User* and the *CreditInstitution*. The input events sent to the system by the customer and the credit institution, as well as the output events generated by the system are depicted as asynchronous messages. When receiving an input event, the system processes it by executing the corresponding *system operation*.

From a concurrency point of view, the interesting information in this diagram is the multiplicity of the actors[1]. The diagram states that there can be any number of customers interacting with the system at a given time. Each customer is autonomous, and can spontaneously send input messages to the system. This is an example of inherent concurrency in the environment. Our system must be capable of handling requests issued by different customers concurrently.

### 3.2 Opening the Black-Box

Now that the system boundary has been determined, the system black-box view is abandoned. As a next step, the conceptual state of the system is specified in the *Concept Model*. It takes the form of a UML class diagram. Exactly how the Concept Model is constructed is out of the scope of this paper, but it is based on the Use Case Model and the Domain Model obtained during requirements elicitation. The Concept Model contains all information required for the purpose of fulfilling the system's responsibilities over time, i.e., the necessary information to process an input message or send out a notification.

Fig. 2 depicts the complete concept model for the *AuctionSystem*.

[1] In UML, the multiplicity on the actors requires that they are in a composition association. The underlying container is the environment, which is implicit and thus not shown for reasons of conciseness.

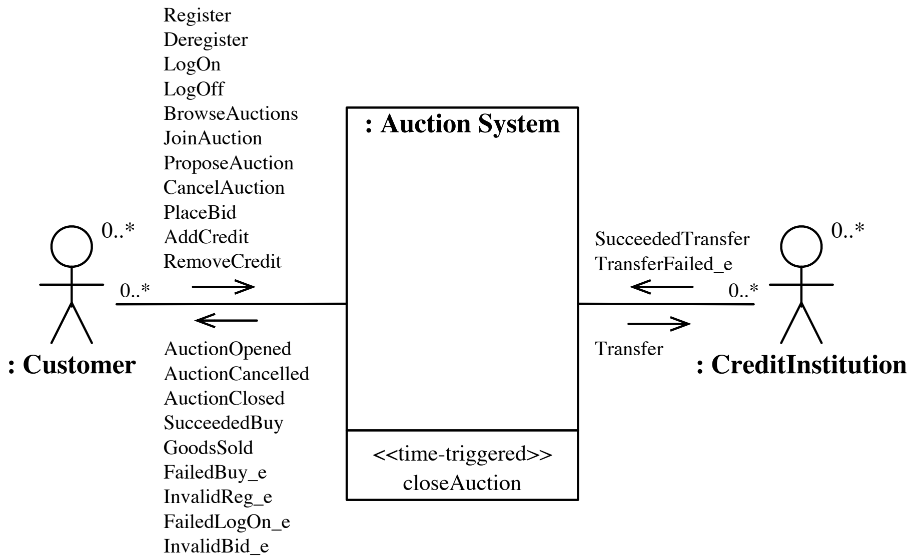


**Figure 1:** Environment Model of the Auction System

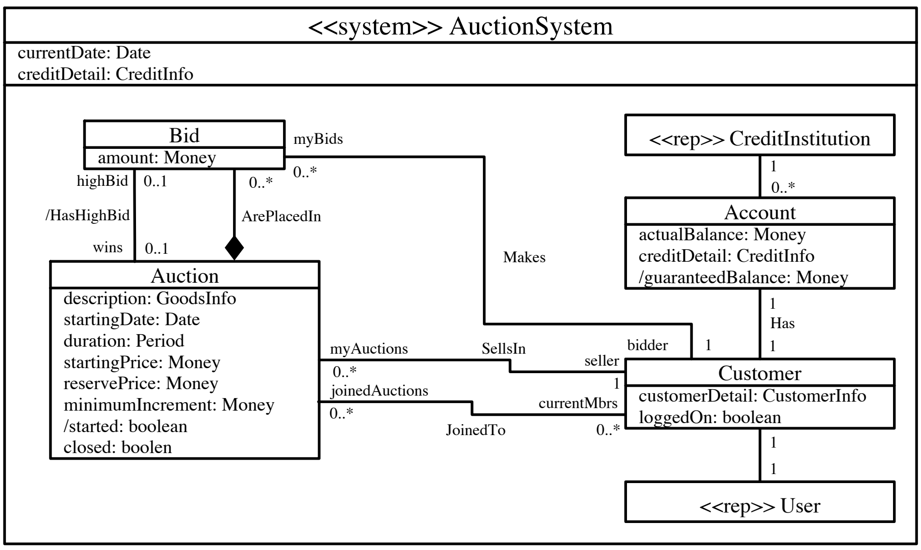


**Figure 2:** Concept Model of the Auction System

Classes stereotyped with `<<rep>>` are classes whose instances represent external actors. In the auction system, a user object is instantiated for each real-world user, and a credit institution object is created for each credit institution that interacts with the system.

One interesting detail is the derived attribute *guaranteedBalance*. In our system we want to guarantee that a bidder can always pay for his bids. The guaranteed balance represents the maximum amount that a customer has available for bidding. It can be calculated by taking the actual balance of his credit with the auction system and subtracting all high bids he has placed in active auctions. Using OCL, the derived attribute is defined in the following way:

**context**: Account::guaranteedBalance : Money
**derive**: self.actualBalance - self.customer.myBids→**select**
(b | b.wins→**exists**(a | **not** a.closed))→**sum**(amount);

### 3.3 Each System Operation as an Indivisible Operation

Since we are developing a reactive system, internal state changes are always triggered by external stimuli. Every input event sent to the system triggers the execution of a corresponding system operation. The complete behavior of the system under development can therefore be specified by describing the effects of each system operation on the conceptual state. In Fondue this is done in the *Operation Model*.

In the Operation Model, each system operation is described in a separate operation schema. As a first simplification step, system operations are considered to be executed *atomically*: they lead the system from some state that satisfies the operation's precondition to a state that satisfies the operation's postcondition. In other words, the execution of a system operation is *instantaneous*: no intermediate state can be observed from the outside. This means also that there can be no interference between system operations: since they execute instantaneously, they are always processed in some sequential order. Hence, it is not necessary to address concurrency issues yet; the developer can focus on the conceptual state changes only.

For the auction system, 13 operation schemas have to be written, one for each of the input messages shown in the environment model (Fig. 1). For space reasons we are going to concentrate on one operation only: the operation *placeBid*. Its operation schema is shown in Fig. 3.

The first line, starting with *Operation*, specifies the context of the operation, here the *AuctionSystem*, the name of the operation, and the parameters.

The *Scope* clause lists all those classes and associations from the concept model that define the name space of the operation, i.e. all classes and associations that are used in the following pre- and post-conditions. The *placeBid* operation

**Operation**: AuctionSystem:placeBid
(a: Auction, c: Customer, bidAmount: Money);

**Description**: A customer requests to place a bid in the given auction. The system must decide whether the bid is valid (i.e. higher than the current bid, above the increment, and that the customer is solvent), and if so make the bid the current high bid.

**Scope**: Auction; Bid; Customer; Account; ArePlacedIn; Makes; Has; HasHighBid; JoinedTo;

**Messages**:¨User::{InvalidBid_e};

**New**: newBid: Bid;

**Pre**: a.currentMbrs→**includes**(c) & a.started & **not** a.closed;

**Post**: **if** bidAmount≥a.highBid.amount + a.minimumIncrement **then**
  **if** c.account.guaranteedBalance≥bidAmount **then** ⋆
   newBid.**oclIsNew**(bidAmount) &
   a.bid→**includes**(newBid) &
   c.myBids→**includes**(newBid)
  **else**
   **sender**^invalidBid_e(Reason::insufficientFunds)
  **endif**
 **else**
  **sender**^invalidBid_e(Reason::bidTooLow)
 **endif**

**Figure 3:** Sequential Operation Schema for *placeBid*

uses the *Customer*, *Bid* and *Account* classes, and navigates through *ArePlacedIn*, *JoinedTo*, *Makes* and *Has*.

The *Message* clause declares the possible output messages that can be output with the execution of the operation. The type and the destination actors of the messages must be specified. In our example, an *invalidBid_ e* exception message might be propagated to the calling user.

The *New* clause provides a declaration of all those names in the operation schema that refer to concept objects of the system that are possibly created with the execution of the operation. These objects are declared to be new in the post clause using the predefined operation *oclIsNew*. The *placeBid* operation potentially creates a new bid.

The *Pre* clause contains an OCL predicate that defines the assumed state of the system and / or parameters before the execution of the operation. In the

| | proposeAuction | joinAuction | placeBid | cancelAuction | closeAuction |
|---|---|---|---|---|---|
| proposeAuction | - | N | N | N | N |
| joinAuction | N | Y | Y | N | Y |
| placeBid | N | Y | Y | N | Y |
| cancelAuction | N | N | N | N | N |
| closeAuction | N | Y | Y | N | N |

**Table 1:** Concurrent Input Events of Auctions

example, the pre-condition states that in order to place a bid, the customer that wants to place the bid must be already joined to the auction, and the auction must be started and not closed.

Finally, the *Post* clause defines the required state of the system after the execution of the operation. Only changes to the conceptual system state must be mentioned here, any unmentioned state remains the same. The post-condition asserts that if the customer has enough money (the *guaranteedBalance* is used here in order to ensure that the customer can pay for all pending bids (see $\star$ in Fig. 3) and the bid is higher than the current highest bid plus the minimum increment, then the bid is made. Otherwise, the user is informed of the exceptional outcome of his request.

### 3.4 Refining Atomicity of System Operations

The assumption made in the previous section, namely that system operations execute instantaneously, is of course not realistic: as soon as time is taken into consideration, the execution of system operations might overlap. Fortunately, not all operations can occur simultaneously due to constraints of the environment and of the problem domain. For example, a single customer can not send multiple concurrent messages, or, an auction can not be closed before it is started. In Fondue, the sequencing of input messages can be specified using UML state diagrams in the *Protocol Model*.

For space reasons, the Protocol Model for the auction system has been omitted. The essential concurrency information for the *placeBid* operation has been extracted and presented in a table shown in Table 1.

The table states that the *placeBid* operation might execute concurrently with other *placeBid*, *joinAuction* and *closeAuction* operations. [2]To take this into account, the sequential operation schema has to be elaborated.

[2] *CancelAuction*, for instance, can never execute concurrently with *placeBid*, because once an auction starts, it can not be cancelled anymore.

The key issue is to identify the conceptual state that is accessed concurrently. The *Scope* section of the sequential operation schema specifies all conceptual state that an operation accesses. For *placeBid*, the accessed concepts are *Auction*, *Bid*, *Customer*, *Account*, and the relations *ArePlacedIn*, *Makes*, *Has*, *HasHighBid* and *JoinedTo* (see Fig. 3). From the protocol model we know that *placeBid* potentially runs concurrently with a *placeBid* issued by a different user. However, the *Bid* concept is not shared, since each *placeBid* creates a new bid! Likewise, the *Customer* is not shared, since each *placeBid* is issued by a different customer. Hence, the *Makes* relation is not shared either. The *started* and *closed* attributes of the *Auction* concept are accessed in "read-mode" only, so we do not have to worry about them for now. The same argument holds for *JoinedTo*. *ArePlacedIn*, however, is modified, since each new bid is added to the list of bids of an auction. *HasHighBid* is also updated concurrently. *Account* is the most tricky one. One might think that it is not shared, since each *placeBid* operation accesses the account of the customer that places the bid only. But this is not true. The *placeBid* operation modifies the *HasHighBid* relation, and therefore modifies the *guaranteedBalance* of the account of the customer that previously was holding the highest bid. The *Account.guaranteedBalance* concept is therefore shared as well.

We must now take into account that the *placeBid* operation also runs concurrently with *joinAuction*. *joinAuction* (operation schema not shown for space reasons) modifies the relation *JoinedTo*, and *placeBid* consults this relation, so *JoinedTo* is shared as well.

At any time, the time-triggered event *closeAuction* might fire, resulting in closing the auction. *placeBid* consults the closed attribute, so it is shared as well.

Once the shared concepts have been identified, they are recorded in a new section of the operation schema entitled *Shared*, as shown in Fig. 4.

Next, the precondition has to be re-examined. In the sequential version, where every operation executes atomically, a precondition such as *not a.closed* is sufficient to guarantee that a bid is placed while the auction is still active. This is different if we consider concurrency. We have to make sure that the auction does not close while we are processing the bid. In implementation terms, this can be achieved by either preventing the auction from closing while there is still bidding activity, by preventing bids if the auction is about to close, or else by undoing whatever partial bid has been made if ever the auction is to close during a bid. Since we do not want to unduly constrain the solution space in the analysis phase, we must allow for all possible (correct) outcomes.

In terms of changes to the operation schema, we remove the precondition *not a.closed* and move it, encapsulated by a *rely statement* [Jones, 1983], to the post condition section (see Fig. 4). The rely statement `rely A then B fail C endre` asserts that either the condition *A* was true during the realization of all the state

**Operation**: AuctionSystem:placeBid
(a: Auction, c: Customer, bidAmount: Money);

**Description**: A user requests to place a bid in the given auction. The system must decide whether the bid is valid and if so make the bid the current high bid.

**Scope**: Auction; Bid; Customer; Account; ArePlacedIn; Makes; Has; HasHighBid; JoinedTo;

**Shared**: Account.guaranteedBalance; Auction.closed; HasHighBid; ArePlacedIn; JoinedTo;

**Messages**:¨User::{InvalidBid_e};

**New**: newBid: Bid;

**Pre**: a.currentMbrs→**includes**(c) & a.started;

**Post**: **rely** **not** a.closed **then**
  **rely** bidAmount≥a.highBid.amount + a.minimumIncrement **then** ⋆1
    **rely** c.account.guaranteedBalance≥bidAmount **then** ⋆2
      newBid.**oclIsNew**(bidAmount) &
      a.bid→**includes**(newBid) &
      c.myBids→**includes**(newBid)
    **fail**
      **sender**ˆinvalidBid_e(Reason::insufficientFunds)
    **endre**
  **fail**
    **sender**ˆinvalidBid_e(Reason::bidTooLow)
  **endre**
**fail**
  **sender**ˆinvalidBid_e(Reason::auctionClosed)
**endre**

**Figure 4:** Concurrent Operation Schema for *placeBid*

changes specified in $B$, or else the state changes specified in $C$ have been realized. In our example this translates to: either a bid has been successfully placed and during that time the auction did not close, or the *auctionClosed* message has been sent to the customer that has requested to place a bid.

It is also important to note that the if statements in the sequential operation schema have been transformed into rely statements to take into account the interpretation for shared resources. For example, we must make sure that no other *placeBid* operation modifies the highest bid while we are accepting a new high bid ($\star$1), and that we continuously have sufficient funds according to our guaranteed balance ($\star$2).

It is interesting to note here also that, when introducing concurrency, we had to add the sending of a new error message *auctionClosed* to the operation schema. In the sequential version, such a message was not necessary.

## 4 Designing with Atomicity

During design, a blue print of a solution that satisfies the requirements defined in the analysis phase must be devised. In object-oriented design, the conceptual state has to be mapped to objects, and then the developer has to decide how the conceptual state changes specified in every system operation are to be implemented by interacting objects at run-time. In Fondue this is done in the *Interaction Model*.

When designing concurrent systems, an additional aspect has to be considered: shared state. In object-oriented programming languages that support concurrency, like for example Java [Gosling et al., 1996] or Ada [ISO, 1995], consistent access to shared data is usually provided by *monitor* objects [Hoare, 1974]. They provide *atomic*, or in this case *uninterruptable*, execution of methods. Changes made by different threads simultaneously are serialized, and hence executed in *isolation*.

In addition to protecting shared state, we must make sure that the *rely* conditions stated in the concurrent operation schema hold during the execution of the respective state changes they belong to. In our example, for instance, we must make sure that the auction does not close, that the bid is higher than the current bid, and that there is enough money in the customer's account, *while* a bid is placed. In other words, the checking and the changing must be made *atomic*.

There are essentially two different ways of achieving isolation and atomicity during execution: using transactions or using monitors.

### 4.1 Transaction-Oriented Design

If the application modifies sensitive or important data, data that persists, or data that must be kept consistent even in the presence of crash failures, then transactions [Gray and Reuter, 1993]should be used to regulate access to shared objects. As mentioned in the introduction, a transaction groups together a set of operations, and gives them the so-called ACID properties. *Atomicity* – either all operations are executed, or none is; consistency – transactions move the application from one consistent state to another one; isolation – concurrently

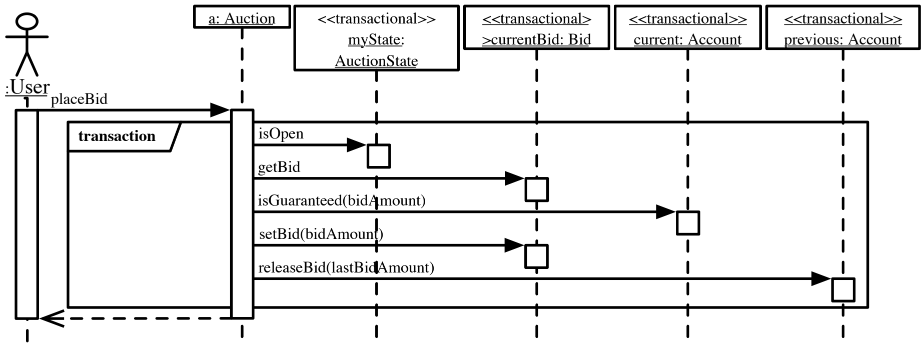


**Figure 5:** Transaction-based Execution of *placeBid*

executing transactions do not see intermediate results of other transactions; and durability – state changes made by a transaction are recorded on stable storage.

The transaction-oriented design of the *placeBid* operation is shown in a sequence diagram in Fig. 5. In order to provide maximum concurrent execution, the auction state and the current bid have been encapsulated in separate objects.

When executing *placeBid*, the auction object starts a new transaction. This is shown in the sequence diagram by a frame labelled *transaction*. As a first step, the auction state is checked. Then, the validity of the bid is checked. Third the bid is deducted from the account, and the current bid is updated. Finally, the account of the previous bidder is credited. All these operations are executed as part of the transaction.

*AuctionState*, *Bid* and *Account* are transactional objects, shown in the diagram by the $<<$*transactional*$>>$ stereotype. Their state is made persistent, i.e. it can even survive crash failures. If any one of the conditions is not satisfied, or if any failures occur during the execution of *placeBid*, the transaction will be rolled back, i.e. all state changes made so far are undone. It can never happen, for instance, that a bid is placed without crediting the account of the previous bidder. Thanks to the isolation property, no other operations will be affected in case of a rollback. Interestingly, the actual way of ensuring isolation is still not specified. It depends on the kind of concurrency control that is used by the underlying transaction support.

In pessimistic, lock-based concurrency control [Gray and Reuter, 1993], shared resources are locked once they have been accessed, and the lock is only released when the transaction ends. In our example, this means that, for instance, the *closeAuction* operation would be blocked until all pending *placeBid* operations have terminated and released their locks on the auction state. On the other hand, optimistic concurrency control [Kung and Robinson, 1981], such as time-stamp

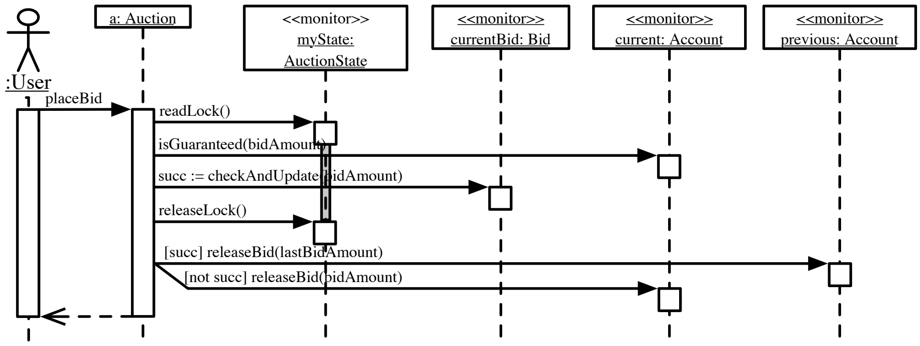


**Figure 6:** Monitor-based Execution of *placeBid*

based versioning, might decide to let the auction close, and abort all concurrently executing *placeBid* operations.

### 4.2 Monitor-Based Design

Transactions require extensive run-time support, and slow down execution significantly. If persistence and tolerance to crash failures is not needed, then a simple monitor-based design can provide the same behavior, with considerably better performance.

The monitor-based design is similar to the transaction-based design. Transactional objects are now monitors, i.e. their methods provide multiple readers / single writers semantics (*synchronized* methods in Java, *protected objects* in Ada). The atomicity needed for implementing the rely conditions is achieved by acquiring read or write locks when checking the condition (similar to lock-based pessimistic concurrency control[3]). A lock prevents other threads from changing the condition while the operation executes. After the state changes that rely on the condition, the locks are released again.

The monitor-based design of *placeBid* is shown in Fig. 6. *AuctionState*, *Bid* and *Account* are now monitors, highlighted in the sequence diagram by the $<<$monitor$>>$ stereotype. When checking the auction status, a read lock is acquired (shown in the figure by a dotted gray activation rectangle ). This would block an attempted concurrent *closeAuction* operation (which would have to acquire a write lock).

An informal analysis reveals that, since the same customer can not place two bids simultaneously, or try to remove credit while placing a bid, the balance of

[3] If read and write locks are not provided by the programming language, they can easily be implemented on top of semaphores.

a customers account can only grow while the *placeBid* operation is executing[4]. Therefore we do not have to acquire a lock to guarantee the balance when accessing the account of the customer that is placing the bid. We can simply check and withdraw the bid amount from the account in one operation (which itself is atomic because accounts are monitors), and then in a similar way check and update the current high bid. Subsequently, we release the read lock on the auction state, and finally release the bid of the previous high bidder. Alternatively, if the bid is invalid, the money has to be put back on the bidders account.

## 5 Related Work

The approach presented in this paper essentially deals with concurrency at the object-level. An alternative approach proposed in [Zorzo et al., 1999] uses the concept of Coordinated Atomic Actions (CA Actions) [Xu et al., 1995] to structure the execution of a safety-critical production cell system.

In CA action-based design, an application is composed of a set of cooperating processes. Processes that want to work together will enter a CA action. Inside the action, the processes can freely communicate with each other, i.e. exchange messages or work on local shared data structures. Objects external to the action can also be accessed in a transactional way. To the outside, the execution of a CA action looks like a transaction, i.e. no intermediate system state is visible: the action executes atomically. If an exception occurs during the execution of a CA action (due to the detection of erroneous state or behavior), then all the participants of the action are involved in cooperative recovery.

The CORRECT project is currently investigating possible ways of extending UML to support CA action-driven development [Guelfi et al., 2004].

Similar to the approach presented in this paper, but focusing more on design, COMET (Concurrent Object Modeling and architectural design mEThod) [Gomaa, 2000] is a development method for concurrent applications, with a particular emphasis on distributed and real-time applications. COMET also starts requirements elicitation with use cases, which are then refined during analysis. During design, the architecture of the system is elaborated, i.e. the system is divided into subsystems, and finally classes, objects and relationships are defined. For concurrent systems, such as real-time or client-server applications, concurrent tasking concepts, such as synchronization and communication, are considered as well.

[4] The guaranteed balance can grow during the operation *placeBid* if, for instance, a customer $A$ bids in auction $a$, and then, while bidding in auction $b$, a customer $B$ overbids $A$ in $a$.

## 6 Conclusion

Modern applications must respond to an increasing amount of demands. Distributed systems, systems serving hundreds of clients simultaneously, systems that interact with real-time devices, or systems that provide interactive user interfaces are forced to operate in a concurrent environment. Complex concurrent and interacting activities, however, make the development, i.e. understanding, analyzing, designing, and implementing, of such systems extremely difficult.

This paper shows that the use of atomicity can considerably ease the development of concurrent software. Atomicity allows the developer to selectively hide the complexity of concurrency by presenting simplified models or views of the system at certain stages of the development cycle. In the auction system example, atomicity is used when establishing the system boundaries and specifying the input and output messages in the Environment Model. Atomicity is considered during initial system operation specification in the Operation Model to abstract away the complexity of concurrency. As the development process goes on, the Operation Model is refined – the system operations are broken up into smaller pieces – to slowly introduce concurrency back into the system. Finally, at the design stage, low-level concepts that provide atomicity, such as transaction or monitors, are used in the Interaction Model to ensure consistent concurrent updating of the application state.

## Acklowledgements

I would like to thank the participants of the Dagstuhl Seminar 04181 for the interesting discussions and feedback. The work presented here has been partially supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

## References

[Best, 1996] Best, E. (1996). *Semantics of Sequential and Parallel Programs.* Prentice Hall, New York, NY.

[Booch, 1994] Booch, G. (1994). *Object-Oriented Analysis and Design with Applications.* Benjamin Cummings, Redwood City, 2nd edition.

[Coleman et al., 1994] Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., and Jeremaes, P. (1994). *Object-Oriented Development: The Fusion Method.* Prentice-Hall, Englewood Cliffs.

[Gomaa, 2000] Gomaa, H. (2000). *Designing Concurrent, Distributed, and Real-Time Applications with UML.* Addison–Wesley.

[Gosling et al., 1996] Gosling, J., Joy, B., and Steele, G. L. (1996). *The Java Language Specification.* The Java Series. Addison Wesley, Reading, MA, USA.

[Gray and Reuter, 1993] Gray, J. and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, San Mateo, California.

[Guelfi et al., 2004] Guelfi, N., Razavi, R., Romanovsky, A., and Vandenbergh, S. (2004). Drip catalyst: An mde/mda method for fault-tolerant distributed software families development. In *OOPSLA & GPCE 2004 Workshop on Best Practices for Model Driven Software Development, Vancouver, Canada, 2004*.

[Hoare, 1974] Hoare, C. A. R. (1974). Monitors: An operating systems structuring concept. *Communications of the ACM*, 17(10):549 – 557.

[ISO, 1995] ISO, editor (1995). *International Standard ISO/IEC 8652:1995(E): Ada Reference Manual*. Number 1246 in Lecture Notes in Computer Science. Springer Verlag.

[Jacobson et al., 1999] Jacobson, I., Rumbaugh, J., and Booch, G. (1999). *The Unified Software Development Process*. Object Technology Series. Addison–Wesley, Reading, Massachusetts, USA.

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

[Kienzle, 2003] Kienzle, J. (2003). *Open Multithreaded Transactions — A Transaction Model for Concurrent Object-Oriented Programming*. Kluwer Academic Publishers.

[Kung and Robinson, 1981] Kung, H. T. and Robinson, J. T. (1981). On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213 – 226.

[Kurki-Suonio and Mikkonen, 1998] Kurki-Suonio, R. and Mikkonen, T. (1998). Liberating object-oriented modeling from programming-level abstractions. (1357):195 – 199.

[Meyer, 1997] Meyer, B. (1997). *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ 07632, USA, 2nd edition.

[Romanovsky, 1999] Romanovsky, A. (1999). On structuring cooperative and competitive concurrent systems. *The Computer Journal*, 42(8):627 – 637.

[Rumbaugh et al., 1991] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, New Jersey, USA.

[Sendall, 2002] Sendall, S. (2002). *Specifying Reactive System Behavior*. PhD thesis, Swiss Federal Institute of Technology, Lausanne, Switzerland.

[Sendall and Strohmeier, 1999] Sendall, S. and Strohmeier, A. (1999). Uml-based fusion analysis. In *UML'99, Fort Collins, CO, USA, October 28-30, 1999*, number 1723 in Lecture Notes in Computer Science, pages 278–291. Springer Verlag.

[Sendall and Strohmeier, 2000] Sendall, S. and Strohmeier, A. (2000). From use cases to system operation specifications. In Kent, S. and Evans, A., editors, *UMLÕ2000 - The Unified Modeling Language: Advancing the Standard, York, UK, October 2-6, 2000*, number 1939 in Lecture Notes in Computer Science, pages 1–15. Springer Verlag.

[Xu et al., 1995] Xu, J., Randell, B., Romanovsky, A., Rubira, C. M. F., Stroud, R. J., and Wu, Z. (1995). Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *FTCS-25: 25th International Symposium on Fault Tolerant Computing*, pages 499 – 509, Pasadena, California.

[Zorzo et al., 1999] Zorzo, A. F., Romanovsky, A., Xu, J., Randell, B., Stroud, R. J., and Welch, I. S. (1999). Using coordinated atomic actions to design safety-critical systems: a production cell case study. *Software - Practice & Experience*, 29(8):677 – 697.