View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Acronym

Integrated Data, Message, and Process Recovery for Failure Masking in Web Services

Dissertation

zur
Erlangung des Grades des Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

vorgelegt von

Diplom-Informatiker
GERMAN SHEGALOV

Saarbrücken, im Juli 2005

Dekan der Naturwissenschaftlich-Technischen Fakultät I:	Prof. Dr. Jörg Eschmeier
Vorsitzender der Prüfungskommission:	Prof. Dr. Joachim Weickert
Erstgutachter:	Prof. Dr. Gerhard Weikum
Zweitgutachter:	Dr. David Lomet
Akademischer Beisitzer	Dr. Ralf Schenkel
Tag des Promotionskolloquiums	26. August 2005

In memory of my father Isaac

To my mother Betti

Abstract

Modern Web Services applications encompass multiple distributed interacting components, possibly including millions of lines of code written in different programming languages. With this complexity, some bugs often remain undetected despite extensive testing procedures, and occasionally cause transient system failures. Incorrect failure handling in applications often leads to incomplete or to unintentional request executions. A family of recovery protocols called interaction contracts provides a generic solution to this problem by means of system-integrated data, process, and message recovery for multi-tier applications. It is able to mask failures, and allows programmers to concentrate on the application logic, thus speeding up the development process.

This thesis consists of two major parts. The first part formally specifies the interaction contracts using the state-and-activity chart language. Moreover, it presents a formal specification of a concrete Web Service that makes use of interaction contracts, and contains no other error-handling actions. The formal specifications undergo verification where crucial safety and liveness properties expressed in temporal logics are mathematically proved by means of model checking. In particular, it is shown that each end-user request is executed exactly once. The second part of the thesis demonstrates the viability of the interaction framework in a real world system. More specifically, a cascadable Web Service platform, EOS, is built based on widely used components, Microsoft Internet Explorer and PHP application server, with interaction contracts integrated into them.

Kurzfassung

Heutige Web-Service-Anwendungen setzen sich aus mehreren verteilten interagierenden Komponenten zusammen. Dabei werden oft mehrere Programmiersprachen eingesetzt, und der Quellcode einer Komponente kann mehrere Millionen Programmzeilen umfassen. In Anbetracht dieser Komplexität bleiben typischerweise einige Programmierfehler trotz intensiver Qualitätssicherung unentdeckt und verursachen vorübergehende Systemsausfälle zur Laufzeit. Eine ungenügende Fehlerbehandlung in Anwendungen führt oft zur unvollständigen oder unbeabsichtigt wiederholten Ausführung einer Operation. Eine Familie von Recovery-Protokollen, die so genannten „Interaction Contracts“, bietet eine generische Lösung dieses Problems. Diese Recovery-Protokolle sorgen für die Fehlermaskierung und ermöglichen somit, dass Entwickler ihre ganze Konzentration der Anwendungslogik widmen können. Dies trägt zu einer erheblichen Beschleunigung des Entwicklungsprozesses bei.

Diese Dissertation besteht aus zwei wesentlichen Teilen. Der erste Teil widmet sich der formalen Spezifikation der Recovery-Protokolle unter Verwendung des Formalismus der State-and-Activity-Charts. Darüber hinaus entwickeln wir die formale Spezifikation einer Web-Service-Anwendung, die außer den Recovery-Protokollen keine weitere Fehlerbehandlung beinhaltet. Die formalen Spezifikationen werden in Bezug auf kritische Sicherheits- und Lebendigkeitseigenschaften, die als temporallogische Formeln angegeben sind, mittels „Model Checking“ verifiziert. Unter anderem wird somit mathematisch bewiesen, dass jede Operation eines Endbenutzers genau einmal ausgeführt wird. Der zweite Teil der Dissertation beschreibt die Implementierung der Recovery-Protokolle im Rahmen einer beliebig verteilbaren Web-Service-Plattform EOS, die auf weit verbreiteten Web-Produkten aufbaut: dem Browser „Microsoft Internet Explorer“ und dem PHP-Anwendungsserver.

Summary

Recovery is the last resort for preserving data and system state consistency in a failure-prone environment. Critical applications use transactional database servers whose data recovery mechanisms establish atomic updates and durability of data in the presence of transient system failures. Unfortunately, data recovery on database servers does not enforce an appropriate exception handling in the other application components. It is the responsibility of every single component in the system to handle all system failures such as message losses, timeouts, and crashes in a correct manner. In a distributed application with a rich state some component interdependences are often overlooked, which leads to incorrect application behavior in that some requests may unintentionally be repeated whereas others may not be executed at all due to message losses.

This has motivated several recovery protocols aiming at masking system failures, and so relieving developers from dealing with them. The *queued transactions* approach has been the most successful industrial solution thus far. It requires that components store their state in transactional input and output message queues mostly residing on a database server, or in a database. In a multi-tier system, a single end-user request incurs a number of instances of the Two-Phase-Commit protocol incurring high logging overhead. Due to an inconvenient programming model and for insufficient scalability in the context of multi-tier applications, queued transactions have not been adopted for Web Services, although most of them are stateful by nature since they require several interactions with the user to accomplish a deal: authentication, catalog search, price negotiation or bidding, and finally committing the deal. This thesis elaborates on a recently proposed framework of interaction contracts geared towards general multi-tier applications that is more efficient than the queued transactions approach, and does not enforce any specific programming style.

This thesis provides for the first time a formal specification for each interaction contract previously only informally described in the original literature. To this end, we adopted the state-and-activity chart language as defined and implemented in the commercial tool Statemate, widely used for modeling reactive systems such as embedded devices in the automotive and airspace industries. Each individual interaction contract is defined by a generic activity that can be easily reused in every application context. We model a complex Web Service comprising several components, which pass messages to each other either in synchronous or asynchronous fashion with the generic interaction contract

activities as building blocks. Most importantly, the Web Service model does not involve any recovery actions other than those defined in the underlying interaction contract activities that are invisible at the application layer.

After completing the formal specification process, we start with verification of the interaction contracts using Statemate's integrated model checker. For this purpose, we formulate crucial safety and liveness properties as temporal logic formulae. As for safety, we show that no message is ever executed more than once. For liveness, we prove that with a finite number of failures each interaction contract eventually terminates, and the corresponding requests are executed exactly once. While the verification of the individual bilateral interaction contracts is straightforward due to their relatively small model size, additional design engineering effort is needed to keep the Web Service model verifiable. We succeed in designing equivalent or more general, verifiable models, whose safety properties carry over into the original specification of the interaction contracts.

Along with the formal specification of the interaction contract framework, in this thesis we describe a prototype Web Service platform called EOS that we built to investigate the framework's viability in a real-world setting. More specifically, we consider two popular products used in the Web Service context: Microsoft Internet Explorer as a browser (user front-end), and a script engine PHP as a Web application server which can be invoked either by a browser or by another application server. We implement the *external interaction contract* to handle interactions between an end-user and her browser. Interactions between a pair of Web application servers, and between a browser and a Web application server run under either the *committed* or the *immediately committed interaction contract*. To this end, we turned the browser and the Web application server into persistent components by equipping them with logging and recovery routines. In accordance with the framework goals, we achieved this without rewriting existing application programs such as PHP scripts and the browser by solely changing their runtime environment. The most challenging part of this work was providing the deterministic replay of the multi-threaded PHP script engine in the business-to-business context, in which the state is shared by multiple sessions and may be simultaneously accessed by several other application servers. Thus, deterministic replay requires logging of original output messages. Enhanced components exhibit acceptable overhead in comparison with the original implementation, which shows their viability in large-scale Web Services.

Zusammenfassung

Die Recovery ist das letzte Mittel, das die Inkonsistenz der Daten und des Systemzustandes in einer fehleranfälligen Ausfürungsumgebung verhindern kann. Kritische Anwendungen benutzen transaktionsfähige Datenbanksysteme, die die atomare Ausführung von mehreren Schreiboperationen und deren Dauerhaftigkeit trotz kurzzeitig auftretender Fehler gewährleisten. Die Daten-Recovery im Datenbanksystem erzwingt jedoch nicht, dass Fehler auch in den anderen Anwendungskomponenten adäquat behandelt werden. Jede Komponente ist selbst dafür verantwortlich, allen möglichen Fehlern wie Nachrichtenverlusten, Wartezeitüberschreitungen und Abstürzen richtig zu begegnen. In einer verteilten Anwendung mit einem großen Zustand werden wechselseitige Abhängigkeiten oft übersehen, was zu einem falschen Systemverhalten führt, in dem manche Operationen unbeabsichtigt mehrmals ausgeführt werden, während die Ausführung anderer Operationen wegen Kommunikationsstörungen gänzlich unterbleibt.

Diese Problematik diente als Motivation für mehrere fehlermaskierende Recovery-Protokolle, die Entwicklern die Behandlung von Fehlern abnehmen. Den bisher erfolgreichsten industriellen Ansatz stellt das Queued-Transactions-Verfahren dar. Es erfordert, dass Komponenten ihren Zustand in transaktionsfähigen, meistens von Datenbanksystemen verwalteten, Ein- und Ausgabewarteschlangen oder in einer Datenbank speichern. In einem Mehrschichtensystem zieht eine einzige Operation des Endbenutzers mehrere Instanzen des Two-Phase-Commit-Protokolls nach sich, was hohe Protokollierungskosten verursacht. Aufgrund des unbequemen Programmiermodells und der für verteilte Anwendungen ungenügenden Skalierbarkeit wurde das Queued-Transactions-Verfahren nicht in den Bereich der Web-Services übertragen, obwohl wir es dort auch mit fehleranfälligen zustandsvollen Anwendungen zu tun haben. Diese Dissertation beschäftigt sich mit dem in den letzten Jahren veröffentlichten Framework der „Interaction Contracts“, das eigens für Mehrschichtensysteme entworfen wurde. Es ist effizienter als das Queued-Transactions-Verfahren und erzwingt keinen bestimmten Programmierstil.

Diese Dissertation präsentiert erstmalig formale Spezifikationen der „Interaction Contracts“, die bis jetzt nur informal in der Literatur eingeführt wurden. Zu diesem Zweck setzen wir mit State-and-Activity-Charts einen automatentheoretischen Formalismus ein. Dieser Formalismus ist implementiert im kommerziellen Tool

Statemate, das eine breite Verwendung in der Automobilindustrie und der Luft- und Raumfahrtbranche hat. Jeder einzelne „Interaction Contract“ wird modelliert durch eine generische Aktivität, die sich leicht in unterschiedlichen Anwendungsszenarien wiederverwenden lässt. Darüber hinaus, wir modellieren einen komplexen Web-Service, der aus mehreren Komponenten besteht. Die Komponenten tauschen mehrere Nachrichten synchron und asynchron aus, jeweils unter Verwendung der generischen Aktivitäten. Besonders wichtig ist, das hierbei keine andere als die für die Anwendungsebene unsichtbaren, in den generischen Aktivitäten definierten Recovery-Aktionen zum Tragen kommen.

Die erstellten Spezifikationen der „Interaction Contracts“ werden mit Hilfe des „Statemate Model Checker“ verifiziert. Dazu formulieren wir wichtige Sicherheits- und Lebendigkeitseigenschaften als temporallogische Formeln. Als eine der Sicherheitseigenschaften beweisen wir beispielsweise, dass keine vom Endbenutzer initiierte Operation mehr als einmal ausgeführt wird. Unter der Annahme einer endlichen Anzahl von Fehlern beweisen wir, dass jeder „Interaction Contract“ terminiert (Lebendigkeit) und die betreffenden Operationen genau einmal ausgeführt werden. Während sich die Verifikation der einzelnen „Interaction Contracts“ aufgrund der verhältnismäßig geringen Modellkomplexität einfach gestalten ließ, erforderte die Verifikation der Web-Service-Anwendung zusätzlichen Aufwand, um analoge verifizierbarere Modelle zu finden, deren Sicherheitseigenschaften sich ins Ursprungsmodell übertragen lassen.

Neben der formalen Spezifikation und Verifikation der „Interaction Contracts“, beschreiben wir eine prototypische Implementierung der Web-Service-Plattform EOS, mit der die Praxistauglichkeit der „Interaction Contracts“ in einer realen Software-Anwendung untersucht wird. Wir betrachten zwei beliebte Web-technologische Produkte: Internet Explorer, den Web-Browser von Microsoft, und den Interpreter der Skriptsprache PHP, die Ausführungsumgebung für Webanwendungsserver. Ein Webanwendungsserver kann entweder von einem Browser oder von einem anderen Webanwendungsserver aufgerufen werden. Wir implementieren den „External Interaction Contract“, um die Interaktionen des Endbenutzers mit seinem Browser zu behandeln. Die Interaktionen zwischen zwei Webanwendungsservern und zwischen einem Browser und einem Webanwendungsserver werden durch den „Committed Interaction Contract“ oder den „Immediately Committed Interaction Contract“ geregelt. Hierzu statten wir den Browser

und den Webanwendungsserver jeweils mit einer Protokolldatei und Recovery-Funktionen aus. Die Änderungen betreffen nur die Ausführungsumgebungen, ohne dass die Anwendungen, d.h. die PHP-Skripte, geändert werden müssen. Die Wiederherstellung der PHP-Ausführungsumgebung im Zusammenhang mit Business-to-Business-Anwendungen stellt eine der größten Herausforderungen dar, weil wir es dort mit gemeinsam benutzten Daten zu tun haben, auf die parallel zugegriffen wird. Die korrekte Wiederherstellung erfordert die Protokollierung von Antwortnachrichten. Die verbesserten Web-Service-Komponenten haben nur geringfügig höhere Ausführungskosten im Vergleich zur ursprünglichen Software und empfehlen sich dadurch für den Einsatz in komplexen Web-Service-Anwendungen.

Acknowledgements

I would like to thank my advisor, Prof. Gerhard Weikum, for his guidance and encouragement during my research that led to this thesis. My interest in transaction processing in general and in recovery technology specifically owes to a very great extent to Gerhard's inspiring lecture on transactional information systems during my master study. Working out the sample solutions for the underlying textbook authored by Gerhard jointly with Prof. Gottfried Vossen made me feel like a real TP expert. I have spent six exciting years with Gerhard's group at the campus of Saarland University learning about the workflow and peer-to-peer technology, the theory of concurrency control and recovery, information retrieval, and so on and so forth.

I take this opportunity to thank Dr. David Lomet for agreeing to act as a second reviewer of this thesis without hesitating to commit to attending my defense in Saarbrücken. I would also like to thank Dave for the opportunity to work with him during my fall internship at Microsoft Research in Redmond in 2002. Sometimes I think I would deserve another university degree given how much I learned from Dave during our weekly meetings about how to implement the database internals right. Our email discussions of several thesis details were always insightful for me.

I would like to thank my mother, Mrs. Betti Shegalova, for all her love and encouragement all my life and especially during my university studies. I learned that the science can be that exciting only because she succeeded in persuading me to pass the entrance examination of the Lyceum of Physics and Mathematics in Saint-Petersburg. I also owe a great debt of gratitude to my father, Dr. Isaac Shegalov, for a happy childhood during the first fourteen years of my life. He died too early and I had no chance to talk to him as to a scientist, which I can only regret because not a few of his students made excellent scientific careers after talking to him.

Last but not least, I would like to thank Janna for her love and patient listening to my complaints about the troubles I was experiencing during completion of this thesis.

Contents

Figures

Tables

1. Introduction

> *"There is nothing quite so bad as bad service, unless it is a bad product too."*
> *- Anonymous*

The Web is currently the broadest available technology on the Internet. Therefore, a steadily growing number of businesses deliver mission-critical applications such as stock trading, airline ticket reservation, procurement, and business accounting systems to their end and business customers in the form of **Web Services**. These applications are often complex, comprise heterogeneous components such as application servers, workflow engines, and databases distributed over multiple layers; they pose strong requirements for service and consistent data availability from both legal and business standpoints. However, since many of those components count many millions of lines of code some bugs pass quality assurance undetected which inevitably leads to unpredictable outages of hardware and software systems at some point.

1.1 Transactional Information Systems

In the last three decades, it has become a common standard to manage the state of critical applications inside a transactional information system [Weikum and Vossen 2001]. An application can declare a sequence of requests to a transactional system as a **transaction** by including it between a "**begin transaction**" and a **"commit transaction"** requests. The transactional system behaves according a contract coined **ACID** (the abbreviation of the guarantees constituting the contract):

- **Atomicity** (**all-or-nothing**)**:** A transaction is executed either completely or not at all. Uncompleted transactions (hit by a failure prior to commit or explicitly aborted by a "**rollback transaction"** request) are undone; this is also referred to as **at-most-once execution**.
- **Consistency:** Transactions violating consistency constraints defined in the transactional system are rejected (i.e., aborted and undone).
- **Isolation:** Concurrency control of the transactional system masks intermediate effects from transactions. From the perspective of applications using the highest isolation level *serializable*, transactions are executed one at a time.
- **Durability (Persistence):** State modifications done by committed transactions survive subsequent system failures.

```
01.  CREATE PROCEDURE money_transfer
02.    @from integer,
03.    @to integer,
04.    @amount integer
05.  AS
06.  BEGIN TRANSACTION
07.    UPDATE ACCOUNTS SET balance = balance - @amount \
             WHERE id = @from
08.    UPDATE ACCOUNTS SET balance = balance + @amount \
             WHERE id = @to
09.  COMMIT TRANSACTION
```

Figure 1: Sample Money Order Transaction

Figure 1 outlines a transactional procedure for transferring funds between two bank accounts managed in an SQL database [Henderson 2000]. Assume that the database relation *accounts* contains integer attributes *id* and *balance*. The transaction (started at line 4) charges the account denoted by the integer variable *@from* and credits the account denoted by the integer variable *@to*. Transaction atomicity guarantees that the account balances are recovered to their state before the transaction begin, if a failure occurs before the transaction commit (line 7) has been executed. After the transaction commit the effect of the money transfer is definite. As you may see in the source code, the application developer is completely relieved from the responsibility of treating intermediate states.

1.2 Problem Statement

Unfortunately, transaction atomicity does not mask failures from the application, which shifts the responsibility to properly deal with them towards the application developers. Moreover, there are pathological situations in which a transactional system is not capable of faithfully reporting the transaction outcome to the application, which may make it erroneously assume the transaction abort case. A subsequent transaction restart leads to a **non-idempotent** request execution as can be demonstrated by the following real-life scenarios:

Scenario 1 (E-Commerce): One of the most prominent examples is unintentional purchase of multiple copies of the same item (e.g., a DVD) in an online store. This may happen when the user sees a browser timeout for the final “checkout” (“place order”) request caused by a short outage or overload of the network or the backend servers. Whereas the request has been successfully albeit slowly processed, the user attempts to send the checkout request once again by hitting the browser “refresh” button, unintentionally buying another copy of the same item.

Scenario 2 (Home Banking): A bank offers home banking where each user is identified by a personal identification number (PIN). The users obtain a list of unique transaction numbers (TAN's). A TAN has to be provided for each user transaction to be accepted and for security reasons each TAN can be used just once. The following problem has happened to customers. After the first attempt to place a money order, the user perceives a long delay. The user re-submits the request and the "resurrected" application responds with "A TAN was used twice. Your TAN list has been frozen. Please visit your nearest branch office to have your TAN's reactivated", which is embarrassing for a service that is referred to as *home banking*.

Scenario 3 (Intranet Application): A friend's family consisting of three persons applied for a new health insurance by sending a filled-out form via conventional mail. After the application form was computerized and reviewed by the insurance company, the friend got back a letter with the positive response. There was nothing wrong with this except for the fact that eight smart cards (insurance ids) were attached to the letter, and five of them were duplicates.

In a complex multi-tier application such as stock trading, a single request is often routed through more than ten system components hosted by different companies depending on the market model. The complexity of failure handling routines in such a system is accordingly high, and the task to cover all possible component interdependences is a real challenge. This motivates a system infrastructure that is able to mask system failures from applications by automatically taking appropriate recovery actions and providing **exactly-once execution**. Such an infrastructure would allow developers to concentrate on the application logic, thus increasing their productivity, and improve application availability, as the application would be able to resume normal operation after a system failure without manual intervention.

1.3 Contribution

This thesis elaborates on the **interaction contracts framework** introduced by Barga et al. [2002, 2004]. The framework is especially geared for Web-based middleware in that, in contrast to the previous solutions, it does not put limitations on programming style, and is much more lightweight in terms of logging costs during normal operation.

In many Web applications, components exchange request and reply messages more than once. For instance, it takes several browser requests to find desired items in an online-

store and add them to the shopping cart, to select the method of payment, and finally to provide the shipping address. A component that remembers the state of the conversation is called **stateful**, as opposed to **stateless** components whose interactions with other components are not related to each other. A client component may have a state as little as the id of the current user. Its server counterpart may maintain shopping profile of the current user as a long-term state and items in her shopping cart as a short-term state. Providing persistence for stateful components and ensuring that each state transition occurs exactly once are among the most important assets of the framework.

The contribution of this thesis consists of the following points:

- Formal specification of the individual interaction contracts with state-and-activity charts in an easy-to-compose manner for usage in a concrete application.
- Formal verification of the formal specifications at the level of the bilateral interaction contracts and their concrete application in a formal specification of a sophisticated Web Service by means of the model checking technology.
- Implementation of the interaction contract framework in the Exactly-Once Web Service (EOS) prototype. It differs from another prototype implementation coined Phoenix/App [Barga et al. 2003, 2004] in that it delivers recovery guarantees to the human end-user by considering the Web browser, an end-user front-end, as part of the framework.

1.4 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 introduces the formal methods used in this thesis: the state-and-activity-chart language for formal specification, a temporal logic CTL used for describing a dynamic computer system behavior, and model checking algorithms utilized for the verification of temporal logic propositions in a formally specified computer system. Chapter 3 puts this thesis into perspective of the state-of-the-art data and application recovery technology for monolithic and distributed applications. In Chapter 4, we provide the formal specifications charts of the interaction contracts in the form of state-and-activity charts and apply model checking to prove that it guarantees exactly-once execution. Chapter 5 deals with a prototype implementation of the IC framework for arbitrarily distributed Web Services. Chapter 6 concludes the thesis and outlines directions for future work.

2. Background on Formal Methods

"Logic! Good gracious! What rubbish! How can I tell what I think till I see what I say?"
- Edward Morgan Forster

This chapter introduces the methods used for the formalization of computational systems exploited for the verification of the interaction contract framework in this thesis. In particular, it deals with the temporal logic CTL used to characterize the behavior of a computational system, the *statechart* formalism, an automata-theoretical approach for abstract specification of a computational system, and *model checking* approach for automatic verification of CTL formulae against formal specifications.

2.1 Computation Tree Logic

For capturing properties of a system's dynamic behavior, variants of **temporal logic** are a well-established formalism [Emerson 1990]. In **linear-time temporal logics,** temporal operators describe events along a single **execution path**, also called a system run. A system reacting to external input has multiple alternative execution paths. The system satisfies a linear-time temporal logic formula if the formula holds in all system runs.

Often the user would like to describe a property that holds only in some specific runs. **Branching-time temporal logics** provide quantifiers for the paths originating from a given state. A system that reacts to external input is considered as a computation tree. The system is a model of a branching-time temporal formula if the formula holds for the corresponding computation subtree. The model checker used in this thesis verifies temporal logic formulae provided in a branching-time temporal logic called **Computation Tree Logic (CTL)**.

CTL uses propositional logic formulae over a finite set of variables as its elementary building blocks. In a given state of a computation, such a formula is evaluated to either *true* or *false* in the usual manner. In addition, CTL allows applying existential and universal quantifiers, denoted *E* and *A*, respectively, to state transition paths originating from a given state. The quantifiers are combined with so-called temporal modalities like *neXt*, *Globally*, *Until*, and *Finally*, abbreviated *X*, *G*, *U*, and *F*. The syntax of CTL is defined as follows:

1. An atomic proposition is a CTL formula
2. If p and q are CTL formulae, then so are $\neg p$, $p \wedge q$, $EX\ p$, $E\ (p\ U\ q)$ and $A(p\ U\ q)$

3. Given the basic operators above the following additional operators can be derived:
$p \vee q \equiv \neg(\neg p \wedge \neg q)$; $AX\ p \equiv \neg EX\ \neg p$; $AF\ p \equiv A\ (true\ U\ p)$;
$EF\ p \equiv E\ (true\ U\ p)$; $AG\ p \equiv \neg E\ (true\ U\ \neg p)$; $EG\ p \equiv \neg A\ (true\ U\ \neg p)$

Let P be a finite set of atomic proposition. The CTL formulae are interpreted over a Kripke structure $K = (S, R, L)$, where S is the finite set of states, $R \subseteq S \times S$ is the state transition relation with $(s, t) \in R$ if t is an immediate successor of s, and $L: S \times P \longrightarrow \{0, 1\}$ is the valuation function. Note that for a software system, the function L is interpreted as a valuation of individual bits of the program variables in the given program state. A computation tree is obtained through unwinding the graph (S, R). A path of the structure K is a potentially infinite state sequence $(s_0, \ldots, s_i, s_{i+1}, \ldots)$ with each successive pair of states $(s_i, s_{i+1)} \in R$.

Whether a current state s of the Kripke structure K fulfills the formula f denoted $K, s \models f$ is recursively defined over the formula structure:

$K, s \models p$	$\Leftrightarrow$	$L(s , p) = 1$, where p is an atomic proposition
$K, s \models \neg p$	$\Leftrightarrow$	$K, s \not\models p$
$K, s \models p \wedge q$	$\Leftrightarrow$	$K, s \models p$ and $K, s \models q$
$K, s_0 \models EX\ p$	$\Leftrightarrow$	for at least one path $(s_0, s_1, \ldots)$ holds $K, s_1 \models p$
$K, s_0 \models E(p\ U\ q)$	$\Leftrightarrow$	for at least one path $(s_0, s_1, \ldots)$ there is an i with $K, s_i \models q$ and for all $j < i$ holds $K, s_j \models p$
$K, s_0 \models A(p\ U\ q)$	$\Leftrightarrow$	for all paths $(s_0, s_1, \ldots)$ there is an i with $K, s_i \models q$ and for all $j < i$ holds $K, s_j \models p$

2.2 Explicit CTL Model Checking

The automatic recursive procedure that verifies whether $K, s \models f$ holds by using the finite state-transition graph (S, R) is called **explicit model checking** [Clarke and Schlinghoff 2001].

For a subset $P \subseteq S$ the set of predecessor states is defined as $Pred(P) := \{ s \mid (s, t) \in R \text{ and } t \in P \}$, and the set of successors is defined as $Succ(P) := \{ t \mid (s, t) \in R \text{ and } s \in P \}$

Let g be a subformula of f and $M_g \subseteq S$ such that $s \in M_g$ if $K, s \models g$. Then, one can recursively apply the following explicit model checking algorithm of Figure 2 that considers seven different cases with regard to the structure of the formula f.

```
01.  if g = p and p is an atomic proposition then
       M_g := ∅
       for all states s_i ∈ S
         if L(s_i,p) = 1 then M_g := M_g ∪ {s_i}
02.  if g = ¬p then M_g := S \ M_p
03.  if g = p ∧ q then M_g := M_p ∩ M_q
04.  if g = EX p then M_g := Pred(M_p)
05.  if g = E(p U q) then
       M_g := M_q
       repeat
         M'_g := M_g
         M_g := M_g ∪ (M_p ∩ Pred(M_g))
       until M'_g = M_g
06.  if g = A(p U q) then
       M_g := M_q
       repeat
         M'_g := M_g
         for each s ∈ M_p ∩ Pred(M_g) ∩ (S \ M_g)
           if Succ({s}) ⊆ M_g then M_g := M_g ∪ {s}
       until M'_g = M_g
07.  if g = f and s ∈ M_g then print K, s |= f
     else print K, s |≠ f
```

Figure 2. Explicit Model Checking Algorithm

This simple model checking algorithm is subject to the state-explosion phenomenon because it requires instantiating of the complete state-transition graph. The state-transition graph of a software system grows exponentially with the amount of memory used to store program variables.

2.3 Symbolic CTL Model Checking

McMillan [1993] developed a more efficient variant of model checking coined *symbolic model checking* that rather than using the state-transition graph considers the Kripke structure encoded in **Quantified Boolean Formulae (QBF)**. Given a finite set of atomic propositions $V = \{v_1, \ldots, v_n\}$, the set $QBF(V)$ of formulae is defined as:

1. The constants *true* and *false* are formulae, i.e., $\{ true, false\} \subseteq QBF(V)$,
2. Each variable $v \in V$ is a formula, i.e., $V \subseteq QBF(V)$
3. If $p \in QBF(V)$ and $q \in QBF(V)$ then $\{p \wedge q, \neg p, \neg q\} \subseteq QBF(V)$
4. Given the basic formulae $p \in QBF(V)$ and $q \in QBF(V)$, and a $v_i \in V$ the following formulae can be derived:
   - $p \vee q \equiv \neg(\neg p \wedge \neg q)$
   - $p(v_i \leftarrow q)$ denotes the formula p in which each occurrence of v_i is substituted by q
   - $\exists v_i.\ p \equiv p(v_i \leftarrow true) \vee p(v_i \leftarrow false)$
   - $\forall v_i.\ p \equiv \neg(\exists v_i.\ \neg p)$

- $\exists(v_i, \ldots, v_j).\ p \equiv \exists v_i.\ \ldots\ \exists v_j.\ p$

For the truth assignment a: $V \longrightarrow \{true, false\}$ we define the evaluation function $eval_a$: $QBF(V) \longrightarrow \{true, false\}$ is defined as follows:

- $eval_a(true) = true$ and $eval_a(false) = false$
- $eval_a(v_i) = b$, if $v_i \in V$ and $a(v_i) = b$
- $eval_a(\neg p) = true$, if $eval_a(p) = false$; $eval_a(\neg p) = false$, otherwise
- $eval_a(p \wedge q) = true$, if $eval_a(p) = eval_a(q) = true$; $eval_a(p \wedge q) = false$, otherwise
- $eval_a(p(v_i \leftarrow q)) = true$, if $eval_a(p(v_i \leftarrow eval_a(q))) = true$

For a vector $W = (w_1, .., w_k)$ with $w_i \in V$, a vector $F = (f_1, .., f_k)$ with $f_i \in QBF(V)$, and a formula $p \in QBF(V)$ we also define the vector substitution: $p(W \leftarrow F) = ((((((p(w_1 \leftarrow f_1))(w_1 \leftarrow f_2) \ldots (w_k \leftarrow f_k)$.

For a formula $q \in QBF(V)$, we define the set of assignments $[q] = \{ a \mid eval_a(q) = true\}$ in which the formula q is *true*. Let $\boldsymbol{S'} = \{a \mid a{:}V \longrightarrow \{true, false\}\}$ be the set of all possible assignments. The set operations can then be expressed in QBF formulae as follows:

- $\emptyset = [false]$; $S' = [true]$; $[p] \cup [q] = [p \vee q]$; $[p] \cap [q] = [p \wedge q]$; $S' \setminus [p] = [\neg p]$.

Now consider a Kripke structure $K = (S, R, L)$ that is defined over V. We are going to encode the states and the transitions by a set of QBF formulae. We occasionally use the notation $v^{false} = \neg v$ and $v^{true} = v$ for convenience. We encode a state $s \in S$ by the set of assignments $[q_s]$, where $q_s = \bigwedge_{v \in V} v^{L(s,v)}$. In order to be able to encode transitions, we introduce another set of atomic propositions V' that is a copy of V. The entire state transition relation is encoded by the formula $R' = \bigvee_{(s,t) \in R} (q_s \wedge q'_t)$, where q_s and q'_t are defined over V and V', accordingly. A binary relation $R_{sym} \subseteq (V \longrightarrow \{true, false\})^2$ is defined by $(x, y) \in R_{sym} \Leftrightarrow x \cup (v'_i := y(v_i)) \in [R']$. We define a derived valuation function $\boldsymbol{L'} : S' \times V \longrightarrow \{true, false\}$ by $L'(s', v) = s'(v)$. Symbolic model checking deals with the derived Kripke structure $K' = (S', R', L')$.

The image set that results from applying the relation R_{sym} to the assignments $[p]$ is represented by the formula: $R'(p) = (\exists V.(p \wedge R'))\ (V' \leftarrow V)$. The following derivation proves this:

$y \in [R'(p)] \Leftrightarrow y \in [(\exists V.(p \wedge R'))\ (V' \leftarrow V)] \Leftrightarrow y \cup \{v'_i := y(v_i)\} \in [\exists V.(P \wedge R')] \Leftrightarrow$ exists an x: $V \longrightarrow \{true, false\}$ such that $y \cup \{v'_i := y(v_i)\} \in \{(p \wedge R')\ (v_i \leftarrow x(v_i))\} \Leftrightarrow$

exists an x: $V \longrightarrow \{true, false\}$ such that $x \in [p]$ and $(\{v_i := x(v_i)\} \cup \{v'_i := y(v_i)\}) \in [R']$ $\Leftrightarrow$ exists an x: $V \longrightarrow \{true, false\}$ such that $x \in [p]$ and $(x, y) \in R_{sym}$ $\Leftrightarrow$ $y \in R_{sym}([p])$

The formula $R^{-1}_{sym}(P) = \exists V'.(P(V \leftarrow V') \wedge R')$ for the inverse image can be proved by a very similar derivation.

Now consider a function τ: $2^S \longrightarrow 2^S$ for the original state set S. Such a function is said to be **monotonic** if for $X \subseteq Y \subseteq S$ the inclusion $\tau(X) \subseteq \tau(Y)$ holds. A subset $Y \subseteq S$ is called a **fixed point** of τ if $Y = \tau(Y)$. For a monotonic function τ there is a **least fixed point** denoted $\mu Y.\ \tau(Y)$ and a **greatest fixed point** denoted $\nu Y.\ \tau(Y)$. For a finite S the least fixed point is the limit of the chain $[false] \subseteq \tau([false]) \subseteq \tau(\tau([false]))$ …, and the greatest fixed point is the limit of the chain $[true] \supseteq \tau([true]) \supseteq \tau(\tau([true]))$ …. Note that since S is a finite set the convergence of both chains is reached in at most $|S| + 1$ steps.

Analogously, consider a CTL formula f defined over $(V \cup V') \cup \{x\}$ with x being an additional non-interpreted atomic proposition. For some CTL formula Y over $(V \cup V')$ we recursively define a sequence of CTL formulae $f_i(Y)$ with $f_o(Y) = f(x \leftarrow Y)$ and $f_{i+1}(Y) = f(x \leftarrow f_i(Y))$. The formula sequence is monotonic if $[f_i(Y)] \subseteq [f_{i+1}(Y)]$ (or $[f_i(Y)] \subseteq [f_{i+1}(Y)]$) for each i. The fixed points $\mu Y.f(Y)$ and $\nu Y.f(Y)$ for this sequence are similarly computed in at most $|S| + 1$ steps. For $f^{E(p \cup q)} = q \vee (p \wedge EX\ x)$ the sequence $f_i^{E(p \cup q)}(false)$ is monotonically increasing: q, $q \vee (p \wedge EXq)$, $q \vee (p \wedge EX\ (q \vee (p \wedge EX\ q)))$, … Similarly we observe that for $f^{A(p \cup q)} = q \vee (p \wedge AX\ x)$ the sequence $f_i^{A(p \cup q)}(true)$ is monotonically decreasing.

From the explicit model checking algorithm the following equivalences can be derived:

$K', s' \models p \quad \Leftrightarrow \quad s' \in [p]$

$K', s' \models \neg p \quad \Leftrightarrow \quad s' \in [\neg p]$

$K', s' \models p \wedge q \quad \Leftrightarrow \quad s' \in [p \wedge q]$

$K', s' \models EX\ p \quad \Leftrightarrow \quad s' \in [\exists V'.(p(V \leftarrow V') \wedge R')]$

$K', s' \models E(p\ U\ q) \quad \Leftrightarrow \quad s' \in [\mu Y.(q \vee (p \wedge EX\ Y))]$

$K', s' \models A(p\ U\ q) \quad \Leftrightarrow \quad s' \in [\nu Y.(q \vee (p \wedge AX\ Y))]$, where $AX\ Y = \neg EX\ \neg Y$

These equivalences define the mapping QBF_{CTL} between CTL and QBF formulae over V. Hence, the original model checking problem $K, s \models p$ can be reduced to the **symbolic model checking** problem of verifying whether $q_s \in [QBF_{CTL}(p)]$. The cost of the

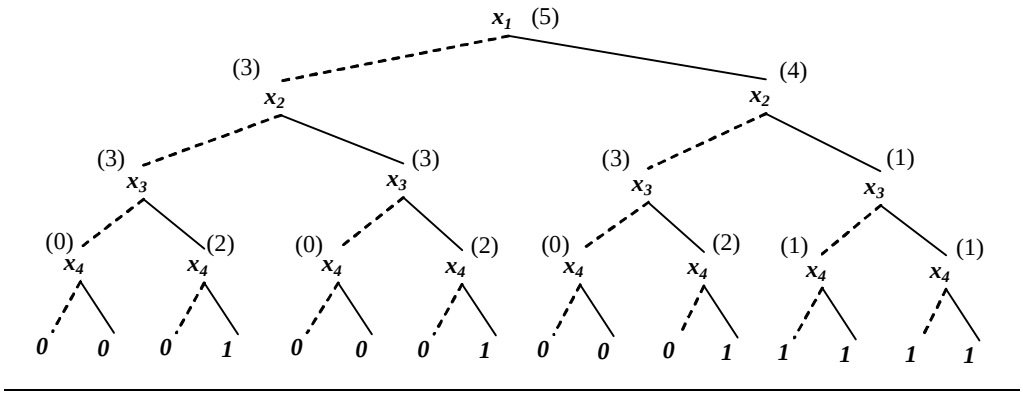

Figure 3: Initial OBDD for (x1 ∧ x2) ∨ (x3 ∧ x4) with $\pi(1) < \pi(2) < \pi(3) < \pi(4)$

symbolic model checking problem boils down to the cost of manipulating QBF formulae by computing their conjunctions, disjunctions, negations, etc.

2.4 Ordered Binary Decision Diagrams

Many symbolic model checkers including that of Statemate use a graph-based data structure coined **ordered binary decision diagram** (OBDD) for representing Boolean functions. Algorithms that allow efficient manipulation (composition, conjunction, negation, etc.) of several Boolean formulae represented as OBDD's are described in [Meinel, C. and T. Theobald 1998].

Consider a Boolean function *f* given by a propositional formula over the set of atomic propositions $X = \{x_1, \ldots, x_n\} \cup \{0, 1\}$. The **Shannon expansion** of *f* around the variable x_i is the function $f' = (x_i \wedge f(x_i \leftarrow true)) \vee (\neg x_i \wedge f(x_i \leftarrow false))$. Clearly, *f* and *f'* are equivalent.

The initial OBDD of the function *f* with respect to a variable ordering π ($\pi(1) < \pi(2) \ldots$) is recursively obtained by applying the following procedure. An OBDD is a binary tree with nodes from *X*. The root node $x_{\pi(1)}$ is associated with the original function *f*. Each node $x_{\pi(i)}$ associated with some function g and *g* is connected by the 0-edge and by the 1-edge to the nodes $x_{\pi(i+1)}$ associated with the left side $g(x_{\pi(i)} \leftarrow false)$ and the right side $g(x_{\pi(i)} \leftarrow true)$ of the Shannon extension of *g*, accordingly. The value of the function for the given assignment *a* can be found through the top-down traversal of the OBDD in that the *0*-edge or the 1-edge are taken at a node x_i if $a(x_i) = false$ or $a(x_i) = true$, accordingly. As an example consider the function $f = (x_1 \wedge x_2) \vee (x_3 \wedge x_4)$ with the ordering $\pi(1) < \pi(2) < \pi(3) < \pi(4)$ whose initial OBDD is depicted in Figure 3 (0-edges are represented by dash lines).

```
01.  the id of a leaf node is its Boolean value (0 or 1)
02.  max_id := 1
03.  for each non-leaf node x_π(i)
04.    if id(0-child(x_π(i)) = id(0-child(x_π(i)) then
05.      id(x_π(i)) := id(0-child(x_π(i))
06.    else if id_table[id(0-child(x_π(i))][id(1-child(x_π(i))] ≠ nil then
07.      id(x_π(i)) := id_table[id(0-child(x_π(i))][id(1-child(x_π(i))]
08.    else
09.      max_id := max_id + 1
10.      id(x_π(i)) := max_id
11.      id_table[id(0-child(x_π(i))][id(1-child(x_π(i))] := max_id
12.    end if
13.  end for
14.  replace all nodes with the same id by a single node
15.  label the two leaf nodes 0 and 1, correspondingly
16.  for k := 2 to max_id
17.    label the node k as x_m where m = max { π(i) | id(x_π(i)) = k}
18.  end for
```

Figure 4: OBDD Reduction Algorithm

An OBDD can be reduced to the canonical form as described by Bryant [1986] (see Figure 4). To this end, each node is assigned a virtual id (parenthesized number in the Figure above) based on their child node values in a bottom-up manner. A new id is created for a node labeled $x_{\pi(i)}$ with a previously unseen child id pair which is captured in a two-dimensional array *id_table*[*0-child*(x_i)][*1-child*(x_i)]. Each id stands for a unique Boolean function, such that several nodes with the same id are replaced by a single node which makes sure that equivalent sub-functions are computed only once by the OBDD.

The resulting reduced OBDD is optimal for the given variable ordering in that it does not contain any isomorphic subtrees computing identical Boolean functions. Figure 5 depicts the canonical (reduced) form of the sample OBDD of Figure 3. An interesting peculiarity of canonical OBDD's consists in that their size highly depends on the chosen variable

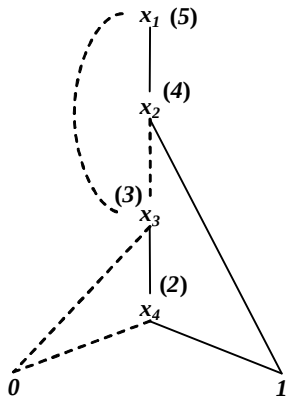

Figure 5: Canonical OBDD for (x1 ∧ x2) ∨ (x3 ∧ x4) with π(1) < π(2) < π(3) < π(4)

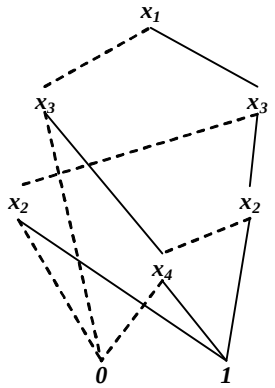

Figure 6: Canonical OBDD for (x1 ∧ x2) ∨ (x3 ∧ x4) with π(1) < π(3) < π(2) < π(4)

ordering. As an example consider the canonical OBDD shown in Figure 6 which computes the same function as the left-hand example relatively to slightly different variable ordering, where x_2 and x_3 are swapped: This ordering is certainly less preferable because the OBDD size increases by two nodes and four edges, accordingly. OBDD-based symbolic model checking works with the OBDD representation of the QBF-encoded Kripke structure *K'*. It outperforms the explicit model checking algorithm when it "guesses" a good variable ordering.

There are several heuristics for finding a good variable ordering to minimize an input OBDD. Among most popular reordering heuristics that are also used in Statemate we would like to mention the **sifting** algorithm [Rudell 1993] and the **window technique** [Fujita et al. 1991]. The sifting algorithm picks one variable x_i and tries out all of its possible orderings relatively to the remaining variables whose order remains fixed (i.e., first $\pi_1(i) < \pi_1(1) < \pi_1(2) < \pi_1(3)$ …, second $\pi_2(1) < \pi_2(i) < \pi_2(2) < \pi_2(3)$ …, third $\pi_3(1) < \pi_3(2) < \pi_3(i) < \pi_3(3)$ …, and so on.). The window technique looks for an optimal OBDD using a sliding window of *k* variables and trying out all *k*-factorial permutations of the variables within the window.

2.5 State-and-Activity Charts

The following brief introduction to state-and-activity charts is based on [Harel and Naamad 1996] (see [Harel and Politi 1998] for the complete Statemate semantics). A system model is based on a hierarchical activitychart, in which the functional capabilities of the system are captured by activities, and the data elements and signals that can flow between them. The semantics of this functional description is that information can flow, but it does not specify what will happen, when, or why. These behavioral aspects except for external, nondeterministic activities are specified in **statecharts,** sometimes called **control activities,** usually one for each activity in the activitychart.

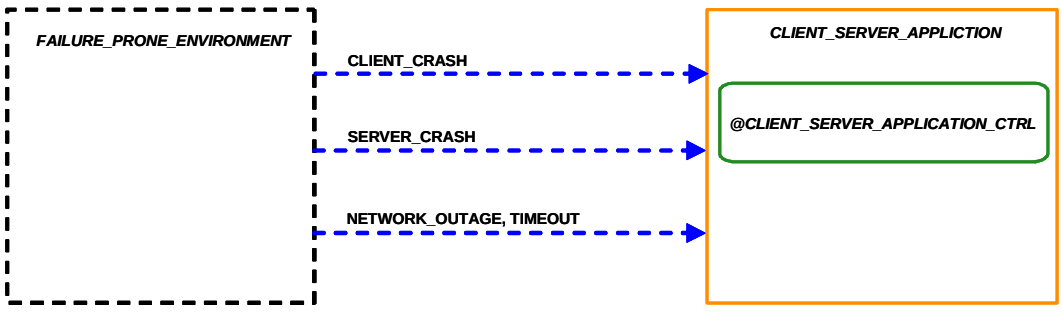

Figure 7: Sample Activitychart

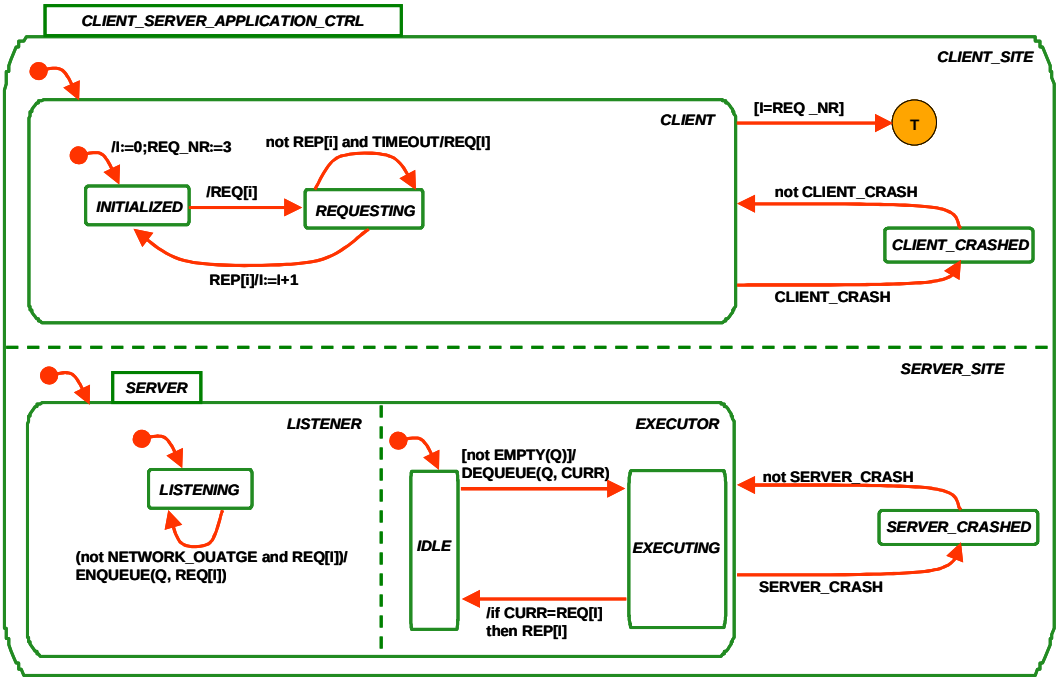

Figure 8: Sample Statechart

Figure 7 shows a sample activitychart. The activitychart consists of the actual activity *client-server-application* and an external activity (the dashed rectangle) that supplies the failure events *client-crash*, *server-crash*, *network-outage*, and *timeout* as nondeterministic input. The behavior of the *client-server-application* is determined by the control activity *client-server-application-ctrl* as indicated by the rounded rectangle. The statechart (see Figure 8) of this control activity is a subchart of the activitychart, which is declared through the prefix "@". The detailed behavior of the statechart will be explained in the following subsections.

Statecharts are finite state automata (FSA) with additional features:

- **Event-condition-action rules** (**ECA rules**, written in the form $e[c]/a$ as annotations of state transitions) determine that in response to an occurrence of the event *e* the system executes within a step the action *a*, moves from the source state to the target state of the transition when the condition *c* is true. ECA-rules can also be associated with a state, which defines its **static reactions.**
- **Nesting** of entire statecharts into subordinate states is a mechanism for specification refinement and composability.
- **Orthogonal components** (essentially cross products of automata) express parallelism of the system.

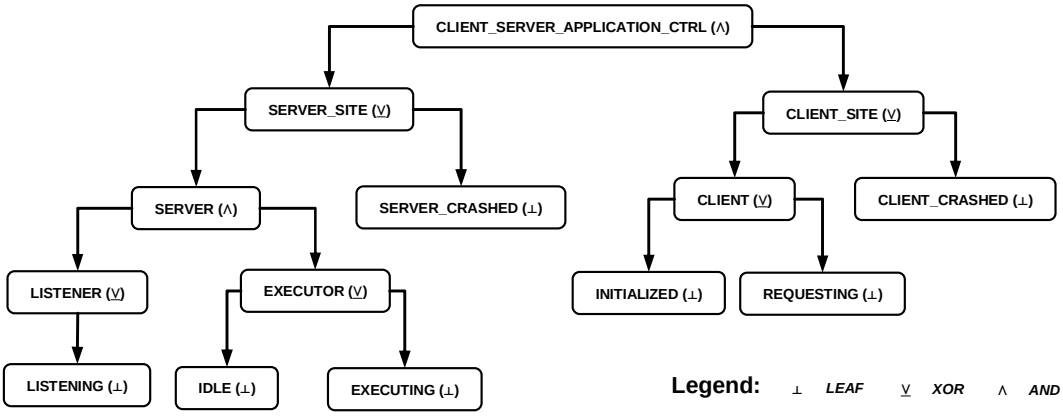

Figure 9: State Hierarchy of the Statechart

For example, in the statechart of Figure 8 the transition of the *executor* from the state *idle* to the *executing* state fires if the condition *not empty*(*q*) holds, and then triggers the action *dequeue*(*q*, *curr*). The entire statechart for the *executor* is a substate of the *server*; and the *executor* and the *listener* are orthogonal components, running in parallel with synchronization based on the events, conditions, and actions on globally shared variables, i.e., the request queue in this particular case.

There are three types of **states** in a statechart: **OR-states**, **AND-states**, and **basic states**. All states together form a tree. OR-states have **substates** that are related to each other by *exclusive or*, whereas AND-states enclose **orthogonal** substates (separated by dash-lined boundaries) related by *and*. An orthogonal substate is an OR-state that usually contains other substates. States that contain substates are sometimes called **superstates**. Basic states have no substates, and are the leaves of the **state hierarchy**. Figure 9 depicts the state hierarchy of the sample statechart specification of Figure 8.

When an AND state is entered, all its orthogonal substates are entered, too. When an OR state is being entered (activated), its **default substate** is being entered at the same time. The default substate is defined by specifying it as a target of a special **default transition** without a source state (e.g., the substate *initialized* of the state *client* in Figure 8). The set of currently entered (active) states is called a **state configuration**. The state configuration is closed upwards in the sense that if a state is active then so is its parent state. Thus, the set of the active basic states called **basic configuration** suffices to describe the complete state configuration.

The formal definition of a statechart is provided as a tuple **(*S, SM, R, SR, DT, T, D, ES*)**, where

- ***S*** is the set of statechart states

- $\boldsymbol{SM} \subseteq S \times \{AND, OR, basic\}$ is the state type relationship
- $\boldsymbol{SR} \subseteq S \times S$ is the substate relationship with $(p, q) \in SR$ if p is a substate of q, we write $p \underset{SR}{\leq} q$ for $(p, q) \in SR$; $\boldsymbol{SR}^*$ is the transitive closure of SR, we write $p \underset{SR^*}{\leq} q$ for $(p, q) \in SR^*$; the root state is $\boldsymbol{R} = \max_{SR^*}(S)$
- $\boldsymbol{DT} \subseteq S \times L$ and $\boldsymbol{T} \subseteq S \times S \times L$ are the sets of default and regular transitions, respectively, with labels of the form $e[c]/a$ from the label set $\boldsymbol{L}$
- $\boldsymbol{D}$ is the finite set of Boolean variables representing individual bits of data items along with events and conditions
- $\boldsymbol{ES} \subseteq D$ is the set of external stimuli out of control of the system.

2.5.1 Statechart State Configurations

Given the statechart state set S, the set of state configurations $SC \subseteq 2^S$ is computed in a top-down manner by flattening the statechart. We define a function *confset*: $S \longrightarrow 2^S$. For a state $s \in S$ the expression *confset*(s) computes the configuration set of the sub-statechart rooted in s. The expression *confset*(R) corresponds to the set of all syntactically possible state configurations of the original statechart.

In the definition of the function *confset* we use a **cross union operator** $\underline{\times}$ that we introduce to compute the set system consisting of pairwise unions of the elements of two other systems A and B, i.e., $A \mathbin{\underline{\times}} B = \{\, a \cup b \mid a \in A \text{ and } b \in B \,\}$

For an $s \in S$, the expression *confset*(s) is recursively computed as:

- *confset*(s) = $\Big(\bigcup_{(s',s) \in SR} confset(s') \Big) \mathbin{\underline{\times}} \{\{s\}\}$, if s is an OR-state.
- *confset*(s) = $\Big(\underline{\bigtimes}_{(s',s) \in SR} confset(s') \Big) \mathbin{\underline{\times}} \{\{s\}\}$, if s is an AND-state.
- *confset*(s) = $\{\{s\}\}$, if s is a basic state.

For instance, the configuration set of the sample statechart of Figure 8 is computed by resolving the following equations:

$\boldsymbol{SC} = confset(client\_server\_application\_ctrl) = confset(server\_site) \mathbin{\underline{\times}} confset(client\_site) \mathbin{\underline{\times}} \{\{client\_server\_application\_ctrl\}\}$

$confset(server\_site) = (confset(server) \cup confset(server\_crashed)) \mathbin{\underline{\times}} \{\{server\_site\}\}$

$confset(server) = confset(listener) \mathbin{\underline{\times}} confset(executor) \mathbin{\underline{\times}} \{server\}$

$confset(listener) = confset(listening) \times \{\{listener\}\}$

$confset(listening) = \{\{listening\}\}$

$confset(executor) = (confset(idle) \cup confset(executing)) \times \{\{executor\}\}$

$confset(idle) = \{\{idle\}\}$

$confset(executing) = \{\{executing\}\}$

$confset(server\_crashed) = \{\{server\_crashed\}\}$

$confset(client\_site) = (confset(client) \cup confset(client\_crashed)) \times \{\{client\_site\}\}$

$confset(client) = (confset(initialized) \cup confset(requesting)) \times \{\{client\}\}$

$confset(initialized) = \{\{initialized\}\}$

$confset(requesting) = \{\{requesting\}\}$

$confset(client\_crashed) = \{\{client\_crashed\}\}$

The **default subconfiguration** of a state is defined by the function $defaultconf: S \longrightarrow 2^S$:

- $defaultconf(s) = \{s\} \cup (\bigcup_{s' \underset{SR}{<} s} defaultconf(s'))$, if s is an AND-state.
- $defaultconf(s) = \{s\} \cup defaultconf(s')$, if s is an OR-state, and s' is the default substate of s (i.e., $s' \underset{SR}{<} s$ and there is a $(s', l) \in DT$).
- $defaultconf(s) = \{s\}$, if s is a basic state.

The initial configuration of the statechart is given by $conf_0 = defaultconf(R)$, which is an implication of entering the root state**.** In Figure 8, the system will enter in the initial step the following states: The root state *client-server-application-ctrl*, its orthogonal substates *client-site* and *server-site*, the default substates of the *client* and *server-site client* and *server* accordingly, the *client*'s default basic state *initialized*, the *server*'s orthogonal components *listener* and *executor*, and finally their corresponding default substates *listening* and *idle*.

2.5.2 Statechart Transitions

There are several subtleties of transition behavior stemming from nesting, i.e., when the source and the target states have different parent states. Entering the target state implies entering of its previously inactive ancestors in addition to the default substate. If the state is left, then so are all its descendant states. This implies that the state can be left, even if it is visually presented as a sink. Moreover, if the transition crosses boundaries of any ancestor states of the source state (i.e., the target state is neither a source state sibling nor a descendant of a source state sibling), these ancestor states are left as well. In order to

describe precisely which states would be left and entered if a transition $tr = (s, t, l) \in T$ would fire in some configuration *conf* with $s \in conf$, we need to provide definitions of the following auxiliary structures:

- The set ***branch*(*s*)** = $\{s\} \cup \{p \in S \mid s \underset{SR^*}{<} p\}$ comprises the nodes of the branch from the root down to the state *s* including *s* on the state hierarchy. The states from *branch*(*s*) are always implicitly active when *s* is active.
- The set ***tree*(*s*)** = $\{s\} \cup \{p \in S \mid p \underset{SR^*}{<} s\}$ consists of the nodes of the complete subtree rooted in the node *s*, and *s* itself. These nodes are implicitly left whenever *s* is left.

The set ***EN*(*tr, conf*)** of the states entered due to *tr* is computed in three steps:

1. $EN_1 := (branch(t) \setminus branch(s)) \cup defaultconf(t)$
2. $EN_2 := \emptyset$; $missed\_orth := \emptyset$;
   for each AND-state $ands \in EN_1$
   $missed\_orth := missed\_orth \cup \{orthc \mid orthc \underset{SR}{<} ands \text{ and } orthc \notin EN_1\}$
   for each $orthc \in missed\_orth$
   $EN_2 := EN_2 \cup default(orthc)$
   (These for-loops are necessary when *tr* directly enters a substate that has one or more AND-states as ancestors because their orthogonal components aside the branch of the target state still need to be activated)
3. $EN(tr, conf) := EN_1 \cup EN_2$

The set ***EX*(*tr, conf*)** of the states exited due to *tr* is defined as:

1. $EX(tr, conf) = conf \setminus (conf \cap EN(tr, conf))$, if $s \neq t$ neither $s \underset{SR^*}{<} t$ nor $t \underset{SR^*}{<} s$
2. $EX(tr, conf) = conf \cap tree(\max\limits_{SR^*} \{s, t\})$, otherwise

2.5.3 Statechart Textual Expression Language

ECA labels are written in the statechart **textual expression language** for event, condition, and action expressions. An event is conceptually different from a condition in that it lasts only for a single step unless explicitly internally re-generated during the step or re-supplied by the environment, whereas the condition keeps its value until explicitly changed. In addition to user-defined events and conditions, Statemate defines a number of system events and conditions, called **condition** and **event operators**, respectively. The

configuration-related events *exited*(*s*) (abbreviated ***ex*(*s*)**) and *entered*(*t*) (abbreviated ***en*(*t*)**) fire when *s* is left and *t* is entered (regardless whether explicitly or implicitly). A counterpart condition *in*(*s*) is *true* if *s* is active. The event-array-related-events ***any*(*arr*)** and ***all*(*arr*)** are syntactical sugar for event expressions *arr*[1] *or arr*[2] … *or arr*[*n*] and *arr*[1] *and arr*[2] … *and arr*[*n*], correspondingly. The activity-related events *started*(*act*) (abbreviated ***st*(*act*)**)and *stopped*(*act*) (abbreviated ***sp*(*act*)**) are generated when an activity is started and terminated by an action part of some transition. We will often use the event *written*(*d*) (abbreviated ***wr*(*X*)**) when an action assigns a value to *X* to show that data is written exactly once.

An **event expression** is defined as propositional formulae over atomic events. A **condition expression** is analogously defined as propositional formulae over atomic conditions. An atomic event is either an atomic proposition (interpreted as "the event has been generated") or an event operator. An atomic condition is an atomic proposition or a condition operator. We consider another type of atomic conditions that are given by the user in the form of comparisons of data-items (=, <, etc) encoded in ALU-style propositional formulae over individual bits. For instance, when two *n*-bit numbers *d* and *d'* need to be tested for equality, this is converted into the formula $\bigwedge_{i=0}^{n} \neg(d_i \veebar d'_i)$. Let *E* and *C* be event and condition expressions, accordingly. The event-condition-part of a transition label is an **event-condition expression (ECX)** whose syntax is recursively defined as follows:

- *E*[*C*] is an ECX
- If *E* is omitted, [*C*] is a syntactical sugar for *true*[*C*]
- If *C* is omitted, *E* is a syntactical sugar for *E*[*true*]
- An empty ECX is a syntactical sugar for *true*[*true*]
- If ecx_1 and ecx_2 are ECX, then so are ecx_1 *or* ecx_2, ecx_1 *xor* ecx_2, and ecx_1 *and* ecx_2.

Note that the expression e_1 *or* $e_2[c_2]$ allows the transition to fire when e_1 occurs or when e_2 is sensed while c_2 is *true*, which is different from the expression $(e_1$ *or* $e_2)[c_2]$ that fires only while c_2 is *true* for either event occurrences.

An action can be as simple as generating an internal output event (written as …/***e***) or a complex action sequence including IF and WHILE statements. Higher level control activities usually orchestrate subordinate activities by starting (…/***st!*(*act*)**) and stopping them (…/***sp!*(*act*)**), accordingly.

2.5.4 Statechart Semantics

The behavior of a system **(*S, SM, R, SR, DT, T, D, ES*)** is a set of possible runs, each representing the responses of the system to a sequence of external stimuli of *ES* (i.e., external events, conditions, and data-items) generated by its environment. A run consists of a series of snapshots of the system's situation; such a snapshot is called a **status** that consists of the state configuration and the **execution context.** An execution context is a valuation of events, conditions, and data-items. That is, a status is element of ***statusset*** = $confset(R) \times \{ val \mid val: D \longrightarrow \{true, false\}\}$. The initial status is given by the initial configuration and the default values of the data-items and conditions. The status changes after executing a **step**. At the beginning of each step, the environment supplies the system under description with external stimuli. These, together with internal changes that occurred in the system during the previous step, trigger transitions between states. Note that from the perspective of model checking that exhaustively tests all possible situations, external stimuli are just convenient syntactical sugar elements. We will show later in this section that we need to consider only internal data and signals because external stimuli are equivalent to internal events, conditions and variable whose values are generated in a nondeterministic way.

To perform the $i+1^{st}$ step, the system evaluates the status $status_i = (conf_i, val_i)$ after the i^{th} step in the following manner. The system identifies the set of **enabled transitions** $ET_i = \{ (source, target, ecx/action) \in T \mid source \in conf_i$ and $eval(ecx) = true\}$, where the function $eval: ECX \times statusset \longrightarrow \{true, false\}$ is defined as follows:

- $eval(ecx_1\ op\ ecx_2, conf_i, val_i) = eval(ecx_1, conf_i, val_i)\ op\ eval(ecx_2, conf_i, val_i)$ for $op \in \{ and, nand, or, nor, xor\}$
- $eval(not\ ecx, conf_i, val_i) = \neg eval(ecx, conf_i, val_i)$.
- $eval(E[C], conf_i, val_i) = eval(E, conf_i, val_i) \wedge eval(C, conf_i, val_i)$
- $eval(in(s), conf_i, val_i) = true \Leftrightarrow s \in conf_i$
- $eval(d, conf_i, val_i) = val_i(d)$, for $d \in D$.
- $eval(const, conf_i, val_i) = const$, for $const \in \{true, false\}$

Clearly, there may be multiple enabled transitions. However, in contrast to an FSA, this does not necessarily imply a nondeterministic situation. An enabled transition that implies leaving a higher-level state of the statechart hierarchy is prioritized over enabled transitions that imply leaving any descendent states (**transition priority rule**). Just enabled transitions whose effect leads to leaving the same state at the highest level are

called to be in conflict, and constitute a nondeterministic situation that is randomly resolved by Statemate. Enabled non-conflict transitions deterministically fire at once (**greediness property**).

For any two different transitions *tr* and *tr'* from ET_i we define the transition priority relation $\underset{TP}{<}$ by $tr \underset{TP}{<} tr'$ if $\max_{SR^*}(EX(tr, conf_i)) \underset{SR^*}{<} \max_{SR^*}(EX(tr', conf_i))$. The transitions *tr* and *tr'* are **in conflict** if $\max_{SR^*}(EX(tr, conf_i)) = \max_{SR^*}(EX(tr', conf_i))$, which we express as $tr \underset{TP}{=} tr'$. If tr_i and tr_j are incomparable, i.e., $EX(tr, conf_i) \cap EX(tr', conf_i) = \emptyset$, then tr_i and tr_j will fire simultaneously unless they are suppressed by some other prioritized transition. The possible maximum subsets of enabled transitions are computed as follows:

1. $\boldsymbol{NST_i} = \{ tr \in ET_i \mid$ there is no $tr' \in ET_i$ with $tr \underset{TP}{<} tr' \}$ is the set of non-suppressed transitions in $status_i$.
2. $\boldsymbol{NST_i[tr]} = \{ tr' \in NST_i \mid tr \underset{TP}{=} tr' \}$ is the equivalence class of transitions that are in conflict with *tr* including the transition *tr* itself. There are $k \leq \boldsymbol{\#NST_i}$ such partitions. If $\#NST_i[tr] > 1$, the statechart is **nondeterministic**. Note that the removal of the suppressed transitions in the previous step was correct by the following argument. If $tr'' \underset{TP}{<} tr'$ for some $tr' \in \#NST_i[tr]$, then the property $tr'' \underset{TP}{<} tr'$ holds for all $tr' \in \#NST_i[tr]$. Regardless of the nondeterministic choice of a transition from $NST_i[tr]$, a suppressed transition always remains suppressed.
3. $\boldsymbol{FIRESETS_i} = \bigtimes_{j=0}^{k} NST_i[tr_j]$ is the set of maximum transition subsets in $status_i$.

Each $fs \in FIRESETS_i$ determines an alternative successor status $(conf_{i+1}, val_{i+1})$ for $status_i$ that is constructed by the function $\boldsymbol{next}(conf_i, val_i, fs)$ as follows:

1. The set $EN_{fs} := \bigcup_{tr \in fs} EN(tr)$ contains the states that were entered through transitions. For each states $s \in EN_{fs}$ we set $val_{i+1}(en(s)) := true$; otherwise, $val_{i+1}(en(s)) := false$
2. The set $EX_{fs} := \bigcup_{tr \in fs} EX(tr)$ contains the states that were exited through transitions. For each states $s \in EN_{fs}$ we set $val_{i+1}(ex(s)) := true$; otherwise, $val_{i+1}(ex(s)) := false$

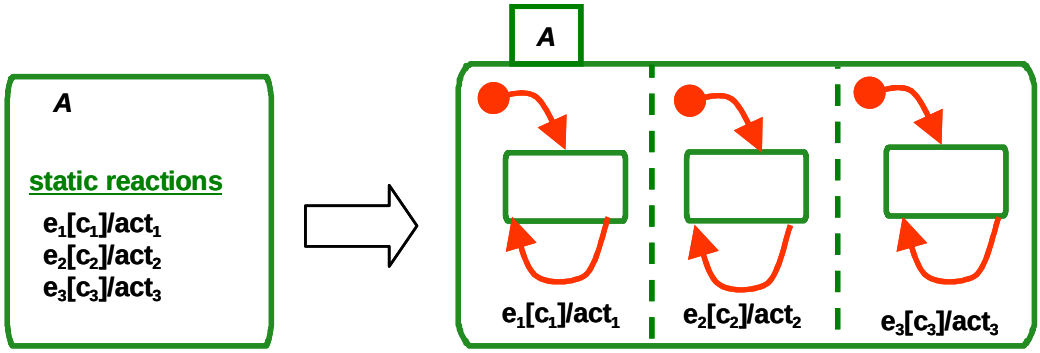

Figure 10: Conversion of Static Reactions

3. $conf_{i+1} := \bigcup_{s \in EN_{fs}} branch(s)$

4. The action set $ACT_{fs} = \{a \mid (s,t, ecx/a) \in fs\}$ determines the rest of the valuation function val_{i+1}. For instance, $val_{i+1}(e) := true$ if e is generated by some $a \in ACT_{fs}$ and $val_{i+1}(e) := false$, otherwise. The valuation of data-items and conditions is obtained by evaluating the action expressions. Clearly, if $d \in D$ is not affected by any action, we obtain $val_{i+1}(d) := val_i(d)$.

The image set $step(conf_i, val_i) = \{ next(conf_i, val_i, fs) \mid fs \in FIRESETS_i\}$ defines possible successor statuses.

The system terminates either explicitly when some higher-level activity calls action *sp!* or implicitly by entering a **termination connector** (a circle labeled *T* in Figure 8). Upon termination, the state configuration becomes an empty set, and the control activity remains terminated until explicitly restarted by some higher-level activity through the action *st!*.

At this point we are able to construct the Kripke structure $K_{sc} = (S_{sc}, R_{sc}, L_{sc})$ of the statechart and apply model checking as described in Sections 2.2 through 2.4:

- $S_{sc} = statusset$,
- $R_{sc} = \{(st_i, st_{i+1}) \mid st_{i+1} \in step(conf_i, val_i)\}$
- $L_{sc} : S_{sc} \times D \longrightarrow \{true, false\}$ with $L_{sc}(conf, val, d) = val(d)$

Last but not least, we show how advanced Statemate features such as external input and static reactions are mapped to basic statechart elements. In Figure 10, you may see the conversion of a state with three static reactions to an equivalent statechart without static reactions. A statechart specification (with the root state *R*) using external events e_1 and e_2 can be converted to an equivalent statechart (with the new root state *R'*) that has an

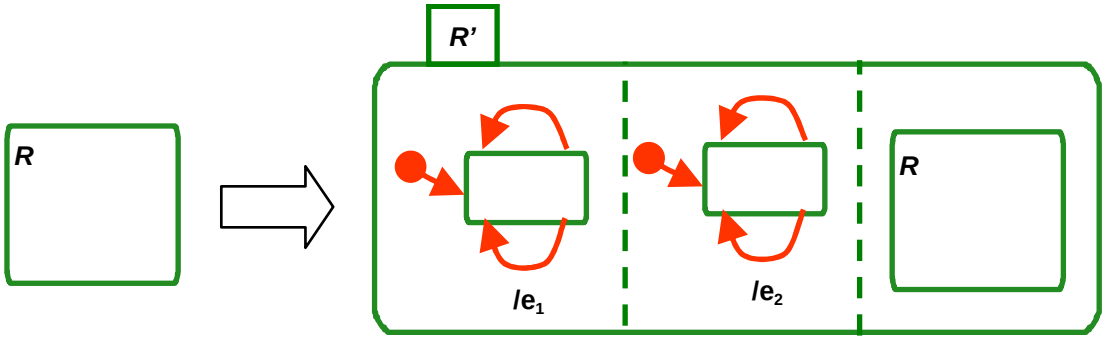

Figure 11: Conversion of External Stimuli

additional orthogonal component for each external event (or data-item bit) that generates them nondetermistically as depicted in Figure 11.

2.5.5 Sample Scenario

Now we are ready to describe how the model in Figure 8 works. The system starts up in the initial configuration as shown above. The single-threaded client is going to submit three requests modeled as a bounded event array *req* to the server, which is controlled by the integer variable *req_nr*. Thus, we can immediately verify a simple CTL formula $AG\ i \leq req\_nr$, i.e., the counter *i* never exceeds *req_nr*. The *server* owns two threads (orthogonal components): the *listener* that accepts and queues client requests, and the *executor* responsible for the queued request execution.

Regardless of the application progress, normal operation of the client and the server is interrupted when the corresponding *crash* events are sensed due to the transition priority rule, which can be expressed in CTL as $AG\ (server\_crash \leftrightarrow AX\ in(server\_crashed))$. Normal operation of the component is resumed as soon as the external environment stops supplying the *crash* event. More precisely, the following CTL formula holds for the server: $AG\ (\ server\_crash \wedge AX\ \neg server\_crash \rightarrow AX\ (AX\ (in(listening) \wedge in(idle)))$.

With the second step during normal operation, the client moves from the state *initialized* to the state *requesting* and generates the 0^{th} element of the event array *req*; if the queue *Q* is not empty, the server dequeues and executes a request in the state *executing* in the very same step. When there is no *network_outage*, the server adds *req*[0] to *Q* with the third step, whereas the server checks if the reply *rep*[0] for the client request is already available and returns it to the *idle* state (note that this is an unconditional transition with a conditional reply action). On the client side, nothing changes until the step after the corresponding reply has been generated by the *server* or the external event *timeout* has been sensed. In the former case, the client prepares the next request and returns to the

state *initialized*; in the latter case, the original request is resubmitted. The statechart terminates when the request counter *i* has reached the value of *req_nr*, and the termination connector is entered. Obviously, there is an execution path with a finite number of external failure events, and the following formula must hold: $EF\ (i{=}req\_nr \wedge in(T))$.

2.5.6 Statechart Time Models

Statemate supports two models of timing: **synchronous** and **asynchronous**. The synchronous time model assumes that the system executes a single step every time unit, reacting to all the external changes that occur in the one time unit that elapsed since the completion of the previous step. Prior to computing a new step the system always senses for external stimuli. The asynchronous time model assumes that the system reacts whenever an external change occurs, allowing several external changes to occur simultaneously and, most importantly, allowing several steps to take place within a single point in time. Such a collection of steps is sometimes called a **superstep**. New external changes are not sensed until the superstep completes.

At first glance, one might think that the asynchronous time model is the best choice for modeling an asynchronous environment like a Web service with virtually all components residing on distinct machines. However, it turns out that the asynchronous model unnecessarily complicates modeling arbitrary failure situations because a superstep, once started, is executed atomically. In the sample statechart of Figure 8, the client would never crash in the state *initialized* because this state is left within the same superstep via the unconditional transition into the state *requesting*, whereas the *server* would never fail in the state *executing* because it would immediately go back to the state *idle*. In a real world model we have to allow failures to occur in each and every state configuration, i.e., even between micro-steps. In order to achieve this with the asynchronous model, we would have to add to each transition an external event in a conjunctive manner, such that the model checker will consider also runs in which a super step comprises only one single step by setting such an event initially to *false*. Besides the fact that this method is far from elegant, one should also consider that the complexity of model checking is $O(n^2)$ in the size n of the specification that in turn grows exponentially with each new Boolean variable. As the result, this discussion suggested using the synchronous model. However, in the specifications we do not assume that any changes on different components have to

take place at the same time, and we do not pass time parameters across component boundaries either. Thus, we do not lose generality.

2.5.7 Generic Activities

Statemate supports modularized specification designs and module reusability. It provides a notion of a **generic activity**. A generic activity exposes a set of formal in- and output parameters, which have to be bound to concrete variables (events, conditions, etc.) whenever the activity is instantiated (reused) just like arguments for a function call in a programming language.

3. Background on Recovery Technology

"It's not whether you get knocked down. It's whether you get up again." - Vince Lombardi

It is sad but true that there is no large-scale software without bugs. The only response consists in modularizing software into components thereby trying to isolate bugs and limit their impact on the overall system. Software designers pay special attention to keeping critical components, e.g., those responsible for durable changes on a hard drive as small as possible thereby reducing the likelihood of bug occurrences in them.

Operations relevant for the component state are logged in the order of occurrence on component's persistent storage. As soon as the failure is detected through the runtime environment (e.g., a virtual machine or a recovery manager) abnormally terminates and restarts the component, i.e., the runtime environment triggers a **soft crash** of the component. A component terminating upon detecting a failure is called **fail-stop**. The task of recovery is to recreate the most recent component state that is consistent with the states of other components.

3.1 Failure Model

It is very important to understand when recovery is applicable. Recovery deals only with **omission failures**, where the system fails to execute some action. From the recovery standpoint, two relevant classes of software bugs may lead to an omission failure [Gray and Reuter 1993]:

- **Bohrbugs** are deterministic programming errors that can be reproduced after their first occurrence repeatedly such that they are normally eliminated during development and testing phases. In some cases, even Bohrbugs pass quality assurance undetected and when such an error occurs in a released production system, recovery will always drive the system into the same problem. This makes it impossible to resume normal operation automatically but the log sequence containing an entire execution path leading to the Bohrbug will help developers rectify the system.
- **Heisenbugs** occur only under specific hard-to-reproduce conditions usually during a peak load, which makes them appear nondeterministic from the system operation and debugging perspectives. Recovery "works" when a Heisenbug does not corrupt stable state and because a Heisenbug is unlikely to recur after the subsequent restart. This does not imply that the system cannot fail during recovery

again but it is almost impossible for the system to fail because of the original Heisenbug several times in a row.

A number of failures are outside the recovery scope and consequently are not covered in this thesis. They are subject to other areas of fault-tolerant computing.

- Recovery does not handle **commission**, so-called Byzantine, faults typically caused by malicious code intrusion of the system [Castro and Liskov 1999].
- Recovery usually does not deal with transient hardware failures such as bit flips, instant network outages etc. These failures are mostly intercepted and masked either by the hardware-integrated self-correcting code or at the operation system level (consider, e.g., RAID storage systems [Katz et al. 1989], and the Internet protocols TCP/IP [Comer 1988]). However, if they are not masked, recovery will trigger a soft crash.
- Human user faults such as intentionally or unintentionally provided inaccurate input normally cannot be handled automatically. Some systems are able to assist system administrators by suggesting compensating transactions that would restore a valid system state [Korth et al. 1990].
- Recovery depends on correctly functioning stable storage hardware. High recovery log availability can be achieved by replication. Recovery cannot be performed if all replicas with state and logging information corrupt simultaneously.
- Faulty recovery code leads to durable system failures. Even worse, it can create an inconsistent state without any further notice. This is why it is so important that recovery algorithms be verified.

3.2 Data Recovery

Database systems provide durable data storage service for applications. They are large-scale multi-user systems that are therefore highly optimized for response time. The most significant performance bottleneck is exhibited by the hardware used to store data persistently. An access to **secondary storage** providing persistence, typically a magnetic disk, also called **hard disk**, is five to six orders of magnitude slower than a **main memory** access. Thus, database systems are geared to reduce the number of disk I/O's.

Database systems **physically** arrange **logical** data units and derived data structures such as indices in uniquely identified fixed-size blocks called **disk pages**. The cost to access a

disk page of several KB is one order of magnitude higher than the cost of fetching it into main memory. A typical disk page size ranges from two to eight KB. In order to save disk I/O's the database system caches a subset of disk pages in main memory and lazily writes them back, e.g., when a cached page needs to be dropped to accommodate a new page. The disk-resident pages constitute the **stable state** of a database. The non-cached fraction of the stable state and the cached pages define altogether the **current state** of the database. The fact that changes made by completed transactions may be missing in the stable state, whereas uncommitted changes may have been made persistent, is dealt with by transaction recovery.

Database systems use high-level logical operations coined access methods for reading and manipulating logical data units (e.g., tuples). A logical operation usually implies accessing multiple pages. For example, an update of a key attribute of a tuple in a sorted table may reposition the tuple on its original page, or move it to a different page, and this also potentially incurs updating pointers in indexing structures.

Transaction atomicity and persistence is usually implemented by logging data update operations. The recovery log is organized as a list of sequence-numbered entries. A log entry contains information needed for redoing and undoing the operation logged. There are three ways to log depending on the system-level of operations [Gray and Reuter 1993]:

- **(Physical)** Logical operations are considered as a sequence of physical page writes (operations). The **before-** and the **after-image** of each page (fragment) are logged for undo and redo, respectively, which makes recovery of a page very simple but log entries consume amount of space that is too large relatively to actually updated parts of the pages.
- **(Logical)** High-level operations are logged along with its parameters. High-level operations have to be designed in a way that an inverse operation for undo is always computable. Logical log entries are compact but redo recovery becomes more complicated since a single operation involves multiple pages, and conventional redo recovery of the given operation requires replaying of all previous logical operations in the original order. Hence, logical recovery incurs a much higher I/O load in comparison to physical recovery.
- **(Physiological)** Physiological logging reconciles both methods above in that it considers high-level operations as a sequence of *logical* operations within a single

physical page each, e.g., update the column *j* of a record at slot *i* on page *p* with the bit vector *v*, etc. Physiological log entries are compact and redo recovery of a physiological operation on page *p* requires replaying of all previous operations on page *p* only.

In the following, we sketch an ARIES-style data recovery algorithm [Mohan et al. 1992] implemented in some commercial database products, e.g., in IBM DB/2 and Microsoft SQL Server.

Recovery after a crash comprises three passes. In the **analysis pass,** the entire log is scanned and ids of encountered transactions are added to the *active transaction table* (*ATT*). When a commit log entry is scanned for a transaction, it is called a **winner**, and its id is removed from the *ATT*. The transactions left in the ATT after the analysis pass are **losers**, i.e., incomplete to-be-undone transactions. During the **redo pass,** recovery performs a forward scan of the log again and applies physiological redo operations regardless whether it belongs to a loser or to a winner, which recreates the database state as of crash. During the **undo pass**, the log is scanned backwards, and each operation belonging to a loser transaction is undone.

Physiological and logical operations are usually not idempotent (e.g., insertion of a tuple into a page). Recovery applied multiple times has to recreate the same committed state (**idempotence**) for the database to survive multiple failures. Redo and undo are designed to test whether the disk page under consideration has already been affected by the given operation. To provide a **testable state**, every data page is stamped with the **log sequence number** (LSN) of the last operation applied to it. Recovery compares the LSN in the page header with the LSN of the given log entry in order to determine whether to apply the logged operation to the page.

As with data pages, initially log entries are created in main memory, in the **log buffer;** they are lazily written to disk (**stable log**) in increasing LSN order. In order to preserve data recoverability, the log manager has to obey certain rules when it flushes log entries from the log buffer up to some LSN to disk, which we call **force-logging**.

- **Write-Ahead-Logging.** Prior to flushing a disk page from the cache, the log manager has to make sure that the log entry for the most recent update to this page and its predecessors are forced to the stable log. Otherwise, undo of uncommitted changes on this page is impossible.

- **Write-On-Commit.** Persistence of a committing transaction is achieved by flushing the commit log entry and all of its preceding entries including those belonging to yet uncommitted transactions. Otherwise, the committed transaction cannot be replayed.

The stable log usually resides on a dedicated disk, and in contrast to the data disks, it is written in an append-only manner. Sequential writes are one order of magnitude faster than random I/O's. This is one of the reasons why managing persistence of committed transactions using a recovery log is more efficient than flushing pages with committed data, potentially scattered all over the disk.

Fast recovery is crucial for high availability. To speed up recovery, the log manager periodically determines the smallest LSN's currently needed for redo and undo, respectively, and truncates the log part containing the operation with the LSN's smaller than the minimum redo and undo LSN's.

3.3 Distributed Transactions

Many real-world applications must be able to perform transactions that involve multiple transactional systems. For instance, there are more cross-institutional money orders than those inside a single bank, and certainly, different banks do not use one central database server. Guaranteeing the transaction atomicity and persistence in a distributed database system is more complex because failures usually affect only one or a few of the participating data servers at once whereas the rest of them remain up and running. A mechanism is needed to prevent a distributed transaction from being committed partially, where some data servers commit while the other ones abort updates of the same transaction. Obviously, the transaction participants have to exchange messages to learn whether they all are able to commit.

A family of **Two-Phase-Commit (2PC)** protocols has been developed as a means to ensure unanimous commit decision among **participants** of distributed transactions. A dedicated process or one of the transaction participants is used as a **coordinator**. When the application finishes updating data on the participants, it sends the commit request to the transaction coordinator. In the first **voting phase**, the coordinator sends a *prepare* message to each participant. When *every* participant has responded with a *yes* message, it initiates the second **commit phase** by sending *commit* messages; otherwise, it sends *abort* messages starting the **abort phase**. The participants report the completion of the second

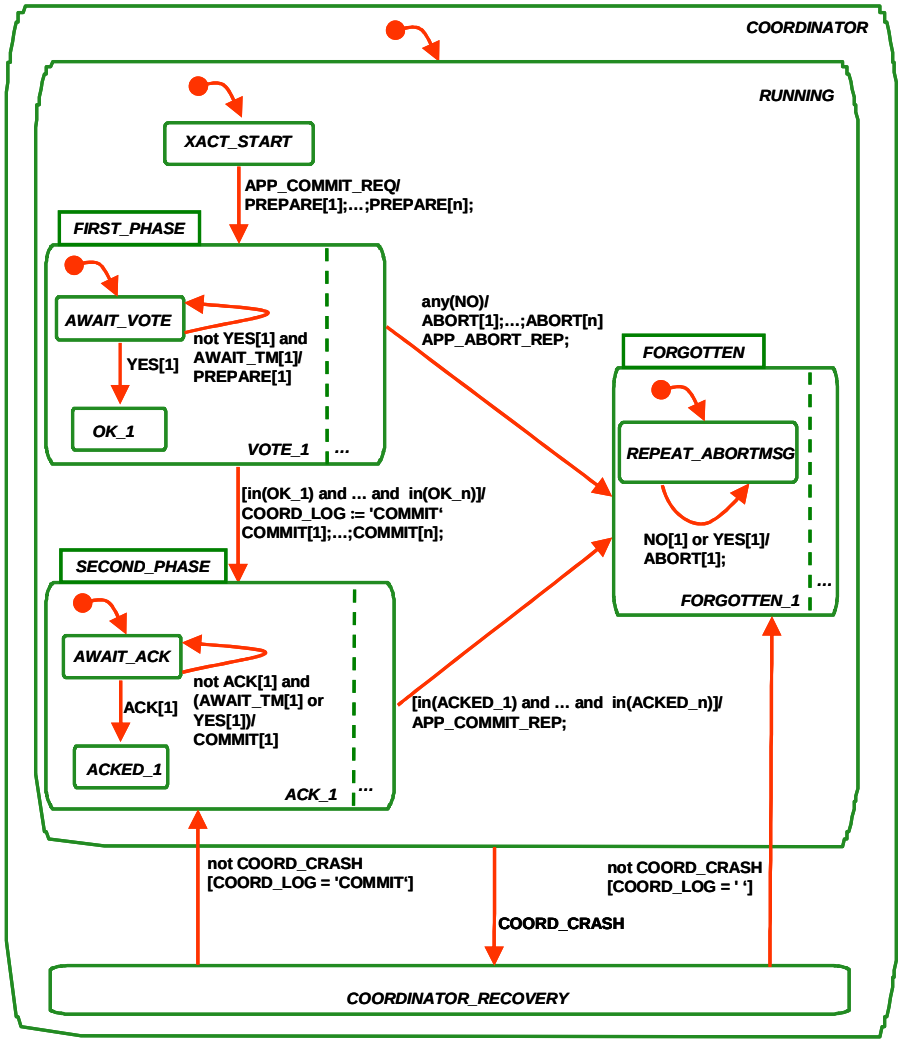

Figure 12: Statechart of the 2PC Coordinator

phase by an *ack* message. A particularly interesting variant of 2PC has been adopted for the industry standard XA [The Open Group 1994].

The statechart specifications of the coordinator and participant behaviors under XA-2PC are provided in Figure 12 and Figure 13, respectively. They are used as two orthogonal components of a single statechart that is left out for a better readability. The coordinator starts the protocol after receiving an *app_commit_req* message from the application, and sends back either *app_commit_rep* or *app_abort_rep* to report the transaction outcome. Protocol messages are modeled by event arrays, where an expression *msg*[*i*] stands for a message sent from the coordinator to the i^{th} participant or vice versa, which becomes clear out of context depending on by what component it is generated. Failures related to the i^{th} participant are modeled analogously. The coordinator log and the participants log are represented by the string variables *coord_log* and *part_log*[*i*], accordingly. A forced write corresponds to an assignment to a log variable. The garbage collector whose effect

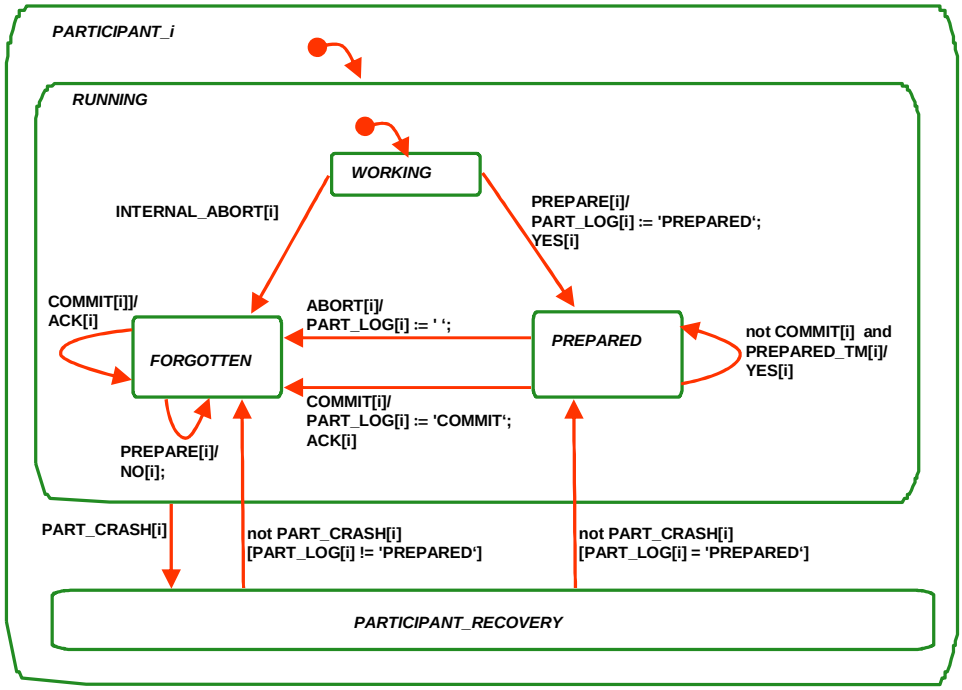

Figure 13: Statechart of the i^{th} 2PC Participant

consists of resetting a log variable is not shown for better readability. However, we do consider the implications of an asynchronous garbage collection, i.e., the situations where only some components truncate their logs whereas the others do not.

The failure model comprises:

- Crashes (occurrences of the external events *coord_crash* and *part_crash*[*i*] standing for coordinator and participant crashes, accordingly),
- Local transaction aborts at the i^{th} participant, e.g., due to concurrency control related issues (the external event *internal_abort*[*i*])
- Timeouts of a message from the i^{th} participant (the external event *await_tm*[*i*]),
- Participant timeouts of the commit decision (the external event *prepared_tm*[*i*]),
- Belated or unexpected receiving of a (possibly duplicated) message due to the periodic resending and crashes.

Prior to sending a *yes* message, the participant forces the log for the first time because it has to be able to redo the transaction when it fails in the prepared state. A crash before this point leads to the transaction abort. A participant cannot recover a prepared transaction autonomously until it learns the global decision from the coordinator, and it is blocked as long as the coordinator is down. To regain the **recovery autonomicity**, the participant forces the log for the second time upon arrival of the decision message, and sends an acknowledgement *ack*[*i*] in the commit case. This terminates the protocol from

the participant perspective, and it does no longer prevent the garbage-collection as indicated by the name of the state *forgotten*.

The coordinator runs the first phase until all *yes* messages have arrived or at least one *no* message (given by the statechart expression *any*(*no*)). The coordinator performs a single forced write only when it is about to broadcast the positive commit decision to the participants. When the coordinator recovers from a crash having occurred during the first phase, it finds no information about the given transaction in the log, and presumes the abort case. This is safe because no participant is committed yet, and a *prepared* participant will eventually receive an *abort* message after resending its positive vote. For this behavior, this protocol is referred as to **presumed-abort 2PC** in the literature. In the commit case, the coordinator is allowed to terminate the protocol by releasing the transaction log entries for the garbage collection only after each participant has acknowledged the commit message. Otherwise, the coordinator would not be able to repeat the commit message for participants who have not received it because of a failure.

The complexity of a failure-free run of an XA-2PC transaction on n participants accumulates to $2n + 1$ forced writes and $4n$ messages. Since the *ack* messages are needed only for the garbage collection at the coordinator site, they can be sent asynchronously, e.g., as a batch, or they can be piggybacked on the vote messages of the next transaction instance. Thus, the communication cost is reduced to $3n$ messages. If one of the participants takes the role of the coordinator, we save one set of messages at the coordinator site, and we save one forced write since we do not need to log transaction commit twice at the coordinator site. Thus, the overhead of a 2PC commit can be further reduced to $2n$ forced writes and $3n-3$ messages.

Another interesting alternative variant of 2PC coined **presumed-commit 2PC** is optimized for a more frequent commit case. When inquired by a participant, the coordinator considers the *garbage-collected* transactions committed, which has the following implications. The coordinator has to be able to distinguish the “forgotten” state from the active voting phase, which is accomplished by forcing the transaction information to the log already at the beginning of the voting phase. The coordinator forces the log to disk once more prior to sending *commit* messages. The participant forces the log on first arrival of a *prepare* message. However, the participant does not have to force the log on arrival of a *commit* message because it can learn from the coordinator about the transaction outcome during recovery when the commit log entry is missing in

its local log. Due to the commit presumption at the coordinator, the participants also do not have to send an *ack* message in response to commit. With one of the *n* participants being in the role of the coordinator, a failure-free presumed-commit 2PC transaction runs with the overhead of $n + 1$ forced writes and $3n - 3$ messages. There are no savings in rare abort cases because rollback log entries have to be forced and explicit *ack* messages are required. Despite the lower complexity of the presumed-commit 2PC in flat database federations, the presumed-abort 2PC protocol has been chosen for the XA standard due to the higher optimization potential in connection with local read-only transactions when used in hierarchical database federations in which a participant may in turn be a federation [Weikum and Vossen 2001.

This consideration motivated the **New Presumed-Commit (NPrC)** 2PC protocol developed by Lampson and Lomet [1993]. In terms of forced writes and messages, NPrC allows the same optimizations for committing read-only transactions as the XA-2PC while having the same cost for committing update transactions as the conventional presumed-commit 2PC. This is achieved at the price of retaining a small amount of non-collectible garbage information in the log after each crash, which is acceptable since crashes do not occur frequently.

3.4 Related Work on Application Recovery

Transaction recovery in database systems guarantees committed data to survive system crashes. However, most components in a multi-tier system including database clients are not transactional and are not provided with fault-tolerance. When the client application crashes and comes up again, the database server will not be able to assist it in finding out whether the crash occurred *before*, *during*, or *after* a certain transaction.

Even if one could disregard client crashes, the database server fails to report the transaction outcome to the clients waiting for a reply to the request submitted prior to or during the server outage. Since the server loses the mapping between network connections and transactions after restart, it normally responds with an error message “Transaction unknown”. When such a reply comes to a request issued before the commit request, the client will correctly conclude that the given transaction has been aborted, and the transaction should be retried. Concerning the final, i.e., the commit request, no clear conclusion is possible because the server may have failed before flushing the commit log entry for the transaction in question (the commit case) or afterwards (the abort case).

This problem may affect many clients simultaneously. For each sequential write, the disk controller has to wait until the position stored in the cursor of the log file rotates under the disk head. Due to this rotational delay, a single sequential write of *n* bytes is faster than multiple sequential writes of the same amount. For this reason, many production servers commit multiple transactions completed within a reasonable time window by a single sequential write [Gray and Reuter 1993]. For instance, with the reference performance of 1 million transactions per minute and the time window of 100 ms, more than 1500 clients may suffer the anomaly described above.

For the true fault-tolerance of multi-tier applications, recovery must be available for each component and cover its **process** state, **data**, and **messages**.

3.4.1 Queued Transactions

Queued transactions have been one of the most successful solutions to this problem. It is supported by virtually all commercial vendors of transactional systems. The basis of this approach is that the client and the server applications communicate through transactional input and output queues, which allows passing messages within a transaction. The client application has to be designed to be stateless. The state information, maintained in the data server, is made available for the client at a specific location or is encoded into server reply messages.

A **transactional queue** *Q* is a persistent structure that consists of a message store and an integer field *lastId* with the *id* the most recent committed message that is/was in the queue. It supports the following opereations:

- *Q.enqueue*(*in msg, in id*) adds a message *msg* to the tail of *Q*. Undoing this operation incurs deleting *msg* from *Q* and restoring the previous value of *lastId*.
- *Q.dequeue*(*out msg, out id*) returns the oldest message and its *id* in the queue *Q*, and subsequently deletes it. Undoing this operation incurs restoring the message in the head of *Q*.
- *Q.getLastId*() returns the current value of *lastId*
- *Q.isEmpty*() returns true when there are no messages in the queue, and false otherwise

```
01.  while(TRUE)
02.  {
03.    while(IQ.isEmpty()) Sleep(pollingPeriod);
04.    BEGIN TRANSACTION;
05.    IQ.dequeue(reqMsg, reqId);
06.    repMsg = executeRequest(reqMsg)
07.    OQ.enqueue(repMsg, reqId);
08.    COMMIT TRANSACTION
09.  }
```

Figure 14: Normal Operation of a Queued Transaction Server

```
01.  user_input_label:
02.  reqMsg = readEndUserInput();
03.  loggedId = readIdFromClientDisk();
04.  id = loggedId + 1;
05.  forceWriteIdToClientDisk(id);
06.  BEGIN TRANSACTION;
07.  IQ.enqueue(reqMsg, id);
08.  COMMIT TRANSACTION;
09.
10.  user_output_label:
11.  while (OQ.isEmpty()) Sleep(pollingPeriod);
12.  BEGIN TRANSACTION;
13.  OQ.dequeue(repMsg, repId);
14.  print(repMsg);
15.  awaitEndUserAcknowledgement();
16.  COMMIT TRANSACTION;
17.
18.  request_recovery_on_any_error:
19.  loggedId = readIdFromClientDisk();
20.  lastInputId = IQ.getLastId();
21.  if (loggedId > lastInputId)
22.  {
23.    print("ERROR: please repeat previous input");
24.    goto user_input_label;
25.  }
26.  lastOutputId = OQ.getLastId();
27.  if (loggedId > lastOutputId)
28.    goto user_output_label;
29.  if (loggedId == lastOutputId && !OQ.isEmpty())
30.    goto user_output_label;
31.  goto user_input_label;
```

Figure 15: Behavior of a Stateless Queued Transaction Client

A client processes some input from the end-user. Instead of directly invoking update routines on the server, the client creates a request message (typically a call to a batch of SQL statements stored on the server), enqueues it into the server input queue *IQ*. The server behavior outlined in Figure 14 is simple: the server periodically checks the input queue for new requests (line 1), extracts the oldest request message, executes it, and enqueue the reply message into the output queue *OQ* (lines 2-6) inside a single transaction. Initially, let us assume the queue objects are stored on the server, and so there

```
01.  while(TRUE) // infinite loop
02.  {
03.    while (OQ.isEmpty()) Sleep(pollingPeriod);
04.    BEGIN TRANSACTION;
05.    OQ.dequeue();
06.    print repMsg;
07.    reqMsg = readEndUserInput();
08.    IQ.enqueue(reqMsg, id);
09.    COMMIT TRANSACTION;
10.  }
```

Figure 16: Normal Operation of a Pseudo-Stateful Queued Transaction Client

are no distributed transactions. If the transaction fails, the server recovery will undo the request execution along with the queue operations. Since the failed request is returned to the head of the input queue during recovery, the request execution will be retried again during normal operation. Hence, a request message that the client has been able to insert into the input queue is executed exactly once without any further client intervention. The client just needs to poll the output queue to pick up the reply when the request execution eventually succeeds.

A more difficult part of this approach is to ensure that the client whose logic is shown in Figure 15 (lines 1-16 for normal operation, lines 18-28 for crash recovery) does not try to insert the same request more than once into the input queue (e.g., when a commit reply is lost). As a fail-stop process, the client executes the recovery code on every failure regardless of the failure type (a server-call-timeout or a real client crash).

In the normal operation mode, the client first prompts the end-user for an input that is used to construct a server request message (line 2). The client reads the id (a sequence number) of the previous request (*loggedId*) from a specific location of the client disk, and force-writes a new id back (lines 3-5). Now, the client is ready to insert the new request into the server input queue within a transaction (lines 6-8). Subsequently, when the server reply arrives in the output queue, the client starts a new transaction (lines 12-16) to obtain the reply, and present it to the end-user. The output transaction is not committed until the end-user acknowledges the output message.

When the client fails before the input transaction is committed, the recovery code will detect that *loggedId* is greater than the last committed request id in the input queue, and it will ask the user to repeat the input (lines 19-25). If the input transaction is committed, there are three cases:

- The server is still executing, detected if *loggedId* is greater than last committed reply id in the output queue (*lastOutputId*) (lines 27-28).
- The server completed the request execution, but the output transaction has not started or it has been aborted if *loggedId* is equal to *lastOutputId* and the output queue is not empty (lines 29-30). Thus, the client may potentially display the output message several times.
- In the remaining case, if *loggedId* is equal to *lastOutputId* and the output queue is empty, the user acknowledged the output. The user prompts the end-user for the next input.

The overhead of such an interaction consist of four forced writes: three transaction commits and a forced write for the request id at the client side.

In a stateful application comprising multiple steps (a **session**), each subsequent request is a function of the preceding server reply and a new end-user input. In this application model, the number of client transactions (and consequently the number of forced writes per request message) can be reduced by combining the output transaction for the preceding reply with the input transaction of the subsequent request. Moreover, the request id logging can be skipped. A new request serves as an implicit acknowledgement of the previous intermediate reply. The client program is called **pseudo-stateful** because it does not maintain its own state although it is used in a stateful application. The server behavior does not change. After a failure, the client simply resumes normal operation (see Figure 16) without any specific actions. Exactly-once execution is guaranteed although some output messages may be displayed multiple times and some input messages may have to be repeated in the course of the application run. The overhead of a queued transaction interaction in the pseudo-stateful setting boils down to two forced writes due to one client-initiated and one server initiated transaction.

Unfortunately, there are two major disadvantages of building applications with queued transactions:

- It requires an extremely inconvenient, unnatural programming style, which makes it unsuitable for state-rich applications such as CAD systems and long-running multiplayer strategy games on the Internet.
- It does not scale with the number of application tiers and components, which is unacceptable for a Web-scale application.

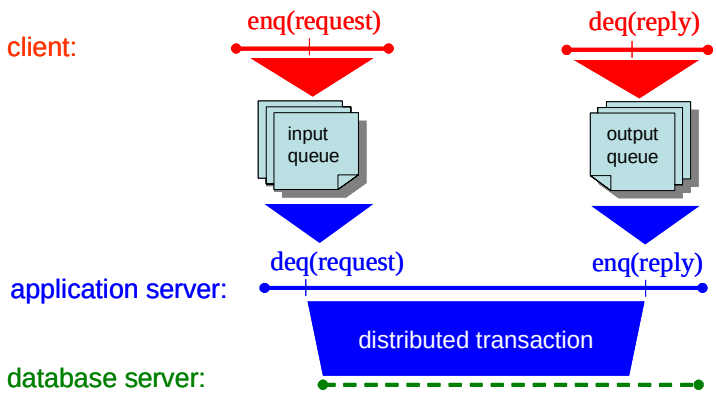

Figure 17: Three-Tier Application with Queued Transactions

The latter can be demonstrated by an example application that uses the popular three-tier-architecture with a client program as a user-frontend, an application server at the middle tier, and a database backend (see Figure 17). No transactional queues are needed between the application server and the database because one can enclose their interactions into a single distributed transaction with the application server acting as both a 2PC coordinator and 2PC participant.

Hence, you see two forced writes for the client output and input transactions, and one forced write of the client request id. The 2PC commit with two participants incurs four forced writes and three additional messages. The overall overhead is too high for a single end-user request.

3.4.2 Stateful Client Server Application

There is some prior work on masking failures and providing recovery for stateful applications, but only in limited contexts. Freytag et al. [1987], Lomet and Weikum [1998], Barga et al. [2000] are focused only on client-server systems, and do not consider multi-tier Internet applications. Other work is restricted to applications embedded in the database server, like stored procedures [Lomet 1998]. It is not obvious how these protocols can be generalized to apply to multi-tier systems. The notion of interaction contracts, developed by Barga et al. [2002], is the key for this generalization.

3.4.3 Fault Tolerance in Web Services and Middleware

Fault tolerance is being discussed also for component middleware like CORBA [OMG 2000] and EJB [Sun 2001], but the focus is on service availability for stateless interactions (i.e., restarting re-initialized application server processes). Products (e.g., VisiBroker, Orbix, BEA WebLogic, or Sun's J2EE suite) at best support simple failover

techniques that do not relieve the application programmer from having to either write failure-handling logic or structure his application as stateless, and are not geared for masking process or message failures to users. More recently, failover techniques for Web servers have been presented in [Luo and Yang 2001], based on application-transparent replication and redirection of HTTP requests.

The need for execution guarantees for Web Services, raised by Tygar [1998], Martin and Ramamritham [1999, 2001], Dutta et al. [2001], Fu et al. [2001], Popovici et al. [2000], Schuldt et al. [2000], has been concerned with specific applications such as payment protocols or mobile data exchange and does not specifically address failure masking in general multi-tier architectures.

Closest to this thesis in terms of objectives is the work by Frølund and Guerraoui [2002] that presents a three-tier protocol for exactly-once transaction execution based on asynchronous message replication and a distributed consensus protocol. However, this work focuses on stateless application servers and does not address the autonomy requirements of components, the optimization of logging, and the need for effective log truncation.

3.4.4 General Process Recovery

Recovery for general systems of communicating processes has been extensively studied in the fault-tolerance community (e.g., Johnson and Zwaenpol [1987], Strom et al. [1988], Cristian [1991], Alvisi and Marzullo [1995], and Elnozahy et al. [2002]), where the main focus has been to avoid losing too much work in long-running computations (e.g., scientific applications), usually using distributed checkpointing. Most of this work does not mask failures. Methods that do mask failures exploit “pessimistic logging” (see, e.g., Huang and Wang [1995]), with forced log I/Os at both sender and receiver on every message exchange. Techniques that are even more expensive, such as process checkpointing (i.e., writing process state to disk) upon every interaction, were used in the fault-tolerant systems of the early eighties [Bartlett 1981, Borg et al. 1989, and Kim 1984]. Thus, failure masking has been considered a luxury affordable only by mission-critical applications (e.g., stock exchanges).

3.5 Related Work on Recovery Verification

Given that state-of-the-art recovery algorithms have evolved to complex, highly optimized procedures, the need of their formalization and verification has been recognized and led to a number of publications.

3.5.1 (Local) Data Recovery

Hadzilacos [1988] developed an abstract formal model of recovery. He stated logging rules in terms of which information *must* be in the log to ensure the redo of the committed and undo of uncommitted transactions in a schedule, and showed how these rules apply to different classes of data managers regarding the usage of the undo/redo paradigms. However, this paper does not verify the behavior of recovery algorithms.

Kuo [1996] used *I/0-Automata* [Lynch 1996] as a formal method for modeling a simplified ARIES algorithm [Mohan et al. 1992], and presented proofs of important invariants for the model that includes checkpoints and redo tests.

There also have been attempts to build formal frameworks to characterize recovery in a more general fashion. However, such models are often too abstract for verifying such a critical property as idempotence. They rather concentrate on showing the atomicity and durability guarantees [Gurevich et al. 1997, Martin and Ramamritham 1997].

Lomet and Tuttle [2003] devised a general recoverability relationship between the log, data, and the recovery managers. The paper shows how the data, installed in the stable storage, affects the operations having to be repeated during the redo recovery. Their *recoverability invariant* can be used for verification of most redo recovery algorithms.

3.5.2 Distributed System Recovery

Chkliaev et al. [2000], used an interactive theorem prover to show the serializability and the durability of distributed transactions when the system deploys the Two-Phase-Locking (2PL) protocol [Weikum and Vossen 2001] for the concurrency control on each site and 2PC as agreement procedure for transaction outcomes. To keep the model verifiable, the authors did not model 2PC messages, and used a finite set of finite schedules.

Close to the formal verification part of the thesis, is the work presented by Younas and Eaglestone [2002]. This paper deals with a 2PC-based protocol that relies on local

compensating transactions to provide the semantic atomicity of Web Transactions. The protocol is formally specified by finite state automata and its key property of at-most-once execution is verified by an automatic theorem prover. The effects of nondeterministic interleaving of concurrent transactions are not considered in this work.

All prior work on recovery verification leaves out important aspects and stays at an abstract level, rather far from the actual implementation.

4. Interaction Contracts Framework

"A good businessman never makes a contract unless he's sure he can carry it through ..."
- Dalton Trumbo

4.1 Computational Model

4.1.1 Components

The framework considers three types of components. Its core is constituted by **persistent components** (**Pcom**) representing any kind of clients, servers or their parts that should provide an exactly-once guarantee for a single request. Furthermore, it includes **transactional components** (**Tcom**) such as database systems that execute a given request at most once because they play a very prominent role in the modern business information systems. Users and systems that provide no guarantees are captured by the notion of **external component** (**Xcom**).

Persistence of Pcom's and Tcom's is implemented by exploiting a *log* and a *recovery manager*. The log is used for capturing nondeterministic events and their order. During normal operation, log entries are initially created in the *log buffer* allocated in the main memory to minimize disk I/O overhead. At certain points, the log entries of the log buffers have to be *forced* to the *stable log* on disk, in order to preserve recoverability. Stable log is used by the recovery manager after a crash for recreating the most recent consistent component state.

4.1.2 Message and Process Recovery Principles

Let us consider a simple application consisting of just two persistent components C_1 and C_2 of Figure 18. The components exchange messages via a network link in a conversational manner. We denote the messages from the component C_1 initiating the conversation **requests** and those from the counterpart component C_2 **replies**. Requests and replies change the state of their corresponding receivers in that they cause data

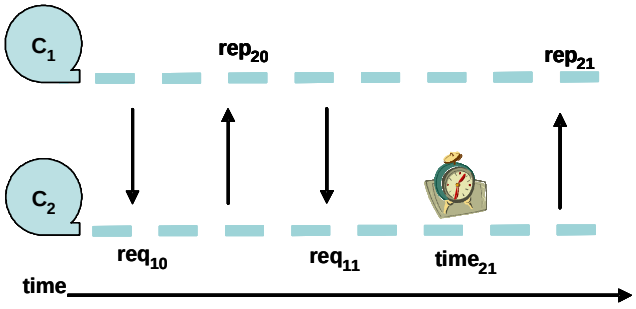

Figure 18: Sample Two-Component System

modifications and have impact on the control flows. The incoming messages are nondeterministic from the receiver perspective. Note that replies to internal system calls are sometimes nondeterministic as well (e.g., replies to calls to timing functions). The outgoing messages result from processing the incoming messages. At any point in time during the conversation, one or both components may crash and the link between them may break down.

Component state recovery: The component has always to provide recoverability of the current state. For this purpose, the component may physically log, i.e., dump a new state to disk after each and every update to it, which is very inefficient since single updates concern only a small portion of the component state. Instead, incremental logical logging of state-relevant nondeterministic events (typically **update requests** from other components) can be used. After a crash, the state is recreated by reading the most recent state dump, called an **installation point,** and replaying subsequent events from the log. Such a replay is deterministic if the component is deterministic up to nondeterministic events out of its control, referred to as **piecewise-determinism**. The component commits its state by either modifying data on local disk or sending to another component a message that already reflects the new state (e.g., a reply to a request). In fact, the former can be seen as a special case of the latter because the file system (or the operation system responsible for it) is simply a different component in the framework view. At any rate prior to sending a message, either a new installation point or the log has to be forced to disk. Otherwise, the system would not be able to resume coherent conversation after a failure.

Output message recreation: The sender may fail before the message transfer is completed. The sender is responsible for message recreation because neither the network layer nor the receiver may have even a chance to capture the message. An output message usually results from the processing of preceding input messages. In contrast to the traditional database recovery, this implies that replies to **read-only requests** have to be captured in the log because the result of a read-only request is in general nondeterministic: the corresponding reply would change through subsequent state updates. During deterministic replay, output messages are recreated and placed into output buffers for delivery.

Message uniqueness: Deterministic replay may lead to a message resend. Therefore, the receiver must be provided with the capability to detect duplicate requests, which is

normally achieved by stamping all messages with a unique *message sequence number*. As in the traditional database recovery, a component that caches update operations in volatile memory must be able to determine whether stable state already includes the effects of the current message in order to skip or to repeat the message execution during recovery.

Input message recreation: We have already shown above that the sender inevitably has to take care of message recreation. This fact makes it possible to defer logging of the message on the receiver component. Only when the receiver in turn is about to send out another message, which reveals its state to the outside world, may forced logging of the incoming message be unavoidable to keep the reply message deterministically replayable. The forced write can be avoided if the receiver component is stateless or when all interacting component pairs work on isolated state portions. As an example, consider a Web-based E-mail system. Each session includes a dedicated Web browser on the client side and a dedicated inbox on the server side. Therefore, there is no interleaving of requests on behalf of different users that would have to be forced to disk for deterministic replay. In the sample application from Figure 18, you may observe that the component C_1 does not need to force log incoming messages rep_{20} and rep_{21} as they will be resent by deterministic replay of C_2 in the original order after C_1 reproduced the message req_{10}. If, for instance, the state of C_1 were also modified by some third incoming message req_{31} between rep_{20} and req_{11} this would be reflected in the state and in subsequent messages req_{11} and rep_{21}. Since the components recover asynchronously after a failure, the original message ordering on C_1 has to be fixed in the stable log once C_1 commits its state.

Periodic message resend: A message can be lost not only because of a sender crash. A receiver crash and the network outage may have an identical effect without any notice provided to the sender. The only way to treat such situations is timeout-based message resend until the receiver will manage to acknowledge the message. Sometimes nothing goes wrong but the receiver executes unexpectedly slowly. Nevertheless, with all messages being unique the receiver is able to eliminate duplicates and remains idempotent.

Autonomous components: In the context of cross-enterprise Web Services, remote components might have poorer connectivity, less satisfactory performance or they may simply enjoy less confidence. In order to keep the local components recovery-independent from the remote ones to the greatest possible extent, the local component must immediately force the log upon arrival of a remote message.

Log truncation: In order to accelerate component restart after a crash, a component should notify its counterparts about the oldest incoming message that has not been captured in the log yet. A counterpart component can consequently determine the oldest output message it would need to replay which allows truncation of the log head up to the most recent installation point prior to the oldest message yet to be recreated.

Resource virtualization: Component crashes in connection with allocated system resources such as file and TCP connection handles conceal another source of nondeterminism. After a crash, the resource handles cannot be reused even if the corresponding structures have not yet been garbage-collected by the operating system because a restarted component appears as a new process to the operating system. Thus, instead of physically logging resource handles, each component resource has to be assigned a virtual id. To this end, file operation log entries should contain the logical file name and each remote operation log entry should include the remote component's virtual id and possibly an explicit session id (e.g., HTTP session cookie) that establishes logical context in which the operation is executed.

4.2 Modeling Issues in Statemate

In this section, we discuss how the key elements of the computational model can be modeled in Statemate. Although Statemate offers several alternatives that are logically equivalent, they may differ in terms of the model and verification complexity.

4.2.1 Stable Log

A stable log is used to recreate the last valid application state as of the time immediately prior to the crash.

Statemate provides an elegant way to remember last active substate configuration of a superstate. When the system enters the superstate through a so-called **history connector** (a circle labeled H), Statemate will activate those substates, which were active as the superstate was exited last time instead of taking the default transitions. Hence, we can simply put states relevant for logging in such a history-connected superstate. Whenever a force-logged operation needs to be executed, the system has to take two transitions consecutively: it first takes a direct transition to the corresponding substate of the log-enabled superstate (i.e., not through a history connector) according to the write-ahead logging rule [Gray and Reuter 1993, Weikum and Vossen 2001], and only after that it takes a transition with the action representing the actual operation. Not that the latter

transition should lead out of the log-enabled superstate, if it does not require force-logging. After a "crash", the system always starts with entering the log-enabled superstate through a history connector. Unfortunately, due to yet immature support of history-connected states in the Statemate verification software, we had to drop this option in the IC specifications.

Instead of implicitly logging by means of history-connected superstates, we introduce a local variable suffixed with "*last_logged*" for each instance of the IC parts running within a component. Whenever we have to force-log the current IC status, we explicitly store the name of the currently active states into the corresponding *"last_logged"* variables. Only a few of states in an IC instance have to be memorized. After a crash, the component first goes into the recovery states of all IC instances it has run. A recovery state is connected to the rest of the chart by transitions each of which handles one particular of all possible "last logged" values. This solution does not use advanced Statemate features, and has not posed any difficulties to Statemate verification tools. In addition, this method allows one to see immediately the global component configuration currently logged by observing all "*last_logged*" variables managed on the component.

4.2.2 Messages and Communication Failures

We model messages as follows: when a sender wants to send a message during the i^{th} step, it does so by generating a corresponding event. According to the Statemate semantics this event becomes visible and can be consumed by a receiver in the $i+1^{st}$ step.

Communication failures of any kind (TCP/IP stack crashes, network interface card or router failures, etc) that lead to a message loss are captured in an external event *link_outage*, where the term link refers to a logical connection between a pair of components rather than to their physical or TCP/IP connections. Transitions of a receiver reacting on a message *m* use a compound event *m_ok* abbreviating $m \wedge \neg link\_outage$. Thus, in the specification a message generated in the i^{th} step is lost due to a communication failure if the corresponding link outage event was sensed between the i^{th} and $i+1^{st}$ step.

From the receiver's perspective, a message loss is not immediately distinguishable from a sender's crash. However, for the sender the former just requires simply resending the message whereas the latter means a complete message and state recovery run through deterministic log replay. A deterministic replay usually takes longer than a transient

network failure, and incurs combinations of component configurations that are more complicated. Because of this fundamental difference, we decided to model communication failures explicitly.

The IC framework requires that messages are unique, in order to detect duplicates. In practice, components would use message sequence numbers to tag messages. In the specifications, message uniqueness is achieved in consequence of that we model ICs as generic activities, such that each individual message in a specification for a concrete application is local to the corresponding instance. This makes the use of additional ids unnecessary. While this modeling technique is equivalent to one using ids, it reduces the verification run time. Although integers do not pose conceptual difficulties from the modeling perspective (they have to be finite to preserve the model finiteness though), they significantly increase verification costs. An event (i.e., a Boolean one-bit variable) clearly adds less complexity to an OBDD than an *n*-bit integer.

4.2.3 Component Crashes

Component crashes are modeled similar to link outages by external events supplied by the runtime environment. All IC statecharts are arranged to terminate interacting components immediately after they are hit by a crash. This is done by enclosing IC's orthogonal components in a superstate with a transition to a termination connector. It fires whenever the corresponding crash event is sensed, and according to the Statemate semantics, it suppresses all enabled transitions inside the superstate. We also model process monitors that are responsible for restarting activities of crashed components. In the real world, process monitors may be part of the operating system or they may run as dedicated heartbeat-checking processes.

4.2.4 Timeouts and Execution Time

System failures often result in message losses: messages are either never generated or they never reach their destinations. To cope with such situations, some interaction contracts require that the sender resends the message periodically until it gets an acknowledgement from the receiver. Clearly, in a concrete application system administrators or developers will have to define appropriate timeout values for each component interaction: when the acknowledgement does not arrive in the given time window, the message will be resent.

Another important aspect of a model for a distributed multi-user environment is that request timeouts on a sender component may occur not only due to failures. Rather, the capacity of the link between the sender and the receiver might be saturated, and thus, messages are delivered too slowly or dropped by a router. On the other hand, the receiver may suffer a load peak and is not able to respond within the expected time window. These issues are the reason par excellence for non-idempotent behavior of Web Services, where the user blindly repeats the request after a timeout. In the formal specifications, we address these issues by having the receiver react on messages originating from other components after some random delay, which we refer to as "message execution time", and which subsumes processing (due to concurrency) as well as delivery (due to network latency) delays.

For these purposes Statemate supports special timeout events of the form *tm*(*e, d*). This event is generated *d* time units after the most recent occurrence of the event *e*. In the synchronous time model, one time unit matches a single step. The way in which the time is advanced in the asynchronous model is under explicit control by the user or execution environment, and can be defined for each activity separately. Simplicity of the timeout semantics is thus another argument for using the synchronous time model for verification.

When the event *e* is generated again earlier than the corresponding timeout event, the timer for the timeout event will be restarted. It is, however, impossible to cancel the timeout event. That is why it is important to pay attention, when modeling crashes, that the system is not confused by timeout events whose timers has been started in previous component incarnations. To avoid such situations, we design statecharts such that on all execution paths starting in the recovery state the original event is used before its timeout counterparts.

In order to keep the specification general, we use external integer variables as timeout values with the range [0...30]. They are read from the execution environment once upon a component (re)start and remain unchanged up to the next crash/restart. The model checker systematically enumerates system runs with all possible timeout values. We could work with this model at the level of every single interaction contract. Unfortunately, the attempts to verify a concrete system with multiple IC instances at a higher level have not terminated even after a week of uninterrupted computations.

Therefore, for application-level verification we replace timeout events by simple atomic external events. Clearly, external events are nondeterministic and in general do not occur periodically in contrast to timeout events. Nevertheless the system runs with periodic generation of such external events is a proper subset of the runs the model checker will have to consider. Thus, a CTL formula we were able to verify in the adapted specification, would be also valid for the original specification. With new specification, we reduced verification time due to a reduced state space from virtually infinity (i.e., more than one week) to one to one and a half hours.

Timeout events in the specifications are suffixed "*_tm*". When we talk about timeouts in the context of an IC specification, we mean compound events referring to timeout expressions. In the context of application-level specification timeouts are nondeterministic external events.

4.3 Statemate Specifications and Verification

4.3.1 Common Design of the IC Specifications

All interaction contracts, i.e., (I)CIC, TIC, XIC input and XIC output, are represented by generic activities. Each generic IC activity x_AC follows an identical pattern. An IC consists of the corresponding control activity x_SC which orthogonally monitors all processes involved in the IC. The control activity starts these processes and restarts them after they have crashed. As an example consider *cic_sc*, the control activity of CIC, whose specification is depicted in Figure 19. While it is running, it checks whether the sender or receiver activity (their behavior is defined in charts *cic_receiver_sc* and *cic_sender_sc* accordingly) has to be launched. This is the case when the execution environment does not supply crash events any longer. Checking the condition "not ac(…)" (i.e., "not active") for IC activities prevents the system from re-launching the

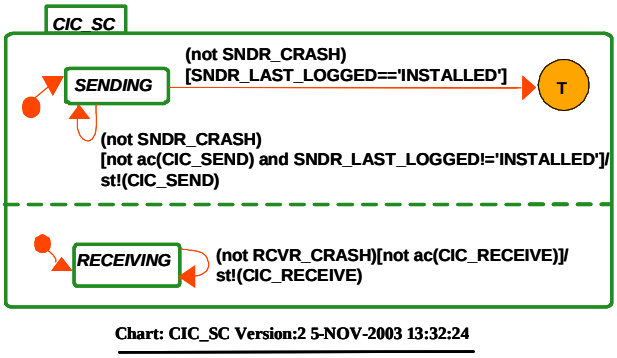

Figure 19: CIC Heartbeat Checker

listening thread during normal operation. Note that this is orthogonal to elimination of message duplicates. After an IC has been installed, all involved components have enough information to recover autonomously, such that the IC may terminate as indicated by entering a termination connector, which stops the given activity along with all its sub-activities. Checking just the sender log for the CIC installation status suffices because the sender's installation always come after that of the receiver as you will see later.

We do not show activitycharts describing the data flow between particular activities because they are straightforward. Instead, we explain these details while describing statecharts where events, conditions, and data are immediately used or generated.

4.3.2 Common IC Properties

As ICs are used in piecewise deterministic components, IC specifications have to be deterministic up to events external to the components involved. As part of the debugging process, we have verified for each specification that it does not contain any nondeterminism, i.e., in a non-orthogonal state there is at most one enabled transition per step.We also have verified that no specifications contain unreachable states. An unreachable state *s* is characterized by the CTL formula $AG\ \neg in(s)$. This is to say that every state makes sense and is used in some situations.

4.3.3 Committed and Immediately Committed IC

According to Barga et al. [2002] a **committed interaction contract (CIC)** between two persistent components (a sender and a receiver) consists of the obligations given in Table 1. An **immediately committed interaction contract (ICIC)** is a committed interaction where sender is released from both message persistence requirements, S2a and S2b when receiver notifies sender (usually via another message) that the message-received state has been installed, without previously notifying sender that its state is stable. Receiver's announcement thus makes the interaction both stable and installed simultaneously [Barga et al. 2002].

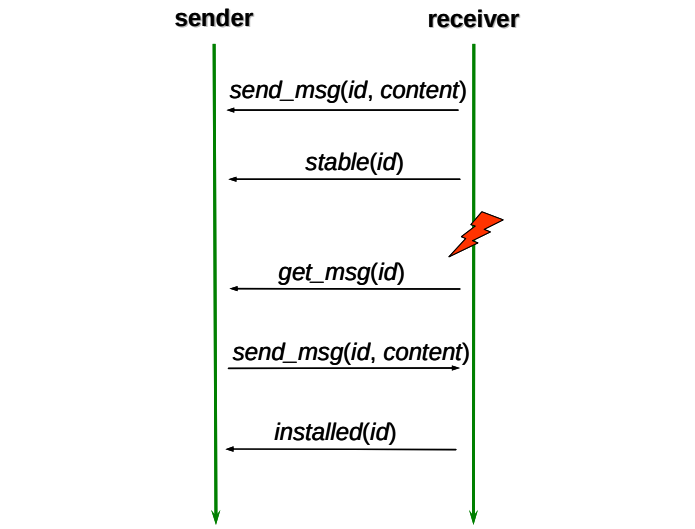

Figure 20: A Message Sequence Diagram of the CIC

Figure 20 shows a message sequence diagram that summarizes the messages exchanged between the sender and the receiver under the CIC in a single interaction. Figure 21 depicts the statecharts that describe the underlying generic sender's and receiver's logics under the CIC and ICIC, whereas the latter is treated as a particular case of the former.

Table 1. CIC Sender and Receiver Obligations

Sender Obligations	Receiver Obligations
S1: Persistent State. Sender promises that its state at the time of the message send or later is persistent. **S2: Persistent Message.** **S2a:** Sender promises to send the message repeatedly (driven by timeouts) until receiver releases it (perhaps implicitly) from this obligation. **S2b:** Sender promises to resend the message upon explicit receiver request until the receiver releases it from this obligation. This is distinct from S2a, typically longer lasting and usually more explicit. **S3: Unique Messages.** Sender promises that its messages have unique contents (including all header information such as timestamps, HTTP cookies, etc.).	**R1: Duplicate Message Elimination.** Receiver promises to eliminate duplicate messages (which sender may send to satisfy S2a). **R2: Persistent State.** **R2a:** Receiver promises that before releasing sender obligation S2a, its state at the time of message receive or later is persistent without the sender periodically re-sending. After S2a release, receiver must explicitly request the message from sender should it be needed. The interaction is **stable**, i.e., it persists (via recovery if needed) with the same state transition as originally. **R2b:** Receiver promises that before releasing the sender from obligation S2b, its state at the time of the message receive or later is persistent without the need to request the message from the sender. After S2b release, the interaction is **installed**, i.e., replay of the interaction is no longer needed.

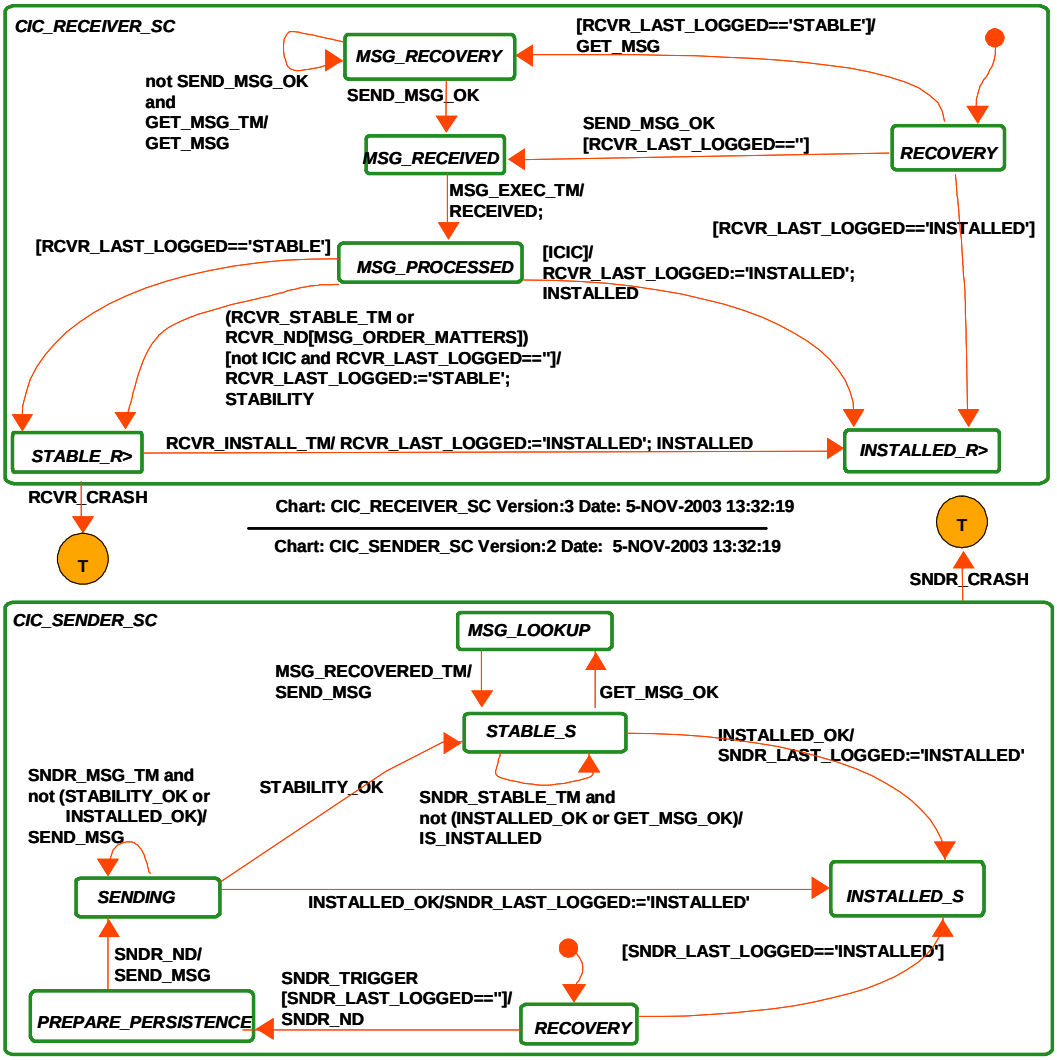

Figure 21: CIC Sender and Receiver

As you may observe in the statechart *cic_sender_sc* the sender starts up in the state *recovery*. Upon (re)start the sender first checks the CIC status in the stable log. When the log turns out to be empty, the sender waits for the event *sndr_trigger*. This event is defined as a formal input parameter. It is supposed to be used as an interface to the application logic of a concrete system specification. It allows me to define in response to which other event the given IC instance is called. Imagine that an instance of *cic_sender_sc* is used to model the HTTP request sending routine of the Web browser. The Web browser is an interactive component and it initiates a request on behalf of its user clicking on a link or typing a new URL. In the browser specification, we would bind the formal parameter *sndr_trigger* to the event *link_clicked* sensed by the browser.

When *sndr_trigger* is finally fired, the sender must prepare its persistence, i.e., make its receiving activities stable, such that the ordering of incoming messages is guaranteed to persist. This will become clearer, when we explain the receiver's behavior. The internal

event *sndr_nd* (a formal output parameter) is generated, while the system is moving to the state *prepare_persistence*. In the next step, the sender sends the message as indicated by generating the event *send_msg*. In the very same step, the receiving threads on the sender have become stable as a reaction on *sndr_nd*. The sender is now in the state *sending*, where it periodically resends the message based on the timeout *sndr_msg_tm* until it receives either *stability* or *installed* notification.

The fact that the sender gets an *installed* notification before a stability notification may indicate that the receiver deploys the immediate variant of CIC or the stability notification has been lost due to previous network failures or crashes. In case of the installed notification, the sender force-logs this by modifying the variable *sndr_last_logged* and moves to the state *installed*, where the CIC actually terminates. In case of the stability notification the sender advances to the state *stable_s*, and stops periodical sending attempts.

When the receiver runs without any problems, it will eventually install the current CIC instance and send the corresponding notification. However, if this final notification gets lost, the sender will need to inquire (*is_installed*) the receiver about the CIC outcome after a relatively long timeout (*sndr_stable_tm*). If the receiver does not find the corresponding CIC id in the list of active interactions, it will respond with an *installed* notification again.

In addition, the receiver may crash, while the sender is in the state *stable_s*. Upon restart, the receiver will inquire the sender about the message content by firing the event *get_msg*. The sender moves to the state *msg_lookup*. After a random amount of time needed for the message recovery (*msg_recovered_tm*) the sender is able to resend the message and moves to the state *stable_s* again. The whole procedure described in this chart is repeated after every crash and restart until the CIC is finally installed.

The statechart *cic_receiver_sc* defines receiver's behavior under CIC. After a (re)start the receiver enters the state *recovery*. When the log is empty, the receiver just waits for the sender to (re)send the message. When this happens, the messages is placed into the receive queue and waits for being processed. The receiver moves to the state *msg_received*. After a random *msg_exec_tm* the message is processed and the system fires the internal output *msg_received* which is an interface to application logic and is normally coupled to some *sndr_trigger* event.

When the receiver uses the immediate CIC variant (i.e., the condition *icic* is true), it will force-log CIC installation and notify the sender by the event *installed* in the same step, in which the Pcom's logic would also react on the message. The receiver advances to the state *installed_r*>. The sign "greater than" in its name indicates that this state has static reactions. While in the state *installed_r* the receiver simply responds to repeated messages or inquiries *is_installed* by anew generating the event *installed.*

With the normal CIC, however, initially nothing happens in the state *msg_processed*. Force-logging that make the current CIC instance stable will be performed: either a) when CIC's instability exceeds (*rcvr_stable_tm*) specified by Pcom's system administrator in order to prevent unnecessary resending of the message (the log is periodically flushed when Pcom is idle) or b) as a reaction on the event *rcvr_nd*. Most importantly *recvr_nd* is coupled together with *pcom_nd* events of all XIC instances, *pcom_nd* output of all TIC client instances (see Section 4.3.4) and *sndr_nd* output of all CIC sender instances running on the same Pcom (i.e., all these formal parameters are bound to the same global event, e.g., *websrvr_nd*). Thus, when the Pcom reveals its state to the outside world by sending a message or by committing a transaction after processing the current CIC message, the message is logged in the proper order and is available for deterministic replay of the Pcom if necessary.

Only a small log entry with the message header (i.e., sender and message ids along with the timestamp) is added to the stable log, when CIC is being made stable. With a full message copy in the stable log, CIC would become installed because the receiver would no longer need the sender to recover.

When the receiver is becoming stable, it writes the corresponding value to the variable the *rcvr_last_logged*, generates the event *stability* and moves to the state *stable_r*. As long as the receiver stays in this state, it responds to periodic send attempts with the stability notification (defined as a static reaction) as the notification might have been lost. During the normal operation, the receiver will remain uninstalled for the user-defined amount of time *rcvr_install_tm*. No other factors have impact on installation because the CIC is already recoverable.

Supposedly, the receiver fails while being stable. Then after restart in the state *recovery*, it finds out that the stably logged status (*rcvr_last_logged*) of the given CIC instance is '*stable*'. Thus, the CIC receiver knows that it has to replay the message, but it does not

have its content though. Thus, the receiver generates the event *get_msg* and moves to the state *msg_recovery*, where it awaits arrival of the message. Message inquiries are repeated based on the user-defined timeout *get_msg_tm*. When the message arrives, the receiver moves to the state *msg_received* where the message is processed (replayed) as originally. The difference arises in the state *msg_processed*: stability of the current CIC instance is not logged once again, and the receiver moves with no action to the state *stable_r*.

We summarize the recovery procedure for the CIC receiver. There are three possible cases:

1. The log is empty: this implies that the receiver is dealing with a completely new message or the message has not left any traces (i.e., neither it has generated any output to Xcoms nor committed a transaction nor stored anything in a local file.
2. The log indicates CIC stability: no local changes have survived the most recent crash but some other components might be aware of the previous message execution because they have received resulting output messages. Thus the message is re-obtained from the sender in order to be deterministically replayed which results in identical state and triggers identical output to the outside world as originally.
3. The log indicates CIC installation: this implies that the receiver can be recovered without the sender because the message content is either part of the stable log or it is reflected in a more recent installation point.

Table 2 summarizes some interesting safety and liveness properties of CIC sender and receiver. Initially, we consider only a simple application specification consisting only of one interacting pair of Pcoms. Thus, we assume that the message being sent is a result of internal computations and is not caused by any external event, i.e., the event *sndr_trigger* is *always* on (i.e., *true*), such that the sender always tries to send a message. The formulae

Table 2. Verified Properties of CIC

Nr	CTL Formula	Res.
F1	AG(¬*sc*) → AG(*rll*=’’ → AF$_{<30}$(*send_msg*))	True
F2	AG(*sll*=’*i*’ → AG (*rll* = ‘*i*’ ∧ ¬*get_msg*))	True
F3	AG(*rll*=’' ⩒ *rll*=’s’ ⩒ *rll*=’i’	True
F4	AG((wr(*rll*) ∧ *rll*=’*s*’) → AG(¬(*rll*=’’))) ∧ AG((wr(*rll*) ∧ *rll*=’*i*’) → AG(¬(*rll*=’’ ⩒ *rll*=’*s*’)))	True
F5	AG((wr(*rll*) ∧ *rll*=*l*) → AX AG (¬(wr(*rll*) ∧ *rll*=*l*)))	True
F6	AF$_{<500}$(AG¬(*sc* ∨ *rc* ∨ *lo*)) → AF$_{<700}$(in(*installed_r*) ∧ in(*installed_s*))	True

presented in this section are valid for both normal and immediate CIC instances unless explicitly stated otherwise.

The sender obligation of persistent state S1 from the CIC definition is outside the specification, when only one protocol instance is concerned. However, it will play a role, when we will verify a complex Web service specification with multiple different protocol instances at the application level (see Section 3). The sender obligation S3 requiring message uniqueness is provided in the specification without any special measures as we have already discussed in Section 4.2.2 above.

As for the sender obligation S2a, we have verified that as long as the sender is running and has not obtained stability or installation notification, it periodically resends the message at least after every 30^{th} step (formula F1 with maximum timeout 30). We abbreviate the event *sndr_crash* by *sc*; *rcvr_last_logged* is abbreviated by *rll*.

In connection to the CIC specification we present here, we reformulate the sender obligation S2b as follows. On all execution paths is true that when the sender is installed, then so does the receiver and no message inquiries occur anymore (formula F2). We abbreviate *sndr_last_logged* by *sll* and '*installed*' is abbreviated by '*i*'.

Formula F3 shows that the CIC receiver log may assume only the following values: '', '*stable*' ('*s*') and '*installed*' ('*i*'). By F4 we show that logging occurs exactly in the given order except that ICIC skips stability, and F5 proves that each log entry *l* out of {'*s*', '*i*'} is created exactly-once given the fact that this happens.

The modality of the form $F_{<n}$ accepted by the Statemate model checker means "eventually after at most *n* steps". F6 demonstrates liveness of the (I)CIC specification: stating that when errors do not occur anymore after at most 500 steps, the sender and the receiver will both install the given (I)CIC instance. In this formula and elsewhere *rc* stands for *rcvr_crash* and *lo* for *link_outage*. Interestingly, however, that the original paper does not say anything, how to make the stable sender terminate, if the final installation notification from the receiver has been lost. This question led to introduction of the new event *is_installed*, by which the sender can inquire the receiver about whether it finally has installed the given interaction. In fact, F6 could not be proved without this event.

4.3.4 External IC

Barga et al [2002] define an **external interaction contract (XIC)** as a contract between a persistent component that subscribes to the rules for an immediately committed interaction, and an external component, which does not. The impacts on external sender or receiver (or users) of Pcom interactions with it are described below. Note that these are impacts on, not obligations of, the external component.

X1: Output Message Send. A Pcom (usually a client machine) sends (displays) a message to an Xcom (e.g., external user), after having logged the message send. The sender Pcom crashes before knowing whether the message was seen. Hence it must re-send the message. Because an Xcom might not eliminate duplicates, a user may see a duplicate message.

X2: Input Message Receive. An external user (Xcom) sends a message, via keyboard, mouse, or other input device, to a (client) Pcom. The Pcom crashes before logging the message. On restart, the user must re-send the message. But the user (an Xcom) has not promised to re-send the message automatically, but rather makes only a "best effort" at this. Moreover, the failure is not masked.

Figure 22 depicts statecharts defining Pcom's behavior during external interactions under XIC. The statechart *xic_sender_sc* shows external output processing on a Pcom. When this activity is started, the system will enter the *recovery* state and check the log (i.e.,

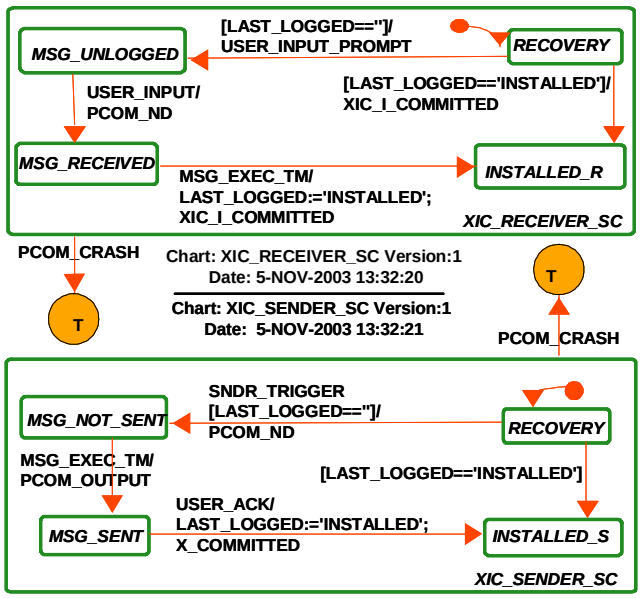

Figure 22: XIC Input and Output

current value of the local (internal) variable *last_logged*). When the log is empty, the system waits for the event *sndr_trigger* to be generated or replayed.

After the *sndr_trigger* has been generated, the output message is moved to the output queue, where it waits for being processed. The system advances to the state *msg_not_sent*. In the very same step, the system generates the event *pcom_nd* (Pcom nondeterminism). This is a formal output parameter of the generic XIC output activity, which signals to the receiver parts of CIC protocol instances running on the same Pcom to prepare persistence of the current Pcom state. This is explained in full detail in Section 4.3.3. When the timeout event *msg_exec_tm* is generated after a random amount of time, the Pcom is ready to display the current message. The system generates the internal event *pcom_output* and moves to the state *msg_sent*, where it waits for the end-user somehow to acknowledge reception of the new output e.g., by editing input elements on the new HTML page in her Web browser. Such kind of events is bound to the formal parameter *user_ack* in a concrete setting. When the Xcom supplies the event *user_ack*, the system will force-log the XIC installation and in the very same step the system moves to the *installed* state. This procedure is repeated upon Pcom restart as long as the external event *pcom_crash* is generated before the system enters the *installed* state for the first time. Otherwise the system always takes the direct transition from the state *recovery* to the state *installed* after a restart.

In the statechart *xic_receiver_sc* you may observe how a Pcom processes input messages originating from an Xcom. Upon a (re)start this activity first lookups its last valid state while in the state *recovery*. When no logging information has been found the system enters the state *msg_unlogged*, and the user is prompted to provide an input as indicated by generating the internal event *user_input_prompt*. After the Xcom has finished inputting data, the execution environment (e.g., the operating system) supplies the external event *user_input*, and input data is placed into the Pcom's input buffer; the internal output event *pcom_nd* is fired. When the Pcom is ready to consume the data after a random amount of time (i.e., when the external event *msg_exec_tm* is generated), it will immediately install the given XIC instance by changing the value of *last_logged*. Should the Pcom fail (i.e., when the external event *pcom_crash* occurs) prior to installing the message but after the Xcom has provided the input, the Xcom will have to repeat the input after Pcom's restart.

Table 3 summarizes interesting properties of XIC in- and output charts that we have tried to verify with the Statemate model checker. For both charts, *xic_sender_sc* and *xic_receiver_sc*, we have the model checker verify that the variable *last_logged* is always either empty or equal to '*installed*' abbreviated '*i*' (F7). Then we want to make sure that each XIC instance is installed at-most-once. This is again identical for both input and output charts. For this purpose, we can use the internal event *wr*(*variable*) that is generated when variable's value is changed (i.e., it is assigned a new value). F8 states that on all execution paths on which *last_logged* is set to '*installed*' *last_logged* is never written again.

We also verify for *xic_output_sc* that as long as the Pcom does not get Xcom's acknowledgement Pcom will keep trying to deliver output after crashes. We actually want to show the formula F9 (*pcom_crash* and *pcom_output* are abbreviated by *pc* and *po* respectively). However, the Statemate model checker accepts only a concrete modality of the form $F_{<n}$. In order to be able to determine when the system achieves some particular progress, we need to know, over which period the system will have to deal with failures. Since we model Heisenbugs, this period must be a finite one. Thus, we pick a reasonably big number (e.g., 500 steps). Note that in a failure-free execution an XIC output completes in maximum 34 steps (i.e., 4 transitions plus maximum message execution time of 30). F10 has been verified by the model checker.

In the same manner, we verify the following liveness property for both, XIC in- and output. When failures no longer occur after 500 steps at latest, the XIC will be finally installed after 600 steps. This is verified with the formula F11.

For an XIC input, we have been interested to make sure that once an input message is captured by the system the user is never asked for this input again as stated in F12. We abbreviate *user_input_prompt* by *uip*.

Table 3. Verified properties of XIC Input/0utput

Nr	CTL Formula	Res.
F7	AG(*ll*='' ⩔ *ll*='*i*')	True
F8	AG(wr(*ll*) ∧ *ll*='*i*' → AX AG(¬wr(*ll*)))	True
F9	AG((¬(*ll*='*i*') ∧ *pc* → AF(*po*))	N/A
F10	$AF_{<500}$ AG(¬*pc*) →AG((¬(*ll*='i') ∧ *pc* → $AF_{<600}$(*po*))	True
F11	$AF_{<500}$ AG(¬*pc*) → $AF_{<600}$ in(*installed*)	True
F12	AG(*ll*='*i*' → AG(¬*uip*))	True
F13	AG(*po* → AX AG(¬*po*))	False
F14	AG(*uip* → AX AG(¬*uip*))	False

As explained by impacts X1 and X2 in the XIC definition above, failures during external interactions are not masked. The Xcom might receive the output message more than once (F13 is *false*). With F14 being *false* it is shown that the Xcom may be asked to provide the same input more than once.

4.3.5 Transactional IC

A **transactional interaction contract (TIC)** between a Pcom client and a Tcom server consists of the obligations given in Table 4. A message sequence diagram summarizing the messages exchanged between the interacting components under the TIC is shown in Figure 23. Figure 24 shows the statecharts which define behavior of a Pcom (*xact_client_sc*) executing a transaction on a Tcom (*xact_server_sc*) under the TIC

Table 4. TIC: Pcom (Client) and Tcom (Server) Obligations

Pcom Obligations	Tcom Obligations
PS1: Persistent Reply-Expected State. The Pcom's state as of the time at which the reply to the commit request is expected, or later, must persist without having to contact the Tcom to repeat its earlier sent messages. **PS2: Persistent commit request message.** The Pcom's commit request message must persist and be resent, driven by timeouts, until the Pcom receives the Tcom's reply message. **PS3: Unique message.** The Pcom promises that its commit request message has unique contents (including all header information such as timestamps, etc.). **PR1: Duplicate Message Elimination.** The Pcom promises to eliminate duplicate reply messages to its commit request message (which the Tcom may send as a result of Tcom receiving multiple duplicate commit request messages sent by Pcom because of PS2). **PR2: Persistent Reply Installed State:** The Pcom promises that, before releasing Tcom from its obligation under TS1, its state at the time of the Tcom commit reply message receive or later is persistent without the need to request the reply message again from the Tcom.	**TR1: Duplicate elimination.** Tcom promises to eliminate duplicate commit request messages (which Pcom may send to satisfy PS2). It treats duplicate copies of the message as requests to resend the reply message. **TR2: Atomic, isolated, and persistent state transition.** The Tcom promises that before releasing Pcom from its obligations under PS2 by sending a reply message, that it has proceeded to one of two possible states, either committing or aborting the transaction (or not executing it at all, equivalent to aborting), and that the resulting state is persistent. **TS1: Persistent (faithful) reply message.** Once the transaction terminates, the Tcom replies acknowledging the commit request, and guarantees persistence of this reply until released from this guarantee by the Pcom. The Tcom promises to resend the message upon explicit Pcom request, as indicated in TR1 above. The Tcom reply message identifies the transaction named in the commit request message and faithfully reports whether it is committed or aborted. **TS2: Unique message.** The Tcom promises that its commit reply message has unique contents (including all header information such as timestamps, etc.)

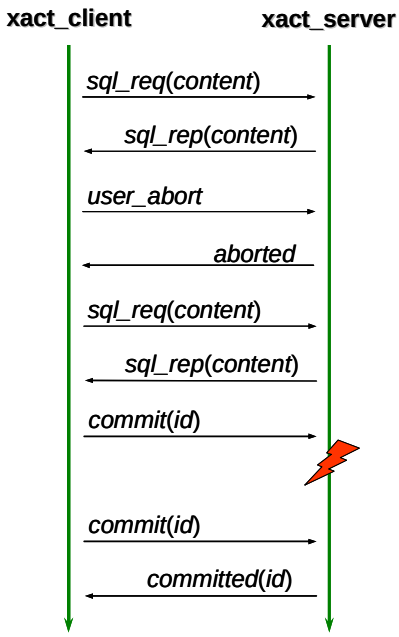

Figure 23: A Message Sequence Diagram of the Transactional Client (Pcom) and Server (Tcom)

protocol.

The client tries to execute a transaction comprising a number of sql statements (input parameter *sql_nr*). A new statement is issued after getting a reply to the previous one. The client (re)starts in the state *recovery*. When its log (*clnt_last_logged*) is empty, it knows that the given transaction has not been committed. After the external event *clnt_trigger* (interface to the application logic) is fired (perhaps replayed) the client advances to the superstate *sql_processing* where it immediately enters the default basic state *xact_start*, and sends the transaction begin event *begin_xact* to the server.

Note that each transaction execution attempt can be aborted by the server anytime for whatever reason. Thus, the client is always prepared to receive the corresponding notification (*aborted*) from the server. In such a case, the client leaves the superstate and enters the state *recovery* again. The same has to be done, when the server does not acknowledge the new transaction by generating the event *begun* for more than the user-defined timeout *begin_tm.*

When the new transaction is started, the client sends the first sql statement i.e., it generates the first element of the event array *sql_req* and advances to the state *querying_updating*. When the client application logic discovers some inconsistency (signaled to the system by event *rollback*) in one of the server replies that are modeled here by the event array *sql_rep*, it will send the event *user_abort* to the server and move to the state *aborting*. The event *user_abort* is periodically generated based on the timeout

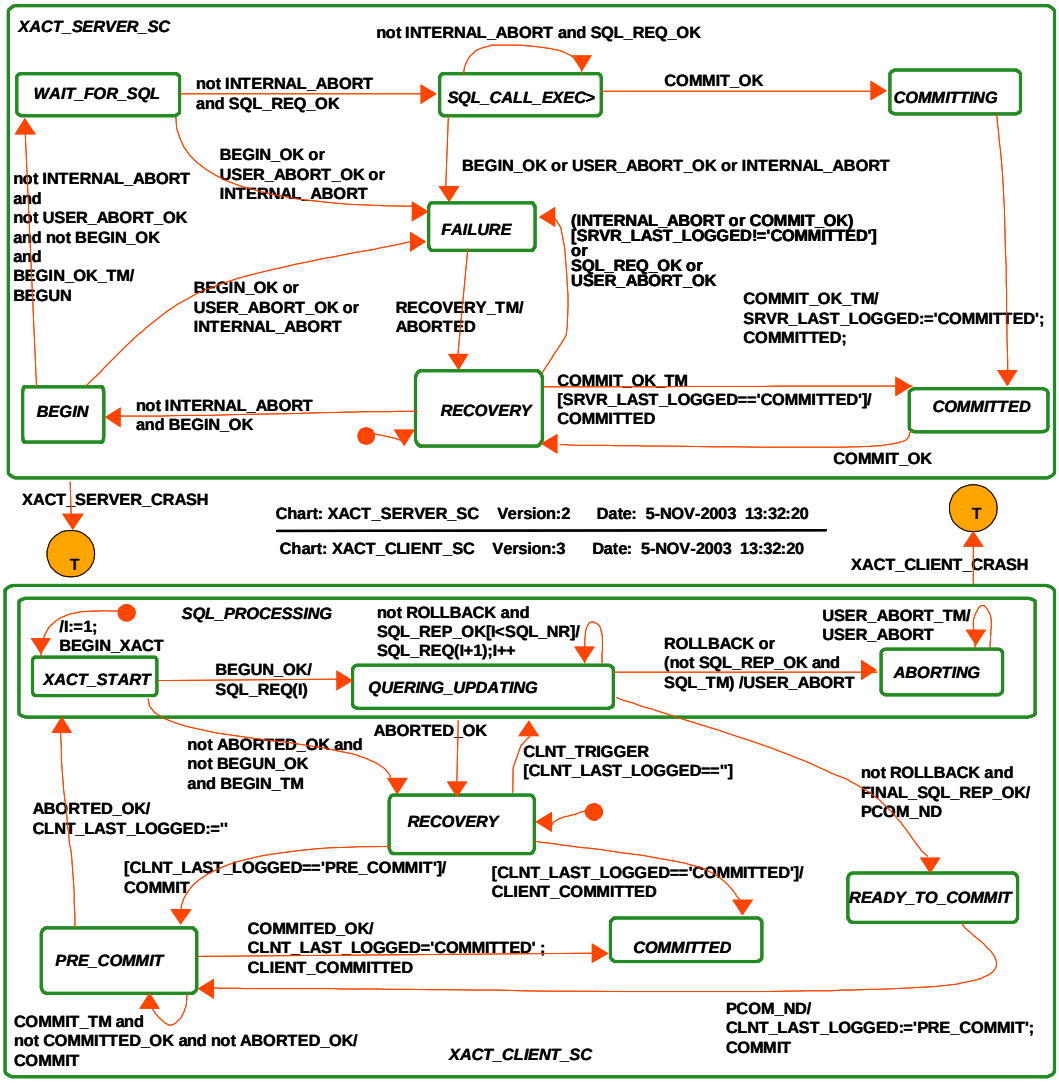

Figure 24: TIC Pcom and Tcom

user_abort_tm until the server does not confirm abortion of this transaction by the event *aborted*. In a normal failure-free run the client is able to collect all server replies as it receives the event *final_sql_rep* (alias of *sql_rep*(*sql_nr*)) while in the state *querying_updating*, and prepares client persistence by generating the formal output event *pcom_nd*. The client moves to the state *ready_to_commit*.

In the next step, the CIC receiver instances running on the same Pcom are given a chance to make the interactions stable. In the very same step the client makes the first forced log write by setting *client_last_logged* to *'pre_commit'*, sends the event commit to the server and advances to the corresponding state. While being in this state, the client periodically resends the commit message based on the timeout *commit_tm* until the servers either confirms by the message *committed* or rejects by *aborted*.

When the transaction is committed, the client installs this fact by changing the value of *client_last_logged* to '*committed*' and moves to the state *committed*. During the same transition it generates the application interface event *client_committed* that normally would trigger some CIC reply to the client on whose behalf the transaction has been executed. If the transaction is aborted, the client will clear the transaction status in the log and start a new attempt to execute this transaction by moving to the superstate *sql_processing*.

Thus, after a crash (*xact_client_crash*) the client considers three possible cases:

1. The log is empty meaning that the transaction needs to be retried.
2. The log contains *'pre_commit'* meaning that the transaction has been executed, and the client has to learn its status from the server by resending the commit message.
3. The log contains *'committed'* meaning that the transaction is definitely committed and the client can recover the reply *client_committed* to proceed with its deterministic replay autonomously.

Now we describe how the Tcom processes transactions. When the Tcom comes up in the state *recovery* and detects that the log (*srvr_last_logged*) for the given protocol instance is empty, it waits for the client to start a transaction. After the server receives the corresponding message from the client, it advances to the state *begin*. Because a Tcom is normally an extensively used multi-user system, the server does not immediately confirm the transaction being started. This rather happens after a random timeout (*begin_ok_tm*) when the Tcom replies with *begun* and advances to the state *wait_for_sql*.

Throughout transaction execution, some events may lead to a transaction abort:

- the nondeterministic event *internal_abort* that models Tcom-initiated aborts e.g., as part of deadlock resolution,
- *user_abort* generated by the client as explained above,
- a repeated out-of-order *begin* message coming from the client implying that it has crashed in the mid of the transaction and is now recovering.

In all these cases, the server moves to the state *failure*. Since transaction rollback is not different from normal transaction execution, it takes some random time *recovery_tm* until the client is notified about the transaction abort. It moves again to the state *recovery* and is ready to accept new transaction execution attempts.

In a normal failure-free run the server advances from the state *wait_for_sql* to the state *sql_call_exec* after receiving the first request, i.e., *sql_req*(*i*) with *i=1*. As you see the state is always exited and more importantly entered again when the compound event *sql_req_ok* (defined ¬*link_outage* ∧ *sql_req*(*i*)) occurs. As a static reaction on entrance of this state the server generates randomly timed replies *sql_rep*(*i*). This seemingly complicated mechanism as opposed to just specifying *tm*(*sql_req*(*i,* …)) is needed in order to prevent confusion by "old" timeouts after a crash as we have already mentioned in Section 4.2.4 above.

After the server receives the event commit from the client, it advances to the state *committing* and waits there for a random period of time *commit_ok_tm*. Then the server commits transaction (i.e., it writes *'committed'* to *sndr_last_logged*), sends the notification *committed* to the client and advances to the final state *committed*. When the client resends the commit message, the server will repeat the event *commited* after the random timeout *commit_tm*.

Table 5 summarizes the properties we have verified for a simple database client-server application. The event *clnt_trigger* is *on* throughout every step because the transaction is a result of internal computations. The client transaction encompasses three SQL requests.

As part of the debugging process, we have verified one of the most important safety properties. F15 proves that the client never even tries to execute already committed transactions. We abbreviate here and in further formulae *srvr_last_logged* by *sll*. The statechart expression *any*(*a*) evaluates to true when one or more elements of the array *a* are generated in the given step. Thus, non-idempotent execution is out of question with this specification.

Now we would like to verify if and how individual TIC obligations are provided by this specification. The Pcom obligation PS1 like S1 can be shown only at the application level

Table 5. Verified Properties of TIC

Nr	CTL Formula	Res.
F15	AG(*sll='c'* → AG(¬any(*sql_req*)))	True
F16	AG(¬cc) → AG ((*cll* = '*p*' ∧ ¬(*c_ok* ∨ *a_ok*)) → $AF_{<30}$(*c*))	True
F17	AG(*cll=''* ⩒ *cll='p'* ⩒ *cll='c'*) ∧ AG(*sll=''* ⩒ *sll='c'*)	True
F18	AG((wr(*cll*) ∧ *cll=x*) → AX AG (¬(wr(*cll*) ∧ cll=*x*)))	True
F19	AG((wr(*sll*) ∧ *sll*=y) → AX AG (¬(wr(*sll*) ∧ *sll=y*))	True
F20	AG(*sll*='c' → AG(¬ (*sll =''*)))	True
F21	AG((*cd* → *sll='c'*) ∧ (*ad* → *sll=''*))	True
F22	$AF_{<500}$(AG¬(*failures*)) → $AF_{<700}$(AG(*cll*='c' ∧ *sll='c'*))	True

as far as the IC framework is concerned. The Pcom obligation PR2 about when to allow the Tcom to drop commit reply message from the stable log goes beyond the interactions within a single transaction and therefore cannot be shown with the given one-transaction-specification.

The Pcom obligation PS2 (periodic resend of the commit message is verified by the formula F16. Assuming that the client does not crash (i.e., *clnt_crash* (*cc*) is false) the following holds: when the client is prepared to commit the transaction (i.e., *client_last_logged* (*cll*) is equal to *'pre_commit'* (*'p'*)) and no reply (i.e., neither *aborted* nor *committed*) reaches the client (i.e.,$\neg(c\_ok \vee a\_ok)$), then the client will resend the message *commit* (*c*) after at most 30 steps.

By verifying the formula F17, we show that

1. the variable *sndr_last_logged* assumes one of the two possible values: empty *''* or *'committed'*,
2. the *client_last_logged* variable may be one of the following: empty *''*, *'pre_commit'*, or *'committed'*.

F18 shows for each possible value *x* that it is written to *clnt_last_logged* at most once. The identical formula F19 is shown for each possible value *y* of *srvr_last_logged.* Thus, one can infer from F19 that the client detects duplicate commit notifications and logs commit only once. Similarly, the server detects commit request duplicates issued by the client and logs the transaction at most once as it follows from F19. F20 proves that a committed transaction is durable. Together with the formula F17, the specification is proved to provide transaction atomicity.

F21 demonstrates that the Tcom always provide faithful reply. The server never generates the message *committed* (abbreviated *cd*) with the transaction outcome log entry being empty and it does not respond *aborted* (abbreviated *ad*) after having committed the transaction.

Finally yet importantly, we present a liveness property of TIC. We show that with a finite number of failures (*xact_client_crash*, *xact_server_crash*, *link_outage*, application-initiated *rollback* due to some inconsistency or *internal_abort* on server) and reasonable server performance (when client requests do not pathologically time out causing transaction restarts) every transaction under TIC is executed exactly-once. The timeouts

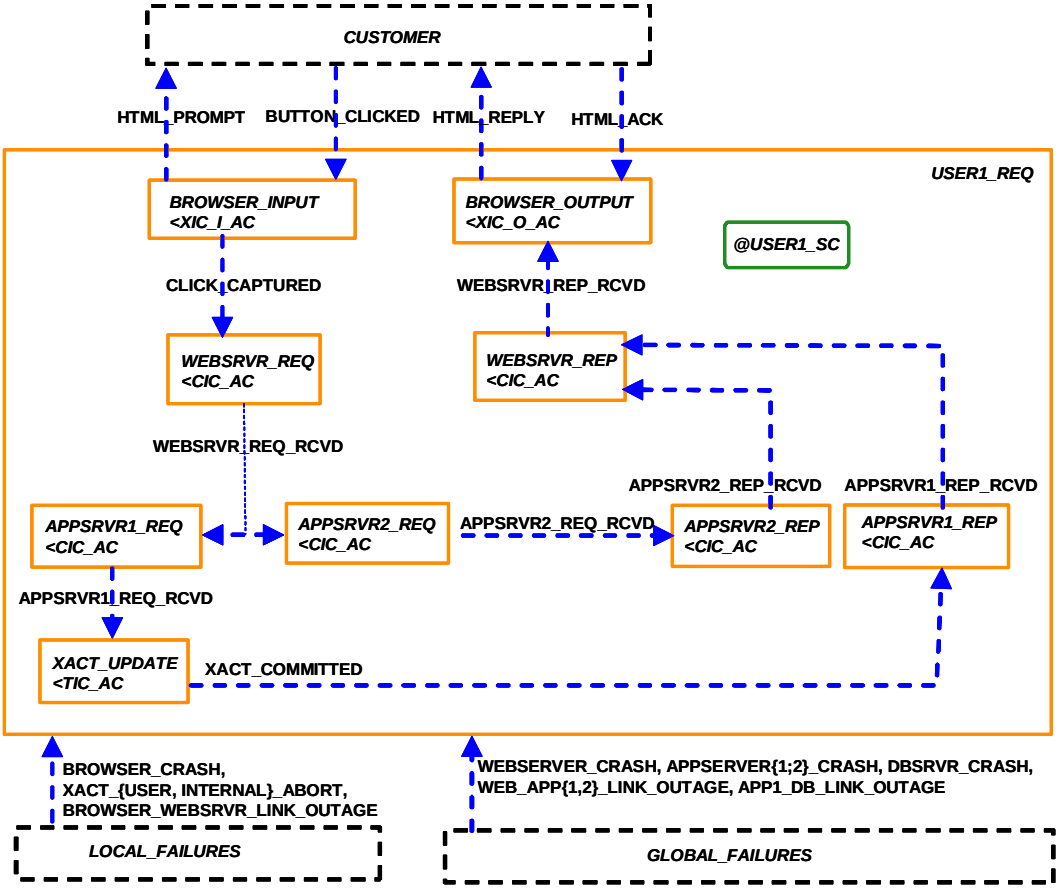

Figure 25: IC Application in Web Service Activitychart

that cause transaction restarts are *begin_tm* and *sql_tm*. They do not occur as long as long as *msg_exec_time* < *client_to* - *RTT* holds for the random execution time on the server (*msg_exec_time*), the current integer value of the client timeouts (*client_to*), and the round trip time (*RTT*) of two steps in the synchronous time model. (When the request message is generated in step *i*, it is seen by the receiver in step *i+1* in which it also generates the reply which is seen by the original sender in step *i+2*.)

Assume that all the failure and timeout events we have just described are OR-ed in the compound variable *failures*. Then the CTL formula F22 can be verified. This formula states that if no failures occur anymore after at most 500 steps, both client and server will commit and install TIC after at most 700 steps.

4.3.6 Sample Application of the IC Framework

Now we are ready to build composed specifications for concrete real-world application scenarios. In this section, we model a sample Web Service that encompasses Web browsers, a Web server, two application servers, and a database server. The browsers, the Web server, and the application servers are Pcoms; the database server is a Tcom; solely

the customers (i.e., end-users) are considered to be outside the framework and are handled as external components. Figure 25 shows the activitychart of an end-user request in the Web Service. In the interest of clarity, the dataflow information in the activitychart is limited to the trigger events that drive the execution of the end-user request, and the failure events. The control activity *user_sc* is not shown here because it consists almost only of orthogonally starting the activities for sending and receiving of the messages defined in the activitychart. The real complexity is hidden in the generic charts of the interaction contracts introduced above. An activity declaration of the form *activity<generic_activity* means that *activity* is an instance of *generic_activity* in the Statemate design.

An informal description of the application logic follows:

1. The user review initial HTML page in her browser (the formal parameter *user_input_prompt* of the XIC input instance *browser_input* is bound to the user-scope event *html_prompt*). The user fills in some data under XIC and clicks on a submit button (the formal parameter *user_input* of the XIC input instance is bound to the user-scope event *submit_clicked*). The installed XIC generates the user-scope event *click_captured*, to which the formal output parameter *xic_i_installed* is bound.
2. In turn, *click_captured* is the *sndr_trigger* of the ICIC instance *websrvr_req* (i.e., the condition parameter *icic* equals *true*) standing for the call of the Web server by the browser. An arrival of the request at the server is signaled by the user-scope event *websrvr_req_rcvd*, to which the formal output parameter *received* of the CIC receiver is bound.
3. The event *websrvr_req_rcvd* is the *sndr_trigger* of the two CIC instances *appsrvr1_req* and *appsrvr2_req* handling the application server requests initiated by the Web server.
4. The output event *appsrvr2_req_rcvd* of the CIC instance *appsrvr2_req* is the *sndr_trigger* for the CIC instance *appsrvr2_rep* dealing with the corresponding reply to the Web server.
5. The output event *appsrvr1_req_rcvd* of the CIC instance *appsrvr1_req* is the *sndr_trigger* of the TIC instance *xact_update* handling a two-statement-transaction on the database server on behalf of the first application server. Once transaction is completed the client part of the TIC instance generates the user-

scope event *xact_committed* to which the TIC client output event *client_committed* is actually bound.

6. The event *xact_committed* is the *sndr_trigger* of the CIC instance *appsrvr1_rep* that handles the reply message of the first application server to the Web server.
7. For the Web server we manage a slightly more complicated application logic that is defined by the orthogonal component *websrvr* of the control activity *user_sc* in Figure 26. The Web server generates the event *websrvr_done* that is the *sndr_trigger* for the CIC instance *websrvr_rep* only after it detects for the first time during uninterrupted normal operation to have gathered both replies from the asynchronously called application servers. As you may see, the Web server loses the information about previously received reply messages on each occurrence of *websrvr_crash* by assuming the default configuration (*wait_app1*, *wait_app2*) again.
8. The user-scope event *websrvr_rep_rcvd* is also the input parameter *sndr_trigger* of the XIC output instance of the browser.
9. Finally, the browser presents to the customer the output *html_reply*, to which the output parameter *pcom_output* of the XIC output is bound which is acknowledged by the user-scope event *html_ack*.

We limit the specification to only **two** users (i.e., parallel end-user requests) due to the high verification costs. However, this suffices to introduce inter-user concurrency into the specification. Random order of the replies resulting from asynchronous calls to the application servers represents an additional source of intra-request nondeterminism.

As for failure events, we distinguish between failures that are local to a particular customer and failures that affect every customer in the system. The global failures are all

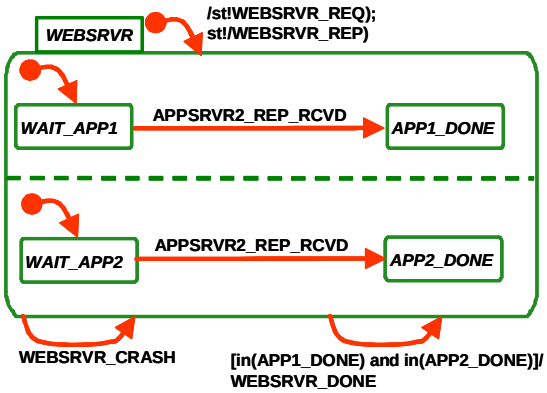

Figure 26: Orthogonal Component of the Web Server Control

server crashes and outages of the network between the servers. Browser crashes and broken internet connection between the browser and the Web server as well as user-initiated and server-enforced transaction aborts affect individual end-users. All failure events are routed correspondingly to the instances of the generic IC activities.

Each Pcom possesses its own nondeterminism alert. On a shared component such as the Web server such an event is shared by all users. To elaborate more on this particular example, *websrvr_nd* is generated as the output parameter *sndr_nd* of the CIC instances *appsrvr*{*1, 2*}*_req*, *websrvr_rep* of **every** user and is consumed as the input parameter *rcvr_nd* by the CIC instances *websrvr_req* and *appsrvr*{*1, 2*}*_rep* of **every** user.

Table 6 summarizes the results of the application-level verification of the IC framework. We start verification by proving that the state corresponding to displaying HTML output to the user is reachable, i.e., there are successful runs in the composite system. The formula F23 is true. Then we have shown that each message is logged at-most-once, i.e., for each instance, for each logging variable *l* and each value *v* that it can assume we have proved the formula F24.

The most interesting safety property of the Web server results from its application logic. The Web server replies to the browser after receiving the browser request and processing the replies from the application servers. Thus, the first attempt to send a reply to the browser commits the exact order of these messages, most importantly that of the asynchronous application server replies. Therefore, for each of the corresponding variables *rcvr_last_logged* (*rll*) and the event *send_msg* from the instance *websrvr_rep* the formula F25 should be true. This has been successfully proved.

Unfortunately, we have not been able to prove any properties for two concurrent users because the model checker has not terminated even after 10 days. On the other hand, it has not produced counterexamples for the safety properties either, and we could demonstrate the correct handling of multiple asynchronous nondeterministic messages with the Web server above in a smaller context.

Table 6. Verified Properties of a Sample Web Service

Nr	CTL Formula	Res.
F23	*EF html_reply*	True
F24	$AG((wr(l) \wedge l{=}v) \rightarrow AX\, AG(\neg(wr(l) \wedge l{=}v))$	True
F25	$AG(send\_msg \rightarrow (rll{=}'s' \veebar rll{=}'i'))$	True

4.3.7 Verification Run-Time

This work has been accomplished on a server with 64x 1.2 GHz UltraSPARC-III+ processors and 184 Gigabytes RAM. The specifications at the level of individual IC contracts result in OBDDs whose size does not exceed 10^5 nodes. It took at most 15 sec to verify the safety properties and 1 to 17 hours were needed for the liveness properties depending on the execution path lengths used for the *Finally* modality, varying between 100 and 700 steps. The OBDD size for the specification of the Web Service model is in the order of 10^7 nodes. The maximum run-time for the safety properties (in the single-user context) is less than two hours.

The verification of liveness in the application-level context has not terminated even after one week. The safety could only be proved for the nondeterministic request execution for a single user. Table 7 gives an overview of model checker run-times for different types of specifications and properties to be proved. Different run-times and OBDD sizes are given for the specification variants with and without integer expressions where available.

4.4 Lessons Learned

Our initial hope for applying formal specification and model checking techniques to recovery protocols was that this would be a nontrivial but not too difficult engineering exercise. However, in the process of working with Statemate and the model checker, we realized that there are many subtleties and modeling choices that can make the difference between a verifiable, readable, and composable specification, and a model that is too complex for the model checker, difficult to match with the informal descriptions of the protocols, or unsuitable for reuse in a more comprehensive application-level model. In the following section, we report some of the lessons learned.

4.4.1 Efficient Verifiability

We faced a number of design choices for modeling the basic elements of the IC framework: messages, failures, logs.

Table 7. Verification Run-Times

Property/Specification Type		OBDD size	Verification Time
IC-level safety	Integers used	~10^4	~5 seconds
	Without integers	~10^3	~1sec.
IC-level liveness	Integers used	~10^6	~10 hours
	Without integers	~10^5	~10 hours
1-user WS safety	Integers used	~10^7	Not terminated
	Without integers	~10^6	~10 hours

As for **messages,** we had to find an appropriate representation for message uniqueness required in the IC framework. The first attempt was to use integer-valued message sequence numbers, but this created tremendous problem for the model checker. Statemate translated (bounded) integers into their bit representations, leading to a state explosion that led to unacceptable, often non-terminating behavior of the model checker in the global context. Instead, we encapsulated all messages used in a given interaction contract as event variables local to the protocol instance. Thus, messages belonging to different protocol instances cannot be confused, and this trick sped up the model checker substantially while retaining the semantics of sequence numbers.

As for **logs**, we considered the following three options: (bounded) queues, history-connected states, and string variables. Queues seemed to be the most intuitive choice, but they are not yet supported by the Statemate model checker. History-connected states were used in the first attempt, but led to disastrous behavior of the model checker. Finally, we resorted to using string variables for explicitly remembering the most recent persistent state of a given protocol instance. This can be seen as an efficient emulation of Statemate's concept of history-connected states.

Another interesting issue was how to capture effects of a message execution since we wanted to show the impossibility of non-idempotent execution with interaction contracts. The solution that led to efficient verification was to show globally that identical values are never written to a log variable. Initially we used integer counters for critical transitions. Showing that these integers can never exceed *one* would also prove the impossibility of non-idempotence at the IC level, but the verification of this at the application level did not terminate. The general insight is that specification tools such as Statemate offer some convenient modeling elements with hidden complexity. They should be used with great caution.

As for **failures**, we wanted to model both component crashes and network problems such as router failures, without having to model a detailed network, as this would have grown the model beyond tractability. We modeled all failures simply as nondeterministic events. As failures of one component lead to timeouts in other components, the time model was a critical issue, too. We found that between the two Statemate options, synchronous or asynchronous time, only the former was suitable and led to readable specifications with clear semantics. For modeling timeouts, we initially used Statemate's native timeout mechanism that uses integer expressions, but this led to unacceptable run-time of the

model checker. The final solution models timeouts as elementary nondeterministic events that do not lead the model checker into complexity pitfalls. The correctness of the original specification follows from the following consideration. Execution paths with properly periodic event recurrence are a proper subset of a complete set of execution paths. With most safety properties being *all-quantified* everything we have proved for the latter will also hold for the former.

We took **request execution times** into account because the model primarily aims at multi-user environments where response times may significantly vary depending on the current load. The ability of the framework to deal with repeated requests caused by very slow request execution in the absence of failures is one of the key features to be verified. It is also well known from empirical studies of so-called Heisenbugs that stress conditions like high load and high variability in the timing behavior of threads may exhibit bugs that do not occur at all under normal conditions. We have reply messages generated at random points triggered by nondeterministic events rather than simply using a default setup with constant execution times. This created a symbolic and exhaustive stress “test” for handling timeouts at all levels of the system. It did slow down the model checker, but this is well justified by the additional confidence in the correctness of the complete system.

4.4.2 Composability

One of the most challenging tasks was to design the interaction contract specifications in an easy-to-compose way. This was achieved by defining IC’s as generic activities. They have a simple interface to the application logic which abstractly defines that receiving a message in one instance (e.g., an application server request) causes a message send in another instance (e.g., the application server reply or a further request). Activities running on the same component share the same failure events. More importantly, they share the same nondeterminism event such as *websrvr_nd* causing a forced write to the corresponding log variables, which is atomically accomplished in a single step.

Verification of an entire composite system is orders of magnitude more complex than for a single IC. In the sample application, one end-user request encompasses five request/reply pairs which results in ten protocol instances running simultaneously; this is doubled in the two-user model. Since each protocol instance is a cross product factor in the overall system, the size of the underlying OBDD increases exponentially with each

new component added to the system. This explains the difficulties we faced during the application-level verification. Nevertheless, the proofs of the underlying IC's already give very high confidence in correctness.

5. EOS: Exactly-Once Web Service

"All is well, provided the light returns and the eclipse does not become endless night. Dawn and resurrection are synonymous. …" - Victor Hugo

In this chapter, we describe another major contribution of this thesis: an efficient implementation of the IC framework for general stateful Web Services. We enhance popular Web technology products: (i) the server-side script language PHP run on Apache Web Server, and (ii) the browser Internet Explorer, to enable the system-failure-resilience of arbitrarily structured Web applications without any coding overhead.

5.1 Introduction

5.1.1 The World Wide Web

The World Wide Web (abbreviated as **WWW**) and often called simply the **Web** for short is a system of interconnected autonomous Internet servers that offer specially formatted interlinked documents. Most documents on the Web are written in **Hypertext Markup Language (HTML)** [W3.org] that is a mixture of plain text and markups defining the text structure, the text layout, and links to other, either local or remote, documents. Figure 27 depicts a sample HTML document with markups displayed in boldface. The scope of a markup is defined by the corresponding start and end tags (e.g., <html> and </html> enclose a complete HTML document). A markup may have additional attributes that are defined in the start tag. In the current example, the attribute *href* of the markup *<a>* provides a reference to a document containing more information on HTML. All objects on the Internet including Web documents have a unique **uniform resource identifier (URI)** [IETF 1998] as e.g., *http://w3.org/*, the value of the attribute *href*.

The Web users access objects via the **Hypertext Transfer Protocol (HTTP)** [IETF 1999]. It was designed as a simple plain-text-based application-level protocol layered just

```
01.  <html>
02.    <head>
03.          <title>Example</title>
04.    </head>
05.    <body>
06.          Hi, I'm a static
07.          <a href=http://w3.org/">HTML</a>
08.          Page!"
09.    </body>
10.  </html>
```

Figure 27: Simple Static HTML Page

on top of the transport protocol stack TCP/IP [Comer 1988] to download static HTML pages from a remote site. Typically, an interactive client program called a **Web browser** translates a URI provided by the end-user into an HTTP request to a **Web server**. The first line of the header request defines the operation on the URI to be executed by the server. HTTP supports several operations with **GET** and **POST** being the most frequently used.

The HTTP GET request was initially used to download static objects such as HTML pages along with multimedia objects embedded in them (all fetched by separate requests). With the need to query the growing Web resources, HTTP allowed adding a **query string** containing parameters in the form of name-value pairs to the URI (e.g., as in *http://google.com/search**?sourceid=navclient&q=application+recovery***). The query string is processed by a program connected to the Web server through the **common gateway interface (CGI)** [W3.org]. Since the query string is physically a part of the URI, its length is limited by the maximum URI length, which has varied over the time from 256 to 4,096 bytes. Although the maximum URI length is not specified in IETF [1999], the HTTP POST request was proposed as a more flexible way of submitting versatile input including large files to Web servers. Instead of encoding parameters into the URI of a CGI resource, an unlimited number of objects can be attached to the request. Moreover, the current HTTP specification recommends that the GET method *should* no longer be used for purposes other than object retrieval, i.e., it should not have any side effects on the server state.

The initial usage of the Web for download of published HTML documents is reflected in that HTTP was designed as a stateless protocol. With the increasing usage of the Web as a gateway for business application, a special mechanism, coined **cookie**, was added to HTTP. The HTTP cookies allow storing the HTTP session state *with the HTTP client*. The client submits the session context with each request; the server produces a new state, and sends it back with the reply such that the server still does not have to maintain the state information. If the application is state-rich, this significantly degrades the end-user experience because of the latency overhead. Currently, cookies are normally used to keep the session identifier only that allows the server to put the current request into the appropriate context. This makes such a Web application truly stateful.

5.1.2 Apache Web Server

Apache is an open-source project focusing on a *complete* and *correct* implementation of the server side part of the HTTP protocol [Apache.org]. According to a recent survey that included responses received from 56,923,737 sites, Apache remains the leading Web server product, having more than 67% percent of the market [Netcraft 2004]. Apache is implemented in *C*, and it is available on most mainstream hardware and operating system platforms including various UNIX and Windows systems. Apache serves as a code base for Web servers shipped with many commercial products, e.g., with Oracle's database and application servers, IBM HTTP Server for iSeries, etc. In the rest of the chapter, we use the term Apache to refer to the Apache 1.3 port for Windows that served as the code base for the prototype. The following review of the Apache architecture follows the documentation from the project Web site [Apache.org]

Although the port for Windows is almost identical with the reference implementation for UNIX, it uses a slightly different process model. Apache on UNIX implements a **pre-forking** server model. The parent process is responsible for spawning child processes that listen for, accept, and execute incoming HTTP requests one at a time. The parent process solely monitors the child processes, and restarts them upon a failure. It also periodically proactively replaces the child processes to prevent them from occupying the entire memory due to potential memory leakages. Thus, the code of the parent process is relatively small and well-tested and its failures have been extremely rare, as reported by the users. The child process logic is much more complex and it usually contains third party code linked either dynamically or statically via **modules**. If a child process becomes corrupted, only one client request will be affected whereas the other requests will continue to be served. The pre-forking model has been criticized for the high overhead of spawning a process, the overhead of context switches between the processes, and missed opportunities for reply caching.

The Apache port for Windows is an experimental multithreaded implementation (a prototype predecessor of Apache 2) that consists of a heartbeat checking parent process and a single child process maintaining a pool of worker threads. It responds to the critic points mentioned above with the penalty of having all concurrent requests interrupted upon a crash.

Apache breaks down processing of a request into the following phases.

1. Translation of the URI into the complete local file path of the requested object.
2. End-user identification (optional).
3. Authorization check of the user for the given URI.
4. MIME type detection of the object requested (e.g., *application/x-httpd-php* for a PHP script, see below). MIME stands for **Multipurpose Internet Mail Extensions** [IETF 1996] and is a standard way to encode arbitrary data using just plain text (i.e., ASCII characters).
5. Detection of an appropriate module that provides a handler for the MIME type. The module takes over request execution and sends a reply back to the client.
6. Logging the request information for statistics (not for recovery), which is optional.

In each of these phases, Apache scans the list of loaded modules, checks if they offer a handler for the current phase, and attempts to invoke the handler should it be there. A handler function may execute the current phase completely, decline execution, or detect an error and stop. Many phases, specifically phase 5, are terminated after the very first module has been found whose corresponding phase handler returns the success code.

5.1.3 PHP and the Zend Engine

The material on PHP can be found on the project homepage [PHP.net]. PHP (recursive acronym for "PHP: Hypertext Preprocessor") is a widely used general-purpose scripting (i.e., interpreted) language that is especially geared for the Web (CGI) application development. PHP is platform-independent and is available as an add-on module for a wide variety of Web servers including Apache, Microsoft IIS, and Sun ONE, to name just a few of them. PHP is used with more than 50% of the Apache installations [SecuritySpace].

The PHP implementation is an open-source project consisting of many **PHP modules** responsible for different PHP function subsets of the PHP language and the Zend engine implementing the language interpreter [Zend.com].

Consider the following example of Figure 28 that produces a simple HTML page displaying the string "*Hi, I'm a PHP script!*" in the browser window. As you may see, PHP code can be embedded into an HTML page (as well as into any other format) as a code island enclosed by special start and end tags("**<?php"** and **"?>"**, respectively) that make the Web server switch into the PHP mode instead of producing output identical to

```
01.  <html>
02.    <head>
03.          <title>Example</title>
04.    </head>
05.    <body>
06.          <?php
07.                echo "Hi, I'm a PHP script!\n";
08.          ?>
09.    </body>
10.  </html>
```

Figure 28: Simple PHP Page

the source page line by line. The server executes the PHP code, and replaces the code island including the enclosing tags (the enclosing tags are not valid HTML tags) by the script output. This differs from Web applications written in conventional programming languages (e.g., C/C++ and Java) where one annoyingly has to *printf* every single HTML line no matter how much dynamic content it contains. Similar technology is used by many other Web scripting languages, e.g., by JSP [Sun] and ASP.NET[Microsoft]. An HTML page may contain several PHP code islands, which is equivalent, up to the HTML output in-between, to the sequential execution of the concatenated code islands as a single code island.

Most of PHP's syntax is borrowed from *C* with a small fraction of elements coming from *C++*, *Java*, and *Perl*. PHP owes its rapid spread to the ease of its syntax and its expression power. In the rest of the thesis, the term PHP is used to refer to PHP 4.0.6 that served as the basis of our prototype.

In PHP, a variable does not require a formal declaration. The variable is allocated automatically when a value is assigned to it for the first time in the script. Variables are prefixed by a dollar sign (e.g., **$varname**). The type of the variable does not have to remain constant: it is determined by the very last value assignment. Variables can be accessed by value and by reference (e.g, **&$varname**). A PHP variable type is one of the following: Integer, Double, Boolean, String, Array, Object, and Resource.

A string is implemented as a *C* structure containing a pointer to the character array and the length of this array. PHP arrays are actually hash tables that allow enumerating values starting with zero as in *C*, indexing values by *strings*, by non-contiguous integers, or by a mix of strings and integers. An object is a hash table mapping the property names to the corresponding values, and the method names to the implementations. Resource variables are created by PHP interface modules for external software such as file system and

network support of the operating system, database connectivity modules, etc. PHP resource variables contain integer keys that are mapped through a hash table to module-specific resource representation (e.g., file handles, sockets, and so on).

PHP relieves developers from manually parsing the HTTP request by providing uniform access to the most important HTTP request parameters through predefined global arrays:

- *$HTTP_GET_VARS* maps the names of the parameter provided in the query string part of the URI to the corresponding values. This also works with a POST request when the requested URI contains a query string.
- *$HTTP_POST_VARS* maps the names of the parameters provided in the body of a POST request except for files.
- *$HTTP_POST_FILES* maps a name of an input field in HTML to the array containing details of the corresponding file POSTed: its name on the client's file system, its MIME type, its size, and its temporary name on the server's file system.
- *$HTTP_COOKIE_VARS* is a mapping of names to the values of the cookies that have been found in the request header.

5.1.4 PHP Session Management

PHP provides a **session module** for maintaining the **PHP application state** across subsequent HTTP requests. The session module supports various methods of storing the session state (e.g., in a file, shared memory, central database, etc.). The state of a PHP application running may be private (e.g., a shopping cart) or shared, concurrently accessed by multiple users (e.g., the highest bid in an electronic auction). The state variables, accessed by their string names through the global session array *$HTTP_SESSION_VARS*, may be of any basic or derived data types except resources. PHP typically uses a cookie to propagate the session (state) id to the client.

The session support is activated either explicitly by calling the function *session_start* somewhere in the script or it is started automatically, if appropriately configured, prior to executing the PHP code for the given request. The session module reads the state associated with the session id provided with the request from the session storage and makes it accessible for the PHP script as the session array. When the request does not contain a session id cookie, the session module generates a new session id (that will be included as a cookie in the reply) and creates an empty session array. The PHP script may

```
01.  <?php
02.    session_start();
03.
04.    if(isset($HTTP_SESSION_VARS["count"]) == FALSE)
05.    {
06.      $HTTP_SESSION_VARS["count"] = 0;
07.    }
08.    $HTTP_SESSION_VARS["count"]++;
09.
10.    echo "Hi, I have been called ";
11.    echo $HTTP_SESSION_VARS["count"];
12.    echo "times\n";
13.  ?>
```

Figure 29: Sample Usage of PHP Session Support

update the session array by changing or deleting the existing entries or by adding new ones. The new state is made available for subsequent requests by either calling the function *session_write_close* or implicitly when the script terminates. Calling this function at the end irrespectively of whether the script has really modified the state is an obvious weak point of the current PHP implementation. The state accesses follow consequently a simple pattern: a read operation followed by a write. As you will see later, we modify PHP to allow scripts logically consisting of a single read.

The sample script of Figure 29 counts how often it has been invoked by a user. It looks up the current state (line 2), zeroes the state variable *count* upon the first access (lines 4-7), and increments this variable for each invocation (line 9). If this script maintains a shared state for several users, their accesses must be serialized for consistency. The session module relies on the session storage module in current use to achieve this. For instance, the standard session storage module that makes use of the server's file system exploits file locking for this purpose. This, however, works only on UNIX systems where requests are executed by different Apache processes. File locking cannot synchronize threads of the same process. Our prototype provides a more flexible concurrency control mechanism coined *latches* to cope with this and some other issues that we describe below.

The shared memory storage module is available solely for Linux. The database storage module is better administrable than the other alternatives, but, relatively inefficient. The most popular session storage module uses the server file system to store session data persistently. It allocates a physical file for each session upon the very first call to the function *session_start* that exists until the function *session_destroy* is called at the end of the session. The original implementation is inefficient because it does not cache session

```
01.  <?php
02.    $b2b = curl_init("http://eos-auctions.com/b2b/");
03.    $params = array(
04.                  "auction_id" => 100232,
05.                  "bid" => 50.74
06.                );
07.    curl_setopt($b2b, CURLOPT_POST, TRUE);
08.    curl_setopt($b2b, CURLOPT_POSTFIELDS, $params);
09.    $b2b_reply = curl_exec($b2b);
10.    curl_close($b2b);
11.  ?>
```

Figure 30: Sample Usage of the CURL Module

data in the main memory, which implies having to perform a random I/O for every access to a particular PHP application state.

5.1.5 PHP Business-to-Business

As in the most other cases, PHP offers several options to interact with (potentially PHP-enabled) Web services. One of the most popular and elegant methods provides the CURL module that allows PHP applications to access remote HTTP, secured HTTP, FTP, LDAP, and some other resources in a uniform fashion. The complexity of the protocols is hidden behind a very simple interface that keeps the coding effort at minimum. This functionality is implemented by the CURL library for *C* developed as an open-source project [Stenberg].

Figure 30 shows a fragment of a PHP script that makes use of CURL to call another Web service to bid for an auction item. It first initializes a resource variable for this operation (line 2) and defines its parameters in an array variable (lines 3-6). Next, the PHP script specifies that the POST method should be used, and associates the parameter array with the request (lines 7-8). Some Web Services are invoked via SOAP, the Simple Object Access Protocol layered on top of HTTP [W3.org]. When SOAP is involved, we would pass a SOAP message as a POST parameter. The HTTP request is sent to the URI provided during the resource initialization and the reply string is assigned to the variable *$b2b_reply* (line 9). When the CURL resource is no longer needed, it can be either explicitly closed (line 10) or it is automatically garbage collected at some point after the termination of the script.

Note that by implementing CIC for CURL and Session modules of PHP, EOS-PHP provides recovery guarantees at the HTTP layer and thus, higher-level applications

including PHP script libraries for SOAP and other protocols over HTTP are relieved from dealing with system errors.

5.1.6 Microsoft Internet Explorer

Internet Explorer (IE) is currently the dominant browser on the Web as recent surveys show [WebSideStory 2004]. The complete documentation on the IE is provided in the Web Development section of the MSDN home page [Microsoft]. The key features of IE6 that served as a client platform for this thesis are the following:

- **Dynamic HTML (DHTML)** [W3.org] allows creating powerful graphical user interfaces (GUI) for the client side of Web applications. DHTML allows client-side scripts (written in JavaScript, VBScript, etc. [W3.org, Microsoft]) defined in *script* markups to interact with the end-user by intercepting her mouse and keyboard input. DHTML allows manipulating the document through its tree-structured representation coined **HTML Document Object Model (HTML DOM)** [W3.org] without contacting the server.
- **User Data Persistence Behavior** allows managing a limited portion of the Web application state on the client's disk. The state is stored as a file in **eXtensible Markup Language (XML)** format that allows describing data objects in a platform-independent manner [W3.org]. XML documents can be manipulated in a similar way as HTML thorough the **XML Document Object Model (XML DOM)**.

5.1.7 Big Picture of EOS

Using the technologies and the components described in the previous section developers are able to build arbitrarily distributed (potentially stateful) Web applications. Figure 31 sketches a sample service configuration of an EOS-based Web application. End-users call Web applications on the server sites PHP 1 and PHP 4 using their browsers. PHP 1 invokes (through the CURL client interface) Web Services on PHP 2 and PHP 3 that in turn call PHP 5 and PHP 6, accordingly. The other frontend Web application server PHP 4 normally follows an identical procedure; occasionally, however, it optimizes this execution path by having the browser immediately invoke PHP 5 from one of the embedded HTML elements. Such a system enabled by EOS provides system-failure-resilience.

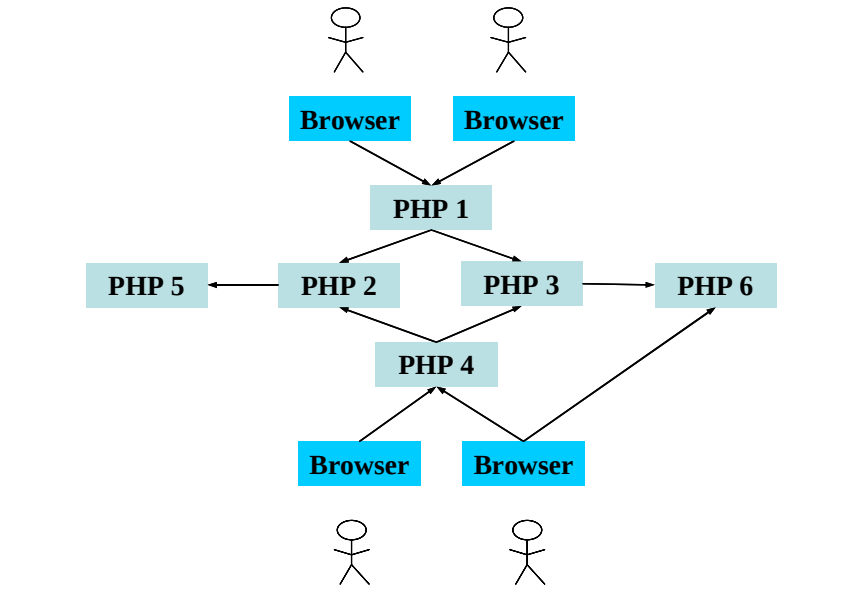

Figure 31: Sample EOS Web Application

In order to comply with the IC framework, the PHP application servers and the browsers must be turned into Pcoms whereas the end-users can only be viewed as Xcoms. The interactions between a browser and an end-user must follow the XIC. PHP servers interact under the CIC or the ICIC.

5.2 Persistent EOS Browser

There are two major design goals of EOS enhancements to the browser. The first is to improve the user experience by saving as much of her input across a failure (such as a browser crash or Web Service unavailability) as possible. This avoids the need for annoying repetition of long inputs, which may happen with lengthy forms such as e-government applications (tax declarations, visa applications, etc.) and e-business applications (e.g., insurance and credit approval requests). This is the task of the XIC implementation for the browser. The second design goal is to give the guarantee to the end-user that all server requests are executed exactly once which is the task of the CIC stub of the browser.

For this purpose, the browser has to be provided with logging capabilities. Several implementation strategies have been considered. Most browsers support proprietary plug-in interfaces for additional modules connected with significant coding effort. ActiveX elements and Java applets are extremely unpopular among home users and limited in intercepting user events generated by HTML elements. As a compromise, we decided to restrict ourselves to using the standard features of the most popular Web browser as of the time we started developing the EOS prototype, Internet Explorer (IE). More radical

```
01.  <ROOTSTUB
02.    MSN="123"
03.    logged_uri="http://localhost/b2c_user1/test7.php"
04.    title="EOS: Test (Screen 7)"
05.    body="escaped HTML string"
06.    user_commit_msn="122"
07.    user_commit_element_id="form1"/>
08.  </ROOTSTUB>
```

Figure 32: XML Store Log

solutions such as building a derivate browser that hosts various Web browser components (e.g., via the WebBrowser control of IE) or extending an open source browser (e.g., Mozilla) would have been feasible as well but they would have consumed more time without significantly enlarging the feature set covered by the implementation described in this subsection. The downside of our approach, however, is that we do not have the full control of some important issues, e.g., when persistent cookies and user data are really flushed to disk.

We do not expect end users to explicitly store our recovery JavaScript on their computers. The server counterpart EOS-PHP adds the browser logging and recovery code as the last step of output processing. Original server scripts do not have to be changed. Logging is done by modifying a so-called *XML store*, an XML structure managed by IE on the client's disk similarly to persistent cookies. Figure 32 depicts a sample instance of an *XML store*. Usage of particular elements of the *XML store* is explained in the following subsections. An *XML store* can keep up to 640 KB per application, which is sufficient because we are going to keep no more than one copy of an HTML tree, typically occupying less than 50 KB. The XML store feature is provided by IE as part of its default persistence behavior called "*userData Behavior*". The recovery code inserted by the EOS-PHP server contains an invisible user-defined HTML element <*sdk:logger id=pagestate style="behavior:url(#default#userData);*"> with attached *userData* behavior. The XML object associated with the XML store can be accessed through a simple interface {*get*,*set*}*Attribute* or using the XML DOM that is natively supported by IE as well. When we need to force the XML store with the logging information to disk we call *pagestate.save*(*XMLStoreName*). For recovery the content of the XML store is fetched by invoking *pagestate.load*(*XMLStoreName*).

5.2.1 Supported Browser Applications

EOS concentrates on providing the recovery guarantees of pure browser applications using solely DHTML, XML, and JavaScript engines of the browser and interacting with

Web servers through HTTP. Enhanced, embedded GUI applications implemented as Java applets or Flash animations are not considered in our prototype. Leading e-business providers try to attract more customers by keeping the software and hardware requirements as low as possible, and do not use these technologies on the client side anyway. Logging is based on intercepting certain DHTML events such as mouse clicks, and keyboard inputs. Events affecting the look of the HTML page (e.g., entering a word into a form field, clearing a checkbox) are handled by the XIC. Events causing the browser to send a request to the Web server (e.g., clicking on a form submit button or a simple link) are subject to CIC. The only way we have found to save the browser state boils down to dumping the DHTML tree to disk.

The browser state may be richer than mere visible and invisible HTML elements. It may also include global JavaScript variables. As we have no access to the IE scripting engine we have the only chance to capture changes to such variables when they are bound to a (potentially invisible) HTML element each. For the prototype we assume that a user event on an element is handled by a single script function or the default browser event processing. We limit the application to changing only a single element on a user event. We can take this for granted for default edits to form fields but not for changing values of global variables. We are not able to provide deterministic replay for event processing functions that affect multiple HTML elements at once between two consecutive end-user events because we may erroneously dump an inconsistent HTML tree. However, the likelihood of running into this incorrectness is really small because typically multiple changes would normally occur in a very small time window as compared to the overall time the user spends on editing the HTML forms of the given page.

We assume that all relevant HTML input elements are manually assigned unique ids using standard HTML attribute *id* because ids generated by the HTML DOM parser are nondeterministic, varying from parse to parse of the same HTML document as stated in the IE documentation [Microsoft].

5.2.2 Unique Identifiers

Since we do not change the browser implementation, we need to consider a couple of technical points to make the browser fit into the IC framework: unique client and message identification in the overall system.

The client identification by the HTTP header field “User-Agent” is not feasible because it usually contains some generic information about the browser software used. The real client IP is often not visible to the servers in the extranet and it is out of the question for multi-user computers in any event because then we need also the physical TCP client port for unique identification. TCP client ports are a typical source of system nondeterminism because they change all the time. We use a persistent cookie *client_id* that is assigned by the Web server to the browser, which is a standard solution in the state-of-the-art Web business. Consequently, a browser is identified differently by different Web applications, but always unambiguously. The value of the cookie *client_id* serves as the name of the persistent XML store used to log interactions with the given Web service.

Similarly, we manage message sequence numbers (MSN) as persistent cookies. A pair (*client_id, MSN*) uniquely tags a request message. To acknowledge this message, the server increments the request MSN and includes it as a cookie into the reply, such that the next client request will be tagged by (*client_id, MSN* + 1). There is a potential problem when only an incomplete reply arrives, e.g., a few HTTP headers including the MSN cookie. When the user or our CIC stub initiates a “refresh” of the Web page by resubmitting the HTTP request, the new MSN value could be used that would violate duplicate elimination at the server. To cope with this, we additionally log the MSN cookie to the XML store, however only after the reply message has been completely received by the browser manifested in DHTML by *document.readyState* having the value *complete*. Thus, when a request is resubmitted, the EOS code checks the logged MSN value against the MSN cookie that is replaced by the logged MSN when the values differ.

5.2.3 URI Logging and Recovery

Another issue arises when treating browser crashes. The operating system can be configured to automatically restart the browser. But after the restart, the browser loads the start page that is stored in the browser settings instead of resuming the interrupted Web session. Clearly, changing the start page of the browser each time the user proceeds to a new Web application step is nonsense. In our prototype, we would like the user to revisit the greeting page of the interrupted Web application to restore her session. Analogously to the MSN logging, we store the URI of the most recent complete page viewed by the user during normal operation in the attribute *logged_uri* of the XML store. When the user calls the greeting page, the Web application will detect that the browser with an active session needs assistance in recovery; based on the session id cookie it compares the

current URI with the URI stored in a special hash table mapping session ids to the start URI's as you will see in Section 5.3. To provide the browser with an appropriate assistance, the Web server replies with an empty HTML page (i.e., HTTP headers plus "<html></html>") supplemented with the browser recovery code inserted on the fly. The MSN cookie is not incremented and a special volatile cookie *uri_recovery* is set to *true*, in order to prevent logging of the start page URI during recovery.

First, the recovery code reads the last complete URI from the XML store and redirects the browser to this page by assigning *logged_uri* to the DHTML property *document.location.href*. This restores the URI in the browser address bar and issues a GET request to the server. It does not matter that the original request may have been a POST because it will not be executed anyway (the server eliminates duplicate requests based on (*client_id*, *MSN*) pairs). The GET request to the server carries the cookie *uri_recovery* equaling *true*. Seeing this, the Web server responds again only with the recovery code without incrementing the MSN and resets the cookie *uri_recovery* to *false*. After these additional steps connected with the recovery of the pre-crash URI, which also involved the server, EOS browser behaves precisely according to the CIC with the server (until the next crash) as explained in the following subsections.

5.2.4 Browser XIC Logging

To implement the XIC between the user and the browser, we monitor all HTML elements (they are enumerated by the DHTML collection *document.all*) for changes while the HTML page is being displayed. A universal event *propertychange* is fired by an element when it is changed either by the user herself or programmatically. Typical changes are modifications of the element attributes and insertions and deletions of child elements. E.g., when the user types into a text input field, she actually modifies its attribute *value*. We register the function *updatePageSnapshot* saving the whole page in the attribute *body* of the XML store to process the property change events for each element *e* by calling *e.attachEvent*('*onpropertychange'*, *updatePageSnapshot*).

We figured out that flushing XML store on each key stroke, while the user is entering her input into a text input field, may significantly degrade the browser performance. To deal with this issue, the user can specify a number *n* of her input losses she is willing to tolerate in the worst case. Setting *n*=1 offers the best possible failure masking. With an increasing *n* the browser performance improves. Flushing the XML store on every 5^{th}

```
01.  var n = 5;
02.  var timer_id = null;
03.  var unflushed_events = 0;
04.
05.  function deferredXMLFlush()
06.  {
07.    unflushed_events++;
08.
09.    if(unflushed_events % n == 0)
10.    {
11.      pagestate.save(client_id);
12.      unflushed_events = 0;
13.      if(timer_id != null) clearTimeout(timer_id);
14.      timer_id = null;
15.      return;
16.    }
17.
18.    // reset 100 ms timeout. Flush after n*100 ms at latest
19.    timer_id = setTimeout(
20.                 'pagestate.save(' + client_id + ')', 100);
21.  }
22.
23.  function updatePageSnapshot()
24.  {
25.    // logging page title and body
26.    pagestate.setAttribute('body', document.body.innerHTML);
27.    pagestate.setAttribute('title', document.title);
28.
29.    deferredXMLStoreFlush();
30.  }
```

Figure 33: JavaScript for XIC Logging

event and every 500 ms worked fine on our test computer (Pentium III, 1 GHz, 256 MB RAM). The source code of the XIC logging is sketched in Figure 33.

5.2.5 Browser CIC Logging

As for the CIC, we need to intercept the DHTML events that precede issuing an HTTP request and switching to the next page. These are *click* events generated by HTML links, and *submit* events generated by HTML forms. The links of an HTML document are enumerated in the DHTML collection *document.anchors*. The HTML forms are listed in the DHTML collection *document.forms*. For every link *l* we register the function *handleCIC* to process clicks by calling *l.attachEvent*(*'onclick'*, *handleCIC*). Similarly, *handleCIC* is attached to processing of *submit* events of each form *f* by calling *f.attachEvent*(*'onsubmit'*, *handleCIC*).

Prior to returning to the default browser event processing, the function *handleCIC* is called. First it writes the current document body (*document.body.innerHTML*) to the

```
01.  var msn_cookie, timeout;
02.
03.  function rollRecovery()
04.  {
05.    var el_id;
06.
07.    if(msn_cookie > pagestate.getAttribute("MSN"))
08.    {
09.      return; //new message – no recovery
10.    }
11.    else if(uri_recovery == true)
12.    {
13.      return; //called only after uri recovery
14.    }
15.
16.    document.body.innerHTML = pagestate.getAttribute("body");
17.
18.    if(msn_cookie == pagestate.getAttribute("user_commit_msn"))
19.    {
20.      el_id = pagestate.getAttribute("user_commit_element_id");
21.
22.      if(document.all[el_id].tagName == 'A')
23.      {
24.        setInterval(el_id + '.click()', timeout);
25.      }
26.      else
27.      {
28.        setInterval(el_id + '.submit()');
29.      }
30.    }
31.  }
```

Figure 34: JavaScript Recovery

XML store because by clicking on a submit button or a link the user commits her edits (in some countries and in some situations, this is even equivalent to signing a contract from the legal perspective). Second, in the XML store, it sets the attribute *user_commit_msn* to the current MSN, updates the attribute *user_commit_element_id* with the id of the form or the link having fired the current event, and forces the XML store to disk. Third, as message and state persistence is already guaranteed, the function *handleCIC* registers the operation associated with the even for periodic repetition. If the event source element is an HTML form with id *f*, *handleCIC* calls the JavaScript function *setInterval*(‘*f.submit()*’, *timeout*). If the event stems from the link *l*, *handleCIC* calls the function *setInterval*(‘*l.click*()’, *timeout*). At this point the control is returned to the browser that generates an HTTP request from the given event.

5.2.6 Browser Recovery

When the browser crashes, it recovers as follows. As the last action of the URI recovery, an empty page with the last valid URI displayed in the browser address bar arrives at the

client. The browser recovery code first replaces the empty HTML body by the value logged in the XML store. At this point the HTML page has the form as of the time immediately before the crash. The MSN cookie that still carries the pre-crash value as ensured by the EOS recovery code is used to compare against the value of *user_commit_msn*. If they differ (i.e., the MSN cookie is greater), the browser resumes normal operation and starts accepting user events. Otherwise, we know that the user has committed her input on the current page. Browser recovery finds the form or the anchor link by calling the function *document.getElementById*(*user_commit_element_id*) and calls the methods *click*() or *submit*() according to the element type. The browser restarts CIC resend and resumes normal operation. Figure 34 shows the most important fragments of the EOS browser recovery code.

5.2.7 Browser Garbage Collection

Garbage collection is not an issue for the browser because we steadily reuse identical elements of the XML store, such that it does not grow in terms of added attributes and elements. The size of the XML store depends only on the size of the last page having been displayed in the browser window.

5.2.8 Future Directions

We have made a small compromise in the implementation of the XIC and the CIC for the browser (by having the user manually revisit the greeting page of the Web application) because we had no access to the browser source code. Note that this compromise does not affect the exactly-once execution guarantee. However, for a rigorous IC implementation, we should consider enhancing an open-source browser. This would also allow us to provide deterministic replay for arbitrary DHTML applications that contain complex JavaScript functions.

5.3 Persistent EOS-PHP

EOS-PHP is the major part of our prototype. It can serve as both an HTTP server and a middle-tier HTTP client at the same time. It transparently implements the (I)CIC stubs for incoming and outgoing HTTP interactions of PHP applications with other PHP applications and Web browsers. EOS-PHP is geared to provide the recovery guarantees for stateful PHP applications. The log is provided as a universal storage for log entries and the session state data. Log access is accelerated by LRU buffers. In addition, EOS-PHP delivers basic concurrency control in the form of latches.

5.3.1 Normal Operation and Logging Issues

When considering a single PHP Zend engine, we can distinguish three relevant system layers from the logging perspective. We observe HTTP requests at the highest level **L2**, individual PHP language statements at the middle level **L1**, and finally I/O calls to external resources such as the file system and TCP sockets (level **L0**). EOS-PHP does not support interactions with the file system, i.e., the PHP file system functions. Instead, EOS-PHP efficiently manages persistent application states stored as session variables. EOS-PHP does not deal with the PHP socket interface. Instead, EOS-PHP supports recoverable HTTP interactions through the CURL module. The purpose of this subsection is to describe HTTP request processing by EOS-PHP and logging that is necessary for correct PHP application recovery.

A request execution by EOS-PHP breaks down into the following stages: client identification (Stage 1), URI recovery (Stage 2 for interactive clients only), reply resend (Stage 3), request execution (Stage 4), output processing (Stage 5). Note that Stages 2 and 3 are EOS-PHP operations needed for client recovery. Prior to the request execution, a shared *activity latch* is obtained for the duration of the request execution. It prevents the garbage collection mechanism that uses this latch in the exclusive mode from physical reorganization of the log file as explained in Section 5.3.7.

Stage 1: Client Identification

During request startup, EOS-PHP identifies the client id information submitted as cookies. If this information is missing the client is assigned a new id and is redirected to the first session URI (interactive clients only). A B2B component (i.e., another EOS-PHP node) autonomously generates its id by concatenating its host name and TCP *listen port* (*socket*) number. Note that this does not incur any nondeterminism since the listen port number is fixed and uniquely identifies a server application on the given host. Web servers typically listen to port 80. A Web application reachable through the URI *http://eosphp.com/* would introduce itself as "*eosphp.com:80*" when calling other Web services.

The following Stages 2, 3, and the state initialization part of Stage 4 are initiated on behalf of the function *session_start*. The request thread acquires an exclusive log latch because all these operations have to be performed atomically.

Stage 2: URI Recovery

Interactive clients (i.e., those whose user agent field submitted with the request header information is different from *EOS_CURL*) need an additional stage for assisting in recovering the last message sent to the EOS-PHP engine. EOS-PHP checks if the current URI coincides with the URI that started the session (i.e., the greeting page URI). If this is the case, we know that this is an interactive client revisiting the greeting page to restore the interrupted session. As described in Section 5.2.3 an empty page containing solely client recovery code is sent to back to the browser without incrementing the MSN cookie. The volatile cookie *uri_recovery* is set to *true*.

Stage 3: Reply Message Resend.

The log is consulted through the request message id lookup in the volatile **input message lookup table (IMLT) (client id, MSN, reply LSN)**, in order to determine if the HTTP reply is already present. In the positive case, the HTTP reply is served right away and the current request is terminated. When the *uri_recovery* cookie is provided, the server knows that the browser is solely restoring the message URI in the address bar without the need for message resend. To save the network latency, the server responds again with an empty HTML page with the browser recovery code as explained in Section 5.2.3. The cookie *uri_recovery* is set to *false*. When the IMLT contains an entry for the request with the reply LSN being invalid, EOS-PHP is dealing with a request message resend: this thread is paused until the reply LSN is set, and the reply can be served.

When the current request is not a duplicate, it is not terminated by this stage. It is important that we hold an exclusive latch for the log at least until the request is registered in the IMLT during the next stage, in order to prevent two identical messages (resends) from being handled both as original requests.

Stage 4: Request Execution

The request execution starts with fetching the PHP application state through the state buffer. The state buffer is latched in the shared mode to find the proper application state. If the entry for the current PHP application state could not be found (i.e., the request initializes a new PHP session), the request upgrades the state buffer latch to the exclusive mode and inserts a newly created empty state into the state buffer. At this point the PHP application definitely has a valid state. A new LSN is generated for the request and EOS-PHP adds an initial log entry to the log buffer that contains PHP representation of the

HTTP request and the translated PHP script file path. (In fact, a PHP script may depend on more than mere HTTP parameters, e.g., when it uses OS shell environment variables or Apache configuration parameters that change over time. If this is the case, an administrator of the EOS-PHP site should mark these variables for logging in the PHP configuration file). An entry is also added to the IMLT containing the client id, the MSN of the message (both as submitted by the client cookies), and an invalid LSN. At this point the request thread latches the PHP application state in the shared or exclusive mode (as specified in the enhanced PHP function *session_start* that now accepts a Boolean flag *$read_only* as an optional argument) and releases the exclusive log latch as well as the shared state buffer latch. In contrast to the original PHP implementation, the ability to access the application state in the shared mode is an appropriate response to the fact that the load of e-commerce sites is dominated by read-only catalog browsing requests. The latch for the application state is held until the script calls the function *session_close* (explicitly or implicitly during the request termination) that replaces the original PHP function *session_write_close* to avoid irritation. If the request has been declared as a *write* by calling *session_start*(*false*), the application state is stamped with the request LSN before the state latch is released, whereas the volatile read LSN field of the buffer cell is updated in any event.

The information logged during request initialization as described above would suffice for deterministic replay of the HTTP request log entries one after another using the high-level routine *zend_execute_scripts* without any further consideration, if we had not to deal with nondeterministic calls throughout request execution. Nondeterministic calls generate further log entries need for a potential replay of the current request. Since EOS-PHP currently can only replay HTTP requests sequentially one after another as opposed to an arbitrary interleaving of PHP statements issued on behalf of distinct HTTP requests, we need to be able to find the needed log entry quickly. For this purpose we link each entry in the log buffer to its successor using the *next_php* pointer of the buffer cell as depicted in Figure 35. Each PHP-level log entry includes the LSN of its predecessor in the *next_php* chain, such that the *next_php* chain can be restored during recovery.

We use a *last_php* pointer to keep track of script-internal nondeterministic events. Note that at the beginning of the request execution *last_php* refers to the very first (HTTP-level) request log entry whose *next_php* field is NULL. The log is consulted upon every statement call that requires logging. If there is a successor of the log entry pointed by

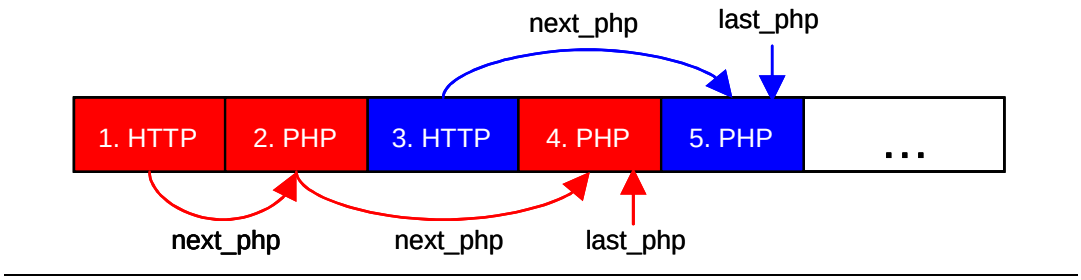

Figure 35: Chained Log Buffer of EOS-PHP

last_php, we deal with a replay. EOS-PHP has the *last_php* pointer refer to its current successor and returns the logged return value from this log entry without executing the statement. If there is no successor of the previously checked entry *last_php* (i.e., *last_php→next_php* is NULL), EOS-PHP either operates normally or completes request execution during the redo phase, such that the statement is executed and a new PHP level log entry is added to the log buffer (using an exclusive log latch). It is connected to the *next_php* chain and becomes new *last_php*. Note that the new PHP log entry contains the LSN of the previous *last_php* that is used during analysis pass to restore the *next_php* chain. Note that log entries of different requests interleave in the log when they use shared latches or an updating request calls *session_close* before its completion.

We are aware that dealing with the *next_php* chains would not be an issue, if we implemented logging with a granularity finer than a single HTTP request. However, a finer-grained logging is more difficult to implement because different thread contexts (consisting of a number of hierarchical function stacks each) would have to be recreated and applied when replaying individual PHP actions (local and global variable reads and writes). Thread memory management of PHP is deeply integrated with system routines that require valid distinct thread handles that are not available during single-threaded redo recovery. For the lack of time, we gave the other issues described in this thesis priority treatment. On the other hand, we still have an argument for the current solution because the simpler recovery is, the more trust we have in its correctness.

Note that when PHP script developers are about to deploy a new script version they need to keep the last version on disk because the ordering of logged operations in the new version may change, and using it to replay HTTP requests would be incorrect. To avoid this, the version number could be attached to the script filename (e.g., *script.php.v2*). After a new version is copied to the script folder, the Apache configuration file needs to be changed to remap the URI to the new script file. At the same time EOS-PHP will

OMLT of eosphp3

URI	MSN	CIC status
http://eosphp1/auctions/	*3*	*installed*
http://eosphp2/books/	5	stable
http://eosphp1/auctions/	6	unknown
http://eosphp1/auctions/	7	installed
http://eosphp2/books/	8	installed

IMLT of eosphp1

client id	MSN	Reply LSN
eosphp3	3	324
…	…	…

IMLT of eosphp2

client id	MSN	Reply LSN
eosphp3	5	324
…	…	…

Figure 36: IMLT and OMLT in Action

always be able to find the proper version to replay a particular HTTP request using the local file path stored in the log.

Nondeterministic functions treated by EOS-PHP include system clock reads (e.g., *time()* returning the current time, random value generators such as *rand(min, max)* generating a random number in the interval between *min* and *max*, and last but not least *curl_exec* returning output of a different Web Service. The routines asking for the system clock and random values are not only interesting because of their potential direct usage in a PHP script, but also because their *C* prototypes are used as input for generating PHP session ids that are pairwise distinct with a high probability. This avoids a potential bottleneck of having a single node in a Web farm assign sequence numbers as session ids to all clients.

The *C* prototype of the function *curl_exec*(*$handle*) implements the CIC transparently to PHP developers. One part of it is implementing periodic resend. The original code reporting failures to the user is replaced by a loop repeating requests on timeouts until the underlying *libcurl* function *curl_easy_perform* returns the success return code *CURL_OK*. When *curl_easy_perform* needs to be retried, we use a new copy of the previous CURL handle containing the same URI and the other original request settings while destroying the old one. Otherwise, *libcurl* would try to recycle its existing sockets, which saves resources and is right with regard to intact connections. However, the socket that timed out is likely to belong to a dead TCP connection and therefore should not be used for retry to avoid blocking.

Since *curl_exec* incurs sending a request message to a different EOS-PHP Web Service, we need to keep track of when the CIC interaction is installed by this counterpart, such that we can move on with the garbage collection at our own discretion. This is done via the volatile **output message lookup table (OMLT)** containing the URI invoked, our

MSN, and the current CIC status. We currently assume that all PHP scripts belonging to the *same Web Application* are stored in *the same Web server directory*. In a B2B Web application we assume that a reply to calling */auctions/bid.php* on behalf of one end-user and */auctions/search.php* on behalf of another user stems from the same EOS-PHP instance. Thus, we store only the URI paths without the script file names in the OMLT.

In terms of a CIC, we need to answer the question of when we have to force the log other than for LRU buffer management as described in Section 5.3.4. In our current prototype, EOS-PHP communication with the outside world is limited to returning HTTP replies and sending CURL requests. Thus, the log is forced prior to sending out these messages because otherwise the interleaving with other requests and nondeterministic values used would not be recoverable. Since we currently log an entire HTTP request, we implemented only ICIC for EOS-PHP. When a reply arrives to an EOS-PHP CURL client, it can mark the interaction as *cic_installed* in its OMLT. Furthermore periodic resend of the *curl_exec* request message is stopped.

Now we need a mechanism of getting rid of unneeded IMLT and OMLT entries. When we deal with a request form a single-threaded browser client, we know that as part of its XIC obligation, it has already installed *all* previous Web server replies. Thus, we can drop all entries stemming from the same client with MSN's lower than we see in the current request. Note that there are no OMLT records for browsers because we record only outgoing requests, but no replies. When we see a request from a multi-threaded B2B client with a particular MSN, it is not even guaranteed that we have already processed its previous requests. Thus, each B2B client includes into each request to a URI an additional cookie containing an *installed-MSN* with the following property: there is no OMLT entry with the same URI and MSN less than this installed-MSN whose CIC status differs from *cic_installed*. Thus, the B2B server is able to drop all entries for the given B2B client in its IMLT with MSN less or equal to installed-MSN of the B2B client. In turn, the B2B client can drop the OMLT entries with the B2B URI and the MSN less or equal to installed-MSN. Figure 36 depicts a scenario illustrating usage of IMLT and OMLT. A B2B client eosphp3 generates calls to applications eosphp1 and eosphp2 on behalf of end-users. When eosphp3 sends the next message to eosphp1, it includes installed-MSN 3 into this request and removes this entry from its OMLT because eosphp1 can garbage collect every interaction with eosphp3 having lower MSN's. As for eosphp2, the entry with the minimum MSN in eosphp3's OMLT is not marked *installed*

yet. Thus, the installed-MSN cookie is not included into the next request message sent from eosphp3 to eosphp2.

To implement this mechanism efficiently, we organize the OMLT as a hash tale that maps URI's to the interaction lists containing (MSN, CIC status) pairs. When the function *curl_exec* is called, it finds the proper list associated with the URI being called, traverses the list while remembering the *maximum installed MSN* seen so far until it finds the first pair belonging to an uninstalled interaction. This pair becomes a new head of the interaction list for the given URI, whereas the other traversed pairs are garbage collected. The new interaction with the CIC status set to *unknown* is appended to the end of the shortened list. The maximum installed MSN will be inserted as a cookie *installed-MSN* into the HTTP request built by *curl_exec*. The IMLT is implemented similar to a PHP two-dimensional array indexed by client id and MSN with the LSN of the corresponding HTTP reply as the content, i.e., as a hash table that is used to find by client id the hash table mapping HTTP request MSN's to HTTP reply LSN's. For reply recovery in Stage 3, we perform a simple lookup IMLT[clien id][msn]. For garbage collection when a new request arrives (Stage 4), we find a pointer to the hash table mapping request MSN's to reply LSN's as referred to by IMLT[client id]. Then we traverse this hash table as a list in the ascending MSN order (buckets in a Zend hash table are linked together as in Java and C# implementations) and throw away all entries with MSN less or equal to the submitted cookie *installed-MSN*.

Stage 5: HTTP Output Processing

When the execution of the request is finished, EOS-PHP updates the reply LSN field of the request entry in the IMLT. In the current prototype solution, the log entries with HTTP output do not require immediate log forcing, since these messages are recreated during deterministic replay. In fact, for *curl_exec* requests EOS-PHP does not even create a log entry with the content of the outgoing message, just the reply is logged to resolve recovery dependency, as you saw above. The point is that EOS-PHP is able to send out the HTTP reply messages prior to forcing them to stable log. Therefore, EOS-PHP can lazily force output messages (from several KB to several MB) that are orders of magnitude larger than preceding log entries of the same request whose sizes range from less than 256 bytes to some KB.

In addition, the browser recovery code that is always cached in the main memory is inserted into original HTTP replies to interactive clients between the opening tag <*html*> and the successor opening tag that is typically <*head*>. This does not have to be logged of course, which otherwise would further increase the size of the output message log entry.

5.3.2 Spinlocks and Latches

Since PHP usually runs in a multithreaded environment, parallel accesses to an identical resource need to be synchronized appropriately. The recovery log is the most frequently used shared resource of the EOS prototype. However, the session state may also be accessed in a parallel manner. First, when the GUI displayed in the Web browser consists of multiple frames, the browser normally loads their sources simultaneously. If the frame sources are PHP scripts using an identical PHP session, the session state will be read and written concurrently. Usually, most popular business Web sites (as e.g., Amazon.com, eBay.com, etc.) refrain from using multiple stateful frames and we can disregard this issue. Secondly and really relevant from our perspective, a session state may be shared by several clients as explained in Section 5.1.4.

Unfortunately, the PHP port for Windows does not offer an adequate means for concurrency control, whereas the UNIX implementation is not sufficiently flexible to distinguish read-only and write requests to allow more concurrency. We resolve these issues by implementing *spinlock*-based *latches* well-studied in the literature on database and operating systems [Gray and Reuter 1993, Silberschatz et al. 2002]. Latches are used when the system disallows by design **deadlock** situations (i.e., there are no cyclic lock waits among threads). In EOS-PHP, a typical request thread has the access pattern given in Table 8. The access pattern is constant except for occurrences of nondeterministic calls in the PHP script. To preserve the deadlock freedom, we cannot allow PHP developers to

Table 8: Request Access Pattern in EOS-PHP

Stage	Action	Latching Order
Stage 2 Stage 3	*session_start*(*$mode*)	latch Log (*X-mode*)
Stage 4		latch State buffer (*SH-mode*) latch State (*$mode*) unlatch Log unlatch State buffer
	session_close()	unlatch State
	a nondeterministic call	latch Log (*X-mode*) unlatch Log
Statge 4/5	*eos_force_log*	latch Log (*X-mode*) unlatch Log

```
01.  #define SPINLOCK_CLOSED 1L
02.  #define SPINLOCK_OPEN 0L
03.
04.  long globalspinlock = SPINLOCK_OPEN;
05.
06.  //call this for a blocking spinlock request
07.  __forceinline void eos_get_spinlock(ulong *lock)
08.  {
09.    long prior_lockstate;
10.
11.    while(TRUE)
12.    {
13.      prior_lockstate = InterlockedExchange(lock, SPINLOCK_CLOSED);
14.
15.      if(prior_lockstate == SPINLOCK_CLOSED)
16.      {
17.        Sleep(0); //yield and retry later
18.      }
19.      else
20.      {
21.        break; //done: the lock was open, current thread closed it
22.      }
23.    }
24.  }
25.
26.  //call this to release the spinlock; doesn't block
27.  __forceinline void eos_release_spinlock(long *lock)
28.  {
29.    *lock = SPINLOCK_OPEN;
30.  }
31.
32.  //sample spinlock usage: money transfer from acc1 to acc2
33.  void moneytransfer(long *acc1, long *acc2, long amount)
34.  {
35.    eos_get_spinlock(&globalspinlock)
36.    *acc1 -= amount;
37.    *acc2 += amount;
38.    eos_release_spinlock(&globalspinlock);
39.  }
```

Figure 37: Spinlock Implementation for Windows in *C*

make nondeterministic calls as long as the state latch is held, i.e., between the calls to *session_start* and *session_close*. Otherwise, we may run into a cycle where one thread (the state latch holder) cannot proceed with a nondeterministic call because the other thread processing the function *session_start* (the log latch holder) cannot release the log latch as long it does not have the state latch.

Typical mutual exclusion mechanisms such as *mutex locks* supported by operating systems provide first-come-first-served (FCFS) access to resources. They need to maintain a queue of lock requests to wake up the thread whose lock resides in the queue head, and to make the others yield. In contrast, a spinlock can be implemented using a

single shared Boolean variable with the values *true* and *false* meaning *locked* and *open*, respectively (lines 1 - 4 in Figure 37). To obtain a spinlock, the current thread has to call the function *eos_get_spinlock*. The gist of this function is that the test for spinlock availability and locking itself are performed without interleaving with concurrent threads to avoid the situation where the spinlock is granted to multiple requestors. Such an atomic test-and-set operation can be implemented either in the scheduler of the operating system or in the hardware. Through the intrinsic interface function *long InterlockedExchange*(*long *target, long value*) (lines 1 - 4 in Figure 37), Windows provides an access to the corresponding multiprocessor-safe instruction *LOCK CMPXCHNG* of the Pentium CPU. This function updates the 32-bit-segment pointed to by *target* with the new value and returns the previous one while preventing other threads (regardless on which processor) from accessing the *target* segment. Consequently, if the previous value of the lock equals SPINLOCK_CLOSED, the current thread returns control to some other thread (by calling *Sleep*(*0*)) and retries when rescheduled. In the jargon of operating systems, the busy-waiting thread is said to spin around the lock, hence the name. The spinlock is released by calling the function *eos_release_spinlock* that uses a simple *C* assignment (line 29) instead of atomic assignment *InterlockedExchange* because the locking thread has already an exclusive access to the *lock* flag. Since Apache assigns an identical *normal* priority to all request threads, there is no starvation of waiting threads. Lines 32-39 show a sample usage of the spinlock mechanism for an exclusive access to two account variables. Spinlocks are lightweight and extremely efficient when they are held for the duration of only a handful of instructions, which prevents high contention of concurrent threads. Their performance increases even further on a symmetric multiprocessor system, where the busy-waiting threads steal less CPU cycles from the locking ones that are thus able to release locks earlier. We use spinlocks to implement a more advanced locking mechanism coined latches.

Unlike spinlocks, latches (see Figure 38) can be used for both shared and exclusive accesses (SH-mode and X-mode, respectively). A latch is requested and released by calling the functions *eos_acquire_latch* and *eos_release_latch*, respectively. Latch requests in SH-mode can be admitted concurrently. To this end, we increment the counter of SH-latch holders *eos_latch.SH_count* (line 35) that is decremented again upon a latch release request (line 90). A shared latch request has to wait for the release of an X-latch. An X-mode latch request is not admitted while the latch is used by other threads in either mode. However, we allow a holder (*eos_latch.owner*) of a shared latch to upgrade it to

the X-mode when there are no other latch holders at this point (lines 47-52). If the latch upgrade is not possible, the thread should release its shared latch and issue a new X-latch request to prevent a deadlock of multiple upgrading threads. A latch downgrade from X-mode to SH-mode is always possible. This functionality is implemented by the function *eos_relatch*(*eos_latch *l*, *eos_latch_mode new_mode*) (not shown in Figure 38).

The latch request and the latch release routines both need an exclusive access to multiple fields of the structure *eos_latch* which is achieved using the spinlock *eos_latch.slock*. Shared latches are well suited to allow concurrent traversals of chained data structures.

```
01.  typedef enum _eos_latch_mode { SH_MODE, X_MODE} eos_latch_mode;
02.
03.  typedef struct _eos_latch
04.  {
05.    long slock;
06.    long SH_count;
07.    long X_count;
08.    long owner;
09.  }
10.  eos_latch;
11.
12.  //a blocking latch request
13.  void eos_acquire_latch(eos_latch *l, eos_latch_mode m)
14.  {
15.    long this_thread;
16.    zend_bool acquired;
17.
18.    if(EOS(in_replay) == TRUE)
19.    {
20.      return; // recovery is a single thread
21.    }
22.
23.    this_thread = tsrm_thread_id();
24.    acquired = FALSE;
25.
26.    while(TRUE)
27.    {
28.      eos_get_spinlock(&l->slock);
29.
30.      switch(m)
31.      {
32.      case SH_MODE:
33.        if(l->X_count == 0)
34.        {
35.          l->SH_count++;
36.
37.          if(l->SH_count == 1)
38.          {
39.            l->owner = this_thread;
40.          }
41.
42.          acquired = TRUE;
```

```
43.         }
44.         break; //done with SH-mode processing
45.
46.       case X_MODE:
47.         if(l->SH_count == 1 && l->X_count == 0 &&
               l->owner == this_thread) // latch upgrade
48.         {
49.           l->SH_count = 0;
50.           l->X_count = 1;
51.           acquired = TRUE;
52.         }
53.         if(l->X_count == 0 && l->SH_count == 0)
54.         {
55.           l->X_count = 1;
56.           l->owner = this_thread;
57.           acquired = TRUE;
58.         }
59.         break;  // done with X-mode processing
60.
61.       default:
62.         assert(m == SH_MODE || m == X_MODE);
63.       }
64.       eos_release_spinlock(&l->slock);
65.       if(acquired == FALSE) // latching failed => spin again
66.       {
67.         Sleep(0); // yield
68.       }
69.       else
70.       {
71.         break; // resources latched, go ahead
72.       }
73.     }
74.   }
75.
76.   // don't forget to release the latch
77.   void eos_release_latch(eos_latch *l)
78.   {
79.     long this_thread;
80.
81.     if(EOS(in_replay) == TRUE)
82.     {
83.       return; // recovery is a single thread
84.     }
85.     this_thread = tsrm_thread_id();
86.
87.     eos_get_spinlock(&l->slock);
88.
89.     if(l->SH_count > 0)
90.     {
91.       l->SH_count--;
92.
93.       if(l->owner == this_thread)
94.       {
95.         l->owner = 0;
96.       }
97.     }
98.
```

```
99.     if(l->X_count == 1)
100.    {
101.      l->X_count = 0;
102.      l->owner = 0;
103.    }
104.
105.    eos_release_spinlock(&l->slock);
106.  }
```

Figure 38: Latch Implementation for Windows in *C*

Special attention must be paid to handling of unexpected thread shutdowns triggered by failure management routines of PHP applications or the Zend engine due to some failures encountered at the level of a PHP statement. When an entire process (i.e., Apache) fails, the operating system is responsible for releasing resources such as allocated memory, file handles, TCP sockets, etc. In a multithreaded server, this task has to be implemented inside the server process. Zend provides this functionality in the form of the memory and resource variable managers. The memory manager of Zend is designed to keep track of resources allocated privately by a thread serving the given HTTP request using a special *zval* structure for bookkeeping. Thus, in order to benefit from Zend resource management, we allocate a *zval* container private to a thread, using the same mechanism that Zend would use to allocate a global PHP script variable (lines 26-27 in Figure 39) having a reference to the global latch. The function *eos_create_latch_resource* provides the calling thread with a *zval* handle that will be used for latching via Zend resource interface (lines 33-53).

During the server initialization, we introduce a new resource type for latches and associate it with a destructor function that in turn calls the function *eos_release_latch* if necessary (lines 5-13). In order to avoid erroneous unlatching of the resource that has never been latched by the current thread, we use the reference count field of the resource variable.

```
01.  eos_latch global_latch;
02.  long eos_latch_rsrc_type_id;
03.
04.  // latch resource variable destructor
05.  ZEND_RSRC_DTOR_FUNC(_eos_latch_cleanup)
06.  {
07.    eos_latch *l = (eos_latch *) rsrc->ptr;
08.
09.    if(rsrc->refcount > 0) // case > 1 implies an SH-latch upgrade
10.    {
11.      eos_release_latch(l);
12.    }
13.  }
```

```
14.
15.  // register destructor during server initialization
16.  void eos_register_latch_type()
17.  {
18.    eos_latch_rsrc_type_id =
       zend_register_list_destructors_ex(_eos_unlatch_if_needed, …);
19.  }
20.
21.  // obtain a handle during request thread initialization
22.  void eos_create_latch_resource(zval **latch_id, eos_latch *l)
23.  {
24.    zval *tmp;
25.
26.    MAKE_STD_ZVAL(tmp); // alloc a zval container
27.    tmp->refcount = 0;
28.    ZEND_REGISTER_RESOURCE(tmp, &global_latch, /* other params */);
29.    *latch_id = tmp; // out: a private handle of the global latch
30.  }
31.
32.  // latch function for resource variable interface
33.  void eos_latch_resource(zval **id, eos_latch_mode m)
34.  {
35.    eos_latch *l;
36.
37.    ZEND_FETCH_RESOURCE(l, (eos_latch *), id, /* other params */);
38.    eos_acquire_latch(l, m);
39.
40.    // need to unlatch upon request script-level failure
41.    zend_list_addref(Z_LVAL_PP(id));
42.  }
43.
44.  // unlatch function for resource variable interface
45.  void eos_unlatch_resource(zval **id)
46.  {
47.    eos_latch *l;
48.    ZEND_FETCH_RESOURCE(l, eos_latch *, id/* other params */);
49.    eos_release_latch(l);
50.
51.    // need to unlatch upon request script-level failure
52.    zend_list_delete(Z_LVAL_PP(id));
53.  }
```

Figure 39: Latches as PHP Resource Type Variables

Disregarding derived data structures such as the log and the state buffers, we have a single object, the PHP application state, for which the request synchronization is needed. With this coarse granularity (that does not allow concurrent accesses to different state variables), simple latching enforces the proper serialization of the request threads in a single EOS-PHP node. Certainly, it is impossible to achieve a request serialization in an arbitrary multi-tier PHP application that would be consistent throughout all the nodes (e.g., request 1 before request 2 at every EOS-PHP node) unless we would employ a sophisticated distributed concurrency control algorithm.

However, EOS-PHP is currently capable of handling the following realistic layered architecture. Web frontend nodes manage application states for individual end-users such as user profiles and shopping carts. End-users access them through a single-framed HTML-GUI, which ensures that no application state is accessed by more than one thread simultaneously. The EOS-PHP nodes at the backend layer manage a number of shared application states such as current auction bids. With each request the end-user accesses her private state at the Web frontend layer and a single shared object at the backend layer that enforces the proper serialization. Some users will just read the current auction state, the others will bid themselves.

5.3.3 Physical Organization of Stable Log

The stable log managed by EOS-PHP is stored as an ordinary file in Windows native file system NTFS. Its layout is shown in Figure 40. The log file begins with a boot sector that contains three 32-bit integer fields. The start position field stores the position at which the recovery manager has to start scanning the log. The start LSN field is used to store last used LSN when the corresponding log entry is garbage collected. The start MSN field contains the last MSN used by the CURL module to tag a request to another EOS-enabled Web Service in order to ensure the proper message duplicate detection after garbage collection. The space following the boot sector is used for storing variable-size log entries. Log entries have the format depicted in Figure 41. First nine bytes constitute the header of the log header including the log entry size, type (*HTTP*, *PHP*, or *state*), and the LSN. In addition, PHP-level log entries contain the LSN of the previous log entry created on behalf of the same HTTP request that may be either a PHP or an HTTP level log entry. HTTP and state log entries contain no back pointers. Information stored in the log entry body is formatted as Zend representation of PHP variables, so-called *zval* containers.

The purpose of HTTP log entries is to store the input of an HTTP request. A PHP level log entry normally stores a nondeterministic return value. The very last PHP log entry stores the complete HTTP reply including the reply header and body. State log entries are normally used to create installation points and to store LRU buffer replacement victims. The LSN field of a state log entry reflects the last HTTP request having updated the state as of the time the state log entry is created as you have seen in Section 5.3.1 above.

In order to ensure the recovery correctness, we need to make sure that log entries are written to disk atomically, i.e., the log file must not contain incomplete log entries. State-

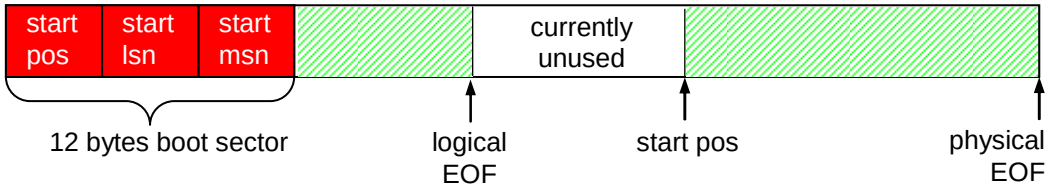

Figure 40: Layout of EOS-PHP Log File

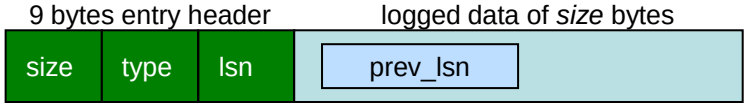

Figure 41: Log Entry Format

of-the-art hard disks support atomic writes of data that fits into a single sector (256 to 512 bytes). Request and state log entries are certainly much bigger since they may occupy from several kilobytes to some megabytes. This, however, does not pose a problem on NTFS due to its transaction support that is based on sufficiently small log entries. In the operating system jargon such a file system is called **journaled** [Silberschatz et al. 2001]. Some file systems offer journaling just for metadata. Windows NTFS logging service, IBM JFS, and Linux ext3 support all file operations. To benefit from NTFS journaling, output to the log file is performed using the low-level I/O functions *_write*(*file_handle, data_buf, data_size*) and *_commit*(*file_handle*). By *committing* the log file after a series of *writes* that encompasses one or more *complete logical units* such as one or more complete log entries, we always keep the log file in the consistent shape. When an error is encountered during an I/O (i.e., an I/O function returns -1), we make the Apache child process crash, and NTFS undoes uncommitted writes.

The log file has a particular size as configured by an administrator, in order to save the cost of claiming new disk sectors from the file system when appending a new log entry. Hence, EOS-PHP maintains its own logical end-of-file (EOF) represented by a signature of four zeroed bytes. Writing the logical EOF always completes an I/O transaction as the very last operation before *_commit*. Then the *log writer pointer* is moved to the real end of the very last log entry such that the EOF signature is overwritten by the successive I/O transaction. When a write operation reaches the physical EOF as managed by the file system, EOS-PHP continues writing from the end of the boot sector of the log file. A log file written in this manner is known as a **ring file**. The ring log file is full when a pending write would move beyond the current start position. Note that the start position is also continuously moved forward in the process of garbage collection with the advancing *minimum redo LSN*. Thus, log truncation does not incur a physical log file truncation on

disk Rather the start position field of the boot sector is updated. When the space in the log file is exhausted, EOS-PHP tries to double the size of the log file and copies the wrapped part of the log between the boot sector and the logical EOF to the area starting with the prior physical EOF inside a single I/O transaction. A log-file-shrinkage occurs in the process of garbage collection if the log data does not wrap and occupies less than a specified percentage (e.g., 10 %) of the physical log file size. To this end, the logical content of the log file is moved back to the end of the boot sector and the start position in the boot sector is updated within the same I/O transaction. Changing the log file size incurs updating the log file position field of the log and state buffer cells. Let *log_file_size* be the size of the log file and *log_file_pos* the offset of an entry **before** the change of the log file size. When the log file grows, the new offset of a moved log entry *log_file_pos'* is given by *log_file_pos* + (*log_file_size* - *boot_sector_size*). When the log file shrinks, the offset *log_file_pos'* of *every* log entry changes to *log_file_pos* - (*start_pos* - *boot_sector_size*).

5.3.4 LRU Buffers for PHP Session Data and the Log

Unlike the original PHP session module, EOS-PHP uses only the log to store PHP session data, which allows limiting interactions with hard disk to solely sequential I/O in most situations. The *PHP application state* and *log buffers* in the main memory with configurable size limits accelerate access to the data stored in the stable log, which is also novel to the original PHP session module that does not cache session data. Distinction between these two buffers arises from the different natures of their usage. The log buffer is normally accessed sequentially in the LSN order and occasionally through a hash table by an LSN as a key. The PHP application state buffer is usually accessed through a hash table by a session id as a key and occasionally by a sequential scan (e.g., during replacement victim selection and creation of installation points). During recovery, the state buffer may be looked up by a key pair (*session id*, *LSN*), which returns the youngest available version of the PHP application state as of *LSN*. All this functionality is provided with the EOS-enhanced Zend hash table implementation that serves as a basic data structure of both buffers. EOS-PHP could not benefit from sophisticated Zend memory manager due to its orthogonal design goal of freeing memory cells allocated during the execution of a request upon termination of the request. Hence, EOS-PHP maintains the buffers separately and provides its own copy constructors for Zend objects to avoid deletions of logged values upon the request completion.

```
01.  typedef struct _eos_log_entry
02.  {
03.    long lsn;
04.    long log_file_pos;
05.    eos_log_entry *next_php
06.    long size;
07.    HashTable *val;
08.    zend_bool is_dirty; // used for only for PHP states
09.    eos_latch lock;     // used for only for PHP states
10.    long read_lsn;      // used for only for PHP states
11.  }
12.  eos_log_entry;
13.
14.  typedef struct _eos_cache
15.  {
16.    long size;
17.    long size_limit; // specified in the php.ini file
18.    HashTable buf;
19.  }
20.  eos_cache
21.
22.  eos_cache eos_state_buf, eos_log_buf;
23.
24.  void eos_select_cache_victims(eos_cache *e, long min_size)
25.  {
26.    eos_log_entry *victm = NULL;
27.    zend_bool commit_pending = FALSE;
28.
29.    while(e.size + min_size > e.size_limit)
30.    {
31.      eos_find_min_lsn(e→buf, &victim);
32.
33.      if(victim→dirty == TRUE)
34.      {
35.        eos_write_log_entry_to_disk(victim);
36.
37.        commit_pending = TRUE;
38.      }
39.
40.      eos_free_hashtable(victim→val);
41.
42.      e→size -= victim→size;
43.    }
44.
45.    if(commit_pending == TRUE)
46.    {
47.      eos_commit_physical_log();
48.    }
49.  }
```

Figure 42: EOS-PHP Log Buffer Management

Figure 42 shows the data structures EOS-PHP uses for LRU buffer management. The structure *eos_log_entry* implements an LRU buffer cell. It stores an LSN field, the log file position of the log entry needed to fetch it from disk in case of a **cache miss** as opposed to a **cache hit** when the *val* field is not *NULL*. The *next_php* field contains a

pointer to the PHP-level log entry created next after the current log entry within the same HTTP request. Clearly, the *eos_log_entries* of type *state* do not use the *next_php* field. The *size* field caches the current amount of space occupied by the log entry in order to save expensive computation of the size that includes recursive traversing of Zend hash tables. For the log entries representing PHP application states EOS-PHP maintains two further fields: an *is_dirty* flag and a latch. The structure *eos_cache* implementing the LRU buffers of EOS-PHP contains an incrementally managed *size* field, a configurable *size_limit* field, and a Zend hash table to access individual log entries. A graphical representation of the log buffer is depicted in Figure 35.

EOS-PHP implements the PHP application state buffer using the **LRU** algorithm selecting the *least recently used* entry (hence the name of the algorithm) as a *replacement victim*. The volatile read LSN stamp of the PHP application state serves as a measure of recency (the higher LSN the more recently the PHP session state has been accessed) as illustrated in the code fragment of Figure 42.

Since EOS-PHP guarantees state recovery through deterministic replay, there only two situations when the state has to be forced to disk: When a particular PHP application state is replaced in the buffer by the LRU algorithm, and when EOS-PHP creates an installation point. Log entries for PHP application states are also written in a write-once manner, which implies that EOS-PHP never updates a log entry physically. Instead a new physical log entry is appended with the effect that the minimum redo LSN for the given PHP application state advances to the LSN of the newly created log entry thus enabling garbage collection of the former physical installation of this state. Note that prior to forcing a new state version to disk the log entries must have been forced to disk. They are copied rather than removed. We incrementally maintain the *last forced LSN* in order to avoid redundancy.

The log buffer of EOS-PHP is implemented in a similar manner with minor exceptions. When the log buffer is not able to accommodate a new entry, it first tries to free a sufficient amount of space by dropping log entries belonging to installed interactions. When the garbage collection of these log entries has not sufficed, LRU is used to select further log entries as replacement victims analogously to the PHP application state buffer.

5.3.5 Failure Detection

There are five classes of detectable system failures from our perspective. The first class includes heavy failures such as power outages that are either handled by the *uninterrupted power supply* hardware or by waiting until the system boots again and the Apache child process can recover pre-crash interactions. Heavy operating system kernel exceptions, causing a reboot analogously, constitute the second class of system failures. Apache process-level exceptions, internal or originating in modules other than PHP, causing restart of Apache processes alone fall into the third class of system failures. The fourth class of system failures originates in and is detected by the Zend engine that terminates only the current request thread. Last but not least, the fifth class of system failures is treated by EOS-PHP. In this subsection, we are going to discuss the last two failure classes.

Whenever EOS-PHP encounters a failure that affects the newly introduced log, LRU buffer, or recovery managers, a soft-crash is issued by calling the *abort()* routine that causes an abnormal termination of the Apache child process running the PHP engine as one of its modules. This is usually encoded as consistency assertions of successful return codes as e.g.: *assert*(*seek*(*log*, *l→log_file_pos*) != *-1*). Typical sources of severe failures that EOS-PHP checks for are I/O operations, memory allocations, and pointer arithmetic. Before a log entry (request or application state) is written to disk, EOS-PHP verifies that it is in a consistent shape having the size matching the cached value of the *size* field of the *eos_log_entry* structure. Aborting the server guarantees that no inconsistent entries can make it to stable log as ensured by I/O transactions. In order to clearly isolate EOS-PHP failures, additional measures are needed to protect the shared EOS data structures in the main memory from erroneous manipulation through straying pointers by other modules. By compiling EOS enhancements to PHP as a stand-alone Windows *dynamic link library* (DLL) that allows to access memory only through the *exported* functions, we can use a Windows mechanism to prevent other modules from manipulating EOS-specific data directly (similar to the process-level address space protection).

The Zend engine distinguishes different levels of failures depending on which part of the PHP runtime is affected and a failure severity. Failure reports are often included into regular HTML output for the end-user. If a failure is severe, the execution of the PHP script is halted. For instance, the Zend engine produces a warning when a script is trying to read an undefined variable without aborting the script. However, attempts to write to a

file using an invalid handle, compilation errors of dynamically included scripts are real hard failures causing script aborts. Note that since in the original PHP engine, script threads do not share memory they can be safely aborted without affecting parallel threads. Thus, since it is not the EOS-PHP routine causing a failure, such a drastic response as shutting down the entire server process is not necessary.

There are two possible situations to consider from the EOS perspective: 1) the failure is detected by the PHP script itself, 2) the failure is detected inside a PHP statement either by the Zend engine or by one of the PHP modules. In the first situation, we are dealing with a so-called **user-level** failure in the PHP jargon caused by an erroneous user input. Since the developer detected this situation, she provides an appropriate output allowing the end-user to correct her input. EOS treats such interactions as failure-free. In the second case however, we are dealing with another transient system failure according to our failure model. EOS-PHP initially suppresses the output to the client and awaits it to resend the request as required by CIC and starts the *retry counter* for this interaction. If the retry counter exceeds the configurable value (the default value is 3), EOS-PHP notifies the client component (a browser or the CURL module of another PHP server) that the current HTML output containing the error message is neither *stabilizing* nor *installing* its original request. The output is not added to the log buffers, neither by EOS-PHP nor by the client component. CURL module ignores this mechanism and simply keeps retrying. Such an error report is rather for convenience of the end-user. She sees that the system is not working smoothly right now, but she will be able to retry again by revisiting the greeting page of the Web Service. The exactly-once-execution guarantee will hold in any event.

5.3.6 Recovery: Analysis and Redo Passes

The analysis pass of the log is performed to initialize the log and PHP application state buffers using *eos_log_entry* elements defined in Figure 42. Just compact log entry headers (32 bytes each) are fetched into main memory, whereas the large *val* fields are skipped, in order to avoid unnecessary buffer cell replacements in this pass.

As for the log buffer, EOS-PHP needs to restore the forward links between the log entries of the same HTTP request using the *previous LSN* field stored with the log entries. As for the state buffers, the log file positions of all available versions of every PHP application state have to be inserted because an HTTP request can be correctly replayed only when it

gets the proper version of the application state, i.e., that is identical to the original execution. If we managed just a single, current version during recovery, replay of an incomplete HTTP request could be confused by an application state version forced by LRU buffer cell replacement after the request end. Log entries of a partially executed HTTP request are usually appended to the stable log for the following two reasons: due to nondeterministic events such as CIC commit of a concurrent request or flushing from the full log buffer.

In an unlikely case, in which EOS-PHP is not able to accommodate every *eos_log_entry* header in the main memory buffers, two B-tree files are generated during the analysis pass: one indexing request log entries using LSN as a key and an other indexing state log entries using a (*state id*, *LSN*) pair as a key (this feature is not implemented yet). Another answer to this problem could be a garbage collection policy that does not allow a number of active interactions that is greater than the number of log entry headers that fit into the log buffer. As an example, the log buffer of 8M can accommodate more than 250.000 log entry headers. When the maximum number is exceeded, EOS-PHP should not admit further requests until it receives a sufficient number of interaction installation notifications from other components.

During normal operation, EOS-PHP concurrency control ensured that the LSN order of HTTP requests coincides with the logical order of accesses to PHP application states. Thus, the redo pass simply replays all HTTP log entries starting with the minimum LSN encountered in the log (which is always an HTTP log entry) using a single thread as sketched in Figure 43. EOS-PHP speeds up the redo pass by checking in the request startup phase whether a replay is really needed (line 19). When the current LSN of the affected PHP application state is higher than the request LSN and the HTTP log entry is complete (i.e., the PHP level log entry with the reply message can be reached via the *next_php* chain), the HTTP request log entry is skipped for replay. When the request has to be replayed, EOS-PHP restores its execution environment including main global variables *$HTTP_{GET,POST,COOKIE,SESSION}_VARS* and *$HTTP_POST_FILES* that constitute PHP language representation of the HTTP request. A proper version of the PHP application state *$HTTP_SESSION_VARS* is recovered when the statement *session_start* is replayed that ultimately translates to calling the *C* function *eos_fetch_state_as_of*(*lsn*, *&app_state*, *&eos_state_buf*).

```
01.  eos_cache eos_state_buf, eos_log_buf;
02.  long min_lsn;
03.
04.  void eos_replay_log()
05.  {
06.    long curr_lsn = min_lsn;
07.    long sess_id;
08.    eos_log_entry *rle, *app_state;
09.    char *http_reply;
10.    char *script_file_name;
11.
12.    while(eos_fetch_log(&rle, curr_lsn, &eos_log_buf)!= END_OF_LOG)
13.    {
14.      sess_id = eos_log_extract_sessid(rle);
15.      http_reply = eos_log_extract_reply(rle);
16.
17.      eos_fetch_current_state(&app_state, &eos_state_buf);
18.
19.      if(http_reply == NULL || app_state→lsn < curr_lsn)
20.      {
21.        script_file_name = eos_log_extract_filename(rle);
22.
23.        // recover $HTTP_{COOKIE, GET, SESSION}_VARS
24.        eos_recover_globals(rle, app_state);
25.
26.        zend_execute_scripts(script_file_name, …);
27.      }
28.      else
29.      {
30.        // NOTHING TO DO
31.      }
32.
33.      curr_lsn = find_next_http_lsn ();
34.    }
35.  }
```

Figure 43: EOS-PHP Redo Pass

5.3.7 Installation Points and Garbage Collection

During normal operation the portion of the log that would need to be replayed after a crash steadily increases due to uninstalled interactions with other components outside our control and because most popular PHP application states mostly remain in the session state buffer without being flushed to disk. In addition, as explained above, browser recovery requires remembering the first URI_0 called to start a PHP session. If we naively always tried to retrieve URI_0 from a regular log entry, we would have to keep the first log entry until the session ends. To avoid this, EOS-PHP maintains the *session-id-*URI_0 table as part of a pseudo PHP application state that does not belong to any particular PHP session. This allows us not only an efficient lookup of an initial URI but also releasing of

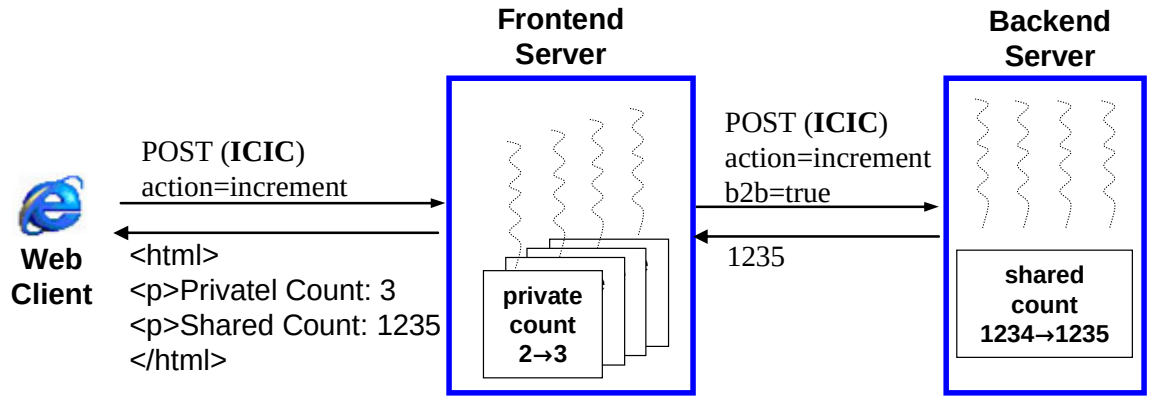

Figure 44: Test Application in the Experiments

log entries for garbage collection before the session ends by forcing the state with the *session-id-URI$_0$* table as a *state-type* log entry to disk.

The procedure of lazily creating installation points and subsequent garbage collection runs in a separate *installation thread* as an infinite loop that is repeatedly resumed after a configurable amount of time.

The installation thread interrupts the normal operation of EOS-PHP by acquiring a special *activity latch* in the *X-mode*. Usual requests obtain this latch in the *SH-mode* before the actual execution and hold it until completion. After successful exclusive latching, it is safe to reorganize the log, and the state of EOS-PHP is consistent because previously running requests are guaranteed to have been completed whereas arriving requests are being queued for later admission to execution.

The loop body of the installation thread consists of the following phases. First a new *minimum redo LSN* is determined by finding the oldest uninstalled interaction with other components in the input message lookup table IMLT. If the minimum redo LSN has not increased since the last check, it is not worthwhile to proceed with the installation such that further actions are skipped and the *activity latch* is released. Otherwise, we proceed with the next phases each of which corresponds to an NTFS I/O transaction. EOS-PHP forces the request log to disk, in the next phase. Subsequently, EOS-PHP flushes dirty PHP application states from the state buffer. The pseudo PHP application state maintaining the *session-id-URI$_0$* table is flushed to disk regardless of its dirty-status in the same phase unless it is empty. Last but not least, the boot sector of the log file is updated with the new start position for the analysis pass and last used LSN and MSN. Note that in rare cases in which all PHP sessions have been terminated and all CIC instances have been installed, the start position field of the boot sector will point to the logical EOF.

5.3.8 Run-Time Overhead

EOS-PHP extensions are implemented in the *C* language and comprise ca. 5500 lines of source code. For the implementation we used as much of the existing efficient Zend engine infrastructure as possible. To evaluate the run-time overhead of EOS-PHP, we performed measurements with Apache/1.3.20 and PHP/4.0.6 running on two PC's each with a 3 GHz Intel Pentium IV processor and 1 GB main memory under Windows XP.

```
01.  <?php
02.  $time = time();
03.  if(isset($HTTP_POST_VARS["b2b"]))
04.  {
05.    session_id("SHAREDSTATE");
06.  }
07.
08.  session_start(false);
09.
10.  if(isset($HTTP_POST_VARS["destroy"]))
11.  {
12.    session_destroy();
13.    exit();
14.  }
15.
16.  if(!session_is_registered["count"])
17.  {
18.    session_register("count");
19.  }
20.
21.  $HTTP_SESSION_VARS["count"]++;
22.  session_close();
23.
24.  if(!isset($HTTP_POST_VARS["b2b"]))
25.  {
26.    $ch = curl_init("http://b2b_server/test.php");
27.    curl_setopt($ch, CURLOPT_HEADER, false);
28.    $params = array();
29.    $params["b2b"] = true;
30.    curl_setopt($ch, CURLOPT_POSTFIELDS, $params);
31.    $b2b_reply = $curl_exec($ch);
32.  }
33.  else
34.  {
35.    echo $HTTP_SESSION_VARS["count"];
36.    exit();
37.  }
38.  ?>
39.  <html>
40.    <p>Private count:
41.            <?php echo $HTTP_SERVER_VARS["count"]; ?>
42.    <p>Shared count: <?php echo $b2b_reply; ?>
43.  </html>
```

Figure 45: Test PHP Script

The call structure of the evaluated application is shown in Figure 44. The load on the frontend Web application server was generated by the synthetic HTTP request generator *Apache JMeter/2.0.3* [Apache.org]. The generator simulated conversations of *n* steps without involving human user interactions. Think times were not simulated.

Both servers deploy the same PHP script outlined in Figure 45. The script calls the nondeterministic function *time()* (line 2), reads the current application state (line 8) and increments the state variable *count* (lines 16-21). On the backend server that receives the flag *b2b* as a POST parameter, the accessed state is shared among all clients as specified by the explicit call to the function *session_id* (line 5). The new value of the state variable *count* is the only content of the HTTP reply body produced by the backend server (lines 35-36). In contrast, the frontend server accesses a private state to increment the variable *count* and invokes another instance of this script on the backend server (lines 24-32). Moreover, the frontend server replies with a complete HTML page containing the shared and private *count* values (lines 39-43) that is returned to the load generator. In the following two experiments we compare the three-tier system of two servers run by the original PHP engine against the equivalent system run by EOS-PHP.

Table 9 shows the total elapsed time, between the first request and the last reply as seen by the client, and the CPU time on the frontend and backend servers for *n = 1, 5, 10* steps, comparing the original PHP engine to EOS-PHP with the rigorous recovery guarantees. The original PHP manages each session in a separate file. Changes to the session variables are made quasi persistent by the original PHP because the session file is written without being forced to disk. The function *_close(int filehandle)* called by the original PHP engine at the request end does not incur a synchronous I/O on Windows. The response time overhead of 135-152% results also from the fact that the original PHP sends the reply before the disk write, the latter being performed during the request

Table 9: 1 Client Experiment

Sessions	1 step	5 steps	10 steps
PHP elapsed time [sec]	0.0480	0.2200	0.4500
EOS-PHP elapsed time [sec]	0.1130	0.5550	1.1000
Overhead [%]	135%	152%	144%
PHP frontend CPU time [sec]	0.0240	0.1625	0.3455
EOS-PHP frontend CPU time [sec]	0.0305	0.2125	0.4636
Overhead [%]	27%	31%	34%
PHP backend CPU time [sec]	0.0050	0.0300	0.0700
EOS-PHP backend CPU time [sec]	0.0090	0.0550	0.1200
Overhead [%]	80%	83%	71%

shutdown. The CPU time overhead on the frontend server is lower than on the backend server by a factor of approximately two due to the file (de)allocation activity of the original PHP when starting new sessions and terminating the old ones by calling the function *session_destroy* (line 12 of Figure 45) on the frontend server, whereas a single file is used all the time on the backend server. Nevertheless, we need to mention that the cost for forced I/O's dominates the overhead because inserting the function *_commit* before the call to *_close* in an additional test made the overhead shrink to less than 20%.

We also performed multi-user measurements by replicating the HTTP request driver on five different client machines generating requests to the frontend server. Table 10 shows the measured average response and CPU times in terms of the simulated *n*-step user sessions. The figures show that the response time overhead decreases in comparison to the one client measurement because concurrency becomes a more significant factor. Although the original PHP does not need concurrency control, the Apache Web server manages a number of shared data structures that are protected by internally implemented *semaphores* and *mutex locks* that increasingly suffer access contention. In this experiment, the overhead on the frontend server is larger than on the backend server due to a higher contention on the log latch which protects two synchronous log writes: one before calling *curl_exec* and another performed prior to sending the output to the client.

The cost for the recovery guarantees is less than factor of two, which is an acceptable overhead. The price is worthwhile given the increased dependability and ease of programming.

Table 10: 5 Clients Experiment

Session	1 step	5 steps	10 steps
PHP elapsed time [sec]	0.1560	0.7900	1.6100
EOS-PHP elapsed time [sec]	0.3140	1.6850	3.1000
Overhead [%]	101%	113%	93%
PHP frontend CPU time [sec]	0.0390	0.2708	0.5727
EOS-PHP frontend CPU time [sec]	0.0815	0.6000	1.1545
Overhead [%]	109%	122%	102%
PHP backend CPU time [sec]	0.0090	0.0550	0.1200
EOS-PHP backend CPU time [sec]	0.0130	0.0750	0.1600
Overhead [%]	44%	36%	33%

6. Conclusion and Outlook

"Real generosity towards the future lies in giving all to the present." -Albert Camus

This thesis has introduced a formal specification of the interaction contract framework in the form of *Statemate* state-and-activity charts. The generic design of the three types of interaction contracts allows a rapid and rigorous specification of complex multi-tier software system architectures. This thesis presents automatic proofs that the formal specifications have the required property of exactly-once execution using the symbolic model checker provided with *Statemate*. Although we were successful in verifying of standalone interaction contracts and we were able to verify a sample Web application model with nondeterminism caused by the parallel asynchronous execution, we observed that the model checker performance does not scale with the number of modeled users. Providing proofs for realistic systems handling hundreds of parallel user session were out of question. The results achieved in this thesis show that mechanical verification technology still requires seeking a compromise between the verifiability and the realism of a model for a complex software system. A promising direction for future work may lie in the combination of model checking with induction proofs (e.g., see the paper by McMillan et al. [2000]).

The second major accomplishment of this thesis is a transparent integration of the interaction contract support with the popular real-world Web technology products: Microsoft's browser Internet Explorer and Zend's server-side scripting engine for the PHP language. Our prototype EOS allows deploying arbitrarily distributed PHP application with the exactly-once execution guarantee. Good performance has been achieved due to efficient log and state data organization, and the LRU buffer management added to PHP by EOS. Experiments show that the rigorous recovery guarantees are provided with acceptable overhead. Performance of EOS-PHP can be further improved by deeper integration of the log and recovery managers with the Zend engine, which would allow replaying the log at the PHP statement level rather than at the HTTP request level. This could make the analysis pass in the current solution obsolete, and make the redo pass simpler. Moreover, such a solution would allow creating installation points in midst of request execution, which would accelerate replaying lengthy scripts during redo. Providing a distributed concurrency control protocol for PHP would be another interesting direction to pursue in future work.

References

Alvisi, L. and K. Marzullo, 1995: *Message Logging: Pessimistic, Optimistic, and Causal*, In Proceedings of the 15th International Conference on Distributed Computing Systems, Vancouver, Canada, May 30 - June 2, 1995, IEEE Computer Society, Los Alamitos, CA, U.S.A., 229-236..... 39

Apache.org: *Apache HTTP Server Project*, http://httpd.apache.org/..... 77, 117

Barga, R., D. Lomet, and G. Weikum, 2002: *Recovery Guarantees for General Multi-Tier Applications*, in Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, U.S.A., February 26 - March 1, 2002. IEEE Computer Society, Los Alamitos, CA, U.S.A., 543-554..... 3, 38, 51, 58

Barga, R., D. Lomet, G. Shegalov and G. Weikum, 2004: *Recovery Guarantees for Internet Applications*, ACM Transactions on Internet Technologies (TOIT) 4(3), 289-328..... 3

Barga, R., D. Lomet, S. Agrawal, and T. Baby, 2000: *Persistent Client-Server Database Sessions*, in Proceedings (Lecture Notes in Computer Science, 1777) of the 7th International Conference on Extending Database Technology, Constance, Germany, March 2000, Springer, Heidelberg, Germany, 462-477... 38

Barga, R., D. Lomet, S. Paparizos, H. Yu, and S. Chandrasekaran, 2003: *Persistent Applications via Automatic Recovery*, in Proceedings of the 17th International Database Engineering and Applications Symposium, Hong Kong, China, July 2003. IEEE Computer Society, Los Alamitos, CA, U.S.A., 258-267... 4

Barga, R., S. Chen, and D. Lomet, 2004: *Improving Logging and Recovery Performance in Phoenix/App*, in Proceedings of the 20th International Conference on Data Engineering, 30 March - 2 April 2004, Boston, MA, U.S.A., IEEE Computer Society, Los Alamitos, CA, U.S.A., 486-497..... 4

Bartlett, J., 1981: *A NonStop Kernel*, in Proceedings (Operating System Review 15(5)) of the 8th Symposium on Operation Systems Principles, Asilomar, CA, U.S.A., December 1981, ACM, New York, NY, U.S.A., 22-29..... 39

Borg, A., W. Blau, W. Graetsch, F. Herrmann, and W. Oberle, 1989: *Fault Tolerance Under UNIX*, ACM Transactions on Computer Systems,7(1), 1-24... 39

Bryant, R., 1986: *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Transactions on Computers, 35(8), 677-691..... 11

Castro, M. and B. Liskov, 1999: *Practical Byzantine Fault Tolerance*, in Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation, February 22-25, 1999, New Orleans, Louisiana, Operating Systems Review, Special Issue, ACM, NY, U.S.A., 1998 and USENIX Association, 173-186... 26

Chkliaev, D., J. Hooman, and P. van der Stok, 2000: *Mechanical Verification of Transaction Processing Systems*, in Proceeding of the 3rd IEEE International Conference on Formal Engineering Methods, September 4-7, 2000, York, England, U.K., IEEE Computer Society, Los Alamitos, CA, U.S.A., 89-97..... 40

Clarke, E. and B. Schlinghoff, 2001: *Model Checking*, in Handbook of Automated Reasoning, Volume 2, Elsevier and MIT, 2001, 1635-1790... 6

Comer, D., 1988: *Internetworking with TCP/IP Principles, Protocols, and Architecture*, Prentice Hall, Englewood Cliffs, NJ, U.S.A., 1988... ..... 26, 76

Cristian, F., 1991: *Understanding Fault-tolerant Distributed Systems*, in Communications of the ACM, 34(2), 56-78... 39

Dutta, K., D. VanderMeer, A. Datta, and K. Ramamritham, 2001: *User Action Recovery in Internet SAGAs (iSAGAs)*, in Proceedings (Lecture Notes in Computer Science 2193) of the 2nd International Workshop on Technologies for E-Services (TES), Rome, Italy, September 2001, Springer, Heidelberg and Berlin, Germany, 132-146... 39

Elnozahy, E., L. Alvisi, Y. Wang, and D. Johnson,, 2002: *A Survey of Rollback-Recovery Protocols in Message-Passing Systems*, ACM Computing Surveys, 34(3), 375-408... 39

Emerson, E., 1990: *Temporal and Modal Logic*, in Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics, Elsevier and MIT, 1990, 995-1072... 5

Freytag, J., F. Cristian, and B. Kähler, 1987: *Masking System Crashes in Database Application Programs*, in Proceedings of the 13th International Conference on Very Large Data Bases, Brighton, U.K., September 1987, Morgan Kaufmann, San Francisco, CA, U.S.A., 407-416..... 38

Frølund, S. and R. Guerraoui, 2002: *e-Transactions: End-to-End Reliability for Three-Tier Architectures*, IEEE Transactions on Software Engineering, 28(4), 378-395... ... 39

Fu, X., T. Bultan, R. Hull, and J. Su, 2001: *Verification of Vortex Workflows*, in Proceedings (Lecture Notes in Computer Science 2031) of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Genoa, Italy, April 2001, Springer, Berlin and Heidelberg, 143-157..... 39

Fujita, M., Y. Matsunaga, , and T Kakuda, 1991: *On Variable Ordering of Binary Decision Diagrams for the Application of Multi-Level Logic Synthesis*. In Proceedings of the European Conference on Design Automation, Amsterdam, Netherlands, March 1991, IEEE Computer Society, Los Alamitos, CA, U.S.A., 50-54.... 12

Gray, J. and A. Reuter, 1993: *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Francisco, CA, U.S.A., 1993..... 25, 27, 34, 46, 99

Gurevich, Y., N. Soparkar, and C. Wallace, **1997**: *Formalizing Database Recovery*, in the Journal of Universal Computer Science 3(4), 320-340... 40

Hadzilacos, V., 1988: *A Theory of Reliability in Database Systems*, Journal of the ACM 35(1), 121-145... 40

Harel, D. and M. Politi, 1998: *Modeling Reactive Systems with State-charts: The Statemate Approach*, McGraw-Hill, New York, NY, U.S.A., 1998..... 12

Harel, D., and A. Naamad, 1996.: *The STATEMATE Semantics of Statecharts*, ACM Transactions on Software Engineering and Methodology, 5(4), 293-333..... 12

Henderson, K., 2000: *The Guru's Guide to Transact-SQL*, Addison-Wesley Professional, Boston, MA, U.S.A., 2000..... 2

Huang, Y. and Y. Wang, 1995: *Why Optimistic Message Logging Has Not Been Used In Telecommunications Systems*, in Proceedings of the 25th International Symposium on Fault-Tolerant Computing Systems, Pasadena, CA, U.S.A., June 1995, IEEE Computer Society, Los Alamitos, CA, U.S.A., 459... 39

Internet Engineering Task Force, 1996: *Multipurpose Internet Mail Extensions*, RFC 2045-2049, http://www.ietf.org/... 78

Internet Engineering Task Force, 1998: *Uniform Resource Identifiers (URI): Generic Syntax*, RFC2396, http://www.ietf.org/... 75

Internet Engineering Task Force, 1999: *Hypertext Transfer Protocol – HTTP/1.1*, RFC 2616, http://www.ietf.org/... 75

Johnson, D. and W. Zwaenepoel, 1987: *Sender-based Message Logging*, in Proceedings of the 7th International Symposium on Fault-Tolerant Computing, Pittsburgh, PA, U.S.A., July 1987, IEEE Computer Society, Los Alamitos, CA, U.S.A., 14-19... 39

Katz, R., G. Gibson, and D. Patterson, 1989: *Disk system architectures for high performance computing*, in Proceedings of the IEEE, 77(12), 1842-1858... 26

Kim, W., 1984: *Highly Available Systems for Database Applications*, ACM Computing Surveys, 16(1), 71-98... 39

Korth, H., E. Levy, and A. Silberschatz, 1990: *A Formal Approach to Recovery by Compensating Transactions*, in Proceedings of the 16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, 95-106... 26

Kuo, D., 1996: *Model and Verification of a Data Manager Based on ARIES*, ACM Transactions on Database Systems 21(4), 427–479... 40

Lampson, D. and D. Lomet, 1993: A New Presumed Commit Optimization for Two Phase Commit, in Proceedings of the 19th International Conference on Very Large

Data Bases, Dublin, Ireland, August 1993, Morgan Kaufmann, San Francisco, CA, U.S.A., 630-640..... 33

Lomet, D. and G. Weikum, 1998: *Efficient Transparent Application Recovery in Client-Server Information Systems*, in Proceedings of 1998 ACM SIGMOD International Conference on Management of Data, Seattle, WA, June 1998, ACM, New York, NY, U.S.A., 460-471..... 38

Lomet, D. and M. Tuttle, 2003: *A Theory of Redo Recovery*, In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, CA, U.S.A., June 9-12, 2003, ACM, New York, NY, U.S.A., 397-406..... 40

Lomet, D., 1998: *Persistent Applications Using Generalized Redo Recovery*, in Proceedings of the 14th International Conference on Data Engineering, Orlando, FL, U.S.A., February 1998, IEEE Computer Society, Los Alamitos, CA, U.S.A., 154-163... 38

Luo, M. and C. Yang, 2001: *Constructing Zero-Loss Web Services*, in Proceedings IEEE INFOCOM 2001 of the 20th Joint International Conference of the IEEE Computer and Communication Societies on Computer Communications, Anchorage, AK, U.S.A., April 2001. IEEE, Los Alamitos, CA, U.S.A., 1781-1790..... 39

Lynch, N., 1996: *Distributed Algorithms*, Morgan Kaufmann, San Francisco, CA, U.S.A., 1996..... 40

Martin, C. and K. Ramamritham, 1999: *Recovery Guarantees in Mobile Systems*, in Proceedings of the ACM International Workshop on Data Engineering for Wireless and Mobile Access, Seattle, WA, U.S.A., August 1999, ACM, New York, NY, 22-29... 39

Martin, C., and K. Ramamritham, 1997: *Toward Formalizing Recovery of (Advanced) Transactions*, in Advanced Transaction Models and Architectures, Kluwer, Norwell, MA, U.S.A., 1997, 213-234... 40

McMillan, K., 1993: *Symbolic Model Checking*, Kluwer , Norwell, MA, U.S.A., 1993... 7

McMillan, K., S. Qadeer, and J. Saxe, 2000: *Induction in Compositional Model Checking*, in Proceedings of 12th International Conference on Computer Aided Verification (CAV), Chicago, IL, USA, July 15-19, 2000, Springer, Heidelberg, Germany, 2000, 312-327... 119

Meinel, C. and T. Theobald, 1998.: *Algorithms and Data Structures in VLSI Design - OBDD Foundations and Applications*, Springer, Heidelberg, 1998... 10

Microsoft: *Microsoft Developer Network Home Page*, http://msdn.microsoft.com/... 79, 86

Mohan, C., D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, 1992: *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*, ACM Transactions on Database Systems, 17(1), 94-162... 28, 40

Netcraft, 2004: *December 2004 Web Server Survey*, http://news.netcraft.com/... 77

OMG 2000: *Fault Tolerant CORBA Spec v1.0*, http://cgi.omg.org/cgi-bin/doc?ptc/00-04-04... 38

PHP.net: *PHP: Hypertext Preprocessor*, http//www.php.net/... 78

Popovici, A., H. Schuldt, and H. Schek, 2000: *Generation and Verification of Heterogeneous Purchase Processes*, in Proceedings of the 1st International Workshop on Technologies for E-Services, Cairo, Egypt, September 2000, 5-22... 39

Rudell, R., 1993: *Dynamic Variable Ordering for Ordered Binary Decision Diagrams*, in Proceedings of the 1993 IEEE/ACM International Conference on Computer-Aided Design, Santa Clara, CA, U:S.A., November 07 - 11, 1993, IEEE Computer Society, Los Alamitos, CA, U.S.A., 42-47... 12

Schuldt, H., A. Popovici, and H. Schek, 2000: *Automatic Generation of Reliable E-Commerce Payment Processes*, in Proceedings of the 1st International Conference on Web Information Systems Engineering, Hong Kong, China, June 2000, IEEE Computer Society, Los Alamitos, CA, U.S.A., 434-441... 39

SecuritySpace: *Apache Module Report*, http://www.securityspace.com/... 78

Silberschatz, A., G. Gagne, and P. Galvin, 2002: *Operating Systems Concepts*, Willey, Indianapolis, IN, USA... ..... 99, 107

Stenberg, D.: *cURL and libcurl*, http://curl.haxx.se/... 82

Strom, R., D. Bacon, and S. Yemini, 1988: *Volatile Logging in n-Fault-Tolerant Distributed Systems*, in Digest of the 18th Annual International Symposium on Fault-Tolerant Computing, Tokyo, Japan, June 1988, IEEE Computer Society, Los Alamitos, CA, U.S.A., 44-49... 39

Sun 2001: *Enterprise Java Beans Specification v2.0*, http://java.sun.com/products/ejb/docs.html... 38

Sun*: Java Technology*, http://java.sun.com/... 79

The Open Group, 1994: *Distributed TP: The XA+ Specification, Version 2*, http://www.opengroup.org/online-pubs?DOC=8095979699&FORM=PDF... 30

Tygar, J., 1998: *Atomicity versus Anonymity - Distributed Transactions for Electronic Commerce*, in Proceedings of the 24th International Conference on Very Large Data Bases, New York, NY, U.S.A., August 1998, Morgan Kaufmann, San Francisco, CA, U.S.A., 1-12..... 39

W3.org: *W3C:World Wide Web Consortium*, http://w3.org/... ..... 75, 76, 82, 83

WebSideStory, 2004: *Browser Trends Survey, Oct. 29th 2004*, http://www.entmag.com/news/article.asp?EditorialsID=6432... 83

Weikum, G. and G. Vossen, 2001: *Transactional Information Systems*, Morgan Kaufmann, San Francisco, CA, U.S.A., 2001... ..... 1, 33, 40, 46

Younas, M., and B. Eaglestone, 2002: *A Formal Verification Strategy for Crash Recovery in Web-Database Applications*, in Proceedings of the 3rd International Conference on Web Information Systems Engineering Workshops, Singapore, December 11, 2002, IEEE Computer Society, Los Alamitos, CA; U.S.A., 113-119... 40

Zend.com: *Zend Technologies, Inc. The PHP Company*, http://zend.com/... 78

Index