STATIC VERIFICATION OF CONCURRENT SYSTEM DESIGN

by

Xun Zhu

School of Computer Science
McGill University, Montréal

June 2007

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Abstract

Elaborating a correct design of a concurrent system is extremely difficult. In part this is due to the infinite number of possible system runtime behaviors that result from the concurrent (or pseudo-concurrent) execution of interacting processes or threads. Data consistency, deadlock, starvation and fairness issues are the most well-known problems encountered in concurrent systems. Accurate concurrent system design verification approaches require an expensive system runtime behavior analysis and consequently result in prohibitively high development costs (i.e. for testing).

In this thesis we try to address this problem by presenting several static approaches that can help the developer of a concurrent system during the design phase. In the first part of the thesis we present an approach that can analyze an existing concurrent system design to detect potential deadlock situations. This is done by mapping object interaction diagrams such as sequence diagrams to *System Synchronization Hasse diagrams*, which are then analyzed to detect deadlock cycles. Since the approach is static, it is overly pessimistic, meaning that it is possible that the algorithm detects a deadlock that, in reality, cannot occur. On the other hand, if the algorithm cannot detect any deadlocks, the developer can be sure that the design is deadlock-free.

In the second part of the thesis we show how a concurrency-enriched specification can be transformed into a system application-level consistent design. The approach starts with concurrency-aware OCL-based operation schemas that describe all system functionality using pre-, rely-, and post-conditions. These schemas are then mapped to *Rely diagrams*. Based on the rely diagrams, sequence diagrams describing the concurrent system design are elaborated. The approach uses locks to ensure consistency and deadlock freedom. We then further show how these locks can be used to enforce certain fairness policies.

The usefulness of our approach is demonstrated by applying it to the design of a non-trivial case study: an online auction system. We first illustrate an original online auction system design that is built according to user requirements elicitation and analysis using the object-oriented development process Fondue; then we check the original design for deadlocks using our first approach. We subsequently modify the original design based on the results.

Résumé

L'élaboration d'une bonne architecture dans un système concurrent est une tâche extrêmement difficile. Ceci s'explique par un nombre infini de combinaison de comportement d'exécution possible causé par l'interaction des processus exécutant de façon concurrente (ou pseudo-concurrente). La consistance de donnée, l'interblocage, l'insuffisance de ressources et le partage juste des ressources sont les problhmes les mieux connus dans les environnements concurrents. La vérification exacte d'une architecture de système concurrent est une solution très dispendieuse, nécessitant des analyses de comportement d'exécution qui s'avère beaucoup trop coûteux.

Cette thèse adresse ce problème en proposant plusieurs techniques statiques qui peuvent aider un développeur d'un système concurrent pendant la phase d'analyse. Dans la première partie de cette thèse, nous présentons une technique qui permet d'analyser l'architecture d'un système concurrent existant pour y découvrir des situations potentielles d'interblockage. Cette analyse nécessité le mappage de diagrammes d'interaction d'objet, tel que les diagrammes de séquence, à des diagrammes de Synchronisation de Systhme Hasse, qui peuvent être analysés à leur tour pour y découvrir des interblockages. Puisque la technique est statique, elle est sur-pessimiste. Donc, elle peut trouver des interblockages qui n'existe pas vraiment. Cependant, si aucun interblockage est trouvé, alors le programmes peut être assuré qu'aucun interblockage ne s'y trouve.

Dans la deuxième section de la thèse, nous démontrons comment des spécifications munies d'information concurrent, peut être transformées en spécification sans problème d'interblockage. Cette technique débute avec des schémas opérationnels OCL munies d'information concurrent, qui décrivent toutes les fonctionnalités du système avec les pré-conditions, les poste-conditions et les conditions de dépendance. Ces schémas sont alors mappés à des diagrammes de dépendance. En utilisant les diagrammes précédents, des diagrammes de séquences décrivant l'architecture du système concurrent sont élaborés. Cette technique utilise des verrous pour assurer un état consistant et l'absence d'interblockage. Nous montrons également comment ces verrous peuvent être utilisés pour assurer les politiques justes.

Le potentiel de notre technique est illustré par son application à l'architecture d'une étude de cas : un système en-ligne d'encan. Premièrement, nous expliquons l'architecture original du systhme d'encan qui a été construit à l'aide d'une analyse des pré-requis en utilisant le processus de développement Fondue. Ceci nous permet d'évaluer l'architecture originale pour la présence d'interblockage, en utilisant notre première technique. Nous pouvons alors utiliser les résultats de l'analyse pour améliorer l'architecture originale.

Acknowledgements

First, I would like to thank my two supervisors (names in alphabetical order): Professor Clark Verbrugge and Professor Jörg Kienzle. This work would not have been done without their constant help, guidance and encouragements throughout my time at McGill. I learned a lot from Clark's insights in concurrency, Jörg's knowledge of high level software analysis and design and the advice from both of them. I appreciate the time we three spent together for the discussion and research.

I would like to thank Professors Clark Verbrugge, Jörg Kienzle, Bettina Kemme and Muthucumaru Maheswaran for the courses they taught in concurrent programming, software design, distributed systems and networking, which established my knowledge basis and interest for this work. I would also like to thank Professor Hans Vangheluwe for his advice on the Software Modeling Analysis and Professor Rob Pettit from George Mason University for his suggestions of COMET development method. Besides, I am very grateful to many professors and the Co-op program at the University of Victoria, which provided me a solid foundation of Computer Science and Mathematics during my undergraduate studies.

Meanwhile, I would like to use this opportunity to thank Laurie Hendren, Clark Verbrugge and Jörg Kienzle for providing very nice research environments in the Sable Research lab and Software Engineering lab, where most of this work has been completed. Additional thanks go to all members from both Sable Research group and Software Engineering group, from which I made a lot of friends. Particularly, I thank Alexandre Denault for his help on the French translation of my abstract, and Haiying Xu for the laughs she brought to the office.

Finally, I would like to give my special thanks to my parents and my cousin sister for their constant love, support, encouragement and patience during my best and difficult times.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1
Introduction

Modern applications must meet an increasing number of demands. Many of these applications are implemented by using concurrent systems due to the fact these systems can serve hundreds of clients simultaneously or they can provide interactive user interfaces. However, concurrent systems are notoriously difficult to design and implement. They can contain errors that are extremely hard to find. There is a large body of work on approaches to detect design errors in concurrent systems; many of them require an expensive runtime behavior analysis after the system's implementation. Some system design issues such as *deadlock*, *system state inconsistency* and *fairness/liveness issues* can cost way more to fix if we detect them after the implementation phase, instead at the design phase. Therefore, we need some efficient, general methods for verifying the concurrent system design before its implementation.

1.1 Motivation

It is evident that practical and truly general approaches to verify concurrent system design do not exist; algorithms capable of dealing with the myriad of diverse possibilities would simply be much too computationally expensive to be effective or are unsolvable. It is our contention, however, that general methods *do* exist for the conservative verification of the system design: we do not need to consider all possible runtime behaviors; instead, a static system design diagram is enough to provide the "reasonably" accurate verification results.

This is the focus of this thesis: by constructing some system design diagrams, we are able to conduct some general verification methods at the system's design phase; as a consequence, the overall cost for the concurrent system development is reduced.

1.2 Contributions

The primary contribution of our work has been to develop several general approaches which we are certain can be efficiently used to overcome the following challenges inherent in the design of the concurrent system.

Deadlock in the concurrent system design

A problem of increasing importance in the design of concurrent systems is the deadlock. This fact arises from the increasing utilization of system resources. Utilizing system resources by distributing them among many concurrently running processes is very common in concurrent systems. But since the processes in concurrent systems can interact with each other while they are executing, the number of possible execution paths can be extremely large, and as a consequence the resulting resources utilization among different processes can be very complex. This complexity in resources sharing can sometimes cause deadlock problems that system developers are not even aware of. In this thesis, we will explain the nature of deadlock state; survey the work that has been done to prevent or avoid deadlock; and propose our own algorithm that helps to guarantee deadlock-free property of a concurrent system design.

System state inconsistency in the concurrent system design

System state inconsistency is another problem in the concurrent system. This kind of problem arises from improper system operation design. A system operation performs several state changes that move the system from one consistent state (that verifies a set of pre-conditions) to a new consistent state (that verifies a set of post-conditions). A concurrent system, however, must be designed to run multiple operations concurrently. In this case, the effect of running one operation can change the system's state in a negative way with respect to the other concurrently running operation. This negative influence from one operation to the other is extremely difficult to discover if we only monitor the system's state from its pre- and post-state of a particular operation. That is, if the negative influence takes effect on the system's state only during the middle of one operation's execution, there is no way we can guarantee the operation(s) affected by the changed state will run properly if we cannot guarantee that the state changes made by concurrently running operations are isolated from each other, for instance by locking the affected system's state within a "safe range". Our approach in this thesis is going to explain how to extract the concurrency information from the system's analysis documents to detect the conflicts in the operation's design. We base our ideas on Sendall's extended Fusion analysis [SS99] and Kienzle's concurrency addressing software development model [KS06]; we also propose a new method to solve the system-state-inconsistency problem.

Fairness/liveness issues in the concurrent system design

Fairness/liveness issues are common in the concurrent system. Liveness issues are sometimes confused with fairness issues for its lack of practical meaning during the system running test [Lam00]. Formal methods for specification and verification of the fairness/liveness properties have been introduced in the past years [AH98, GPSS80]. A key component in many formal approaches is the mathematical modeling of concurrency. However, verification with mathematical modeling is non-trivial, although it can provide accurate verification results. A simple and general approach to verify the fairness/liveness properties is difficult to design. There are two reasons we believe that cause such difficulty: one is due to the subjective nature within the fairness; and the other is because liveness properties are inherently difficult to verify. In this thesis we discuss and develop a new simple and yet still general approach to help the developers to verify the system's fairness/liveness properties. We inject the subjective fairness constraints into the meaning of liveness property for each individual system, and we are therefore able to approximate the liveness property to the property we really care about and that the corresponding running test is feasible. We provide a general approach to find a list of system components that may affect the system's performance in terms of any kind of fairness constraints imposed on it. With the approximation within the liveness property, the new list-generation approach also applies to the verification of the liveness property.

Keep the development cost low

Designing complex systems is not trivial, and designing concurrent systems is extremely challenging. The cost to fix the design issues varies tremendously depending on the time when these issues are detected. It has been shown in [Dav93] that the cost of fixing a design error in later phases (coding, unit testing, acceptance testing, maintenance) can be up to 40 times more. It is therefore desirable to detect design errors as soon as possible. The techniques proposed in this thesis aim at preventing concurrency-related errors (deadlocks, consistency and fairness issues) from occurring, and hence reduce the development cost of concurrent systems.

Figure 1.1 is a modified software development waterfall model. It is augmented by our verification methods at its design phase. Since we are performing static verification, our approach is overly conservative. In other words, there might be other designs that would also be correct, but that allow more flexible concurrent executions.

1.3 Thesis Organization

We begin with Chapter 2, which outlines the background of our research. It contains a general introduction of interaction operators in Unified Modeling Language (UML), the Fondue method and the Object Constraint Language (OCL). This chapter also presents the related work people have done to incorporate concurrency into software system design and to verify the concurrent system design.

Chapter 3 gives an overview of an online auction system that is used as a case study throughout the thesis. The overview introduces the architecture, services and rules of the online auction system. Also, the original analysis and design of the online auction system using the Fondue method is presented.

Chapter 4 introduces the approach we adopt to check for deadlocks within a concurrent system design. The approach is based on the *Hasse Diagram*, a simple picture of a finite

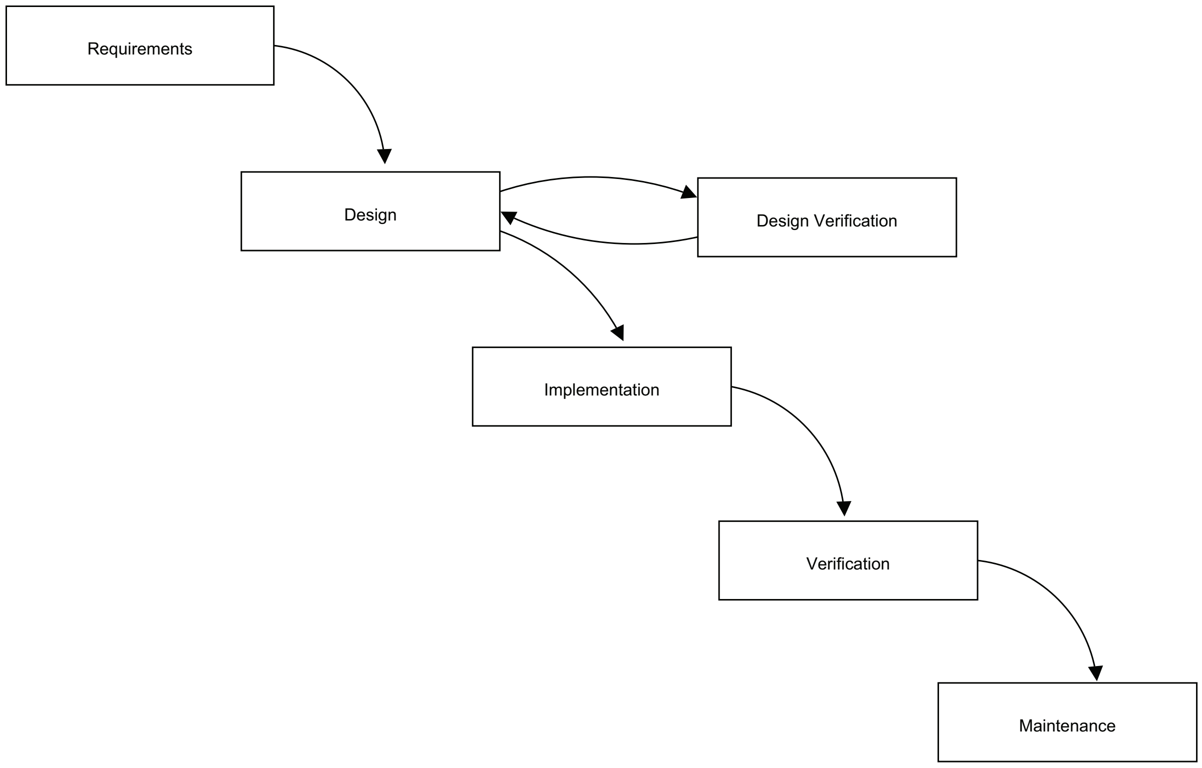


Figure 1.1: *Extended Software Development Waterfall Model*

partially ordered set, from which we develop a new diagram called *System Synchronization Hasse Diagram (SSHD)*. By properly representing the system's components in the SSHD, we can run a cycle detection search in the SSHD in order to determine if the system is deadlock-free. If the SSHD is cyclic, some further analysis methods are introduced in order to refine the obtained result. We present our design, example usage, and related work in this area.

In Chapter 5, we show how to construct a correct concurrent system design, in which individual system operations perform their state modifications in a consistent and deadlock-free way, even in the presence of other concurrently executing system operations. Based on a concurrent system specification obtained using the Fondue development method, we transform the rely conditions of the operation schema to *rely diagrams*. Based on the *rely diagrams*, we elaborate *sequence diagrams* that describe the design of the system.

These *sequence diagrams* contain *critical regions* in accordance with the *rely diagrams* that guarantee the state changes are performed in a consistent way. The resulting design is therefore “correct” by construction. In this chapter, we describe our technique and also relate it to other related work on transaction systems.

Concurrent systems that have been designed without taking fairness into account can exhibit undesirable behavior. In Chapter 6, we explore this kind of issue by suggesting an approach that does not solve fairness issues, an extremely difficult problem, but points out fairness sensitive resources, i.e. resources whose access-policies have an important impact on the fairness of the overall system. The approach works for the liveness issues also, as we explain why and how to bind liveness with fairness at runtime verification.

Finally, in Chapter 7 we draw some conclusions and present future work on our proposed approaches. We find that the verification and generation algorithms we have developed can be used effectively to cope with common concurrent system design issues; the online auction system case study demonstrates both the feasibility and applicability of our methods.

Chapter 2
Background and Related Work

There has been a lot of research into addressing concurrency in software design. The following sections build background knowledge for the work presented in this thesis; first we discuss the general interaction semantics of the Unified Modeling Language (Version 2.0). The next section introduces the Fondue method, a software engineering method that evolved from the Fusion method, and explains how it is extended to address concurrency. The whole section is based on the work that have been done by Kienzle and Sendall [KS06]. In Section 2.3, we describe various other approaches that can address concurrency. Finally, in Section 2.4 we present a survey of the previous work on the verification of concurrent system design.

2.1 UML Interaction Operators

The Unified Modeling Language (UML) is widely used for the description of object-oriented designs and therefore provides an excellent environment for specifying, visualizing and documenting models of software systems. Its newest version is 2.0.

UML supports behavioral modeling by providing several forms of interaction diagrams; a very intuitive way to describe object interactions is the sequence diagram, which is adopted throughout the thesis to describe system designs. Interactions can be used during the detailed design phase, where the precise inter-process communication must be set up according to formal protocols [UML04]. There are several interaction operators provided in UML 2.0 sequence diagrams that are commonly used in the system design and here is a list of them [UML04]:

alt

The interaction operator **alt** represents a choice of behavior. By using **alt**, within the same interaction fragment at most one behavior will be chosen. For each chosen behavior, there is an explicit or implicit guard expression that evaluates to true at this point in the interaction. An implicit true guard is implied if the behavior has no guard. The **else** guard represents a guard that is the negation of the disjunction of all other guards in the enclosing combined fragment. Figure 2.1 shows a situation in which, when `X` is larger than 0, `foo()` will be called; otherwise, `bar()` will be called instead.

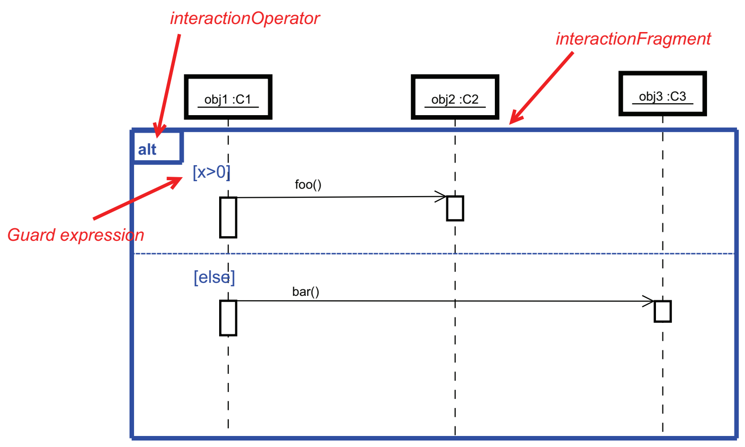


Figure 2.1: *Example of Using Interaction Operator **alt***

par

The interaction operator **par** represents a parallel merge between the behaviors. The event occurrences can be interleaved in any way as long as the ordering imposed by each operand of the `par` operator is preserved. The example displayed in Figure 2.2 shows that the phone operators can receive different incoming calls simultaneously; but for each call, the operator always handles it in the same way, regardless of the current situation of other calls. That is, the operator first receives the call from the caller and then forwards the call to the corresponding callee; the order is never changed.

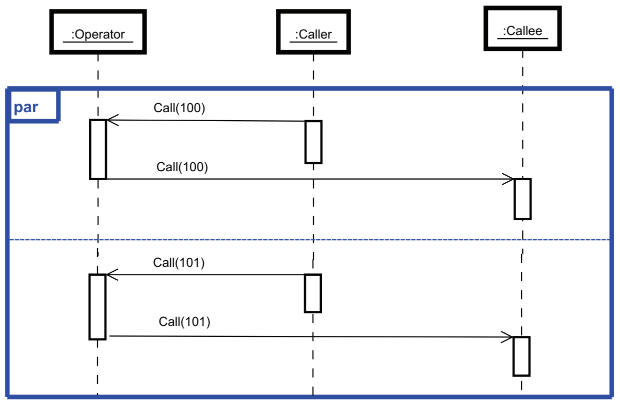


Figure 2.2: *Example of Using Interaction Operator* ***par***

loop

The interaction operator **loop** represents a loop behavior. The loop behavior will be repeated a number of times. The guard expression attached to the operator **loop** may include a lower and an upper number of iterations, which means that a loop will iterate at least the “lower” and at most “upper” number of times. The guard expression may also include a Boolean expression; in this case, the loop will terminate when the Boolean expression is false. The example displayed in Figure 2.3 shows that the `foo()` will be kept invoked until the value of `X` is not positive.

critical

UML 2.0 allows a developer to visualize isolation using the interaction operator called *critical region*. A critical region is a region within which all operations are executed atomically. A critical region, represented as a fragment with the keyword **critical**, implies that even though there could exist other **par** interaction operators within the same or other sequence diagrams, the parallelism is prevented on the objects within the critical region.

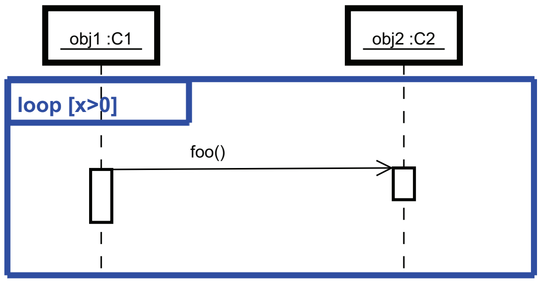


Figure 2.3: *Example of Using Interaction Operator **loop***

Figure 2.4 shows an example of how to use a critical region fragment in the sequence diagram. According to Figure 2.4, any 911-call must be handled contiguously; the operator has to forward the 911-call before doing anything else. The normal calls, however, can be interleaved freely.

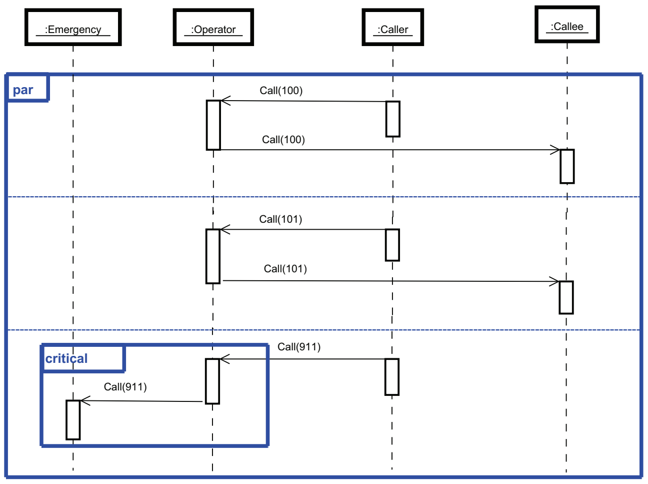


Figure 2.4: *Critical Region Example*

Of course, there are more interaction operators; for a complete reference, we can read Section 14 in [UML04]. In this section, we only give a brief introduction of some of them, which are also the ones we chose to use in our online auction system design case study.

2.2 The Fondue Development Method

Fondue is an OOSD method [SS99], which itself is based on the Fusion method [Col94]. In this section, we are going to first describe the Fusion method, which is followed by the introduction of the original and extended Fondue method.

2.2.1 The Fusion Method

Fusion [Col94] is one of several interesting recent methods for OOSD. Very generally, like those OOSD methods, Fusion breaks system development into several stages, which cover the phases of analysis, design and implementation. A special characteristic of Fusion is that Fusion has no requirements elicitation phase; this special property makes Fusion relatively simple; but as a purported method for global system development, Fusion is clearly not in line with the current ideas which reduce the emphasis on design and implementation and emphasize instead the understanding of organizational environment and needs, and the requirements collection and analysis processes [PdBZ95]. Moreover, comparing with other OOSD methods, the Fusion method is relatively limited in its application scope; for example, Fusion does not deal with synchronization in concurrent systems.

2.2.2 Object Constraint Language

Object Constraint Language (OCL) is a formal language for writing expressions whose principles are based on set theory and first-order predicate logic [JBW03]. OCL can be used in conjunction with UML models as a supplement to the UML diagrams.

One of the common usages of OCL is defining pre- and post-conditions for operations. Pre- and post-conditions come from the work on program correctness [R.F67, Hoa69, E.D76]. They were originally proposed as a pair of assertions in order to prove the correctness of the programs. Over time, pre- and post-conditions were used as the design tools, offering a means to constrain the required behavior of an operation. The Fondue method uses pre- and post-conditions specified by OCL in its operation model to represent the effects of each system operation.

2.2.3 Original Fondue Method

Fondue is based on Fusion; but Fondue overcomes the limitations inherent with Fusion by extending it in many ways. The main improvements consist of:

1. Fondue uses UML as its notation, which makes Fondue widely understandable.
2. Fondue includes the requirements elicitation phase in its system development process; the needs of the system stake-holders are addressed in this phase.
3. Fondue introduces pre- and post-conditions in its operation schema by using OCL [JBW03]; this new addition to the operation schema makes Fondue more precise in its formal specification.

Fondue analysis models

During Fondue analysis phase, the system and its environment are described with a collection of models, each model describing a different aspect or view. The resulting description is of course limited to the expressiveness of the model's building blocks and our perception of the problem domain [KS06]. The models used in Fondue analysis include [KS06]:

- *Environment Model:* It precisely describes the system boundary, and shows how the system interacts with its environment. The environment is represented by a set of actors, which are autonomous entities external to the system.
- *Concept Model:* It offers insight into the problem domain. It takes the form of a UML class diagram, and provides a description of the concepts of the problem domain relevant to the application under development, by representing the concepts as classes, attributes and associations between classes.
- *Protocol Model:* It allows the analyst to specify the ordering in which input events occur for the system under development by using UML state diagrams.
- *Operation Model:* It represents the effects of the system operations on the conceptual states specified in the concept model. For each operations, a separate operation

schema is written in OCL. The template for the original Fondue operation schema is shown in 2.5.

Operation: system class:operation name (list of parameters, if any)
Description: description of operation, including purposes and effects
Notes: additional comments of the operation (optional)
Use Cases: list of related use cases (optional)
Scope: list of all classes and associations involved in the operation
Message: list of message types output by the operation together with their receiving actor classes
New: list of objects that will be created by the operation
Alias: list of names that act like aliases (optional)
Pre: precondition; a Boolean expression that must be met before operation starts
Post: post-condition; a Boolean expression that represents the effects of the operation;

Figure 2.5: *Sequential Operation Schema Template*

Detailed examples of how to use each of these analysis models will be explained in Chapter 3, where we can also see how these models help on the system design.

2.2.4 Concurrency-Addressing Fondue Method

The original Fondue method does not address concurrency in the system development; Kienzle and Sendall improved the original Fondue in several ways by extending it in order to record the inherent concurrency of the system [KS06].

First, the use case template of the Fondue method is extended to include a *Frequency & Multiplicity* section. The concurrency requirements from the users therefore can be encapsulated in this section.

Second, the Protocol Model is extended by two techniques. One technique for partitioning the concurrency of the system is the *divide-by-actor* technique. This technique recommends that the interaction protocol between the system and each actor type is described separately using a composite state. Such a state is referred to as an *actor-activity-state*, and it takes a *...Activity* suffix. The other technique, *divide-by-collaboration*, can be used to specify the interaction protocol between the system and its actors in terms of distinct types of collaboration between them. Such a collaboration is represented by a state named with a *...View* suffix, referred to as a *view-state* [KS06]. Comparing to actor-activity-state, view-state puts more restricts on the concurrent behaviors of the actors.

Third, in order to specify the inherent concurrency captured in the Protocol Model, the original operation schema has been elaborated to its concurrent version. Figure 2.6 shows the concurrent operation schema.

Operation: system class:operation name (list of parameters, if any)
Description: description of operation, including purposes and effects
Notes: additional comments of the operation (optional)
Use Cases: list of related use cases (optional)
Scope: list of all classes and associations involved in the operation
Shared: list of all shared concepts
Message: list of message types output by the operation together with their receiving actor classes
New: list of objects that will be created by the operation
Alias: list of names that act like aliases (optional)
Pre: precondition; a Boolean expression that must be met before operation starts
Post: post-condition; a Boolean expression that represents the effects of the operation;
rely-condition is part of post-condition; rely-condition is a Boolean expression that must be met during the execution of the operation

Figure 2.6: *Concurrent Operation Schema Template*

A new clause called *Shared* has been added in the concurrent operation schema; the *Shared* clause is used to identify all shared concepts, i.e. classes or attributes or associations that are used by potentially concurrent executing operations. Once all shared concepts have been identified, we can update the Concept Model by adding <<*Shared*>> stereotype to all shared classes, shared associations and shared attributes.

In the original sequential Fondue, analysis operations had instantaneous semantics. The concurrent version, however takes into account the time it takes to execute an operation. Therefore, the effects specified in the post-condition of the concurrent operation schema no longer possess instantaneous semantics; pre-conditions or *if* statements with conditions that include shared concepts can change their values during the execution of the operation. To deal with this new problem, a new construct called a *rely expression* can be used in the post-condition. A rely expression is a Boolean expression with the following form, where the *fail* part is optional:

rely rely-condition **then**
    rely-effect-expression
fail
    fail-effect-expression
endre

The *rely-condition* within the rely expression defines a condition that can be relied upon to stay true during the execution of the running operation (or fulfillment of an effect). The meaning of the above rely expression is that the running operation should fulfill the rely-effect-expression if and only if rely-condition stays true during the execution period of the running operation, even in presence of other concurrent operations. If, during the execution period, the contract imposed by the rely-condition cannot be maintained, then there is an obligation for the running operation to fulfill *fail-effect-expression*. This implies that, if any other concurrently running operation changes the system to a state that violates the rely-condition, then the running operation has to carry out the fail-effect-expression.

There are two notes for the concurrent operation schema. First, it is important to be aware that the *if* expressions in the sequential operation schema that test shared concepts have to be transformed into *rely* expressions. Second, in the concurrent operation schema, pre-conditions that are based on shared concepts in the sequential operation schema may also have to be moved to the post-condition; this is because there could be multiple operations executing concurrently, it may be not enough to check the condition only once at the beginning of the operation.

This section only briefly describes the main features of the new concurrency-addressing

Fondue method. The further discussion on how to use these newly added features will be in Chapter 3 and Chapter 5.

2.3 Other Concurrency Addressing Approaches

There have been several approaches that also address concurrency in system design; this section presents a survey of some of the previous work and compares each of the approaches with the Fondue method.

2.3.1 COMET

COMET (Concurrent Object Modeling and architectural design mEThod) is a development method for concurrent applications [Gom00]. It also has a requirements elicitation phase and analysis phase; comparing to Fondue method, COMET emphasizes more on the design phase. The system design is well elaborated by two phases. In the first phase, the system under study is structured into subsystems, classes, objects. Interfaces between subsystems are identified. The second phase is focusing on task structuring; concurrent tasks such as synchronization and communication are considered in this phase. Moreover, in the second phase, task clustering criteria is used for system control flow; and the interfaces between tasks are identified.

2.3.2 Rely-Guarantee-Conditions

In [Jon83], Jones proposed rely- and guarantee-conditions as an extension to operation specifications that use pre- and post-conditions. A specification that uses rely- and guarantee-conditions consists of four assertions: pre-condition, rely-condition, guarantee-condition and post-condition. The meanings of pre-, rely- and post-condition are very similar to those in the Fondue method; the guarantee-condition specifies the condition that the running operation must satisfy when it changes the system state. There are several approaches that make use of rely- and guarantee-conditions, such as the Catalysis approach [DFD98].

Comparing to the Fondue method, the rely-guarantee-conditions apply to the whole execution period of a running operation; while the rely expressions in the concurrency-addressing Fondue method are used within the post-conditions and they have scope over only a subset of the running operation's effects. Another difference is that in Fondue method, the rely expression may have a *fail* part encapsulating the fail-effect-expression for the case when the rely-condition cannot be maintained during the execution of the operation. Therefore, the Fondue method has more descriptive capability and consequently provides the designers more possibilities without restricting the valid designs.

2.3.3 Concurrency Pattern in UML

In [CC01], Crinchton and his researchers proposed a pattern of usage for the UML. Their work was intended to describe the systems in which two or more operations may be acting concurrently upon the same object. Their pattern addresses two common problems - inadequate models, and complicated state diagrams - with a simple separation of concerns. Changes in attribute state and changes in operation state are described separately, using two different types of diagrams.

The similarity between Crinchton's pattern approach and the concurrency-addressing Fondue is that they both analyze the operation individually; the pattern approach uses operation diagrams while Fondue uses operation schemas to describe each operation. However, to perform detailed concurrent analysis on operation diagrams, additional help from formal tools such as CSP [Hoa78] is needed; the concurrency-addressing Fondue method can use intuitive graphical methods, such as rely diagrams, to conduct the concurrency analysis. The details of how to use rely diagrams will be described in Chapter 5.

2.4 Approaches to Verify Concurrent System Design

There have been plenty of approaches dedicated to detect concurrency related errors in the system design. This section divides these approaches into three categories: deadlock detection approaches, detecting inconsistent state and fairness/liveness verification approaches.

2.4.1 Deadlock Detection

Modeling concurrent behavior using abstract models such as process algebra including CSP [Hoa78], CCS [Mil89], the π- calculus [Mil99] or Finite State Process(FSP) [JM06] directly exposes concurrent behavior. For example, the Labeled Transition System Analyzer (LTSA) [JM06] is a verification tool for concurrent systems. The tool supports FSP specifications for concise descriptions of component behavior and allows the Labeled Transition System(LTS) corresponding to a FSP specification to be viewed graphically. This produces a mechanical way of checking that the specification of a concurrent system satisfies the properties (including deadlock-freedom property) required of its behavior. This kind of approach, although is general enough, requires a deep knowledge of the abstraction being used and consequently is difficult to use for the practitioners.

Kaveh and Emmerich offered a new way to translate UML diagrams into behaviorally equivalent process algebra representations and then use model checking techniques to find potential deadlocks [KE01]. Their approach is realized by suggesting UML stereotypes to represent the different synchronization behaviors in the UML model, and defining the semantics of the UML stereotypes by using an abstraction model (FSP in their demonstrated tool). Therefore, the verification of the UML model can be carried out on the generated formal specification. However, their research concentrates on synchronization behavior in object-oriented middle-ware systems only.

vUML [LP99] developed by Lilius is a tool using the SPIN model checker to perform the verification of UML models. vUML is designed to convert the UML state charts diagrams into a PROMELA (a PROcess MEta LAnguage) specification first and then invokes SPIN; the verification is then performed by SPIN without the necessity for the user to know the knowledge of SPIN or the PROMELA language. In case the verification fails, vUML generates a counter-example that shows how to reproduce the error in the model. vUML, comparing to LTSA, releases the burden from the users by not requiring them to know how to use SPIN or the PROMELA language; compared to Kaveh and Emmerich's approach, vUML has a broader application scope. But, it assumes that each instance of the modeled class runs in a separate process, something not always true in practice.

2.4.2 Detecting Incorrect System Conceptual State

System design errors are defined as violations of requirements expressed as properties of the system. Among all of such errors, violation of the constraints imposed on the system conceptual state are hard to detect, especially for concurrent systems. State changes in a concurrent system are particular dangerous operations. That is, if the state-change is not atomic then there exists a time point where a partially changed state can be fetched by a concurrent actor, which can in turn apply a state-change-operation. This may lead to a complex interleaving of partial state changes.

Temporal logic is a type of modal logic that is used for reasoning about changing properties with time, and naturally people have used it as a formalism to describe how a system/program state will change over time. Working with assertions, which can express system or progress invariants, temporal logic is a basis for many tools that are used to verify system's consistent state. Among them, SPIN [Hol97] is one of the most famous tools. Many system state verification tools were developed based on SPIN, such as HSF-SPIN [ELL01], extended PROMELA and SPIN for real time [TC96], and vUML [LP99]. SPIN is an example of a Finite State Verification (FSV) approach. There are two main drawbacks to the FSV technique: first, FSV obtains the result from the model of the system/program, therefore the quality of the results are depending on the models, which can be very difficult to construct to be small enough to be tractable. Second, for concurrent system/programs, the number of states can increase exponentially with the number of processes. This well known "state explosion" can introduce significant scalability problems.

Although database and operating systems do not relate to concurrent systems directly, the transaction processing techniques that are deeply ingrained in these two fields are inspirational to our approach. Transaction processing is designed to monitor, control and update information in the database or modern system to ensure them remain in a consistent state; it allows multiple operations to be bound together as a single, atomic transaction. The transaction-processing system ensures that either all operations in the same transaction are completed without error, or none of the operations are completed and the system rolls back to its previous valid state. There exist many advanced technologies based on transaction processing. For example, Park proposed a distributed group commit protocol [PY], which identifies a consistent set of transactions that have to be committed together. The new protocol avoids consistency or system performance problem by propagating the transaction precedence information and log contents on normal transaction messages. After reviewing several transaction processing approaches, we found though transaction processing is a promising approach that gives us a lot insights in how to keep system state consistent, the space cost by using checkpoints (or log) and communication overhead to keep consistency information up to date are not exactly what we want in our own cost-effective approach.

2.4.3 Verify Fairness/Liveness Properties

There are several general approaches to verify system liveness properties. Again, temporal logic is one of them. Liveness properties can be specified by assertions in a simple temporal logic. SPIN [Hol97], a versatile verification tool we just introduced, can also be used as an efficient on-the-fly verifier for the liveness property. State reach-ability is another way for liveness verification. Cheung and his researchers have extended the conventional Compositional Reachability Analysis (CRA) by extending the analysis to check liveness properties which may involve actions that are not globally observable [CGK97]. Verification of fairness, on the other hand, is very subjective. Each system may have different fairness constraint from its users requirements. Fairness properties are easily confused with liveness properties for the reasons we will explain in Chapter 6. Therefore, the techniques to verify fairness properties are very similar to those used on liveness properties.

2.4.4 Summary

We address concurrency issues with the ideas from Kienzle's concurrency-addressing Fondue method [KS06] and transaction processing. We are able to analyze and construct the system design to ensure its state remains application-level consistent, and our techniques to help developers verify system's fairness/liveness properties are derived from and integrate with our initial design work. Our approach to fairness/liveness verification is quite different from the conventional techniques we surveyed in the Subsection 2.4.3. As a whole, our three approaches to deadlock detection, verification of system application-level consistency, and fairness/liveness properties are all linked together and can be used as a simple, general and fairly accurate verification package for the concurrent system. More related work and knowledge of these three topics is discussed in each of the corresponding chapters.

Chapter 3
Overview of Online Auction System

As product functionality increases, modern applications must be capable of responding to an increasing number of requirements, including the need for better user interfaces and interactions with real time devices, e.g. sensors. Multiple user situations are also common, and in general the software integrated within is required to be capable of performing several operations concurrently in order to provide prompt response to external stimuli.

This chapter gives a general introduction of a practical example with multiple demands. Our online auction system introduction includes physical architecture, services, rules and the design of an auction system. Since online auction systems are an example of dynamic systems with cooperative and competitive concurrency, we use it throughout the thesis as our use case study. The informal description of the online auction system presented in this chapter is inspired by the auction service examples presented in [KSR02, Xio04], which in turn are based on various live examples that can be found from Internet websites, e.g. www.ebay.com, www.ubid.com.

3.1 Online Auction System Architecture

The physical online auction system architecture is shown in Figure 3.1. From Figure 3.1, we see there are three main component types existing within online auction systems:

1. System Server (C1): it responds to the requests from its external users. The server can connect to the credit institutions via the network. The number of servers varies

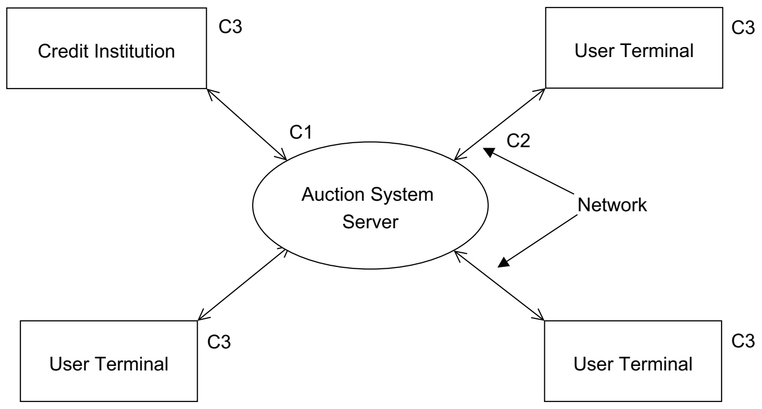


Figure 3.1: *Physical Architecture of Online Auction System*

depending on the system design.

2. Network (C2): it connects external user terminals and the credit institutions with the system server.

3. External nodes (C3): they can be either user terminals or credit institutions. User terminals are usually personal computers supporting different operating systems and Graphical User Interfaces (GUI). The credit institutions are institutions that can provide users credit information.

3.2 Online Auction System Services and Rules

Like eBay, the online auction system we are using in this thesis only allows its registered users to buy or sell items. To make our case study more complex, the registered users are only allowed to use money from their auction system accounts to make deals. On the other hand, at any time, the registered users can deposit into their auction accounts by transferring money from their credit institutions.

Services

The auction system provides basic services to allow its members to log in, browse auctions, bid in auctions and start new auctions. The system should terminate an auction automatically when the corresponding auction passes its expiration time. The expiration time of an auction can be determined either by a fixed duration or by a pre-defined time out. A fixed duration could be 24 hours, 3 days, etc.; a pre-defined time out could be a one-day idle since the last bid. The system can automatically withdraw or deposit money to its member's auction system account. Upon request, the system can connect to a user's credit institution to transfer money between the user's credit account and the user's auction system account. For each auction, the system preserves a bid *history* object for recording purpose and the *highest bidder account* to track the last highest bidder's account. The system is capable of handling concurrent bids on the same auction item.

Rules

There are certain rules users should follow in order to use the online auction system. To start a bid, the seller must provide the detailed information about the item. This information should include the starting price, minimum increment amount and the duration of the auction. The seller has the right to cancel the auction anytime before its initiation, but not after. The bidder must not be the owner of the auction item; the bidding must take place while the auction is still in its open status. The initial bid amount must be at least large as the starting price of the auction. Each new valid bid must be higher than or equal to the sum of last highest bid amount and the minimum increment amount. The user is allowed to make as many bids as he/she wants. But a user's account balance should remain at least as much as is required to cover his/her pending bids and the new bid. When there is a higher bid, the last highest bid amount should be returned to its corresponding bidder's account.

3.3 Online Auction System Requirements Elicitation

Use cases are a widely used formalism for discovering and recording behavioral requirements of software system [Lar02]; it is an effective communication means between the technical developer and the non-technical stake-holder of the software. Use cases are in general text-based, but their strength is that they can scale up or scale down in terms of sophistication and formality, depending on the need and context [KS06]. For simplicity purpose, this section only describes the use case of *Buy Item under Auction* by using the Fondue method.

Fondue is one of the software development methods that have their own use case templates. The following is the *Buy Item under Auction* use case written in the Fondue use case template.

Use Case: Buy Item under Auction
Primary Actor: Customer
Intention: The intention of Customer is to buy an item by following an auction of an item that meets his/her criteria.
Frequency & Multiplicity: Customer may bid simultaneously in several auctions, and several Customers can bid in the same auction simultaneously.
Main Success Scenario:

1. Customer searches for an item under auction.
2. Customer requests System to join the auction.
3. System presents a view of the auction to Customer.

Step 4 and 5 can be repeated according to the intentions and bidding strategy of the Customer.

4. Customer makes a bid on the item.
5. System validates the bid, and updates the view of the auction for the item with the new high bid to all Customers that are joined to the auction.

Customer has the high bid for the auction.

6. System closes the auction with a winning bid by Customer.

Extensions:

3a. System informs Customer that auction is closed; use case ends in failure.
5a. System determines that bid does not meet the minimum increment. System informs Customer; use case continues at step 4.
5b. System determines that Customer does not have sufficient funds to guarantee the bid. System informs Customer; use case ends in failure.

6a. Customer was not the highest bidder. System closes the auction; use case ends in failure.

Buy Item under Auction Use Case

The section *Frequency & Multiplicity* in this use case is used to help the developer discover concurrency requirements for each use case. For example, in the *Buy Item under Auction* use case, a customer can participate and bid in several auctions simultaneously.

3.4 Online Auction System Analysis

The Fondue analysis phase comes right after requirements elicitation. During analysis, a black-box specification of the system under development is constructed. As mentioned in Chapter 2, in Fondue the system and its environment can be described in a collection of models. Based on the work done in [KS06] and for space reasons, we only briefly go through these different models, each of which describes a different view of the online auction system.

Environment model

The *Environment Model* of the online auction system, as shown in Figure 3.2, precisely defines the online auction system boundary and how it interacts with its external environment. The multiplicity of the actors and interactions states that there can be any number of system users, of which any number can be interacting with the system at any given time. This setting in the *Environment Model* ties closely to the requirements specified in the section *Frequency & Multiplicity* in the *Buy Item under Auction* use case.

Concept model

The *Concept Model* of the online auction system, as shown in Figure 3.3, offers insight into the problem domain; it excludes the objects, classes and relationships that belong to the environment. The *Concept Model* is really straightforward to understand. It does not address concurrency issues in itself; however, it forms the base for identifying possible shared data structures [KS06]. For an instance, since *Auction* in Figure 3.3 has a *one-to-*

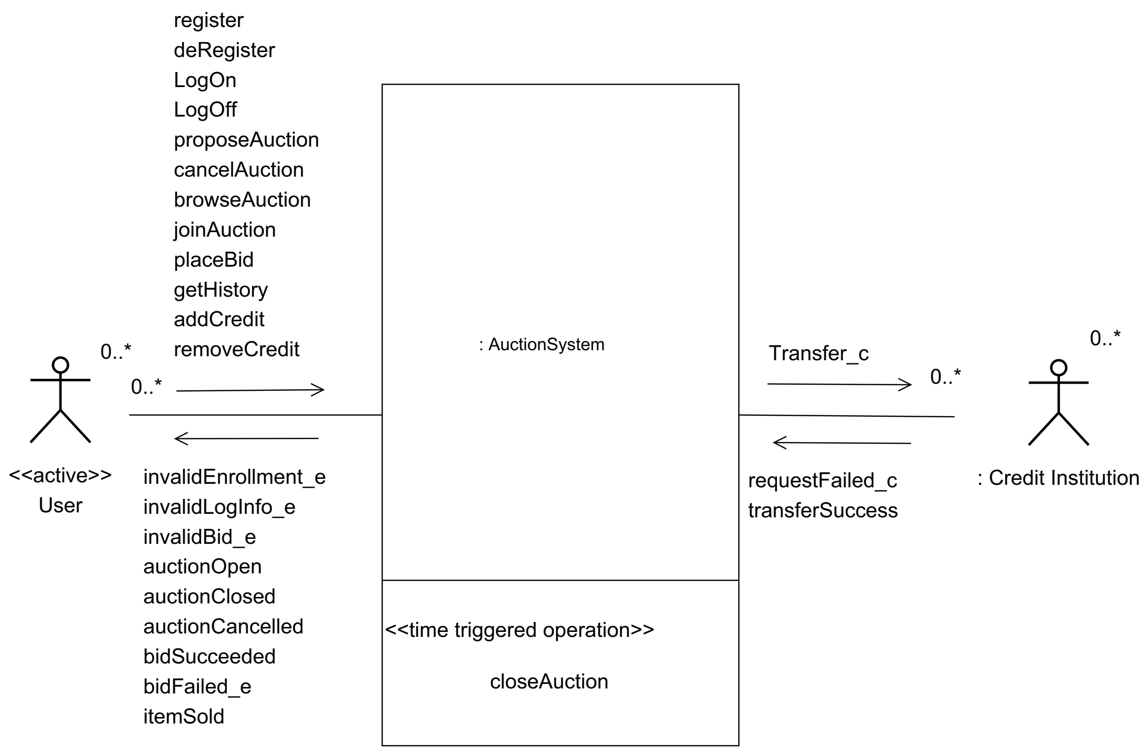


Figure 3.2: *Online Auction System Environment Model*

many association with *Bid* and each *Bid* has only one *Customer*, therefore we can determine that *Auction* is the shared data structure.

There is one interesting detail about the derived attribute *guaranteedBalance* of *Account* in Figure 3.3. In order to guarantee that a bidder can always pay for his/her outstanding bids, we let *guaranteedBalance* represent the maximum amount that a customer has available for bidding. Since the attribute *guaranteedBalance* is derived type, its value is calculated automatically after each valid bid. The calculation is conducted by subtracting all high bids the customer has placed in the active auctions from the actual balance of his online auction system account.

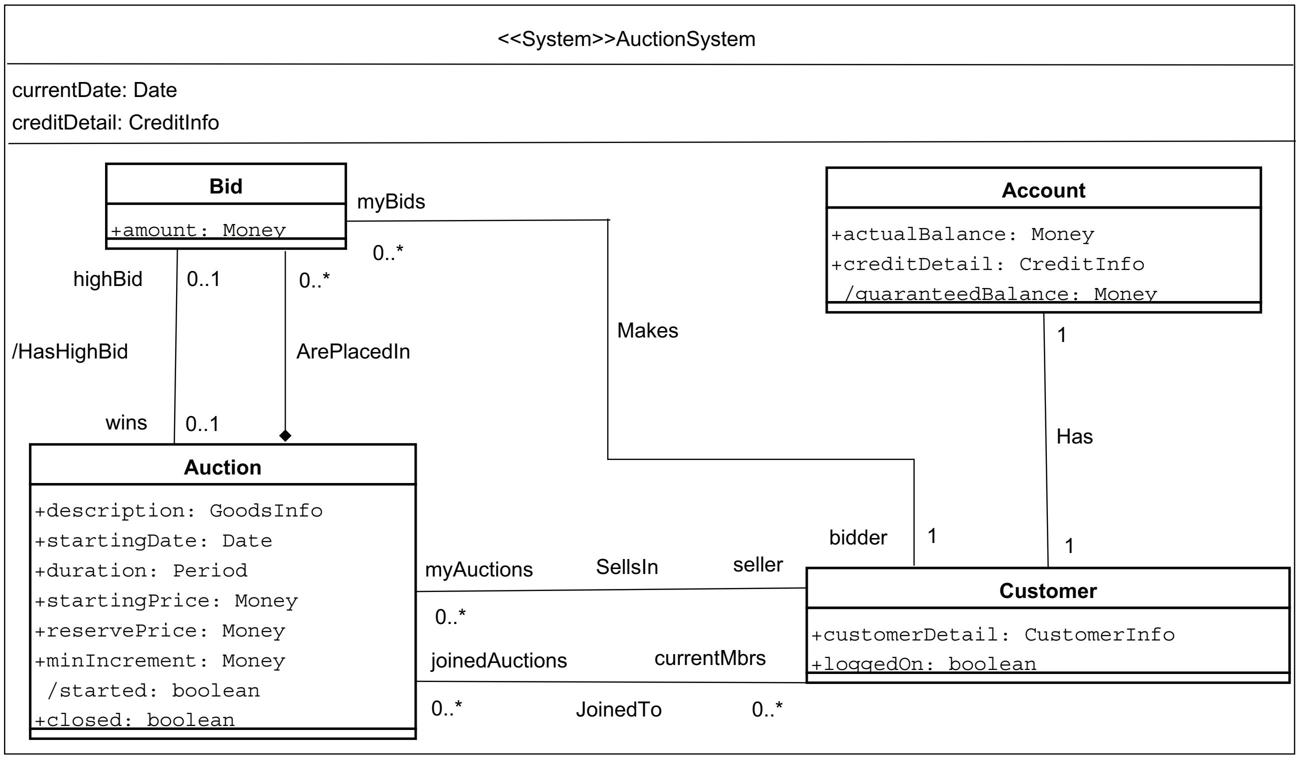


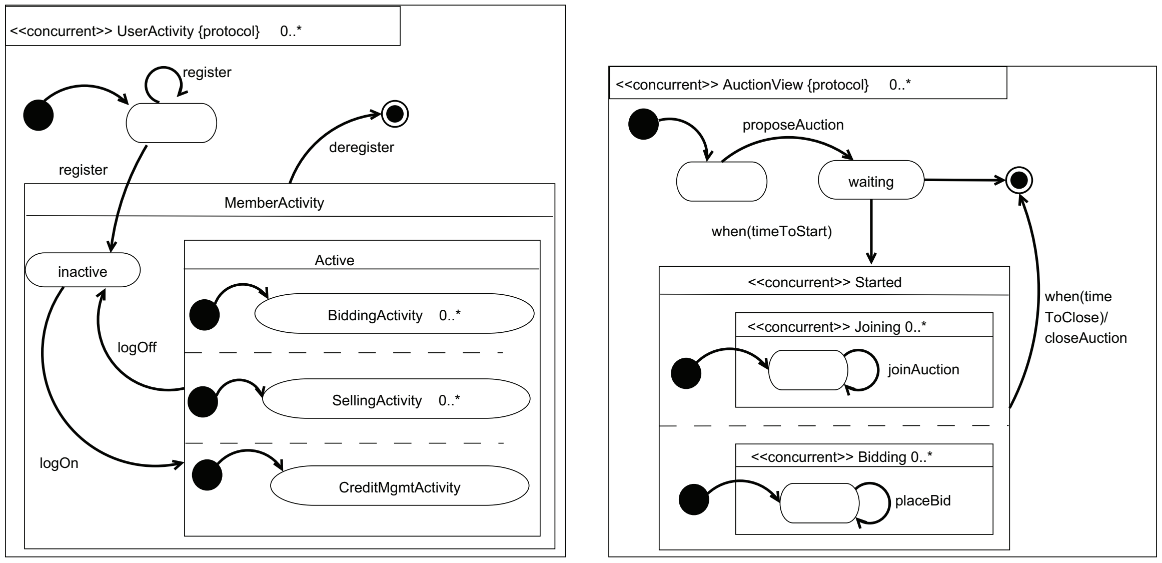
Figure 3.3: *Online Auction System Concept Model*

Protocol model

As mentioned in Chapter 2, the *Protocol Model* is constructed by using state diagrams to specify the ordering in which input events occur for the system under development. Realizing the online auction system is a highly dynamic system, we use both *divide-by-actor* and *divide-by-collaboration* techniques to describe the inherent concurrency of the auction system.

Figure 3.4(a) shows *UserActivity* state by using the *divide-by-actor* technique, which models the protocol that the system has with *User* actors. In Figure 3.4(a), the <<*concurrent*>> stereotype notation is added to highlight events in the model which can be invoked concurrently. Since the number of clients is dynamic, it is modeled as an *auto-concurrent* state, represented by adding a multiplicity of 0..* to the state. Each concurrent state encapsulates the interaction between the system and a particular *User* actor. The *Active* state is not concurrent, since a single user cannot physically perform multiple activities in parallel.

Further analysis on the online auction system tells us the protocol established in *User-Activity* state is too general. It allows more concurrency than what is likely to be supported by the implementation. Figure 3.4(b) shows *AuctionView* state by using *divide-by-collaboration* technique. It represents the collaboration between the various parties to take an auction from start to finish. *AuctionView* state, comparing to *UserActivity* state, puts more constraints on the concurrency level of the system. For example, from Figure 3.4(b), we can see that joining and bidding is only permitted once the auction has started, and as long as it is not closed.



(a) The UserActivity State (b) The AuctionView State

Figure 3.4: *Online Auction System Protocol Model*

Operation model

For each of the system operations, a separate operation schema is needed. For simplicity reasons, in this section we are concentrating on the operation `placeBid` only.

The protocol model (Figure 3.4(b)) states that the `placeBid` operation might execute concurrently with other operations. Therefore, we should use the concurrent operation schema instead of sequential operation schema, which we introduced in Chapter 2, to specify the effect of `placeBid` on the auction system. Figures A.1 and Figure A.2 in Appendix A are the sequential and concurrent operation schemas for operation `placeBid`.

With the rely-conditions specified in Figure A.2, we understand that `placeBid` operation relies on the auction state being active, the bid amount is high enough and the balance in the user's account is sufficient to pay the bid during the execution of the operation.

3.5 Online Auction System Monitor-Based Design

After requirements elicitation and system analysis, we need to design the system that satisfies the requirements defined by the analysis models. This section briefly describes a monitor-based design of the online auction system.

Figure 3.5 is our original proposed online auction system design by following the object-oriented design method. We map the system's conceptual state to objects, and therefore the developers can decide how the conceptual state changes specified in every system operation are to be implemented by interacting objects at runtime. In our sequence diagram design, we choose to use dark grey color as the background of the *monitor* object. The atomicity needed for implementing the rely expressions is achieved by acquiring read or write locks on the monitor objects when checking the condition. A lock prevents other threads from changing the condition while the operation executes. After the state changes that rely on the condition have been performed, the locks are released again.

In Figure 3.5, *auctionState*, *currentBid*, *currentAccount*, *previousAccount*, *history* and *newBid* are now monitors. A read lock is acquired while checking the auction status; this blocks a potential concurrent attempt to execute the `closeAuction` operation. A careful analysis shows that the balance of a customer's account cannot decrease since the same customer cannot place two bids simultaneously, or try to remove credit while placing a bid[1]. Therefore, it is not necessary to acquire a lock to guarantee the balance when accessing the account of the customer who is placing the bid. We can simply check and withdraw the bid amount from the account in the operation `isGuaranteed`, which itself is atomic

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

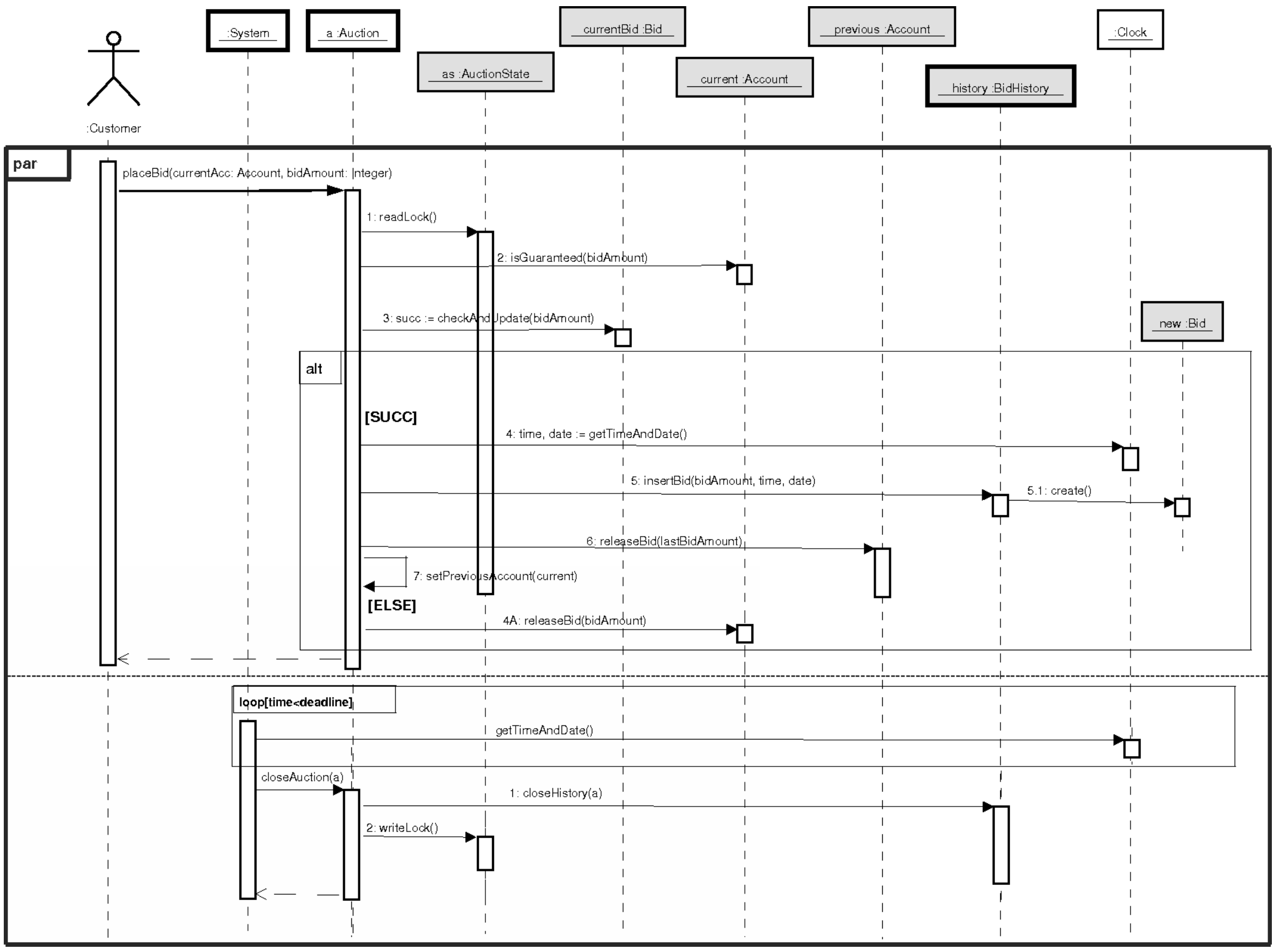

:Customer
:System
a :Auction
as :AuctionState
currentBid :Bid
current :Account
previous :Account
history :BidHistory
:Clock
new :Bid
par
placeBid(currentAcc: Account, bidAmount: Integer)
1: readLock()
2: isGuaranteed(bidAmount)
3: succ := checkAndUpdate(bidAmount)
alt
[SUCC]
4: time, date := getTimeAndDate()
5: insertBid(bidAmount, time, date)
5.1: create()
6: releaseBid(lastBidAmount)
7: setPreviousAccount(current)
[ELSE]
4A: releaseBid(bidAmount)
loop[time<deadline]
getTimeAndDate()
closeAuction(a)
1: closeHistory(a)
2: writeLock()

Figure 3.5: *Monitor-based Execution of* `placeBid` *and* `closeAuction`

because accounts are monitors; and then in a similar way check and update the current high bid (`checkAndUpdate` operation). Subsequently, we release the bid of the previous high bidder and update the *previousAccount*. Alternatively, if the bid is invalid, the money has to be put back in the bidder's account.

Chapter 4
Detecting Deadlock within System Design

In concurrent systems, processes or threads tend to communicate and interact by accessing shared resources. Concurrent modifications to shared data structures can lead to state inconsistencies or data corruption. To prevent this, modifications have to execute with mutual exclusion: when a thread tries to modify state that is already being modified by some other thread, the first thread has to wait until the second one is done with the modification.

Due to the inherent non-determinism of concurrency, the number of possible interleaving of resource accesses at run-time is extremely large, and it is therefore hard for a developer to see the big picture and make sure that all interleavings lead to correct results. In certain situations and circumstances it is unfortunately possible that a *deadlock* occurs, which prevents the application from making further progress. In this chapter, we explain the nature of deadlock, survey the work that has been done to prevent or avoid deadlock, and propose our own algorithm that allows a developer to analyze his design for potential deadlock situations.

4.1 Deadlock in Concurrent System

In order to understand how our proposed *deadlock* detection algorithm works, we first need to have an overview of *deadlock* in the concurrent system. This section is structured to introduce the following: what is a deadlock state in our thesis, different types of resources and resource request models and historical approaches to prevent and avoid *deadlock*.

4.1.1 Conditions for Deadlock

Historically, there are at least five different situations that have been termed as deadlock [Lev03]:

1. An infinite waiting state containing a nonempty set of processes.
2. An infinite waiting state containing a nonempty set of processes that make no progress according to some definition of progress.
3. An inactive infinite waiting state containing a nonempty set of processes that make no further progress.
4. An infinite waiting state containing a circular chain of processes.
5. An infinite waiting state containing a circular chain of processes, each of which is holding a non-preemptive resource while waiting for another held by the next process in the chain.

These definitions are overlap and ambiguous and hence it is not trivially clean how to define a precise deadlock detection algorithm. For the purpose of this thesis, we will use the *Coffman conditions* to determine whether a deadlock situation exists or not. Coffman stated four necessary conditions for resource deadlock to occur within a system [CES71]. These four conditions are:

1. *Mutual exclusion condition:* a resource is either assigned to one process or it is available.
2. *Hold and wait condition:* processes hold resources while waiting to acquire others.
3. *No preemption condition:* only the process holding the resource can release it.
4. *Circular wait condition:* two or more processes form a circular chain where each process waits for a resource that is held by the next process in the chain.

We use *Coffman conditions* as our necessary conditions to deadlock state in concurrent system design. That is, if there is a deadlock in the system design, these four listed conditions should be true. In most cases it is quite common for processes in the concurrent system to satisfy mutual exclusion, hold and wait and no preemption conditions, therefore only the circular wait condition is left for us to check in order to ensure that the system does not deadlock.

Deadlock vs. Livelock / Starvation

Coffman conditions allow us to resolve the ambiguities within the deadlock definitions presented above. According to the *Coffman conditions*, only the fifth definition in Section 4.1.1 is considered a deadlock. Situations 1 and 2, for instance, do not explicitly mention a ”hold-and-wait” or ”circular wait” condition, and hence are not considered deadlocks in the context of this thesis. In the literature, such situations are sometimes referred to as *livelock* or *starvation* situations.

Another similar example is found in [Hol72]. Holt stated the following three-line PL/I program would cause a system deadlock if the event in line 2 never occurs.

```
1  REVENGE: PROCEDURE OPTIONS(MAIN, TASK),
2  WAIT(EVENT);
3  END REVENGE;
```

Listing 4.1: *Holt's PL/I Example*

The code segment shown above would fall into situation 3 described in Section 4.1.1. But according to the *Coffman conditions*, deadlock cannot happen with only one process. This conclusion has also been widely accepted in modern textbooks [SGG04]. Nowadays, people usually refer to infinite suspension of processes, such as the REVENGE process in the Pl/I example program, as a *liveness* issue rather than deadlock.

4.1.2 Resource Request Models

To understand why *Coffman conditions* cannot be used as sufficient conditions for deadlocks, we need to review different resource request models. Due to the reasons we explained before, we are only focusing our discussion on the circular wait condition to understand this topic. To make the idea easy to comprehend, we are using *Wait-For Graph*. The Wait-For Graph (WFG) is a directed graph, where nodes are processes and a directed edge from P → Q represents that P is blocked waiting for Q to release a resource. If there is a "circular wait condition" within the system, then there is necessarily a cycle within the corresponding system wait-for graph.

There are four different kinds of resource requests that can all lead to a deadlock. They are explained below:

1. Single-Unit Request Model: in this model, a process requests one single unit of a resource and every requested resource has only one single copy or unit. If the resource is available then the process acquires it; otherwise, the process is blocked until the resource becomes available. When we represent this situation using a wait-for graph, there will maximally be one outgoing edge for each process node. In this case, the sufficient condition for deadlock to happen is a circular wait-for graph among processes. Figure 4.1 shows such an example, where process P1 requests a resource held by process P2; and process P2 requests a resource held by process P1. In this model, since there is only one outgoing edge for each node, there are never any branches in the path following the outgoing edges. The length of any detected cycle is proportional to the number of computational steps that follow along the blocked nodes.

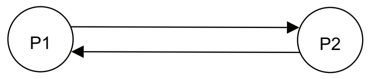


Figure 4.1: *A Deadlock Situation in the Single-Unit Request Model*

2. AND Request Model: in this model, a process needs more than one resource to

proceed. Again, each requested resource only has one copy. For example, a student might need a pizza and a drink in order to continue to work. To represent this model in a wait-for graph, more than one outgoing edge per process node is needed. With multiple outgoing edges per node, it is possible for one process to be involved in more than one deadlock at a given time. Since the complexity of cycle-detection algorithms is linear with respect to the number of edges in the graph, an execution of such an algorithm will most likely to visit large numbers of nodes that may not be part of the deadlock cycle. Figure 4.2 shows a possible deadlock situation for this model. P1 is involved in deadlocks with processes P2 and P3. Like in the single-unit model, a circular wait-for relationship in the graph is a sufficient condition for deadlock to occur.

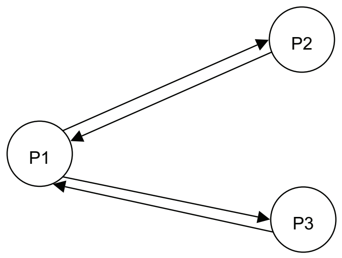
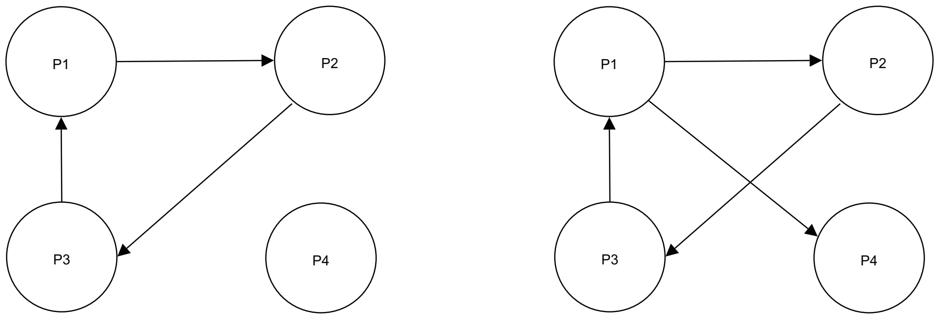


Figure 4.2: *Deadlock in AND Request Model*

3. OR Request Model: in this model, a resource may have more than one copy that can be requested. For example, a student might need food such as a pizza or a sushi in order to continue to work. According to Gertrude, the copies for the requested resource are fungible [Lev03]. That is, in the example mentioned above, pizza and sushi are interchangeable units; both can be regarded as a copy of the requested resource, in this case food. The wait-for graph representing this model looks identical to the wait-for graph in the AND-model, i.e. with potentially multiple outgoing edges per node. But the interpretation of the outgoing edges is different. In this model, a mere presence of a cycle does not necessarily signify a deadlock; instead, a *knot* is required for deadlock to happen. A *knot* in a graph is a subset S of nodes such that the reachable set of each node in S is exactly S. For example, in Figure 4.3(a) processes

P1, P2 and P3 form a cycle and also a *knot*; while P1, P2 and P3 in Figure 4.3(b) do not form a *knot* although they are in a cycle.



(a) Cycle with *Knot*　　(b) Cycle without *Knot*

Figure 4.3: *OR Request Model with/without Knot*

In Figure 4.3(b), even if a wait-for cycle is found, a deadlock is not bound to occur. This is because process P1 may get its requested resource from process P4 instead of P2, therefore once process P1 proceeds and terminates, P2 and P3 will no longer be blocked. Algorithms to detect *knots* within a wait-for graph are more complex than those that detect cycles.

4. AND-OR Request Model: this model is a combination of the AND-request model and the OR-request model. For example, a student might need a pizza, at least one kind of a drink, such as a Pepsi or a Sprite, in order to continue to work. Since the OR-request model is part of this model, a knot is also required for deadlock to occur in the AND-OR request model.

4.1.3 Kinds of Resources

There are two kinds of resources: reusable and consumable. Each kind has its own attributes:

- *Reusable resource:* the number of such resources is fixed; the resources cannot be created or destroyed; when the resource is allocated to a process, it is held until the process is done with the work and then released. Examples of such kind of resources are CPU, memory and printer.

- *Consumable resource:* the number of such resource varies; there is a producer process producing such resources; when the resource is allocated to a process, it is consumed and ceases to exist. Examples of such kind of resources are messages, interrupt signals and V operations in a semaphore.

In the context of the *Coffman conditions*, the resource kind we are interested in for our thesis research is the *reusable* resource. That is, since *consumable* resources are not necessarily unique or even bounded in number, we cannot detect deadlock caused by a process requesting this type of resource. This constraint on the resource type is consistent with the conclusion we made from listing 4.1 in Section 4.1.1: infinite wait for a consumable resource such as a message or an event is treated as *liveness* issue in our thesis context.

4.1.4 Deadlock Prevention and Avoidance

There are several algorithms available to prevent or avoid deadlocks. Although they are both deadlock handling strategies, there are differences between these two.

Deadlock prevention is done by constraining how processes' requests for resources can be made in the system and how they are handled. Some fairly common algorithms to prevent deadlock from happening are:

- impose a linear ordering for the access to resources for each process;
- release a resource before the next resource request; and
- request all resources together and release all together after work is done.

Any of these situations mentioned above result in the ”circular-wait condition” being avoided, and as a result, the potential deadlock is prevented. A reasonably common goal behind designing such prevention algorithms is to make it impossible for one or more of the *Coffman conditions* to occur.

Deadlock avoidance, on the other hand, is performed by the system dynamically. At run-time, every resource request is considered and a decision has to be made whether it is safe to grant the request at this moment. To do this, the system requires additional *a priori* information regarding the overall potential use of each resource for each process. One famous avoidance algorithm is called the *Banker's algorithm*. It avoids deadlock by simulating the allocation of pre-determined maximum possible amounts of resources. The basic idea behind the *Banker's algorithm* is to conservatively keep the system always in a safe state. A safe state means that all processes within the system can eventually terminate. The safe state is determined by knowing how much of each resource the system has available, how much of each resource each process is concurrently holding, and how much of each resource each process is possibly going to request in the future. With the knowledge of all these three, the *Banker's algorithm* can determine if a system state is safe by finding a hypothetical set of requests by the processes that would allow each to acquire its maximum resources and then terminate with returning its held resources to the system. Any state within which there exists no such set is considered to be unsafe.

The *Banker's algorithm* is overly pessimistic, since it always assumes each process will attempt to acquire its stated maximum number of resources. The advantage of using deadlock avoidance over deadlock prevention is that often more concurrency can be achieved, at the cost of constantly requiring dynamic information for each request. The difference between deadlock prevention and deadlock avoidance can be compared to the ways a traffic light or a police officer direct the traffic. Traffic lights use pre-designed rules to control traffic, while police officers can direct the cars dynamically according to the different situations on street.

However, these algorithms have drawbacks [Bel87]. For example, the *Banker's algorithm* requires a huge number of messages to be exchanged in a concurrent system in order to determine safe or unsafe states. This can be especially expensive in a distributed system. It also requires a linear order when acquiring resources, which is sometimes infeasible or impractical.

Other algorithms, such as *wait-die* and *wound-wait* [RSI78], have been suggested in a context when it is possible to abort or undo and restart processes. In the *wait-die* scheme, a process with smaller (older) time stamp is allowed to wait for the resource that is held by a process with a larger (younger) time stamp; or the waiting process should be aborted and restarted. In the wound-wait scheme, a process with a larger time stamp is allowed to wait for the resource held by a process with a smaller time stamp; the waiting process (older process) preempts the resource holding process (younger process) and the latter aborts and restarts. However, these algorithms could cause many process abortions. There are some algorithms that improve the abortion rate such as the deadlock avoidance algorithm proposed by Wu, Chin and Jaffar [WCJ02]. But their algorithm only works in the context of the resource AND-request model.

4.2 Our Deadlock Detection Algorithm

From previous examples of available deadlock prevention and avoidance algorithms, we find there is no perfect approach to prevent or avoid deadlock in different systems. We only can say there are different algorithms for different needs, but for each algorithm there is typically limitation to using it. Since a static approach is our adopted style in this research, we feel instead of preventing or avoiding deadlock it is more reasonable to propose an algorithm that can be used to guarantee deadlock-free property within the system design. In this section we present our design-based approach.

There are some specific advantages to using our deadlock detection algorithm:

1. Being a static approach, the algorithm is a general method; it can apply to any system that needs to be checked for deadlock under *Coffman conditions*.
2. To use our approach, there is no restriction on the type of resource request model.
3. Our design can be a foundation for algorithms to verify system application-level consistency and help check fairness/liveness properties.
4. Our design can be used as an extension to high level modeling methods, such as the Fondue concurrency addressing model introduced by Kienzle and Sendall [KS06].
5. The technique itself is easy and cheap to use.

4.2.1 System Synchronization Hasse Diagram

In order to guarantee a deadlock-free property, there must be at least one of *Coffman conditions* being violated. As we mentioned previously, in most concurrent systems it is common to allow processes to have mutual exclusion, hold-and-wait and no preemption abilities when they attempt to acquire resources. As a necessary condition to all models of deadlock, the circular wait condition is the mostly easily modified property not allowed to happen in deadlock-free systems. Since we aimed to develop a static approach that should apply to any concurrent system, we chose to focus on the circular wait condition in order to detect potential deadlocks within system design. Below we give the theoretical basis for our approach and show how it can be applied to actual concurrent designs.

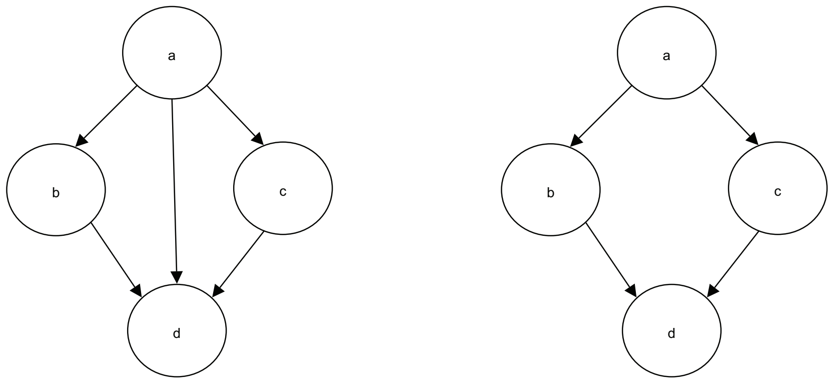
The Hasse Diagram

Helmut Hasse, a German mathematician, once introduced a simple picture of a finite partially ordered set. This picture, called as *Hasse diagram*, represents partially ordered set as a directed graph. In a *Hasse diagram*, elements of the set are represented by vertices of the graph, and there is a directed arc from x to y if and only if y is "larger than" x.

As mentioned, Hasse diagrams represent partially ordered sets. A partially ordered set, abbreviated as *poset*, is a set equipped with a partial order relation. A *partial order* is a binary relation R over a set S which is reflexive, anti-symmetric, and transitive. For example, if *a, b* and *c* are all in *S*, by definition of *partial order*, we have:

- Reflexive: aRa
- Anti-symmetric: aRb and bRa then $a{=}b$
- Transitive: aRb and bRc then aRc

The Hasse diagram actually draws the *partial order* in its *transitive reduction form*. The *transitive reduction form* of a binary relation R on a set S is the actual smallest relation R on S such that the *transitive closure* of R is the same as the actual *transitive closure* of R with all edges added, even "redundant" ones. To understand this better, consider the following example. If you think S is a set of airports and aRb means there is a direct flight from airport *a* to airport *b*, then the *transitive closure* of *R* is the relation it is possible to fly from *a* to *b* in one or more stops. With the knowledge of all above, we can see Figure 4.4 shows the graph of non-transitive binary relation on the left side and its transitive reduction form on the right.



(a) Non-transitive (b) Transitive Reduction Form

Figure 4.4: *Transitive and Non-transitive Graphs*

4.2.2 Using the Hasse Diagram to Detect Circular Wait Conditions

One important prerequisite for circular wait condition is to have two or more processes form a circular chain where each process waits for a resource that is held by the next process in the chain. So in order to detect a circular wait condition, we need to have two kinds of key information:

1. We should know during each operation's execution: the objects that need to be acquired, the length of time each object is "held" by the operation, and the ordering of the objects to be acquired by the operation.

2. We should know the concurrency relationship among different operations in terms of their executions; that is, can operation A execute in parallel with operation B?

If we represent the time sequence of all objects to be requested by one operation as a *partial order* relationship, then in a *poset* of all objects we can assume the objects required later are covered by the objects that are required earlier and still being held by the operation. By knowing first type of key information, we can determine the partial ordering relationship for all objects to be acquired by one specific operation; by knowing the second type of key information, we can uncover what are the shared objects between every two concurrently executing operations. A Hasse diagram easily encodes this information. Nodes represent object sets, and edges represent precedence in object acquisition.

In order to get detailed information about how objects are acquired by one operation during its execution, especially the sequence of the objects being acquired, we can consult with operation *sequence diagram*. The *sequence diagram* consists of vertical lines called lifelines. Each lifeline element represents the life of a given object. Lifelines are connected by horizontal lines containing messages that pass from one object in the operation (scenario) to the next object in the operation (scenario). By following the first horizontal line that represents the initialization of the operation until the end of the operation, the sequence of all objects being acquired during operation execution can be determined by checking the vertical positions of the incoming messages to these objects. That is, if the horizontal incoming message to object A is lower than the horizontal incoming message to object B, then it means the operation acquires object B before object A during its execution. Figure 4.5 shows one example, in which operation `foo` is started by an actor sending a synchronous `foo()` message to object A first. In other words, operation acquires object A first, and then a synchronous message `bar()` is sent from object A to object B in order to acquire the latter. Since both messages are of synchronous type, `foo()` cannot complete until `bar()` finishes. (Dark grey scale in the object's background represents a monitor lock on this object, therefore object A and object B are both locked during the execution of the operation.) Thus, in `foo`, acquisition of A precedes acquisition of B.

We can represent the example shown above as a Hasse diagram, shown in Figure 4.6. To distinguish from a normal Hasse diagram, which more generally represents ordering relations, we call the Hasse diagram used to represent the objects locking sequences as the *System Synchronization Hasse Diagram* or *SSHD*.

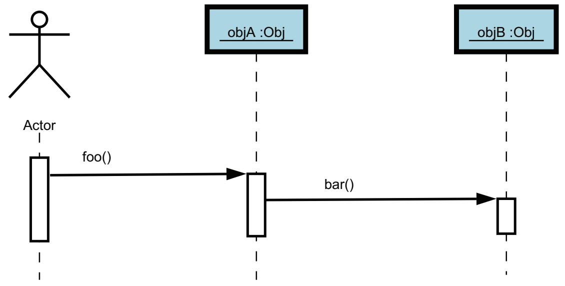


Figure 4.5: *Ordering of Objects Acquaintance for Operation* `foo`

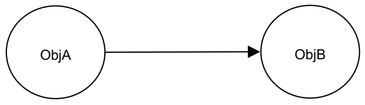


Figure 4.6: *SSHD of Objects Locking Sequence*

The second key piece of information is to find out all the operations that could execute concurrently. Fortunately, by using Fondue Protocol Model, especially by using *divide-by-collaboration* technique, the essential concurrency information can be extracted and presented in a table [KS06], such as Table 4.1. Table 4.1 lists all possible messages (operation invocations) that can appear in parallel during the execution of a particular system. The *Y* cell represents the messages on its row and column can occur concurrently; the *N* cell represents the messages on its row and column cannot occur concurrently.

	input message1	input message2	input message3	...
input message1	Y	Y	N	...
input message2	Y	Y	N	...
input message3	N	N	N	...
...	...	...	...	...

Table 4.1: *Potential Concurrent Input Messages for a Given System Template*

With both key pieces of information, we can use our proposed SSHD to verify the deadlock-free property of the system under study. First, we can represent each system object that will be locked during the execution of any operation as a vertex. By checking an operation's *sequence diagram*, we can find the operation's locked objects and the sequence of these objects to be acquired by the operation. Among all the vertices, draw lines from the objects that are locked earlier and are still held by the operation to the objects that are locked later by the same operation. There are several notes on our proposed System Synchronization Hasse Diagram:

1. The System Synchronization Hasse Diagram differs from normal Hasse diagram by the fact that its vertices are not necessarily connected in their *transitive reduction form*.

2. We use different types of lines to represent different resource request models. For example, as shown in Figure 4.6 we use the solid line to show AND request model; we also use the dashed line to show OR request model, and AND-OR request model can use both types of lines. For example, Figure 4.7(a) shows that the resources can be requested as: R1 and R4; or R1 and R2 and R3.

3. Redundant links are only displayed once. In case the dashed line (OR request) overlaps with the solid line (AND request), we always draw the solid line instead of the dashed line. The reason why we let AND dominate OR is because the AND request model has more restrictions on the resource acquisition. In order to guarantee the deadlock-free property, we always assume the system is running in the most restrictive way; this gives us a conservative solution. For example, assume there are three processes with one requesting resources R1 and R2; one requesting R2 and R3 and R1; and the third process requesting R1 and R2, or R1 and R4, then the System Synchronization Hasse Diagram for this example is shown in Figure 4.7(b). Note that the R1→R2 edge is solid despite the final OR requirement.

After all vertices are connected properly according to the system operations, we can use any searching algorithm such as *Depth First Search* to find potential cycles within the System Synchronization Hasse Diagram. If the searching result shows the diagram is

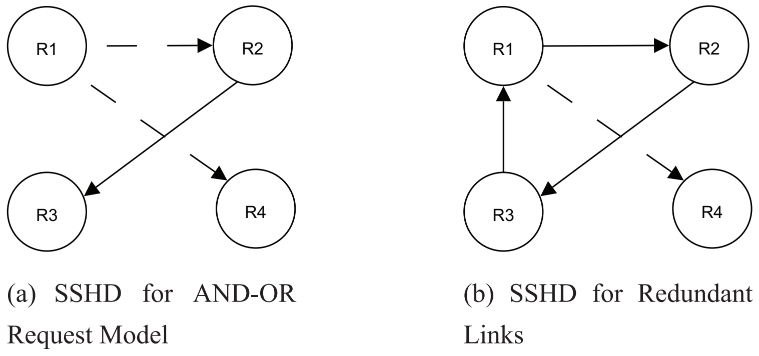

R1
R2
R3
R4
(a) SSHD for AND-OR Request Model
R1
R2
R3
R4
(b) SSHD for Redundant Links

Figure 4.7: *SSHD Examples*

acyclic, then we can guarantee the system is deadlock-free. Alternatively, if a cycle is found we need to do some further investigations to determine if the found cycle really means there is a deadlock flaw in the system design. Depending on the types of resource request model, the possible further investigations are:

- if the resource request model is OR request model or AND-OR request model, then we need to use available knot detection algorithms, such as the ones introduced in [MC82, CJ86], to determine if the system really has deadlock(s).
- if the request model is AND, then we can use critical regions to translate the SSHD to its transformation form and check cycles again. The details about the transformation will be discussed in chapter 6.

4.2.3 Applying the SSHD to Dining Philosophers Problem

The Dining philosopher problem is an illustrative and classic example of a computing problem with potential deadlock concerns in concurrency. In order to verify that our proposed System Synchronization Hasse diagram works, dining philosopher problem is one of the best cases to study first.

The dining philosopher problem was invented by E. W. Dijkstra in 1971. The problem consists of *n* philosophers who spend their lives just thinking and eating in a dining room with a circular table. All these *n* philosophers are sitting around the circular table with *n* chopsticks on it. The following Figure 4.8 shows the positions of the philosophers and the chopsticks, when *n=5*.

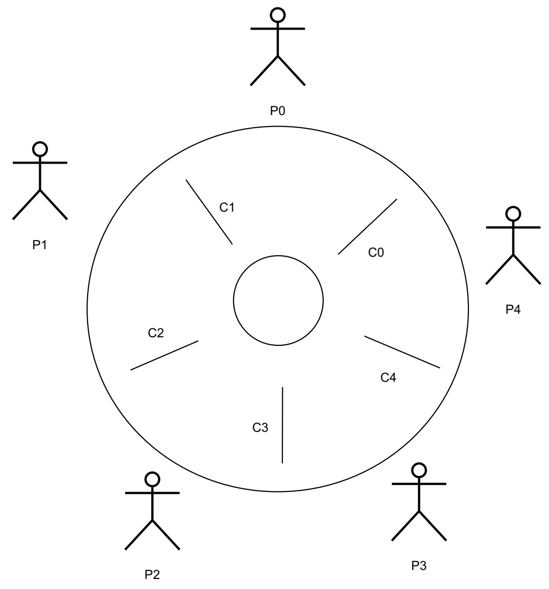


Figure 4.8: *Dining Philosopher Problem*

Each philosopher thinks. When he gets hungry, he sits down on his assigned chair and picks his left hand chopstick followed by his right hand chopstick. If a philosopher can pick both chopsticks then he can eat. After a philosopher finishes his eating, he puts down the chopsticks and goes back to thinking again. In principle, at least two philosophers should be able to eat concurrently. The problem happens when all *n* philosophers decide to sit down and pick their left hand chopsticks simultaneously. A deadlock can appear at that moment since no philosopher can find his right hand chopstick and meanwhile no one is willing to put down his left hand chopstick. Figure 4.9 represents dining philosophers `eat` operation in a sequence diagram. In this sequence diagram, the `eat` operation is executed by sending two synchronous messages to two chopsticks laid on both sides of the philosopher. Since both messages are synchronous type, then only when both chopsticks are picked up can the eat operation be completed.

The previous textual description gives the key information of the first type. For the key

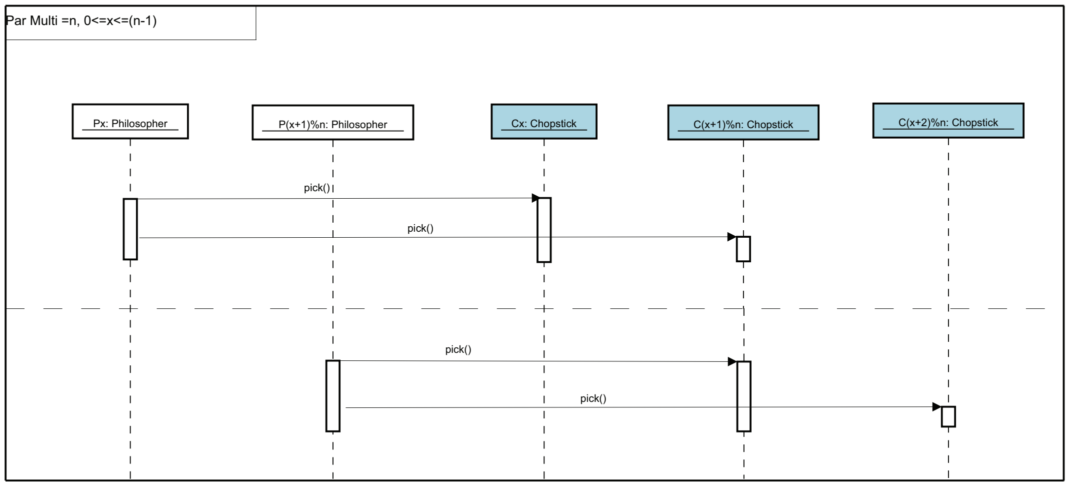


Figure 4.9: *Dining Philosopher Sequence Diagram for* `eat` *Operation*

	`pick()`
`pick()`	Y

Table 4.2: *Potential Concurrent Input Messages for* `pick()`

information of the second type, in this example it is obvious that there is only one `eat` operation; and the `eat` operations from different philosophers can be executing concurrently. The table of concurrent input messages for dining philosopher problem is quite trivial, and is shown in Table 4.2.

With the necessary key information at hand, we can start to draw SSHD for the dining philosopher problem. First, each requested resource, which is chopstick in this example, is irreplaceable. For example, when philosopher A wants to pick his left hand chopstick, no other chopstick can substitute his choice. Plus, each philosopher needs two chopsticks to eat, therefore the resource request model here follows the AND request model. Each chopstick is represented as a vertex in the SSHD since it is locked during eat operation. Furthermore, each eat operation with acquaintance of both left hand side and right hand side chopsticks can be represented as an arrow coming from a vertex representing the left chopstick to the vertex representing the right chopstick, as illustrated in Figure 4.10. Similarly, Figure 4.11 displays the SSHD integrating all philosophers `eat` operations.

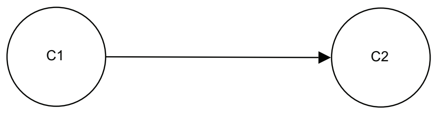


Figure 4.10: *SSHD for Philosopher1 (P1) Eat Operation*

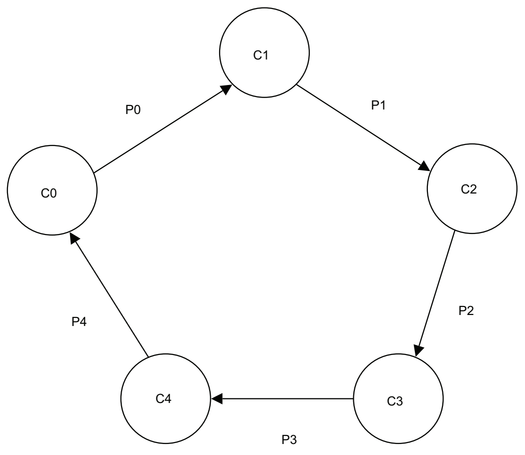


Figure 4.11: *SSHD for Dining Philosopher Problem*

The extra message besides each arrow in Figure 4.11 denotes which philosopher executes the matching eat operation. Very obviously, there is a cycle within Figure 4.11. In other words, there is a "circular wait condition" which occurs within the dining philosopher problem. Since all synchronous messages `pick()` can be sent concurrently as shown in Table 4.2 and the resource request model is AND request, there is no need to do any further investigation of our found result. We can conclude that dining philosopher problem is not deadlock-free.

4.3 Online Auction System Case Study

Having demonstrated the basic principles, we now show our system applied to a much more complex and realistic example, our online auction system. Our original online auction

	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 4.3: *Potential Concurrent Input Messages for Online Auction System*

system design was proposed in chapter 3 without verifying its deadlock-free property. In this section, we are going to verify this property by using the System Synchronization Hasse Diagram.

For simplicity purpose, we assume the resource request model in online auction system is AND request. To be consistent with Kienzle and Sendall's studies [KS06], we are focusing our verification based on `placeBid()` and `closeAuction()` operations. From the *AuctionView* state displayed in Figure 3.4(b)(Page 30), the potential concurrent input messages for online auction system generate Table 4.3 [KS06].

In order to know what objects are needed by `placeBid()` and `closeAuction()` and the sequence of these objects being acquired, we can consult with original system design, as displayed in Figure 3.5 (Page 32).

From Figure 3.5, we can see objects *auctionState*, *currentBid*, *currentAccount*, *previousAccount*, *history* and *newBid* are all locked by using monitors. Thus, only these objects need to be mapped as vertices in the SSHD. The length of each object being held by each operation is represented by the length of the bar in objects lifelines. *Auction* and *System* objects differ from other objects by their autonomous properties. That is, *Auction* object serves as a control center that receives different `placeBid()` messages from different users and then sends out messages to different objects in order to fulfill users request; the *System* object automatically sends out message to the *Auction* object in order to close the auction when time is up. Figure 4.12 shows the SSHD for `placeBid()` and `closeAuction()`.

From Figure 4.12 we can find a cycle between *auctionState* and *history* objects. With all

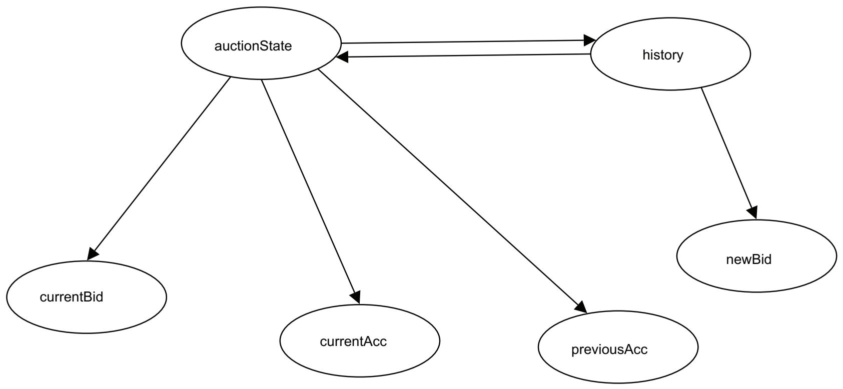


Figure 4.12: *SSHD (with Deadlock) for* `placeBid` *and* `closeAuction`

the information we have so far, we can conclude that the original online auction system design is not deadlock-free. The deadlock happens if `placeBid()` and `closeAuction()` are executing concurrently with `placeBid()` holding *auctionState* object and waiting for *history* object held by `closeAuction()`, and `closeAuction()` holding *history* object and waiting for the *auctionState* object held by `placeBid()`. There are several techniques we can use to break this circular wait condition. The reason why we designed `closeAuction()` to let it lock the *history* object is to guarantee that by holding the lock on *history*, no more `placeBid()` can be made after the auction time has expired. But since this design decision led to deadlock, we changed our design solution to the one as shown in Figure 4.13. In the new design, we exempt `closeAuction()` from locking *history* during its execution. Meanwhile we still can guarantee that no `placeBid()` is going to be successful after auction time is expired. This is realized by adding a new check on time before new bid is inserted into *history*. The corresponding SSHD (Figure 4.14) confirms that the new system design is deadlock-free.

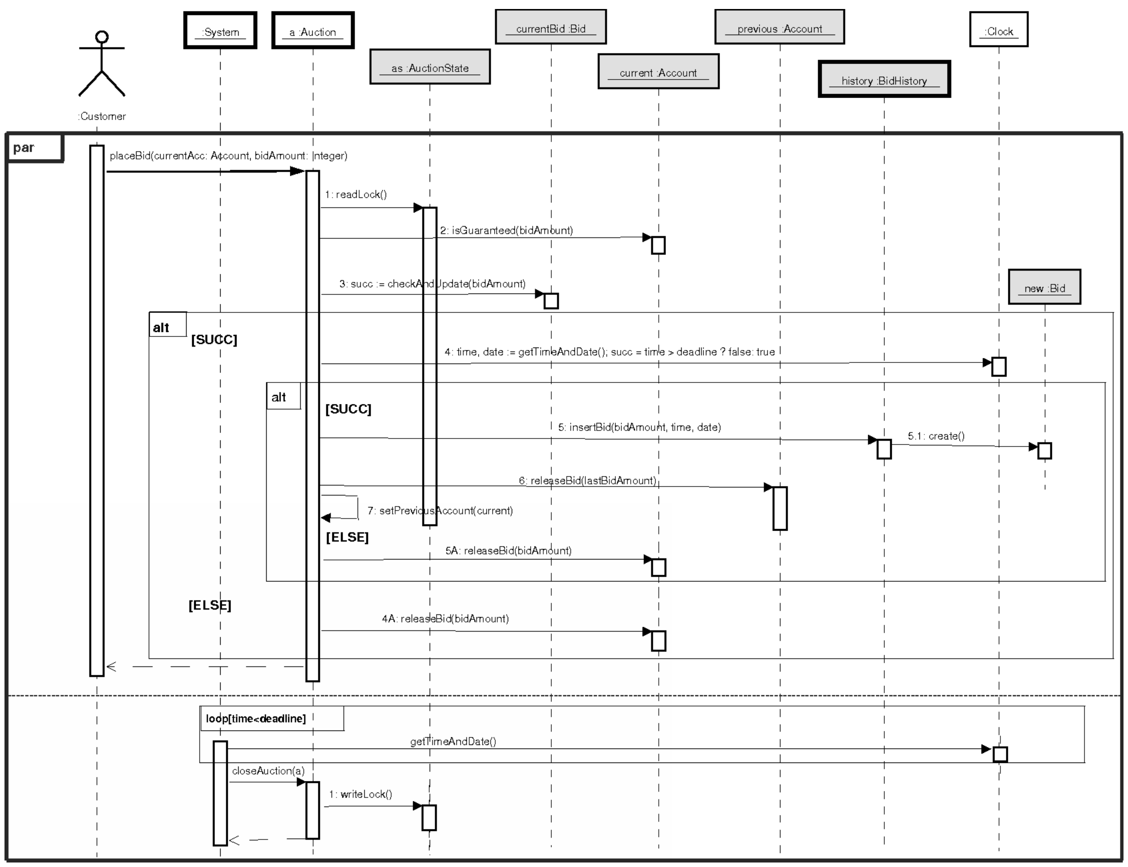

:Customer
:System
a :Auction
as :AuctionState
currentBid :Bid
current :Account
previous :Account
history :BidHistory
:Clock
new :Bid
par
placeBid(currentAcc: Account, bidAmount: Integer)
1: readLock()
2: isGuaranteed(bidAmount)
3: succ := checkAndUpdate(bidAmount)
alt
[SUCC]
4: time, date := getTimeAndDate(); succ = time > deadline ? false: true
alt
[SUCC]
5: insertBid(bidAmount, time, date)
5.1: create()
6: releaseBid(lastBidAmount)
7: setPreviousAccount(current)
[ELSE]
5A: releaseBid(bidAmount)
[ELSE]
4A: releaseBid(bidAmount)
loop[time<deadline]
getTimeAndDate()
closeAuction(a)
1: writeLock()

Figure 4.13: *Deadlock-free Sequence Diagram for* `placeBid` *and* `closeAuction`

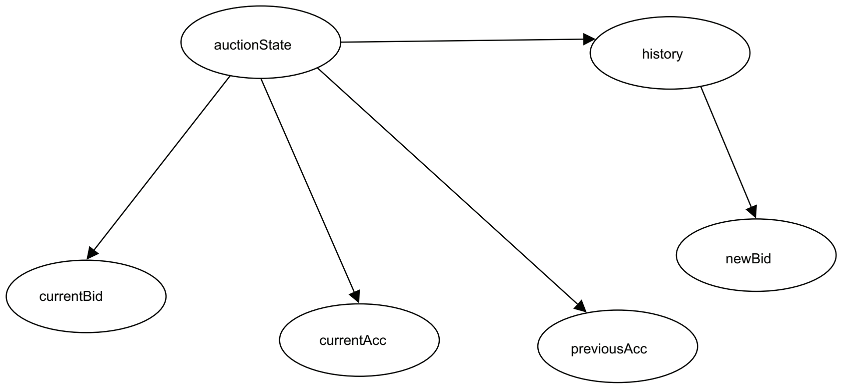


Figure 4.14: *SSHD (without Deadlock) for* `placeBid` *and* `closeAuction`

Chapter 5
Deriving a Correct Concurrent System Design

One of the requirements of system design verification is to check the validity of the conceptual system state. For the non-concurrent system, C.A.R Hoare once elucidated how to use sets of axioms and rules of inference in proofs of the properties of the system state or computer programs [Hoa69]. His introduction included a new notation: $P\{Q\}R$. This notation may be interpreted as the triple $P\{Q\}R$ is true if, assertion P is true before initiation of a program Q and assertion R will be true on the program's termination. According to the *Fondue operation schema* introduced in Chapter 2, P and R are specified as pre- and post-conditions of the running program.

Verification of the conceptual state of the concurrent system, however, is non-trivial. Besides the system states before and after the running program, we also have to consider the state during the execution of the program. Kienzle and Sendall integrated this concern into the software development model by extending the original Fondue development method to its concurrency addressing version [KS06]. Following the new Fondue method, the developers can have a set of system operation schemas that specify the effects of every system operation on the conceptual system state. These schemas contain concurrency-related information as well, since they explicitly tag all conceptual system state that is accessed by operations, which can potentially execute concurrently. That is, when a set of state modifications can only be executed under certain conditions, and if these conditions are likely to change because of other concurrently executing operations, then these conditions are tagged as *rely conditions*; and the state modifications that have to be performed “atomically” with respect to these conditions are packaged within *rely conditions*. This chapter describes how to use the new Fondue method to verify the conceptual state of the concurrent system and use the verification result to come up with a design that implements the required state changes in a correct way, even if the other operations are executing concurrently.

5.1 Application-level Consistency and Concurrency

The state of an application or system is composed of many data structures and objects. Depending on the application, the state stored in one data structure might be linked to the state in some other data structure. For example, in an application where a customer owns a set of accounts, a design might store in the customer object a list of account references, one for each account the customer owns. Vice versa, each one of the account objects stores a reference to the customer that owns the account.

In general, a consistent change to the application state involves many individual state modifications on data structures and objects. Usually, a system operation groups together these individual operations, and hence a successful execution of a system operation results in a consistent application state change. For instance, if an account changes owner, not only the owner reference in the account must be changed, but the corresponding account lists in the new owner and the old owner must be updated in order to achieve application-level consistency.

5.1.1 Definition of Application-level Consistency

In the database world, transactions guarantee the ACID properties (Atomicity, Consistency, Isolation and Durability) for accesses made to transactional objects [GR93]. One of the properties, *consistency*, states that a transaction produces consistent results only; otherwise it aborts. A result is consistent if the new state of the application fulfills all the validity constraints of the application according to the application’s specification.

Unfortunately, this requirement is very hard or even impossible to verify. The state of an application tends to be very complex, and the number of possible consistency constraints among data items is huge. In order to still guarantee consistency, current transaction systems rely on the application programmer to only commit a transaction if the application state has been updated in a consistent way. A transaction must be written to preserve consistency. That is, each transaction expects a consistent state when it starts, and recreates that consistency after making its modifications, provided it runs to completion. Note that the intermediate states produced by a transaction during execution of its individual operations need not necessarily be consistent. The transaction system guarantees only that the execution of a transaction will not erroneously corrupt the application state.

Inspired by the transaction world, we will refer to the property of a state that fulfills all the validity constraints of the application according to the application's specification as *application-level consistency* in this thesis. A system operation moves the system from an *old application-level consistent state* that satisfies the validity constraints of the application to a *new application-level consistent state*. In the Fondue development method, the conditions under which a system operation can execute are described in the pre-condition, and the state changes that define the desired new consistent state are given in the post-condition. During the execution of an operation, the application state might however not always be consistent.

In the presence of concurrency, special care must be taken to insure that system operations that execute concurrently do not interfere with each other, i.e. do not see intermediate or temporary inconsistent application state. To make this easier, the sequential operation schema of Fondue (see chapter 2) has been extended. It now allows the use of a *rely expression* within the post-condition:

rely rely-condition **then**
    rely-effect-expression
fail
    fail-effect-expression
endre

It prescribes that the new application state satisfies *rely-effect-expression* if and only if *rely-condition* was satisfied during the time in which the required state changes were performed; otherwise there is an obligation on the operation to fulfill *fail-effect-expression*.

5.1.2 Motivation

It is highly non-trivial to come up with a design that correctly implements a system operation with rely conditions in a concurrent environment. The purpose of the following sections in this chapter is to present a design technique that helps the developer to derive a correct concurrent design from a concurrent system operation schema. In addition, our algorithm can also be used to detect *fairness sensitive components* that can be used to impose sophisticated resource sharing policies to prevent starvation and other fairness issues. This, however, is presented in chapter 6.

5.2 Rely Diagram

The ultimate goal of the design phase is to define how objects interact at run-time to achieve the specified system functionality. This can be done, for instance, using interaction diagrams, such as sequence diagrams. To make the jump from the declarative operation schema to the interaction diagram easier, we first describe in this section how to extract concurrency-related information from a concurrent operation schema. We show how to transform the operation's rely-conditions to a *rely diagram*, and then finally how to use the rely diagram to construct the sequence diagrams for each system operation that satisfy application-level consistency.

5.2.1 Extracting Concurrency-related Information

The original Fondue operation schema only describes the system state before and after the execution of a system operation. In Chapter 2, we have shown how to extend the original Fondue operation schema from its sequential version to its concurrent version. From the concurrent operation schema template, we can see that in order to guarantee the post-condition to be true, not only the Boolean expression specified in the effects of the operation should be true, but also the rely-condition should be met during the whole execution period. In other words, the statements specified in the post-condition of the concurrent operation schema no longer possess instantaneous semantics.

Figure 5.1 shows a concrete example that follows the concurrent operation schema template. The operation is designed to withdraw a certain amount of money from an account. In our case, there is no condition that could prevent the operation from executing, so the precondition is always true. However, during the execution of the `withdraw` operation we require the account balance to remain equal or larger than the amount to be withdrawn. This is due to the fact that other system operations could potentially access the same account object concurrently. For instance, other concurrent withdraw operation executions could also want to take money from this account. The rely expression makes sure that either the requested amount of money was successfully withdrawn from the account and during the execution of this state change no other operation modified the balance in such a way that there was not enough money left on the account, or else the *InsufficientFund* message has been sent to the customer.

5.2.2 The Rely Diagram

This subsection describes the generation of the rely diagram based on the rely-conditions. The rely diagram can then in turn be used to construct the design of an application-level consistent operation.

Constructing a Rely Diagram

By looking closely at the rely expression specified in section 5.1.1, we can see that each rely expression has a rely-condition, which is a predicate that can contain assertions on multiple objects or object attributes. We will call the objects and attributes listed in the rely condition the rely expression's *dependant objects*. Correspondingly, the operation that has the rely-condition specified in its operation schema is called the dependent object's *owner operation*. For example, in Figure 5.1 the dependent object of its owner operation `withdraw` is *acc.balance*.

In order to construct a rely diagram, we can put the owner operation name on the top of the diagram; then a dashed arrow is drawn from the rely expression to each of its dependent objects. Figure 5.2 is a simple rely diagram for the `withdraw` operation.

It is important to note that, especially when transforming *if expressions* from the se-

```
Message (type) declarations:
        InsufficientFunds_e(); DispenseCash(amount: Money);
        //define the messages that are to be used during the operation
Operation: Bank::withdraw (acc: Account, request: Money);
Description: Request from an ATM of some amount to be taken
        from a given account. Cash is dispensed only if account has sufficient funds.
Scope: Account;
Shared: acc.balance;
Messages: ATM::{InsufficientFunds_e; DispenseCash;};
        // messages types to be output by the operation
Pre: true;
Post:
        rely (acc.balance@pre ≥ request) then
            acc.balance = acc.balance@pre - request &
            sender ˆ dispenseCash (request)
        fail
            sender ˆ insufficientFunds_e ()
        endre;
```

Figure 5.1: *Concurrent Operation Schema for* `Withdraw`

quential operation schema to *rely expression* in order to take into account the shared resources [KS06], there could be *nested* rely expressions in the same concurrent operation schema, such as Figure 5.3.

In Figure 5.3, we use numerical indices to show the depths of the nested rely expressions. In the concurrent operation schema with nested rely expressions, we call the inner rely expression the *dependent rely expression* of its outer rely expression. Of course, the inner rely expression still has its own dependent objects. For each rely expression in the rely diagram, there is at most one direct inner dependent rely expression, while the number of its dependent objects varies. Figure 5.4 shows the rely diagram of the concurrent operation

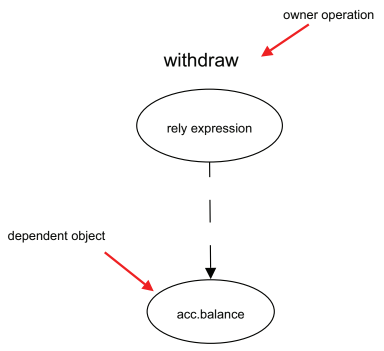


Figure 5.2: *Rely Diagram for* `withdraw`

schema with three nested rely expressions.

The Dependent-status-passing Rule

It is very important to be aware that the objects in the design solution domain are not the same as the concepts in the problem domain. That is, migrating from analysis to design can result in that some concepts may be implemented using several objects; or alternatively, some concepts may be implemented as attributes of classes. The granularity of objects can affect the system in several ways. A too fine-grained decomposition can lead to system with thousands of objects, which can result in the extreme difficulty in system analysis and huge communication overhead; a coarse decomposition can lead to a bulky architecture, and objects within have unclear responsibilities [KS06].

Due to the objects mapping from problem domain to solution domain, it is highly possible that some new objects that only appear in the Design Class Model (class diagrams generated based on the system design) are not treated as the dependent objects, which they should be. In this case, if a dependent object or its mapping object in the Design Model has a one-to-one association with some other "new" object that is only in the system Design Class Model, then the "dependent" status of the first object is automatically granted to the second object, and we should include the new dependent object in the rely diagram. We name this as *dependent-status-passing rule*. For example, Figure 5.5 is a simple Design Class Model. In the figure, if the *customer* object is only in the solution domain, then the

```
rely rely-condition then
    rely rely-condition then            (1∗)
        rely rely-condition then        (2∗)
            rely-effect-expression
        fail
            fail-effect-expression
        endre
    fail
        fail-effect-expression
    endre
fail
    fail-effect-expression
endre
```

Figure 5.3: *Nested Rely Expression*

dependent object status of the *account* will be automatically passed to the *customer*, which should also be included in the rely diagram. Of course, during the time when we use Design Class Model to include new dependent objects in the rely diagram, all the dependent objects included in the rely diagram are updated to system design objects; this implies the old dependent objects coming from problem domain should also be replaced by their mapping objects in the Design Class Model. As displayed in Figure 5.1, the rely expression in the concurrent operation schema shows that *acc.balance* is `withdraw` operation's dependent object. Since in the Design Class Model (Figure 5.5), every *Account* object associates itself with only one *Customer* object by the *own* relationship; therefore, according to our proposed dependent-status-passing rule, the *customer* who owns the account is also the dependent object of `withdraw` operation. As a result of being dependent objects, both the *account* and its owner cannot be changed during the execution of `withdraw` operation.

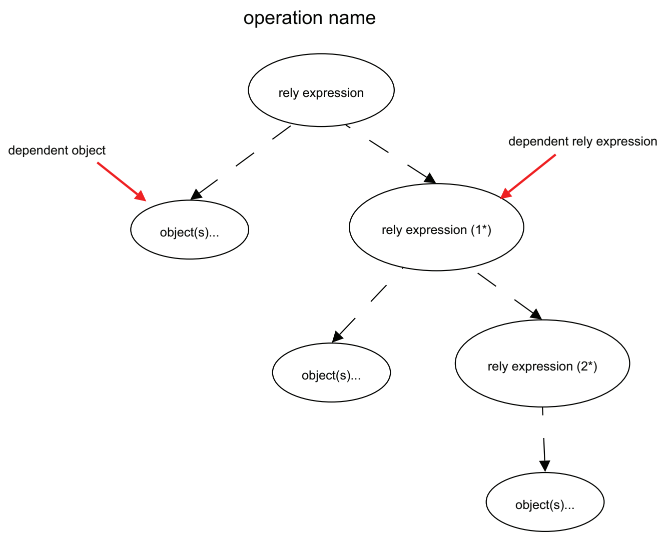


Figure 5.4: *Example of Rely Diagram Structure with Nested Rely Expressions*

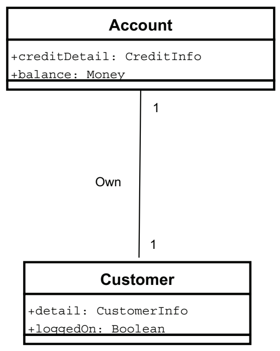


Figure 5.5: *Design Class Model of Own Relationship*

Change Dependent Object's State in Two Ways

It is the nature of the concurrent systems to have more than one operation execute simultaneously and hence the state of one dependent object can be altered by another concurrently running operation during its owner operation's execution. The change of dependent object's state can either be "positive" or "negative" with respect to the owner operation's rely-condition. Change in a "positive" way means the state of the dependent object is changed in a way that favors the owner operation. In Figure 5.1, if during the execution of `withdraw` operation there is another operation depositing money into the same *account*, then we say this `deposit` operation changes *acc.balance* in a "positive" way, since more money in the account does not prevent the withdraw from succeeding. Change in a "negative" way means that the state of the dependent object is changed in a way that could decrease the chances of the owner operation running successfully. For an instance, in Figure 5.1, if during the execution of `withdraw` there is another operation transferring money from the same *account* to somewhere else, then we say this `transfer` operation changes *acc.balance* in a "negative" way. Transferring money decreases the balance of the *account* and consequently lowers the chance to pass the *rely-condition*: acc.balance $\geq$ request.

In order to find the objects whose states could be altered in a negative way during the execution of an operation, we can consult the system concurrency table that is built during the Fondue analysis phase. An example of such a table is shown in Table 4.1(Page 45). By looking at the *Shared* clauses of all concurrently executing operations, we can find out which dependent objects can potentially be altered by other operation(s) during the execution of the owner operation; by examining the post conditions of concurrently running operations, we can determine in which way (positive or negative) can the state(s) of the owner operation's dependent object(s) be affected.

Refining the Rely Diagram

The initial rely diagram can be refined based on the effects from all other concurrently running operations, therefore it can display which dependent objects could be potentially negatively affected. Figure 5.6 is one example of a refined rely diagram.

The difference between the general rely diagram and its refined form is that in the refined version, there are some *"-"* notations besides some of the dependent objects. This *"-"* notation indicates that the associated dependent object might be changed in a "negative" way during the execution of the owner operation. For the dependent objects without *"-"* notations, these are the objects that either will not be altered during the owner operation's execution, or they could be altered in a way that favors the owner operation's execution.

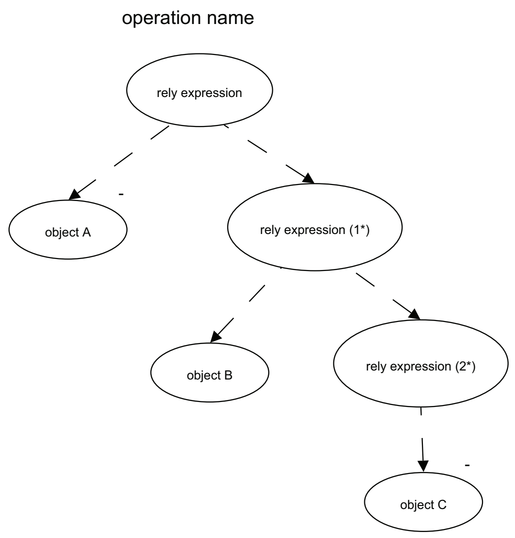


Figure 5.6: *Example of Refined Rely Diagram Structure*

5.3 Guaranteeing Application-level Consistency Using Critical Regions

This section describes how to elaborate sequence diagrams that contain critical regions in accordance with the rely diagram to obtain a system design, which ensures that application-level consistency is always guaranteed.

5.3.1 Difficulty to Verify Rely-Condition

As we introduced in section 5.1.1, in order to guarantee application-level consistency, the system has to satisfy its running operation's pre-condition, rely-condition and post-condition. Verification of the pre-condition or post-condition is relatively easy; as these conditions can be examined before and after the operation's execution. For example, if the system operation interaction is designed using sequence diagrams, we can use the interaction operator **alt** together with different guards to model different conditions that direct different control flows.

However, verifying a rely-condition in the sequence diagram is non-trivial. A rely-condition, unlike a pre-condition or a post-condition, does not have instantaneous semantics. This implies that it is not sufficient to verify if a rely-condition holds by checking only once during the execution of the operation. In fact, in order to ensure the system always stays in its application-level consistent state, the rely-condition should be examined continuously during the execution of the operation. This is, of course, impossible from a computational point of view. It can also not be represented as such in a sequence diagram. For example, interaction operator **alt** in the sequence diagram cannot model continuous condition checking.

5.3.2 Using Critical Regions to Guarantee Non-Interference with a Rely-Condition

One way to solve the problem of guaranteeing that the rely-condition holds during the system operation execution is to check it, and then prevent it from being changed. This can be achieved by looking closely at the information found in the refined rely diagram. We absolutely do not want the operation's dependent objects' states to be changed in a "negative" way during the operation execution; however, a change in a "positive" way is allowed.

According to our approach, we only need to check the rely-condition once before executing the changes specified in the rely-effect expression; once the check is made, the system is designed in a way that "isolates" those dependent objects from being changed in a negative way until the end of the execution of the rely-effects. Based on the refined rely diagram, it is easy to identify the group of dependent objects that have to be isolated: the group consists of all objects with "-" notations besides.

With the property of the critical region as we mentioned in Chapter 2, it will be safe to cover the owner operation together with its dependent objects within the same critical region, without worrying about its dependent objects being altered during its execution. Again, please note it is not wrong to cover all dependent objects in one critical region. But this is a conservative approach. To achieve maximum concurrency in the system's performance, we should not include in the critical region the dependent objects without "-"

notations.

5.4 Online Auction System Case Study

In Chapter 4 we have refined our online auction system design to ensure it is deadlock-free; in this section we are going to verify its application-level consistent state. In order to do so, let's first look at the original concurrent operation schema and the related pseudo codes for `placeBid` operation; these are listed in Appendix A.

Problematic Scenario

By comparing the `placeBid` concurrent operation schema and the pseudo code created based on the design presented in Chapter 4, we discovered an interesting problematic scenario.

Let's imagine that Mike, a bidder, makes a first bid of \$50 for the auctioned item. At that time, the auction's previous highest bidder account is set to Mike's account. Later on, Larry, a different bidder, joins the auction and over bids Mike by \$100. The system accepts Larry's bid and sets the amount of money to be released by Larry's `placeBid` operation to \$50. But before updating the auction history object and setting auction's previous highest bidder account to Larry's account, Bill, yet another bidder, joins the auction. Bill, who makes an even higher bid of \$150, has his bid accepted by the auction system. Consequently, the amount of money to be released by Bill's `placeBid` operation is set to \$100. However, after Bill's bid is accepted by the auction system, the system does not go back to complete what was left for Larry's `placeBid` operation immediately. The system keeps proceeding Bill's `placeBid` operation by updating the auction's history object. Thus, the new previous highest bidder's account is set to Bill's account. Then, the system goes back to work on Larry's operation. Since Larry's bid has already been accepted by the auction system, its validity is not checked again before the auction's history object is updated to set the new highest bidder's account to Larry's account. The whole scenario is represented in Figure 5.7.

The application state is now not consistent anymore. First, when the system set Bill to

The order of bidders who make bids and the amount of each bid:

Mike (\$50) → Larry (\$100) → Bill (\$150)

The amount of money to be returned by each bidder's `placeBid` operation:

Mike (\$0) → Larry (\$50) → Bill (\$100)

The order of the highest bidders stored in the auction's history:

Mike → Bill → Larry

The amount of money the auction system will return to each bidder:

Mike (\$100) → Bill (\$50) → Larry (\$0)

Final bid winner:

Larry

Figure 5.7: *Mike, Larry and Bill Bidding Example*

be the highest bidder, it released the last highest bid's amount of \$100, which came from Larry's bid, to Mike's account. Then, when the system set Larry to be the highest bidder, it released the last highest bid's amount of \$50, which came from Mike's bid, to Bill's account.

Guaranteeing application-level consistency by using rely diagram

All the errors coming from the problematic scenario suggest that we may have "forgotten" to list *ArePlacedIn*, or its mapping design object *Auction.theHistory*, in the `placeBid's` rely-conditions. Using scenario analysis is one way to find such errors; but rely diagram analysis can give us a more systematic way to find application-level inconsistency errors. In the rely diagram analysis, `placeBid` is an owner operation that relies on the states of its dependent objects. According to the concurrent operation schema A.2, these dependent objects (after being mapped) in the system sequence diagram are: *auctionState*, *highBid* and *currentAccount*. Following the general rely diagram construction rule , we can represent these `placeBid's` rely-conditions in the Figure 5.8.

Figure 5.9 is part of the Design Class Model for `placeBid` operation. In the figure,

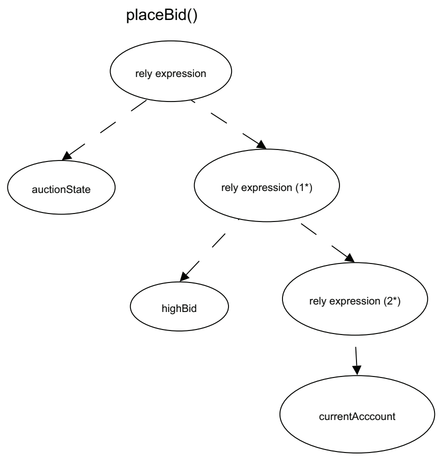


Figure 5.8: *General Rely Diagram for* `placeBid`

each *Auction* owns one *history* object and has zero or one *highBid*; the *history* consists of all previous valid bids and is the superset of the *highBid*. Since the change in the *history* can lead to the change of *highBid* (a change in the *history* implies a new higher bid is inserted into the *history*), then there is a one-to-one association between each specific *highBid* and each particular *history*. As the *highBid* is identified as a dependent object, then in order to prevent the *highBid* from being changed during the execution of `placeBid`, its superset *history* should not be changed either. According to the dependent-status-passing rule, the dependent object status is automatically passed from the *highBid* to the *history*. On the other hand, since each bid is made from only one account, therefore the depend object status of the *highBid* is passed to the *previousAccount* object, from which the *highBid* comes from.

As explained above, by applying the dependent-status-passing rule, we can find the hidden dependent objects even they are not originally specified in the operation's rely expression(s). Figure 5.10 is a new rely diagram for `placeBid` that includes all hidden dependent objects.

In Figure 5.10, all dependent objects have "-" notations besides, indicating they might be changed in a "negative" way during the execution of `placeBid`. Therefore, when we design our auction system in the sequence diagram, operation `placeBid` ought to be

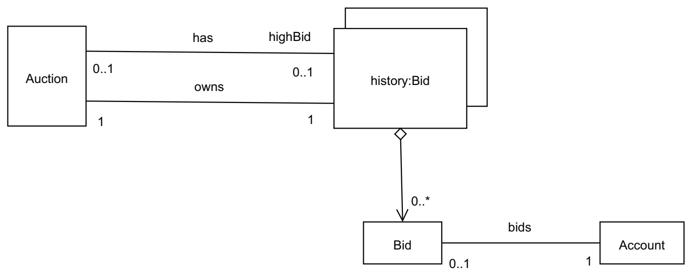


Figure 5.9: *Design Class Model (partial) for Auction and history*

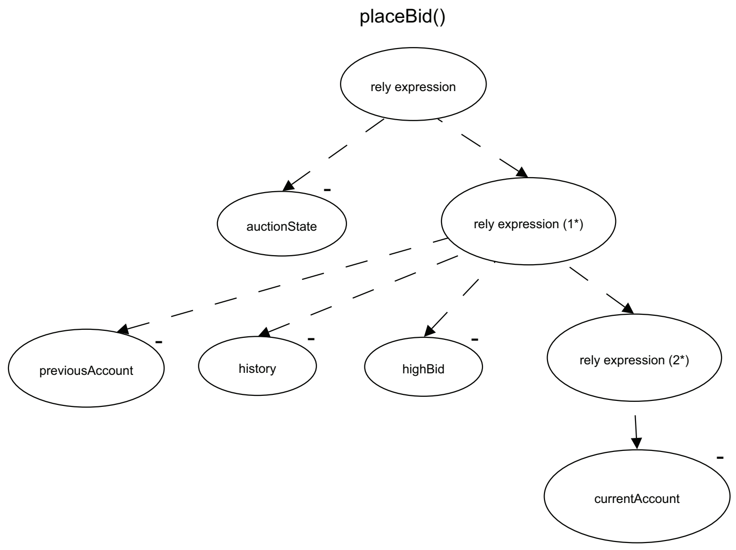


Figure 5.10: *General Rely Diagram (with Hidden Dependents) for* `placeBid`

covered together with these dependent objects within the same critical region in order to avoid the errors we presented in the problematic scenario. In our final design solution, however, *currentAccount* object is left outside the critical region. There is one reason for this design decision. We know from the requirements elicitation phase that the system user should be able to make several bids simultaneously. If we leave the *currentAccount* in the `placeBid's` critical region, then the user will not be able to make multiple bids at the same time. In order to resolve the difficulty just mentioned, the pseudo code of `isGuaranteed` (Figure A.4) is designed in a way to freeze the bid amount of money within the user's account; so as soon as `isGuaranteed` is passed successfully, there's no way for the current account balance to be lower than required during the rest of execution of `placeBid`. Although there is still possibility that another concurrent `placeBid` would negatively affect the *currentAccount's* balance before `isGuaranteed` is executed, in order to achieve more concurrency in the system's performance, leaving *currentAccount* out of the critical region is a tradeoff we have to afford in our system design. Therefore, based on the deadlock-free online auction system design (Figure 4.13) and the rely diagram analysis, we can derive an application-level consistent system design by grouping `placeBid` operation with its dependent objects except *currentAccount*, in the same critical region. The resultant design is shown in Figure 5.11.

Our case study does not follow entirely with the rely diagram analysis result due to the external user's requirement imposed on the system's concurrency performance; however, rely diagram can guide us to a correct system design decision that ensures system's application-level consistency property.

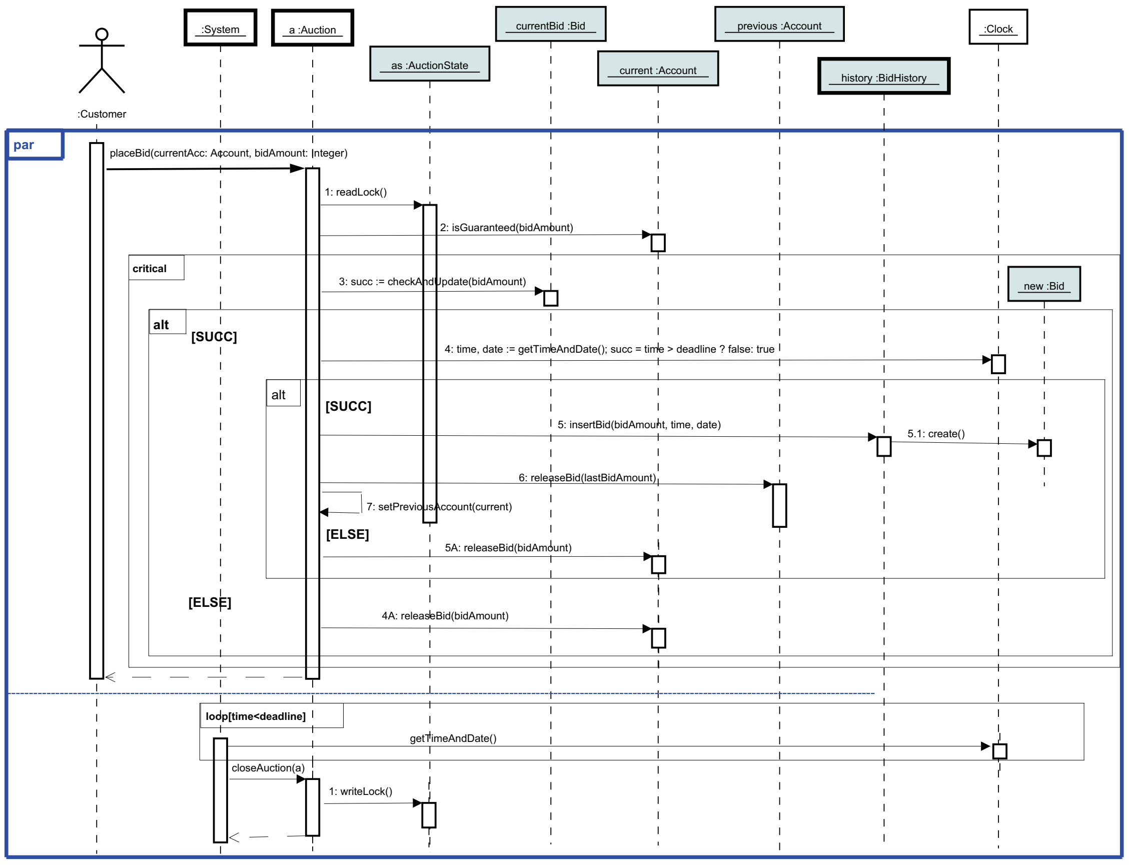

:System
a :Auction
as :AuctionState
currentBid :Bid
current :Account
previous :Account
history :BidHistory
:Clock
:Customer
par
placeBid(currentAcc: Account, bidAmount: Integer)
1: readLock()
2: isGuaranteed(bidAmount)
critical
3: succ := checkAndUpdate(bidAmount)
new :Bid
alt
[SUCC]
4: time, date := getTimeAndDate(); succ = time > deadline ? false: true
alt
[SUCC]
5: insertBid(bidAmount, time, date)
5.1: create()
6: releaseBid(lastBidAmount)
7: setPreviousAccount(current)
[ELSE]
5A: releaseBid(bidAmount)
[ELSE]
4A: releaseBid(bidAmount)
loop[time<deadline]
getTimeAndDate()
closeAuction(a)
1: writeLock()

Figure 5.11: *Application-level Consistent Sequence Diagram for* `placeBid` *and* `closeAuction`

Chapter 6
Fairness and Liveness in System Design

Problems of *liveness* and *fairness* have been studied intensively for concurrent systems. Fairness within a concurrent system ensures each process or thread makes progress whenever possible. There are many specific fairness definitions describing when and how often processes or threads are guaranteed to get a turn to run. For example, do we expect each thread to execute as often as others; or is it acceptable if one thread runs a thousand times for each run of another thread? Definitions of fairness for different systems may vary significantly, and so consideration of user specified fairness constraints during the verification of the concurrent system design is an important part of reviewing a system design in order to ensure the proposed design ties to the original user's requirement. In this chapter, we present a static approach to addressing fairness issue within a concurrent system design. In addition, since fairness assumptions can be used to prove systems liveness properties [KZ05], it is also possible that our fairness verification approach will ultimately help to address liveness issues.

6.1 Fairness/Liveness in Concurrent System

Fairness and *liveness* are two important concepts of concurrent systems. Although they are technically two different properties, people often find distinguishing them difficult. In this section, we review these two concepts by focusing on their definitions, the reason why people can get confused while distinguishing them, and finally the necessities and difficulties of verifying them.

6.1.1 Definition of Liveness/Fairness

When we verify a concurrent system design, it is useful to categorize the properties of the system into two classes: *safety* properties and *liveness* properties. Informally, a safety property stipulates that some "bad thing" does not happen during system execution [Lam79]. The properties of deadlock freedom and system application-level consistent state are the examples of safety properties which have been discussed in the previous chapters. On the other hand, liveness properties stipulate that some "good thing" happens during system execution [Lam79]. Liveness properties, which relate very closely to fairness properties, include the following [Lam89]:

1. *Starvation freedom:* asserting each waiting process will eventually make its progress.
2. *Termination:* asserting a program must eventually generate an answer; or the program will eventually terminate.
3. *Guaranteed service:* asserting every request will eventually be satisfied.

In concurrency theory, fairness is a concept used to define different levels of *timeliness* constraints placed on concurrent actions. In a multiprogramming environment, fairness abstracts the details of admissible schedulers; in a distributed environment, fairness abstracts the relative speeds of processors [AH98]. Although there is more than one meaning of fairness in computer science [Hoa78], usually fairness is defined in one of these two forms:

1. *Weak fairness:* also called *justice* [KPRS06]; all processes are executing infinitely often; put differently, no enabled system state transition is postponed forever.
2. *Strong fairness:* also called *compassion* [KPRS06]; if a process requests a resource infinitely often, then it should be allowed to get the resource infinitely often.

As mentioned previously, in reality the meaning of fairness to a specific system can be greatly extended. It can be covered in the user's specific system requirements. For example, it is fair to design the call switch system to forward all emergency (9-1-1) calls before it deals with the normal calls. In this example, the additional fairness constraint on the switch system is to allow 9-1-1 calls to be served earlier than normal ones. Hence, when it comes to a specific system, the meaning of fairness for this system is very subjective; it may include complex notions of priority in addition to simply being equitable. In order to verify a system in terms of its fairness, we should also put those system-customized requirements into consideration.

6.1.2 Fairness/Liveness Issues in Concurrent System

It is hard to enumerate all system specific fairness issues since each system may be expected to have different requirements and behaviors from this point of view. However, for system liveness issues, there are some general forms. Before we go any further, it is necessary to have a review of these common forms.

Lack of liveness will usually be manifested by one of the following forms:

1. *Starvation:* a waiting process will never get into its critical section or progress.
2. *No termination:* a process never terminates.
3. *No guaranteed service:* following a request the corresponding service is never performed.

For example, Listing 6.1 shows a code segment. If we look closely at the codes, we can find if process *p0* exits its critical region and goes back to check the *f1* value quickly enough, then process *p1* will never get a chance to check *turn's* value in its inner while loop. As a consequence, process *p0* will again gain control on the critical region. Since theoretically, this situation can happen repeatedly, process *p1* may be in a starvation status since its request to get control of the critical region is never satisfied. As a result, the system that is implemented by this code segment may eventually suffer a starvation issue.

Although the definitions of different types of liveness issues are clear, to prove them for a particular program or system is hard. It may not, at a practical level, even be meaningful to verify a system's liveness property, something Lamport points out in a later article:

```
f0 := false
f1 := false
turn := 0 // or 1
p0:
    f0 := true
    while f1{
       if turn != 0{
          f0 := false
          while turn != 0{}
          f0 := true
       }
    }
   // critical section
   turn := 1
   f0 := false
   // non−critical section
```

```
f0 := false
f1 := false
turn := 0 // or 1
p1:
    f1 := true
    while f0{
       if turn != 1{
          f1 := false
          while turn != 1{}
          f1 := true
       }
    }
   // critical section
   turn := 0
   f1 := false
   // non−critical section
```

Listing 6.1: *Code Segment with Starvation Issue*

> The question of whether a real system satisfies a liveness property is meaningless; it can be answered only by observing the system for an infinite length of time, and real systems don't run forever. Liveness is always an approximation to the property we really care about [Lam00].

As the comments shown above, in reality liveness property is usually tested as an approximation to the property that we really care about. For example, we may set our system's *"guaranteed service"* property to guarantee any of its processes to progress within a hundred hours instead of infinitively long. In such a case, the new constraint on the *"guaranteed service"* property is obviously tied uniquely to our specific system. In theory, however, it is possible for us to prove system liveness properties by using fairness without using approximation [Sis94], or use fairness to allow us to capture some liveness properties [AS84]. In the context of our thesis research, we aim to help verification of a system's liveness property in a realistic way, based on conservative approximations of system fairness.

6.1.3 When to Verify Fairness Properties

Fairness property can be verified at different software development phases. But verification at different phases may need different verification approaches and result in different result accuracy and verification costs. In this section, we are going to discuss when to verify the system's fairness property.

As discussed above, fairness is system implementation dependent. Sometimes even if the system passes verification at its design phase; it still may behave "abnormally" at runtime after its implementation. This is because the system implementation can include a scheduler that determines how to interleave the threads, and the scheduler may or may not provide any fairness guarantees; in most systems few actual guarantees are made. Lack of fairness or other scheduler guarantees can result in unexpected and undesired behavior, potentially as bad as any violated safety property. Due to this reason, we believe dynamic verification of fairness property at runtime is necessary. There are also some other reasons. First, it may happen that during the system's implementation developers tend to pay more attention to more obvious and critical system properties, such as safety, than the subtle notions of fairness and liveness. Verification of fairness after implementation could help to find the errors caused by the common focus on more obvious and deterministic bugs. Second, the user might change his or her fairness requirements for the same system without changing the system's original design. In this case, usually a different implementation may be necessary to satisfy the new requirements. Therefore, only verification after the new implementation can ensure the new requirement is fulfilled. Third, in extreme, safety-critical cases such as NASA projects or nuclear plant systems only verifying systems at runtime can give the most accurate guarantees on whether the required fairness properties are satisfied.

Besides verification after system's implementation, we can also statically verify fairness properties at the design phase. However, there is a tradeoff between the static and dynamic verification approaches. Static approaches at the design phase typically provide results conservative with respect to any possible input. This provides strong guarantees, but can also easily be too conservative to be useful. Runtime dynamic verification is more accurate, but may incur significant overhead and provide results either specific to a particular (set of) inputs, or too late to be useful. Specific problems will benefit more or less from either technique.

We concentrate here on verification in the late design phase. This allows maximal design information to be used, without necessarily requiring a complex runtime fairness monitoring system. Additionally, we feel fairness verification at the design phase can serve as a supporting step for the runtime verification, which is inevitable.

6.2 Verifying Fairness in Concurrent System Design

As mentioned in the previous section, a system's static verification of fairness property may not provide accurate enough results. In this section we introduce our static approach, attempting to minimize such problems. The approach itself is based on the critical region concept discussed in Chapter 5. To explain it, we first introduce why we chose to use this approach followed by the detailed description of the algorithm. Finally we will list some extra advantages made possible by using our approach.

6.2.1 Overview

In Section 6.1.3, we have mentioned the necessity to verify system's fairness property after its implementation. The whole fairness verification process is shown in Figure 6.1. Figure 6.1 first abstracts the work we have introduced in the previous chapters, which includes requirements elicitation, Fondue analysis with its four different models, system design with its derived SSHD and the rely diagram derived from the Operation model; Figure 6.1 also shows that with SSHD and the rely diagram, we can make a checklist for the system fairness verification purpose. Different fairness policies can be injected into these selected components. If the system does not pass its runtime verification in terms of its fairness property, the developers should go through the checklist and change the system implementation accordingly.

The idea behind this fairness verification process is that our research aims to provide a supporting and reasonably accurate verification method; therefore, we choose to focus our static verification approach on not providing “pass” or “fail” outcomes; instead, our approach is set to provide the developers a checklist before they implement the system. The checklist should be system dependent. It should contain all the system components that developers should pay attention to if they want to implement the system with its particular fairness constraint(s). Therefore, even the system fails to meet its fairness constraint at runtime, then the developers only need to go through and re-specify what is on the checklist and how the system enforces fairness for the same constraint. As a consequence, the whole repairing process will be convenient and cost effective.

6.2.2 Detecting Fairness Sensitive Components

In theory, every component in the concurrent system can affect system’s fairness performance. That is because there are multiple processes or threads contending to use virtually every component within the system. Any component can be designed to favor any specific process or thread. However, it is unusual and impractical to design a system with the concern of fairness for every component within the system. Experience tells us only a limited proportion of all system components have a substantial impact on the system’s fairness performance. If we can find such components and inject application-specific fairness policy into them during the implementation, then the system can run in a way closer to its user’s requirements. Even later on the fairness requirements on the system may be changed; we still can easily adjust the system by injecting a different fairness policy into those discovered system components. For the present purposes, we call these kinds of components *fairness sensitive components*. This section describes how to find such fairness sensitive components within the system design. This section first describes a small example that explains the common features of the fairness sensitive components; it is followed by the definition of two types of such components and the ways to find them.

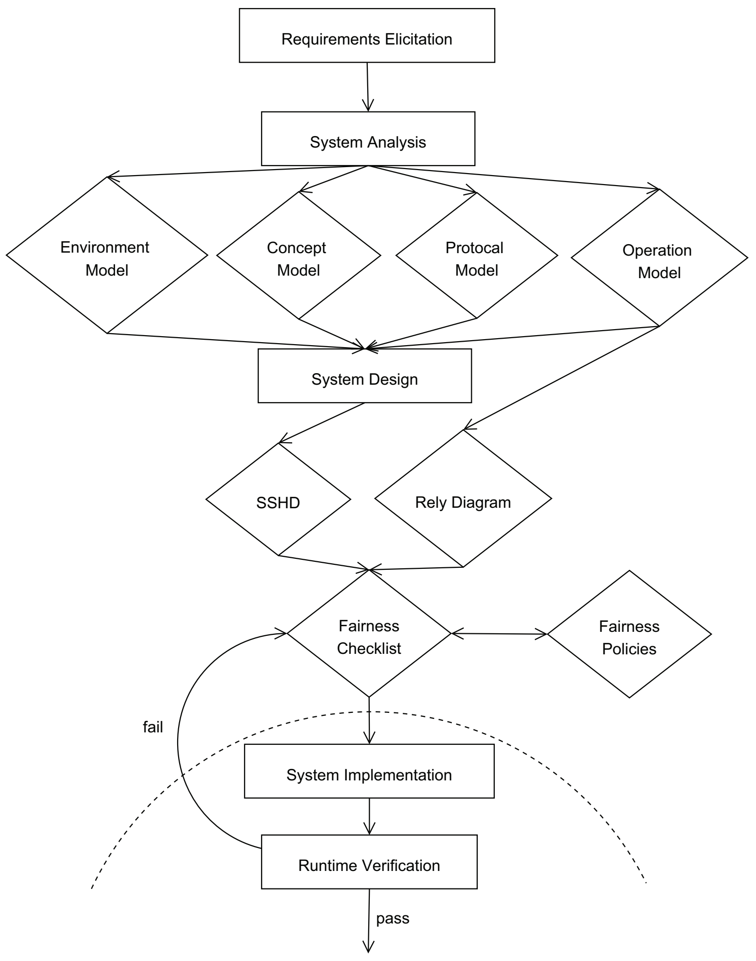


Figure 6.1: *System Fairness Verification Process*

A small example

In order to extract the system's fairness sensitive components, we need to know their common traits. Let's consider the following example first.

Two philosophers are sitting together with three chopsticks available. These three chopsticks are laid down as shown in Figure 6.2. In order to eat, each philosopher should pick up the two chopsticks laid on both his left hand side and right hand side. Assuming the fairness constraint for this example restricts philosophers to eat in an alternative order; now the question is: which chopstick is the one that can cause execution to be "unfair"? Obvi-

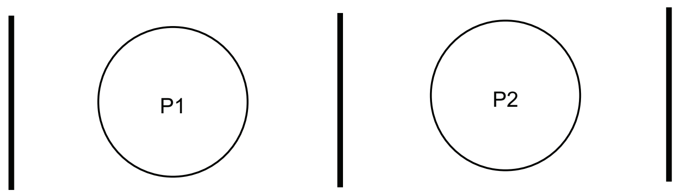


Figure 6.2: *Two Philosophers with Three Chopsticks*

ously, the answer for this question is the chopstick laid between two philosophers. This is because only the middle chopstick is contended by both philosophers. If the acquisition of the middle chopstick is out of alternative order between these two philosophers, then the fairness constraint imposed in this example is violated.

Although the example shown in Figure 6.2 is very simple, it reveals one important property of the fairness sensitive component. That is, a fairness sensitive component is a system component contended or shared by multiple processes or threads, which are running the operations that do not require exactly the same resources.

Types of shared objects

Different shared objects may affect the system's fairness property differently. In order to distinguish such difference, we divide each operation's shared objects into two groups:

Holding group: objects in the holding group are locked by the thread for the period longer than necessary. In sequence diagrams, even if such objects are no longer needed by the operation, the lock on it will not be released until the control flow passes its covering critical region.

On-call group: objects in the on-call group are locked by the thread for the period exactly as long as it is needed. When the related work is done with that object, even if the overall operation is not complete at that moment, it still releases the lock on the on-call object. In sequence diagrams, the on-call objects are not covered by the critical region.

Both groups of shared objects can act as fairness sensitive components. However, objects in the holding group possess extra values for fairness property if the underlying operation has timeliness constraints on it, or is time critical. To understand this statement better, let us consider the following example. Peter may want to use the computer to sell his stocks online within one minute or he may lose his current good selling price; his daughter, Rachael, is using the same computer to do an online interview and cannot be interrupted. In this example, selling stocks within one minute is a time constraint request from Peter's side, but the shared object - the computer is held by Rachael until her interview is over. If Peter and Rachael can foresee this situation and therefore let Peter use computer first before Rachael's interview, then both will be satisfied at the end. From the example, we find that in order to satisfy a time critical operation such as selling stocks at real time, finding its shared objects is important since these shared objects could be one of the holding group objects of another concurrently running operation.

Finding shared objects

To find shared objects is trivial. We can use the concurrency table such as Table 4.3(Page 51) to find out all possible concurrent operations. Then we can use the concurrent operation schemas to find the required objects for each of the found operations. By crosschecking each concurrent operation's required objects, we can filter out all of the shared objects and their corresponding sharing operations. In order to find the shared objects in the holding group for each operation, we can use the critical region. That is, if the operation has a critical region within its sequence diagram, then all shared objects within the same critical region are the operation's holding group shared objects; the rest of the shared objects are in the operation's on-call group. The characteristic of the critical region tells us the shared objects in the operation's holding group will not be released by the operation until all executions covered by the critical region are over.

Find holding group shared objects using SSHD

The algorithm just described to find holding group shared objects can be visualized. Given an SSHD, each system synchronization object is mapped as a vertex in the diagram. As we mentioned in Chapter 4 if we consider all possible control flows and map all potential synchronization objects into one SSHD the resultant diagram will be a static graph. That is to say, since all possible synchronization objects are considered and displayed within the same SSHD, then even if there is non-determinism existing for the runtime behaviors, it will not change the structure of the diagram. If we can represent the critical region concept in SSHD, then any shared objects within the operation's critical region in SSHD are the operation's holding group shared objects. On the other hand, for each specific operation's synchronization object, if it is not covered by any critical region belonging to this operation in the SSHD, then it is the operation's on-call group shared object.

Figure 6.3 shows one SSHD example for Peter and Rachael's example as explained previously. We assume in order to sell his stocks, Peter needs the computer and the calculator,

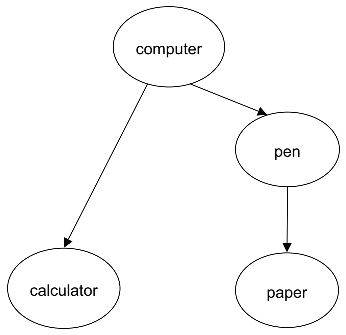


Figure 6.3: *Peter's Stock Selling and Rachael's Online Interview SSHD*

and in order to do her interview Rachael needs the computer, pen and the paper. Since the computer is required for the entire stock selling and online interview, the computer is a holding group object for both the stock selling and interview processes. For simplicity, we also assume the calculator is held by Peter until he sells the stocks; pen and paper are held by Rachael until she is done with her interview. So as seen in the sequence diagram shown in Figure 6.4, Peter's stock selling operation will be represented as a critical region that covers both computer and calculator; and Rachael's interview operation will be represented as a critical region that covers computer, pen and paper.

Please note we use different colors (or grey-scales) to represent different critical regions in Figure 6.4. This color setting also makes it easy for us to recognize the different critical

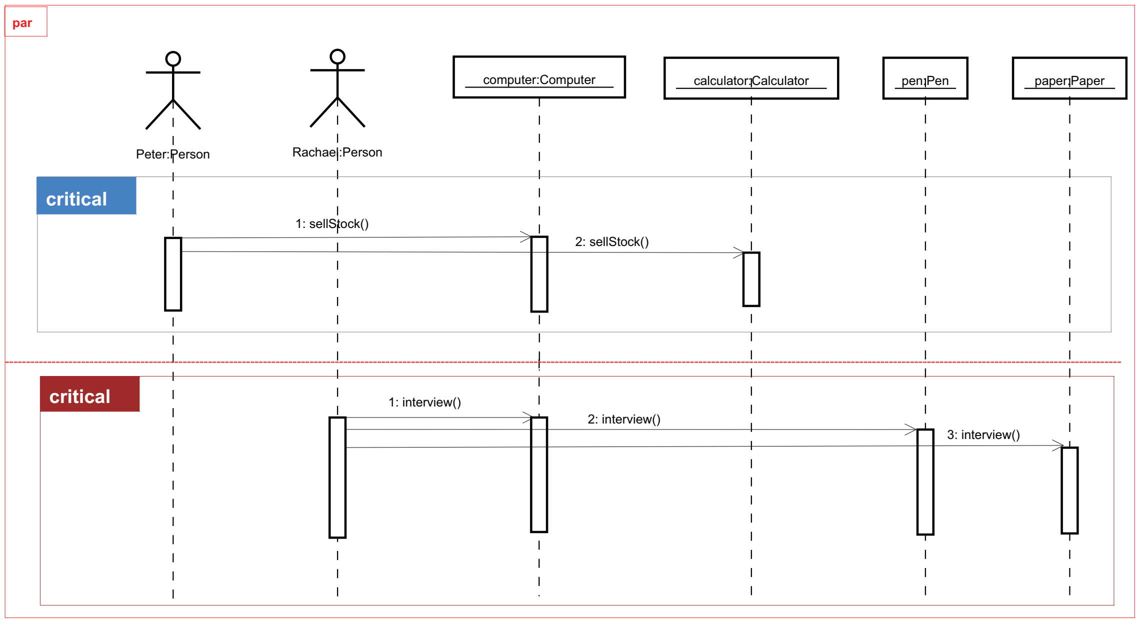


Figure 6.4: *Peter's Stock Selling and Rachael's Online Interview Sequence Diagram*

regions in SSHD after it is generated from the sequence diagram; the corresponding SSHD for this example is displayed in Figure 6.5. Here, each object is assigned the same color

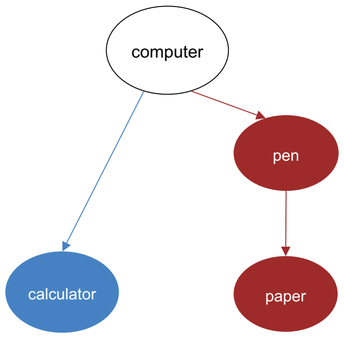


Figure 6.5: *Peter's Stock Selling and Rachael's Online Interview Colorful SSHD*

as its covering critical region, except the `computer` object. We cannot determine which color should be assigned to the `computer` object, since it belongs to two critical regions in the original sequence diagram. In reality, since it is impossible for the `computer` object to be used by more than one operation at the same time, the `computer` can only have one color assigned to it at anytime. Consequently, Figure 6.5 must exist in one of the following two forms (Figure 6.6):

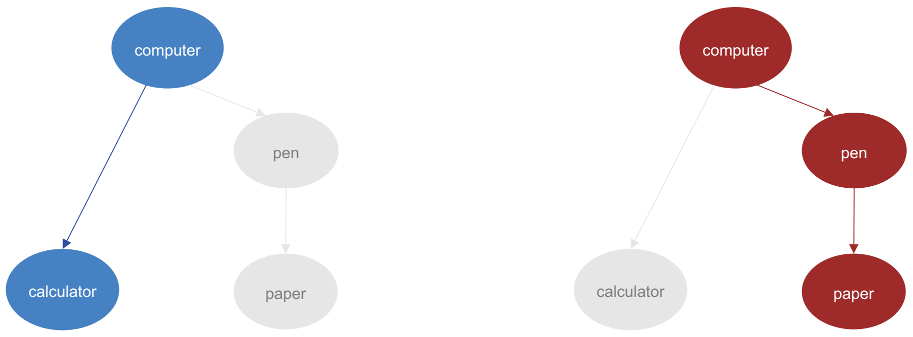


Figure 6.6: *Peter's Stock Selling and Rachael's Online Interview Colorful SSHD Transformation Forms*

In Figure 6.6, we greyed out the operation that cannot coexist with the other currently running operation. That is, if Peter is selling stocks then Rachael cannot be having the interview; if Rachael is having the interview then Peter cannot be selling his stocks. Therefore, the color of the `computer` object can be uniquely determined. We call this kind of diagram a *transformation form* of the SSHD. Since selling stock is time critical, it is necessary to find out which objects can hinder its processing. From Figure 6.5, we found the `computer` object is the only object shared by both the stock selling operation and the interview operation. Moreover, from Figure 6.6, we found the `computer` object belongs to the holding groups of both operations. So, in order to resolve conflicts between Peter's and Rachael's requests on the `computer` object, they should negotiate beforehand to decide who uses the `computer` first.

When translating the original SSHD to its transformation forms, we should only consider the activities that can run concurrently with the operation under study. Therefore, with the different operation under study, the same SSHD can result in different transformation forms.

6.2.3 Additional Benefits

By sorting out all of the fairness sensitive components, we can make the verification and implementation of the system design a much easier job for the developers. That is the primary goal we can achieve. Nevertheless, there are also several side benefits we can gain from detecting such fairness sensitive components at system design phase. These include the effects to refine deadlock detection results and some other aspects of fairness.

Refining deadlock detection

First, the same color setting of vertices within the same critical region in SSHD helps to refine deadlock detection results. SSHD represents the concurrent needs of all operations together. The transformed SSHD, however, has only a subset of vertices colored; this subset may or may not include a cycle otherwise in the old SSHD.

For example, assume there are two threads. One thread executes an operation that requires the sequential locks on objects A, B and C, while the other thread executes a concurrent operation that requires the sequential locks on objects C and A. We assume each operation is covered by its own critical region, and the critical region covers all of the objects required by the operation. Based on these assumptions we can establish the corresponding SSHD as shown in Figure 6.7. Please note that the two critical regions in the figure are represented by two different colors. Since there is a cycle: A→B→C→A detected in Figure 6.7, then there is no guarantee of deadlock-free property. But since objects A and C belong to the holding groups of both operations, Figure 6.7 can be transformed to the two forms shown in Figure 6.8.

Each form displays one of two possible synchronization sequences. Since at any point of time, only one of the two forms can exist and neither has a cycle within, then we can use this refined result to guarantee that the original system design is deadlock-free.

Liveness

Our second side benefit is that the discovery of shared objects, especially the ones in the operation's holding group helps to solve liveness issues. As we mentioned earlier in this

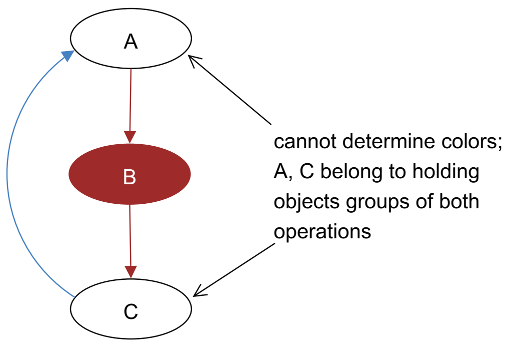


Figure 6.7: *Colorful SSHD with Deadlock Detected*

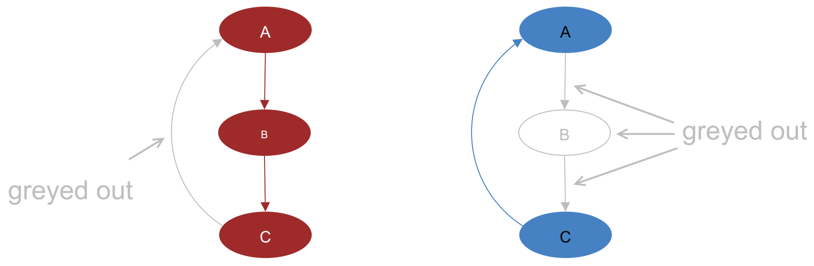


Figure 6.8: *Colorful SSHD without Deadlock Detected*

chapter, liveness properties are usually tested as an approximation of the properties we really care about; similarly, we can use the same approach to improve understandings of liveness issues by providing a checklist.

For example, if you find there is an operation that has been delayed infinitely long (e.g., longer than the period that the user can tolerate), then the starvation issue occurs. In order to solve this issue, you should check the operation's shared objects first. Shared objects, especially the ones in the holding groups of other concurrently running operations, might delay the operation infinitely long. So for each of the shared objects, we have to consider carefully the operations that might use it concurrently with the operation under study. After the analysis, we can make a list of possible concurrently running operations. With this list, it is much easier for the developers to pinpoint what might cause the violation of the liveness property.

All in all, identifying fairness sensitive components definitely can be the first step to assist solving liveness issues.

Flexibility

Our discussion implies considerable flexibility in implementations, and different choices can achieve the same or different final fairness outcomes.

For example, if the blue (light grey-scale) critical region is chosen in Figure 6.8, then the possibility for object B to be locked later is zero. Similarly, during the execution of the process/thread, every choice of the operation on the fairness sensitive component changes the odds for the subsequent operations being executed. This interesting discovery inspired us to inject the fairness policy to the fairness sensitive component given a specific user requirement.

The information from the concurrent operation schemas and SSHD is not likely to change due to the expense, but the user's fairness requirement for the system may change from time to time. In order to cope with user's changing requirements, we need a way to inject different fairness constraints to the same system without changing its structure and operation schema designs. To understand this, let us consider the following example first.

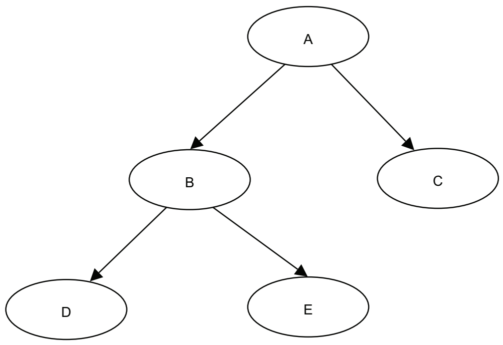


Figure 6.9: *Choice and Control Flow Example*

Assume there are four operations; each requires its specific objects to execute. The detailed concurrency information is shown in Table 6.1; and the corresponding SSHD is shown in Figure 6.9. From Table 6.1, we know that object A is a shared object con-

`Operation Name`	`Synchronized Object(s)`	`Concurrent Operation(s)`
`operation1`	A, B	`operation2`
`operation2`	A, C	`operation1`
`operation3`	B, D	`operation4`
`operation4`	B, E	`operation3`

Table 6.1: *Operation Table*

tended by `operation1` and `operation2`; object B is a shared object contended by `operation3` and `operation4`. Assume uniform choices are employed and either `operation3` or `operation4` comes sequentially after `operation1` depending on which operation acquires the lock on object B first; and the initial odds to acquire the shared objects by concurrently running operations are evenly distributed. The odds of executing `operation1` are thus 50%; and consequently the odds to execute `operation3` are 25%. Now if the user specifies in his system requirements that he wants the overall odds to execute `operation3` to be 10% and he does not care what are the chances for other operations to execute, we have at least two choices for the developers to implement the system in order to satisfy the user's requirement. First, the developers can inject a specific scheduler to set the chance for `operation3` to acquire object B to be 20% and `operation4` to acquire object B to be 80%. Second, the developers can choose to leave the odds to acquire object B unchanged (50%). Adjusting the chance for `operation1` to acquire object A to be 20%, then results in the chance for `operation3` to be executed 10% of the time. Of course, actual calculations to force or determine scheduling behaviors are complex, and beyond this study.

The example presents the influence from the fairness sensitive components to the chances of running operations. By advising the developers with a good checklist of fairness sensitive components, we can help them to make implementation decision to fulfill the user's specific requirements, at least as much as the actual scheduling system allows.

6.3 Online Auction System Case Study

In previous chapters, we have verified the online auction system's deadlock-free and application-level consistency properties. In this section, we are going to use our proposed method to verify the system's fairness property.

Since our approach aims to provide the developers a list of shared objects that can affect system's performance in terms of its fairness property, we first need to find the auction system's fairness sensitive components. Once the fairness sensitive components are found, we can modify our old system design in order to let it fit the user's requirements better.

By reviewing our online auction system sequence diagram design, the only fairness sensitive component is *auctionState* object since it belongs to the holding groups of both bidder's `placeBid` operation and the auction system's `closeAuction` operation. Additionally, since the auction system should close auction immediately when the auction's expiration time is reached, then `closeAuction()` operation is a time critical operation.

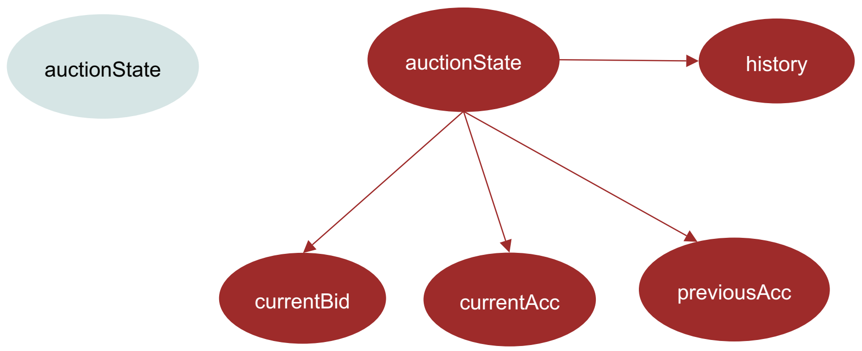


Figure 6.10: *Online Auction System Colorful SSHD Transformed Form*

Figure 6.10 is a transformed form of original auction system's SSHD. The two forms in Figure 6.10 display that the operations `placeBid` and `closeAuction` cannot execute simultaneously since they both have *auctionState* in their holding groups. As a time critical operation, `closeAuction()` should take place as soon as auction's expiration time is reached. That is, the auction system should close the auction even if there are still active `placeBid()` operations. But considering the fairness requests from the client's side, we should always allow the pending `placeBid()` operations initiated before the auction's expiration to accomplish eventually. However, there is one dilemma doing so. Since `closeAuction()` requires a write lock on the auction state, then as long as new `placeBid()` operations keep coming to the system, the auction system cannot close the auction since all `placeBid()` operations share and hold read locks on the *auctionState* object. A potential consequence of this dilemma is that even though the time has passed for the auction's expiration, clients still can place bids on the auction since the auction system cannot close it. To solve this problem, we let the system check the time stamp before each bid is placed into auction's *history*. By checking the time stamp, every bid initiated after auction's expiration will be rejected by the system. As a result, even if `closeAuction()` does not execute precisely on time, clients still cannot make new bids after auction's expiration time.

The design decision to use a time stamp check is an arbitrary choice, and there are many other ways to solve the same problem. The key point here, however, is that by identifying fairness sensitive components within the system design, we can help the developers to implement the system that satisfies its users' specific fairness requirements.

Chapter 7
Conclusions

Concurrency is inherent in the domain of many problems that are addressed by software solutions. It is important to have some general approaches to treat the many issues that arise in such environments. This thesis illustrated a number of systematic approaches to verify the design of a concurrent system.

In the first step we developed a technique that can verify if a design is deadlock-free by mapping synchronization objects from design sequence diagrams to a specialized Hasse diagram that we called the *System Synchronization Hasse Diagrams (SSHD)*. The SSHD approach is static, and hence avoids having to monitor complex resource sharing at run-time to cover all possible execution paths; it detects the potential deadlocks (that can occur when the *Coffman conditions* are satisfied), regardless of the resource request model. The described approach, however, is overly pessimistic due to its static nature. It is nevertheless viable, particularly when the system under study has to guarantee absence of deadlock.

The second approach described in the thesis shows how to use the concurrency-related information extracted from a concurrency-aware system analysis and specification to derive a system design that ensures that the system state is always application-level consistent, i.e. fulfills all the validity constraints of the application according to the application specification. At the specification level, a system operation schema defines pre-conditions, post-conditions and the rely-conditions that specify properties that have to be valid before, after and *during* the operation execution. Violation of any of these three conditions can lead the system into an application-level inconsistent state. Verification of pre- and post-condition is trivial. We propose to generate a rely diagram, which is generated from the concurrent operation schema, to help ensure the rely-condition. We list the dependent objects that are specified in the rely-expressions as the nodes in the rely-diagram. In order to come up with a correct concurrent system design that implements the required state changes in a correct way, we use *critical regions* to group all the dependent objects of the operation in the sequence diagram. With further analysis, we refined our general rely diagram by tagging the dependent objects whose states might change during the execution of the operation by some other concurrently executing operations in a way that would break the rely-condition. Only the tagged objects have to be grouped within the critical region to ensure application-level consistency; grouping other dependent objects into the critical region is unnecessary, and could slow down the system's performance.

Once deadlocks are eliminated from a concurrent system, fairness issues are the next big concern. It is non-trivial to analyze fairness at the design phase. We show how our approach can facilitate reasoning about fairness by detecting a set of shared resources, whose access-policies can have important impacts on the system's fairness property. Shared resources are identified using concurrency tables and the concurrent operation schemas. In order to distinguish the different influences of the resources on the overall fairness, we divide the shared resources into on-call group and holding group. Theoretically, every shared resource can affect the system's fairness; further analysis reveals that shared resources within the holding group can cause larger impacts, especially if the operation under study is time critical.

We have used the online auction system as the main case study throughout the thesis to illustrate our approaches. At very beginning, the original online auction system design is shown; based on the original design, we used our approaches to verify and modify the system design in order to ensure deadlock-freedom and system application-level consistency; the *auctionState* object was identified to have a significant effect on the auction system's fairness performance. Besides this main case study, we have also illustrated the usage of our approaches on the classic dining philosopher problem and other simple small examples. All these case studies have demonstrated that our approaches can produce good quality verification results during the design phase. Meanwhile, our approaches are intuitive and algorithmically simple, compared to other methods that were designed to address similar issues.

7.1 Future Work

First of all, it would be interesting to see how our approaches perform when applied to a real-world concurrent system development project by comparing the performance of a development team using our approach with a development team using standard development approaches. It would be interesting to determine how many concurrency-related problems can be detected early on during design, and therefore how much development costs can be saved overall by using our approaches. This would be of course, a difficult and expensive test to implement.

Considering the non-determinism of the concurrent system's runtime behavior, the results coming from our conservative static verification method might be prohibitively strong. SSHDs can be used effectively to prove deadlock-freedom, but not to show the existence of a deadlock. In case of the occurrence of a cycle in a SSHD, which indicates a potential deadlock, we could find ways to extend our approach to get a more fine-grained analysis. This could be done, for instance, by coloring critical regions in the SSHD that correspond to different concurrently running operations. Our initial design is straightforward, but perhaps could be refined through further investigations.

A finer granularity analysis of SSHD might also be feasible. One of the promising directions is to take different runtime "phases" into consideration. A sequence diagram, for example, can be divided into several time phases, with each one having a partial system's call graph. If there is a way to find such a partial system call graph, then we can split the original SSHD into several diagrams, one for each different time phase. As a consequence, the deadlock detection results are a lot more precise. The viability of this work is unknown due to the extreme complexity of the concurrent invocations of the different operations; it is unknown to us if the concurrent system call graph can be split based on the different time phases. However, this topic is definitely worth investigating.

The quality of the list of system fairness/liveness sensitive components can be improved. The technique described here is the basis for producing a list of potential system components that the developers should pay attention to in order to fulfill fairness requirements. However, we feel there are still differences in the degree that each individual shared resource can affect the system's overall fairness property. We have tried to use on-call group and holding group to distinguish such difference; our SSHD could be extended to add some degrees to the edges, in a way to help distinguish the different levels of influences from different shared resources.

Rely diagrams provide a conservatively correct solution; however, for performance reason and especially to improve concurrency, more optimistically speculative decision may still be adopted. In addition to the verification usage, rely diagrams can also be used for system design construction purpose. Investigating how to derive a systematic optimization approach that takes both the rely diagram and user requirements into consideration, and how to construct a correct application-level consistent system design based on the rely diagram generated from the analysis are our future work.

Appendix A
Operation Schemas and Pseudo Codes for `placeBid`

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;

Message: User::InvalidBid_e;

New: newBid: Bid

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

```
Post: if bidAmount ≥ a.highBid.amount + a.minimumIncreasement then
           ifc.account.guaranteedBalance ≥ bidAmount then
             newBid.oclIsNew(bidAmount) &
             a.bid → includes(newBid) & //update auction history
             c.myBids → includes(newBid)
           else
             sender^invalidBid_e(Reason::insufficientFunds)
           endif
           else
             sender^invalidBid_e(Reason::bidTooLow)
             endif
```

Figure A.1: *Sequential Operation Schema for* `placeBid`

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;

Message: User::InvalidBid_e;

New: newBid: Bid

Pre: a.currentMbrs$\rightarrow$ **includes**(c) & a.started;

```
Post: rely not a.closed then
          rely bidAmount $\geq$ a.highBid.amount + a.minimumIncreasement then (2$*$)
              rely c.account.guaranteedBalance $\geq$ bidAmount then          (3$*$)
                  newBid.oclIsNew(bidAmount) &
                  a.bid $\rightarrow$ includes(newBid) & //update auction history
                  c.myBids $\rightarrow$ includes(newBid)
              fail
                  sender^invalidBid_e(Reason::insufficientFunds)
              endre
          fail
              sender^invalidBid_e(Reason::bidTooLow)
          endre
      fail
          sender^invalidBid_e(Reason::auctionClosed)
      endre
```

Figure A.2: *Concurrent Operation Schema for* `placeBid`

```
**Operation** Auction :: placeBid( currentCus: Customer, bidAmount : integer )
    currentAcc := currentCus.getAccount();
begin
    if currentAcc.isGuaranteed(bidAmount) then
        if checkAndUpdate(bidAmount) then
            if  getTimeAndDate() $\leq$deadline
                theHistory. insertBid(bidAmount);
                if  lastBidAmount $>$ 0 then //if not first bid
                    previousAcc.releaseBid(lastBidAmount);
                endif
                previousAcc := currentAcc;
                lastBidAmount := bidAmount;
            else
                currentAcc.releaseBid(bidAmount);
                Exception("auctionClosed");
            endif
        else
            currentAcc.releaseBid(bidAmount);
            Exception("invalidBid");
        endif
    else
        Exception("invalidBid: insufficientFunds");
    endif
end placeBid
```

Figure A.3: *Pseudo Code for* `placeBid`

```
Operation Account::isGuaranteed( bidAmount : Integer)
        OK : boolean;
begin
        if currentAcc.actualBalance - bidAmount $\geq$ 0 then
              currentAcc.actualBalance = currentAcc.actualBalance - bidAmount;
              OK = true;
        else
              OK = false;
        endif
return OK;
end isGuaranteed
```

Figure A.4: *Pseudo Code for* `isGuaranteed`

```
Operation BidHistory::insertBid(bidAmount : Integer, time : Time, date : Date)
        bidList : Stack; //the Stack is like normal stack data structure
begin
        newbid = new Bid(time, date, bidAmount); //create new bid object
        bidList.add(newbid); //insert
end insertBid
```

Figure A.5: *Pseudo Code for* `insertBid`

Bibliography

[AH98] Rajeev Alur and Thomas A. Henzinger. Finitary fairness. *ACM Trans. Program. Lang. Syst.*, 20(6):1171–1194, 1998.

[AS84] Bowen Alpern and Fred B. Schneider. Defining liveness. Technical report, Ithaca, NY, USA, 1984.

[Bel87] Ferenc Belik. Deadlock avoidance with a modified banker's algorithm. *BIT Numerical Mathematics*, 27:290–305, 1987.

[CC01] Alessandra Cavarra Charles Crichton, Jim Davies. A pattern for concurrency in uml. *Programming Research Group Research Report*, 2001.

[CES71] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971.

[CGK97] Shing Chi Cheung, Dimitra Giannakopoulou, and Jeff Kramer. Verification of liveness properties using compositional reachability analysis. *SIGSOFT Softw. Eng. Notes*, 22(6):227–243, 1997.

[CJ86] I Cidon and J M Jaffe. Local distributed deadlock detection by knot detection. In *SIGCOMM '86: Proceedings of the ACM SIGCOMM conference on Communications architectures & protocols*, Stowe, Vermont, United States, 1986, pages 377–384. ACM Press, New York, NY, USA.

[Col94] Derek Coleman. *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1994.

[Dav93] Fred D. Davis. User acceptance of information technology: System characteristics, user perceptions and behavioral impacts. *International Journal of Man-Machine Studies*, 38(3):475–487, 1993.

[DFD98] Alan Cameron Wills Desmond Francis D'Souza. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley Professional, 1998.

[E.D76] E.Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[ELL01] Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Directed explicit model checking with HSF–SPIN. *Lecture Notes in Computer Science*, 2057, 2001.

[Gom00] Hassan Gomaa. *Designing Concurrent, Distributed, and Real-Time Applications with UML*. Addison-Wesley Professional, 2000.

[GPSS80] Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Las Vegas, Nevada, 1980, pages 163–173. ACM Press, New York, NY, USA.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, California, 1993.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[Hol72] Richard C. Holt. Some deadlock properties of computer systems. *ACM Comput. Surv.*, 4(3):179–196, 1972.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[JBW03] Anneke G. Kleppe Jos B. Warmer. *The Object Constraint Language*. Addison-Wesley Professional, 2003.

[JM06] Jeff Kramer Jeff Magee. *Concurrency: State Models and Java Programs*. Wiley; 2Rev Ed edition, 2006.

[Jon83] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, 1983.

[KE01] Nima Kaveh and Wolfgang Emmerich. Deadlock detection in distribution object systems. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, Vienna, Austria, 2001, pages 44–51. ACM Press, New York, NY, USA.

[KPRS06] Yonit Kesten, Amir Pnueli, Li-On Raviv, and Elad Shahar. Model checking with strong fairness. *Form. Methods Syst. Des.*, 28(1):57–84, 2006.

[KS06] Jörg Kienzle and Shane Sendall. Addressing concurrency in object-oriented software development. In *CASCON '06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, Toronto, Ontario, Canada, 2006, page 15. ACM Press, New York, NY, USA.

[KSR02] J. Kienzle, A. Strohmeier, and A. Romanovsky. Auction system design using open multithreaded transactions. *Proceedings of the 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'02), San Diego, CA, USA*, pages 95–104, 2002.

[KZ05] A. Koprowski and H. Zantema. Proving liveness with fairness using rewriting, 2005.

[Lam79] Leslie Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Trans. Program. Lang. Syst.*, 1(1):84–97, 1979.

[Lam89] Leslie Lamport. A simple approach to specifying concurrent systems. *Commun. ACM*, 32(1):32–45, 1989.

[Lam00] Leslie Lamport. Fairness and hyperfairness. *Distrib. Comput.*, 13(4):239–245, 2000.

[Lar02] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd edition*, 2002.

[Lev03] Gertrude Neuman Levine. Defining deadlock. *SIGOPS Oper. Syst. Rev.*, 37(1):54–64, 2003.

[LP99] Johan Lilius and Ivan Porres Paltor. vUML: a tool for verifying UML models. Technical Report TUCS-TR-272, 18, 1999.

[MC82] Jayadev Misra and K. M. Chandy. A distributed graph algorithm: Knot detection. *ACM Trans. Program. Lang. Syst.*, 4(4):678–686, 1982.

[Mil89] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[Mil99] R. Milner. *Communicating and Mobile Systems: the* Π*-Calculus*. Cambridge University Press; 1st edition, 1999.

[PdBZ95] Alain Pirotte, Thierry Van den Berghe, and Esteban Zimányi. The fusion object-oriented method: an evaluation. In *SOFSEM '95: Proceedings of the 22nd Seminar on Current Trends in Theory and Practice of Informatics*, 1995, pages 437–442. Springer-Verlag, London, UK.

[PY] Taesoon Park and Heon Y. Yeom. A distributed group commit protocol for distributed database systems.

[R.F67] R.Floyd. *Assigning meaning to programs*, volume 19. American Mathematical Society, 1967.

[RSI78] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis II. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems (TODS) archive*, 3(2):178–198, 1978.

[SGG04] Avi Silberschatz, Peter Baer Galvin, and Greg Gagne. Operating system concepts, seventh edition. 2004.

[Sis94] A. Prasad Sistla. Safety, liveness and fairness in temporal logic. *Formal Asp. Comput.*, 6(5):495–512, 1994.

[SS99] Alfred Strohmeier Shane Sendall. *UML-Based Fusion Analysis*, volume 1723. Springer Berlin / Heidelberg, Fort Collins, CO, USA, 1999.

[TC96] Stavros Tripakis and Costas Courcoubetis. Extending promela and spin for real time. In *Tools and Algorithms for Construction and Analysis of Systems*, 1996, pages 329–348.

[UML04] *UML 2.0 Superstructure Specification*. Object Management Group, Framingham, Massachusetts, October 2004.

[WCJ02] Hui Wu, Wei-Ngan Chin, and Joxan Jaffar. An efficient distributed deadlock avoidance algorithm for the and model. *IEEE Trans. Softw. Eng.*, 28(1):18–29, 2002.

[Xio04] Jie Xiong. Addressing concurrency using uml-based software development. *Master Thesis*, 2004.