An Event Algebra Based System for Verifying E-Commerce Transactions

Renyi Zhao and V. Lakshmi Narasimhan
School of Electrical Engineering and Computer Science
University of Newcastle, NSW 2308, Australia
Email: narasimhan@cs.newcastle.edu.au
Tel.: +61 2 4921 6953
Fax: +61 2 4921 6929
URL: www.cs.newcastle.edu.au/~narasimhan

This paper describes extensions to a Behavioural Description Language (BDL), which was originally proposed to characterize concurrent behaviour of simple objects and a group of objects. One of the novelties of this paper is its application to the field of E-Commerce transaction systems. Based on the BDL, we propose new concepts, namely, transaction patterns and transaction architectures, which have event-based semantics to describe large-scale transaction systems. Furthermore, the transaction architecture is introduced as a unified medium for specifying and verifying distributed, heterogeneous and complex E-Commerce transaction processes. It is also illustrated as a powerful modeling technique which is easy-to-use, flexible and promotes re-usability.

ACM Classification: I.1.3 (Symbolic and Algebra Manipulation: Languages and Systems – special-purpose algebraic systems), D.2.4 (Software/Program Verification: Model Checking), K.4.4 (Computers and society; Electronic Commerce)

1. INTRODUCTION

Over the years, E-Commerce has become a rapidly expanding area leading to a significant increase in Internet based commercial transactions. A report for the United States Embassy on E-Business predicts (US Embassy Report, 2000) that by the year 2010, the net value of E-Commerce transactions will exceed $2 Trillion. E-Commerce is a part of E-Business (Adam et al, 1999) and it has several categories, such as (Chan et al, 2001), Business to Business (B2B), Business to Consumer (B2C), Customer to Customer (C2C), etc. A transaction system is the crucial part of an E-Commerce system. The development process of E-Commerce transaction systems can be a highly complex process, mainly because of the concurrent and distributed computing requirements of these systems. In order to prevent execution of conflicting operations, the management of such systems may also include multiple and heterogeneous transactions which involves complex workflows.

Workflow is defined by WFMC (Workflow Management Coalition) (Lawrence, 1997) as “The automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules”. The correctness of flow control can be guaranteed by a workflow management system which
emphasizes the task dependencies. Recently, a number of published papers put forward workflow modeling technologies and languages. For instance, Graphical modeling approaches are adopted in FlowMark (Alonso and Mohan, 1997), where vertexes and arcs are used to represent activities and controls. The Action Port Model, which is based on the DFD tradition, is described by Carlsen (1998). One of the drawbacks of these approaches is that there is no strict defined semantics. Formal approaches, such as Petri Nets, can also be used to model workflow. The main merit of using Petri Nets method is that large-scale workflows can be modeled and analyzed because it has a rigorous mathematical foundation (Coves et al., 1998; Yuhong and Bejan, 2001). However, the model developed may be too complex and unmanageable for an enterprise manager to handle. Techniques employed for workflow modeling can also be used for modeling E-Commerce transactions. Although Ngu et al. (1996) separated tasks and transactions to model the workflow environment while describing their CBORD system, most of the work in the workflow-modeling area demonstrates that there is no distinction between tasks and transactions. In other words, transaction management is basically similar to workflow management (Alonso et al., 1996). However, E-Commerce transaction systems have some of its own features (see Section 2 for more details). For instance, *rollback* is a transactional concept which helps system to recover from errors and exceptions. We have noticed that in Edelweiss and Nicolao (1998), the TF-ORM workflow-modeling technique was extended to represent the occurrence of failures and exceptions and define the information for system recovery, but there is a lack of a unified medium which can be used to model, verify, compile and manage distributed and heterogeneous E-Commerce transaction processes.

A Behavioural Description Language (BDL) introduced in this paper provides a good starting point to achieve these goals mentioned above. Bubler and Jablonski (1994) have described the behavioural element in their ABS Workflow Model, which controls the flow of execution and decides when executables can be performed. From the behavioural perspective, a condition method is first associated with all sub-executables of a composite executable. Then the condition method will be evaluated to determine the processing of sub-executables. The sequence, enabling conditions, constraints and possible outcomes were also described in the behavioural perspective in the DWM Modeling Framework developed by Kwan and Balasubramanian (1997). The BDL described in this paper is different from those employing activity-based methodologies in that our approach extends the BDL developed by Bertrand and Augeraud (1999), which has event-based semantics, for E-Commerce transaction processing. Further, the work addresses research issues that relate to automation technologies that can be applied to E-Commerce systems. We choose event-based BDL because it provides the following advantages: i) ease of modeling distributed, heterogeneous and complex business transaction processes, ii) flexibility to describe diversity in design and good expressive power to let analysts specify a transaction system easily, and iii) ready verification of the transaction processes through formal analysis.

The rest of the paper is organized as follows: Section 2 describes the problem space, while Section 3 provides the notations and the event-based semantics employed in our BDL. In Section 4, we introduce two new concepts, namely, transaction patterns and transaction architectures along with relevant examples. A complex example is demonstrated in Section 5, followed by a more in-depth presentation of the related research work in Section 6. The conclusion summarizes the paper and provides pointers for further research in this area. The Appendix provides a quick technical review of the concepts, mechanisms and models for constructing a transactional-based business.
2. BUSINESS TRANSACTIONS AND LONG TRANSACTIONS

A business transaction is built on the basis of business workflow protocols which implement a long transaction.

2.1 Long Transaction

Long transactions (also known as Sagas) are transactions that may take days, months or years before the final commit e.g., the transaction system may wait for information a user will input to complete the transaction. It has two major features: i) they involve multiple organizations participating in multi-steps transactions, and ii) there exist multi-rollbacks in the long transaction.

2.2 Amortization

It is possible to combine short-running ACID transactions into a long-running transaction. ACID transactions were first illustrated by Härder and Reuter (1983) and Gary and Reuter (1993), who define an ACID transaction as “a unit of operation on data and system resources, which are strictly governed by the four properties – Atomicity, Consistency, Independence, and Durability”. We have provided a brief description on the details of ACID transactions as an appendix. In order to model more complicated business transactions, the concept of local transaction is introduced, which is defined as a middle-level unit of work that encapsulates a single or a combination of ACID transactions to group the internal stage. It has two obvious advantages: i) according to the business logic, it defines a boundary for the recovery of partial failure of sub-ACID transactions and ii) non-conflicting local transactions can be executed concurrently. Figure 1 shows the relationship between the overall long transaction T and the local transactions: T1, T2, T3 and T4. T1 and T4 are local transactions that have only one ACID transaction (circles in Figure 1) separately. The ACID transaction may be flat, chained, nested or distributed transactions. T2 is a local transaction which comprises two ACID transactions in a certain sequence. T3 is a more complex local transaction which consists of two local transactions in a parallel structure.

A business transaction always involves a multi-step workflow structure defined by business rules. Each step of the workflow can be represented by the local transaction in an overall long transaction. The final commit of the overall business transaction can be reached through realizing the local transactions step by step. At the end of each local transaction, there is a save point which is recoverable (e.g., P1 in Figure 1).

Figure 1: An Overall Long Transaction with Local Transactions and ACID Transactions
2.3 Compliance and Synchronization

From the business transaction point of view, since it is a multi-organization transaction structure, the data passed need to be synchronized between parties. A business transaction will be finally committed, or we can say the business trade is completed, only when all the trade parties agree to implement the transaction according to the business rules. Here we use the concept of contract which acts as the agreement between business parties.

2.4 Compensation Scheme

If the entire business transaction or the local transactions fail, while the embedded ACID transactions have been committed, the effect of the ACID actions needs to be undone. As a consequence, compensation schemes are introduced when the transaction fails at a specific point. While ACID transactions always compensate through the rollback function, a business transaction creates compensation balance to reverse the results of business activities done previously – e.g., a business compensation to cancel the invoice sent to a customer is done by issuing another letter saying the invoice is not valid any more. Here the compensation is semantically to undo the result. In other words, this means that the compensation does not need to reverse to the state from which the local transaction begins. Because of the multi-rollback feature of the business transaction, the compensation scheme can be carried out at different levels, namely, at the ACID level, at the local transaction level, at the combination of the local transactions and at the overall transaction - the compensations C1, C2, C3 and C4 for the local transactions T1, T2, T3 and T4. If all of the local transactions commit and the overall transaction completes, the sequence will be: T1, T2, T3, T4 respectively as shown in Figure 1. If the local transaction T4 fails and the transaction needs to reverse the overall transaction, the sequence will be:

\[ T1, T2, T3, T4, C4, C3, C2, C1. \]

In addition to the scheme, here we introduce the concept of compensatory dependency (Warne, 1993). If one of the local transactions fails, the compensation sequence is carried out to reverse the effects of committed local transactions. The compensation sequence of the committed local transaction (predecessor) is held by the following local transactions (successor).

2.5 E-Commerce Transactions – Differentiating Properties

The long transaction can be recognized as the crucial part of the E-Commerce system. Although commercial transactions on the Internet are only a part of E-Commerce transactions, they have considerable impact on Business to Business (B2B), Business to Customer (B2C) and Customer to Customer (C2C) types of E-commerce, because long transactions underpin these operations. A typical example is the online auction, whose bid-offer-acceptance-settlement process usually takes several days to complete and involves several participants.

Besides the capability for a transaction to be executed for a period of time, there are more issues arising in an E-Commerce transaction. For instance, where multiple parties participate in the transaction, there is no guarantee that they will respond in a fixed order. In addition, not only are there rules about completing a transaction, but also there are rules about aborting or reversing part or all of the transaction.

While an E-Commerce transaction can be defined as a long transaction, it is possible to divide the overall transaction into several local transactions and ACID transactions (zur Muehlen, 2004). The next part will introduce the presentation and semantics of the Behavior Description Language, whose power will be demonstrated in the context of an E-Commerce transaction.
3. PRESENTATION OF THE BDL

3.1 BDL Syntax

We propose our BDL to be a unified medium for specifying and verifying E-Commerce transaction systems. BDL is an expressive language with similar features as in Talpin et al. (1998). It is also flexible and powerful in order to model complex, distributed and heterogeneous transaction systems. The BDL transaction system is structured so that the model can be verified through formal techniques. Additionally, the model can be theoretically exploited to correct and detect potentially errors and to improve the reliability at the early stages of software development. Cassez and Roux (1995) had proven that the ELECTRE reactive language, which is based on an asynchronous approach to real-time systems, can be compiled into a finite-state machine. Bertrand and Augeraud (1999) extends and combines the fundamental operators in Cassez and Roux (1995) and gives the concrete event-based semantics. The events exist when a method is invoked between the client object and server object. Since events are generated and received independently – which is different from a “message”, as used in typical Object Orient languages – they can make the control of concurrent objects easier. Hoare’s Communicating Sequential Processes CSP (Hoare, 1985) is a formal language that addresses sequential and concurrent characteristics of computing systems. Compared with CSP, the event notations have two major differences: 1) The notations used in CSP provide the mathematical theory for a wide-range of computer applications from process control to shared-resource operating systems, while BDL operators are intuitive and the operational semantics of the Electre language for real-time systems (Cassez and Roux, 1995) is used to build a model for execution. 2) Different classes of events in CSP, which are the foundation of mathematical models, are depicted to analyze different process properties such as sequence, concurrency, non-determinism, communication and shared resources. Events in BDL emphasize the formal semantics of the notations and the intuitive notations can be concretely rewritten to produce an automation.

The BDL used in the E-Commerce context consists of expressions for defining the business workflow of an E-Commerce transaction. It has three types: unique method or activity identifier, BDL operators and constraints. The BDL expression is the crucial part of the transaction architecture. Table 1 gives the syntax for the E-Commerce transactions in our BDL. Some of the

| E ::= E*               | repetition             |
| E; E                   | sequentiarity          |
| E || E                 | weak parallelism       |
| E || E                 | strong parallelism     |
| E | E                    | mutual exclusion       |
| E # E                  | priority               |
| (E)                    | parenthesis            |
| Identifier             | method or activity identifier |
|                       | guards                 |
|                       | selection              |
|                       | strong synchronization  |
|                       | Minimum try times      |

Table 1: Syntax of the BDL
operators have been defined by Bertrand and Augeraud (1999) and have the same semantics. However, several new operators have been defined in the context of an E-Commerce transaction system. The meanings of all the operators are described in the next section.

3.2 Semantics of BDL Operators
The operators and constraints of the BDL syntax are described below:

1. The ‘*’ repetition operator specifies the set of methods or business activities that can be repeated for a finite (pre-defined) numbers of times, but not indefinitely.
2. The ‘;’ sequentiality operator specifies the method or activity on the left must be executed first before the ones on the right.
3. The ‘||’ weak parallelism operator specifies that the left and right methods or activities can be executed in parallel. However, it is possible to let the parallel structure complete when one method or activity has achieved the execution, while the other one has not started yet.
4. The ‘||’ strong parallelism operator specifies that the left and right methods or activities can be executed in parallel. However, the parallel structure must finish when all of the methods or activities have achieved their execution.
5. The ‘|’ mutual exclusion operator specifies that the execution of mutually exclusive methods or activities. Only one of them can be selected to execute.
6. The ‘#’ priority operator specifies that if the method or activity on the right is triggered, the requests for the left terms will not be satisfied.
7. The ‘( )’ parenthesis operator specifies that the process sequence of the BDL can be redone in the modified order.
8. There are two types of identifiers: method identifier and activity identifier. These will be used in the transaction pattern and architecture separately (see later in Sections 4.1 and 4.2).
9. The ‘◊’ guard constraint is a Boolean expression which specifies that only when the Boolean expression is true, the following method or activity will be executed. If it is false, an abort or rollback method will be triggered. The guard constraint will be numbered if they are more than one guard in the BDL expression.
10. The ‘◊’ selection constraint is also a Boolean expression, which specifies that if the Boolean expression is true, the alternative BDL will be executed.
11. The ‘‖’ strong synchronization is used to synchronize the strong parallelism to specify that all of the methods or activities inside the parallel structure have to complete execution and have execution results at this point.
12. The ‘③’ minimum try number specifies that the method or activity should be executed until the indicated number of tries is achieved. In a normal situation, the abort or rollback will be executed immediately if the method or activity fails.

3.3 Illustration of the Use of the BDL Operators
In this section, a real life online payment processing example (Online Payment Processing, 2004) will be used to demonstrate the use of the BDL notations.

3.3.1 Example 1
VeriSign Inc. (Online Payment Processing, 2004) promises to provide online stores with easy and secure online payment processing services. The basic service workflow is as follows: when customers visit the online store and decide to make a purchase, the website will request for their
credit card information. Regardless of the complex connections and processing behind, VeriSign has to verify the customer. If the customer’s payment information is valid, VeriSign will arrange to settle the transaction and pass the result to the merchant. Based on the transaction result, the merchant can choose to reject the transaction or prepare to ship their goods. The basic transaction payflow can be presented as follows:

\[
\text{(Submit; validate; settle)} \# \text{abort; notify)} \ast
\]

We use a priority operator to indicate that if there is a crash when submitting, validating or settling the payflow, the transaction will be aborted and the merchant will be notified.

Besides credit cards, VeriSign can also accept debit cards and checks for validation. The payflow can be represented as:

\[
\text{(Submit; (vldt\_CC | vldt\_debit | vldt\_check); settle)} \# \text{abort; notify)} \ast
\]

-- vldt\_CC is the method identifier for validating credit cards
-- vldt\_debit is the method identifier for validating debit cards
-- vldt\_check is the method identifier for validating checks
-- We use mutual exclusion here to indicates that each transaction will be validated with only one type of payment

In order to protect online store owners’ business from fraud, credit cards’ number should be validated. We use a guard to express this constraint in order to indicate that the transaction can be settled only if the guard is true.

\[
\text{(Submit; (vldt\_CC | vldt\_debit | vldt\_check); } \text{\textasciitilde} \text{settle)} \# \text{abort; notify)} \ast
\]

\[\text{constraint: \textbullet} -- \text{The credit card number and expire date is valid and there is enough money left for this transaction}\]

VeriSign Inc. can also provide enhanced fraud protection: Card Security Code (CSC) and Address Verification Service (AVS). In addition, an abnormal order can be checked before the validation of credit cards. These additional services will also be represented by guards:

\[
\text{(Submit; } \text{\textbullet(vldt\_CC | vldt\_debit | vldt\_check); } \text{\textbullet settle)} \# \text{abort; notify)} \ast
\]

\[\text{constraint: \textbullet} -- \text{The order amount is less than a maximum limit and the order amount is bigger than a minimum limit and the order quantity isles than a maximum limit}\]

\[\text{\textbullet} -- \text{The credit card number and expire date is valid and there is enough money left for this transaction and the CSC and AVS is valid}\]

A more comprehensive example illustrating the full power of BDL to a real world application is given in Section 4.2.2.

4. TRANSACTION ARCHITECTURE
The BDL by itself cannot be used for modeling long transactions, because it does not address other E-Commerce transaction issues, such as, semantic compensation, multi-organization compliance and synchronization. As a consequence, appropriate structures to combine atomic transactions into a long or complex transaction, thereby reflecting the realities of typical E-Commerce applications.
need to be developed. We propose two entities, namely, transaction pattern and transaction pattern architecture as a means to address these issues. In addition, the transaction pattern and the transaction architecture support code re-use and make modeling easier and more flexible.

4.1 Transaction pattern
A transaction pattern represents the lower level of the overall business transaction. We usually define the short transactions as building blocks for a complete business transaction mansion. Table 2 gives the syntax of the transaction pattern.

4.1.1 Syntax for Expressing Transaction Pattern

\[
\text{pattern :: =}
\]

\[
\text{pattern identifier is type_pattern}
\]

\[
\text{pattern_expression}
\]

\[
\text{identifier ::= pattern identifier}
\]

\[
\text{type_pattern ::= flat -- flat transaction model}
\]

\[
\text{| chained -- chained transaction model}
\]

\[
\text{| nested -- nested transaction model}
\]

\[
\text{| 2PC -- two phase commit protocol}
\]

\[
\text{for distributed transaction}
\]

\[
\text{pattern_expression ::=}
\]

\[
\text{| method_identifier -- method identifier}
\]

\[
\text{| bdl_operator -- BDL operator}
\]

Table 2: Syntax of the Transaction Pattern

4.1.2 Illustration of the Use of the Transaction Pattern

A. Flat Transaction Pattern

\[
\text{pattern money_transfer is flat}
\]

\[
\text{(begin; \text{\#debit; credit; commit}) \text{\# rollback}}
\]

\[
\text{constraint:}
\]

\[
\text{\# -- the debit account has sufficient money}
\]

\[
\text{\circ -- minimum 3 tries before rollback if any method crashes}
\]

In the above pattern, money_transfer is the name of the transaction pattern. flat is the type of the transaction. (begin; debit; credit; commit) \# rollback is the BDL expression in which begin, debit, credit and commit are the method identifiers and ‘;’, ‘( )’ and ‘#’ are the BDL operators.
B. Chained Transaction Pattern

We take an example from Vogel and Rangarao (1999) about the updating of 100,000 accounts using the chained transaction model.

```plaintext
pattern account_updating is chained
.......  
◆ ( (begin8; update8; commits) #rollback8 );
◆ ( (begin9; update9; commit9) #rollback9 );
....... 
constraint:
◆ -- The previous partial transaction has completed
```

Assume that each sub-transaction will update 10,000 accounts. We can now build a chained transaction which has 10 sub-transactions to update the 100,000 accounts. Each sub-transaction has one updating method. Therefore, the update\textsubscript{8} and update\textsubscript{9} methods respectively are responsible for updating the 80,000 to 89,999 and 90,000 to 99,999 accounts separately. According to the features of the chained transaction model, only the work that is being committed can be aborted. Thus, if there is a crash at update\textsubscript{9} when updating the accounts from 90,000 to 99,999, the rollback\textsubscript{9} method will be triggered and the updating will be rolled back to the beginning state of begin\textsubscript{9}. However, the result of committed update\textsubscript{8} will be preserved.

C. Nested Transaction Pattern

![Diagram of Nested Transaction for Travel Reservation](image)

Figure 2: A Nested Transaction for Travel Reservation
Figure 2 shows a travel agent application, where a customer wants the airline, hotel and car reservations to be done at one time (Vogel and Rangarao, 1999). This is an example of a nested transaction, whose transaction pattern BDL expression is given below:

```
pattern booking is nested
  begin;
  ( ( ( begin_airline; book_airline; commit_airline) # abort_airline )
  || ( ( begin_hotel; book_hotel; commit_hotel ) # abort_hotel )
  || ( ( begin_car; book_car; ◇ commit_vehicle) # abort_vehicle ) )
  ;
  commit }
  # rollback
constraint:
  ◇ -- synchronization point for 3 sub-transactions
  ◇ -- if it is not succeed → abort_car; begin_limo; book_limo;
```

According to the nested transaction rules, the air ticket, hotel and car reservations can be executed in parallel. A nested transaction needs a synchronization point to let the parent transaction to get access to the result of its sub-transactions. Only when all the sub-transactions finish, can the parent transaction go ahead to commit.

We use the ◇ notation to give the alternative arrangement to the customer to rent a limo if the car rental is not successful. The logic expressed by the pattern is as follows: if the book_car method succeeded, commit_vehicle will be executed. If the book_car crashed, the abort_vehicle need not to be executed. Instead, an alternative arrangement will be done: abort the car reservation, start a new limo reservation, book a limo reservation. If the alternative is successful, the commit_vehicle will be executed. If it crashes, now the abort_vehicle can begin to execute.

D. Two Phase (2PC) Transaction Pattern
A simple distributed debit/credit transaction pattern taken from Casati et al (2004) is elaborated below:

```
pattern distributed_transfer is 2PC
  begin;
  ( ( debit_A || credit_B ) |; ( prepare_A || prepare_B ) | )
  ) # RollBack ;
{ (Commit_A || Commit_B)
constraint:
  ◇ -- both debit_A and credit_B not crash
  ◇ -- both prepare_A and prepare_B not crash
```

From the above pattern, we can see that the method debit_A and credit_B can be executed in parallel, but the two methods need to be synchronized before the start of the prepare_A and prepare_B methods. Any crash at this stage will trigger the rollback method to be executed. After the prepare methods have completed and there are no crashes, the commit methods can be executed in parallel.

4.2 Transaction Pattern Architecture
A transaction pattern architecture is built on top of patterns to describe their structure, integrity and business rules, which are expressed by a set of dependency and compensation schemes.
We define three kinds of architectures: the local, composite and workflow architecture. A local architecture comprises transaction patterns and a composite architecture consists of well-defined local architectures. The workflow architecture describes the top-level business workflow.

### 4.2.1 Syntax for Expressing Transaction Pattern Architecture

The syntax for expressing transaction pattern architecture is given in Table 3. A travel-booking example is given in the next section to illustrate the various architectures.

```
architecture :: =
with pattern_identifier or architecture_identifier
architecture name is
  architecture_expression
dependency
commit: architecture_identifier
rollback: architecture_identifier
Compensation
pattern_identifier:
end name;
architecture_identifier ::= transaction_pattern_identifier
architecture_identifier ::= - transaction architecture identifier
architecture_expression ::= or architecture BDL Expression
```

### Table 3: Syntax of Transaction Pattern Architecture

#### 4.2.2 Example 1: A Travel-Booking Business

Let take the travel-booking business transaction as an example of a long transaction. Figure 3 shows the workflow of the business transaction (zur Muehlen, 2004). The blue boxes can be modeled as the transaction patterns which are short and atomic. Assume they are already defined with the following pattern names: `select_accom`, `select_trans`, `calc_cost`, `send_ack`, `book_trip`, `prep_doc`, `prep_inv`, `send_inv`, `check_paym`, `send_doc`. As mentioned before, a local transaction can consist of one or several ACID transactions, or even several local transactions. The red boxes are the local transactions that are made of ACID transactions. We will use the transaction architecture to model these local transactions.

**A. The Sales Architecture**

The `sales` architecture is a local transaction with three transaction patterns. The save points of this transaction architecture is shown in Fig.3.

```rust
with select_accom, select_trans, calc_cost
architecture sales is
  (select_accom; select_trans; ◇ calc_cost)*
constraint:
  ◇ -- both select_accom and select_trains have completed
dependency
  commit: book
  rollback: t0
```
Compensation

\[
\begin{align*}
\text{select\_accom:} & \quad \text{cancel select\_accom} \\
\text{select\_trans:} & \quad \text{cancel select\_trans, cancel select\_accom} \\
\text{calc\_cost:} & \quad \text{cancel calculation, cancel select\_trans, cancel select\_accom}
\end{align*}
\]

end sales

Comments:

-- The commit dependency is the book transaction architecture. The sales architecture will be committed and recorded only if the book architecture commits and records without subsequent compensation.

-- The t0 is the initial state of the whole business transaction.

-- When one of the transaction pattern crashes, the following actions will be taken as compensation inside the architecture, e.g. the compensation for calc\_cost, if the cal\_cost pattern crashes, is the cancel calculation, cancel select\_trans, and then cancel select\_accom.

Figure 3: Travel Booking Business Transaction

B. The Book Architecture

The book architecture is a local transaction with two transaction patterns.

\[
\begin{align*}
\text{with book\_trip, send\_ack} \\
\text{architecture book is} \\
\text{book\_trip; send\_ack} \\
\text{constraint:} \\
\text{-- book\_trip is finished and successful}
\end{align*}
\]
An Event Algebra Based System for Verifying E-Commerce Transactions

dependency
commit: prepare, invoice
rollback: sales

compensation
book_trip: cancel book
send_ack: send again, cancel book
end book

Comments:
-- Only when the prepare and invoice architectures are both committed, the book architecture can be committed.
-- The rollback dependency hold by this architecture is sales architecture.
-- The compensation for the send_ack is sending again.

C. The Doc Architecture
The doc architecture just has one transaction pattern and the reason to promote it to an architecture is that we can define the dependency and compensation inside the architecture.

with prep_doc
architecture doc is
prep_doc
dependency
commit: prepare (parent architecture)
rollback: book, payment (successor)
compensation
prep_doc: cancel preparation
doc
end doc

Comments:
-- The prepare architecture is the parent architecture of doc. The commit dependency rule is: only when the parent architecture prepare commits, this doc architecture can commit.
-- Here we define a parallel rollback dependency: if the payment architecture crashes, the doc architecture should be compensated through the scheme.

D. The Invoice Architecture

with prep_inv, send_inv
architecture invoice is
prep_inv; send_inv
constraint:
-- prep_inv is done
dependency
commit: prepare
rollback: book
compensation
prep_inv: cancel preparation
send_inv: send invoice again, cancel preparation
end invoice

Comments:
-- The commit dependency of the invoice architecture is still prepare architecture.
E. The Payment Architecture

```plaintext
with check_paym
architecture payment is
  check_paym
dependency
  commit: prepare
  rollback: invoice
compensation
  check_paym: check again until reach to minimum try number
end payment
```

Comments:
-- Here we define a compensation scheme according to the assumed business rules: if the payment checking crashes, the architecture will not roll back immediately. Instead the architecture will check again until the minimum number of tries has been attempted.

F. The Prepare Architecture

The `prepare` architecture represents for a composite transaction which has three local transactions (child transaction architectures): the `doc`, `invoice` and `payment` architecture. Inside the `prepare` architecture, `doc` architecture can be executed in parallel with the other two architectures.

```plaintext
with doc, invoice, payment
architecture prepare is
  doc || (invoice; payment)
dependency
  commit: send
  rollback: book
end prepare
```

G. The Send Architecture

Suppose the business rule is that the travel-booking business transaction can be completely committed after sending the documents.

```plaintext
with send_doc
architecture send is
  send_doc
dependency
  commit: commit
  rollback: prepare
compensation
  send_doc: send again
end send
```

Comments:
-- The commit dependency of this architecture is `commit` which means it is commit-independent.
-- The rollback dependency is present in the `prepare` architecture, which means the whole architecture along with its child architectures, have to perform a rollback.
H. The Workflow Architecture

The workflow architecture is the highest architecture which consists of one or more architectures and/or composite architectures as shown in Figure 4.

```plaintext
with sales, book, prepare, send
architecture workflow is
    sales; book; prepare; send
compensation
    sales: sales
    book: book; sales
    prepare: prepare
    send: send
end workflow
```

Comments:
-- We define book as a complete compensation (Figure 5). A complete architecture is defined as an architecture such that if it crashes, the system compensates to the beginning. In this example, if the book crashes, the system compensates to the sales.
-- The compensation for the prepare architecture is a partial compensation. If a partial compensation happens, the whole transaction system will not be rolled back to the
beginning. For instance, the system will only compensate to the book architecture if the prepare architecture crashes and it may build a new prepare architecture later with the previous architectures protected.

Fig. 5 shows the compensation diagram for this business transaction.

5. A DETAILED EXAMPLE: AN E-MOVE TRANSACTION
We take the example scenario from Cassati et al (2004) which describes a platform for a composite e-move service. Suppose there are three companies A, B and C that specialize in their own e-business area (the services they provide can be seen from Table 4). There is an E-Move company (hereunder referred to as “E-Move”), which provides reallocation service to customers on the basis of the three business companies. In other words, E-Move is new service provider whose service is a composite of existing e-services. This means that E-Move will connect the three companies and compose the operations needed for truck rental, furniture shipment, address change, billing, etc, according to the customer’s requests from the level of local move to domestic move.

5.1 E-Move Version I: The Local Move
The local move service includes: collecting the customer’s information, submitting the credit card information to company B for verification and appropriately notifying all concerned parties about the customer.

Figure 6 shows the transaction relationship between E-Move and company B and C. From the point of transaction pattern view and architecture view, there are five major transaction architectures in the E-Move, namely, collect, billing, prepare, notify and send. Each of the architecture has one or more transaction patterns as defined below.

<table>
<thead>
<tr>
<th>Company</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service List</td>
<td>Truck Rental</td>
<td>Truck Rental</td>
<td>Credit Card Verification</td>
</tr>
<tr>
<td>Storage Space Rental</td>
<td>Storage Space Rental</td>
<td>Credit Check</td>
<td></td>
</tr>
<tr>
<td>Airline Shipment</td>
<td>Personal Contact Address Notification</td>
<td>Credit Card Payment</td>
<td></td>
</tr>
<tr>
<td>Naval Shipment</td>
<td>Banking Address Notification</td>
<td>Billing</td>
<td></td>
</tr>
<tr>
<td>Railway Shipment</td>
<td>Internet Accounts Address Notification</td>
<td>- - -</td>
<td></td>
</tr>
<tr>
<td>Airline Reservation</td>
<td>House Utility Address Notification</td>
<td>- - -</td>
<td></td>
</tr>
<tr>
<td>Customs Paperwork</td>
<td>- - -</td>
<td>- - -</td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Service List of Companies A, B and C

From Figure 6 we can see that E-Move transaction connects company B and company C through transaction contracts. A contract is the compliance and synchronization of a multi-organization transaction (see Section 2.3). Inside the company B, it is a transaction architecture which is made
of three transaction patterns, where each transaction pattern is a flat style transaction. The E-Move company does not have to launch the company B’s transaction architecture, but E-Move does have to synchronize the data flow with company B and get the result according to the contracts between them. It is the same situation with company C, wherein there is a nested transaction to perform the necessary notification service. The E-Move just needs to synchronize with company C and get the result. In summary, at the side of E-Move, the transaction architecture will just include the synchronization of data and results in order to decide the necessary dependency scheme and compensation scheme.

The following is the E-Move Version I transaction expression. We will start from defining transaction patterns, then the transaction architectures. Note that the expression for defining the workflow architecture is provided towards the end.

```plaintext
pattern data_collection is flat
  (begin; collect, record; commit) # rollback
constraint:
  -- the information collected is valid
  -- record the customer information, update the system state to “information recorded”

pattern pass_cc_info is flat
  (begin; send; update; commit) # rollback
  -- send the information to company B, update the system state to “information sent”

pattern check_payment is flat
  (begin; check; update; commit) # rollback
constraint:
  -- the payment is received
  -- check the payment, update the system state to “payment received” if check is successful

pattern prepare_invoice is flat
  (begin; prepare; update; commit) # rollback
constraint:
  -- the preparation is successful
  -- update the system state to “invoice prepared”

pattern pass_cust_info is flat
  (begin; send; update; commit) # rollback
constraint:
  -- the information is successfully sent
  -- send the notification information to company C, update the system state to “payment received & notification information sent”

pattern check_notif_result is flat
  (begin; check; update; commit) # rollback
  -- check the notification service result, update system state to “result checked”

pattern inform_cust is flat
  (begin; inform; update; commit) # rollback
  -- inform the customer the service done, update system state to “customer notified”
```
pattern send_invoice is flat
  (begin; send; update; commit) # rollback
  --send the invoice and update system state to “invoice sent”

with data_collection
architecture collect is
  data_collection
dependency
  commit: billing
  rollback: t0
compensation
  data_collection: cancel collection, recover the system to the
  beginning
end collect

with pass_cc_info, check_payment
architecture billing is
  pass_cc_info; ◆check_payment® constraint:
  ◆ -- the payment result is available
dependency
  commit: prepare, notify
  rollback: collection
compensation
  pass_cc_info: pass the information again
  check_payment: check the payment again with a minimum 8 tries
end billing

with prepare_invoice
architecture prepare is
  prepare_invoice
dependency
  commit: send
  rollback: billing
compensation
  prepare_invoice: prepare again
end prepare

---with pass_cust_info, checknotif_result
architecture notify is
  pass_cust_info; ◆checknotif_result® constraint:
  ◆ -- the notification result is available
dependency
  commit: send
  rollback: billing
compensation
  pass_cust_info: pass the information again
  checknotif_result: check the result again with 8 minimum tries
end notify
An Event Algebra Based System for Verifying E-Commerce Transactions

with send_invoice
architecture send is
send_invoice
dependency
commit: commit
rollback: prepare, notify
compensation
send_invoice: send the invoice again
end send

with collect, billing, prepare, notify, send
architecture workflow1 is
collect; ◆billing; ◆ ( prepare || notify ); ◆send
constraint:
◆ -- data collection is valid and enough
◆ -- payment is received
◆ -- invoice prepared and notification service is done
compensation
collect: delete collection
-- complete compensation
billing: terminate billing service; delete collection
-- complete compensation
prepare: cancel preparation
-- partial compensation
notify: terminate notification service
-- partial compensation
send: cancel sending
-- partial compensation
end workflow1

Figure 6: E-Move transaction architecture

NB:
1* Personal Contact
   Address Notification Service
2* Banking Address
   Notification Service
3* Internet Accounts
   Address Notification Service
4* House Utility Address
   Notification Service
5.1.1 Discussion

1. We define that the E-Move service terminates after sending the invoice. The whole transaction will be committed right after the invoice is sent. The after-sale service will be regarded as a new transaction and it will not be covered in this transaction expression.

2. Before the system’s billing architecture, if any failure happens, the system will execute a complete compensation to rollback the system state to the beginning. However, after the billing architecture, the system will execute partial compensation to retain the potential profit. The complete and partial compensations can be seen from Figure 7.

5.2 E-Move Version II: The Domestic Move

E-Move Version II is about a domestic move which may involve truck rental, packing and shipping furniture. It may also include booking airline tickets to let the customer travel to the new location. From the modeling point of view, we still need the E-Move Version I architecture to provide the address change transaction service. In fact, we can reuse any of the architectures defined before. The billing service is different from the Version I billing since customer can decide on the service necessary for them. We do the credit card checking in advance and, after the shipment service is done, we then calculate the cost and ask for the subsequent company to do billing service. Figure 8 shows the E-Move Version II workflow, which is on the basis of Version I.

Compared with E-Move Version I, we can see from Figure 8 that some architecture are the same as the Version I, such as collect, notify, prepare and send. The verify and billing architectures
has different meanings as E-Move Version I. First, we check the customer’s credit details and at the end of the workflow, the customer is charged according to the actual rental and shipment service.

Company A provides the rental and shipment service. From the E-Move point of view, the detailed transaction will be performed by Company A. The E-Move Company just needs to synchronize the information provided and the results received. That means, in the E-Move transaction architecture, we do not have to express the detailed rental and shipment transactions. The following is the E-Move Version II transaction architecture. The bold architecture names are reused from E-Move Version I.

```plaintext
pattern pass_cc_info is flat
  (begin; send; ◆ update; commit) # rollback
  -- send the information to company B, update the system state to “information sent”

pattern validate_CC is flat
  (begin; check; ◆ update; commit) # rollback
  constraint:
  ◆ -- the credit card is valid
  -- check the credit card verification result, update the system state to “credit valid”

pattern pass_rental_info is flat
  (begin; send; ◆ update; commit) # rollback
  constraint:
  ◆ -- the rental information is successfully sent
  -- send the rental information to company A, update the system state to “credit valid & rental information sent”

pattern start_rental is flat
  (begin; update; commit) # rollback
  -- update system state to “rental started”

pattern check_result is flat
  (begin; check; ◆ update; commit) # rollback
  constraint:
  ◆ -- all of the rental & shipment requests get responded
  -- check the rental & shipment result, update system state to “result received”

pattern pass_charge_info is flat
  (begin; send; ◆ update; commit) # rollback
  constraint:
  ◆ -- the charge information is successfully sent
  -- send the charge information to company B, update the system state to “charge information sent”

pattern check_payment is flat
  (begin; check; ◆ update; commit) ◆# rollback
  constraint:
  ◆ -- the payment is received
  -- check the payment, update the system state to “payment received” if check is successful
```
An Event Algebra Based System for Verifying E-Commerce Transactions

with pass_cc_info, validate_cc
architecture verify is
  pass_cc_info, ◆ validate_cc
constraint:
  ◆ -- the validate result is available
dependency
  commit: notify, rental
  rollback: collect
compensation
  pass_cc_info: pass the information again
  validate_cc: cancel validation
end verify

with pass_rental_info, start_rental, check_result
architecture rental is
  pass_rental_info; ◆ start_rental; ◆ check_result®
constraint:
  ◆ -- the information passing is completed
  ◆ -- all of the rental requests have results
dependency
  commit: billing, prepare
  rollback: validate
compensation
  pass_rental_info: pass again
  check_result: check again until the minimum try number
end prepare

with pass_charge_info, check_payment
architecture billing is
  pass_charge_info; ◆ check_payment®
constraint:
  ◆ -- the payment result is available
dependency
  commit: send
  rollback: notify, rental
compensation
  pass_charge_info: pass the information again
  check_payment: check the payment again with a minimum 8 tries
end billing

with collect, verify, notify, rental, prepare, billing, send
architecture workflow2 is
  collect; ◆ verify; ◆( notify || rental ); ◆( prepare || billing ); ◆ send
constraint:
  ◆ -- data collection is valid and enough
  ◆ -- credit checking is valid
  ◆ -- notification and rental service are done
  ◆ -- invoice prepared and payment received
compensation
  collect: delete collection
    -- complete compensation
  verify: terminate verification service; delete collection
    -- complete compensation
An Event Algebra Based System for Verifying E-Commerce Transactions

rental: cancel rental; terminate verification service; delete collection
-- complete compensation
notify: terminate notification service
-- partial compensation
prepare: cancel preparation
-- partial compensation
billing: cancel billing, cancel preparation, cancel notification service, cancel rental, delete collection
-- complete compensation
send: cancel sending
-- partial compensation
end workflow2

Figure 8: E-Move Version II Transaction Architecture Diagram
6. RELATED RESEARCH WORK

Related research work will be addressed along aspects as given by: i) transaction modeling, ii) business process modeling and iii) BDL.

In the field of transaction modeling, a number of approaches have been adopted in the literature. For instance, dynamic transaction schema (Ngu, 1989), transaction interoperability (Tarr and Sutton, 1993) and dependency rules (Gunthor, 1993) have been investigated in previous work. In later research, transactional technologies were introduced into workflow management systems to express business processes because of the lack of failure semantics and recovery features in these systems (Alonso et al., 1996; Georgakopoulos et al., 1996; Chen and Dayal, 1996). With the dramatic growth of the Internet, more and more issues are discussed about E-Commerce transactions (Sandholm, 1997; Li et al., 2001). For example, in Nektarios and Christodoulakis (2002), a high-level Unified Transaction Modeling Language is proposed recently to model complex web transaction design. However, few transaction modeling techniques demonstrate very complex long transactions which are more and more identified in real world and keep the modeling process flexible and extensible in order to support existing and future transaction models. In addition, these research works do not support verification and validation, which is required in the E-Commerce context. Furthermore, they do not provide an executable framework for scenario-based checking. These aspects motivated us to find a new approach, which can verify the transaction processes before a complex E-Commerce system is built and this has led to the proposed transaction patterns and architectures in this paper. It then becomes a good starting point if one first develops an effective high-level modeling tool to facilitate future work.

Recently, a rich set of notations, called the Business Process Modeling Notation (BPMN), has been released recently by bpmi.org (BPMN, 2004), which acts as a standardized bridge for all business users including business analysts, technical developers and business people to model from internal business processes to B2B processes. Flowcharting technique is used in the Business Process Diagram (BPD) in which graphical elements are categorized to enforce simplicity, but with additional macro operations, complex system modeling can also be handled. In the complete set BPD, an event, compensation and exception notations are included, as shown in Figure 9 (adapted from White, 2000). The example models both a simple transaction and a complex long transaction, which deals with a number of exceptions and compensation schedules among different business entities. Compared with BPMN, realistic and complex E-Commerce transaction constraints can be effectively captured by the transaction patterns and architectures. As mentioned in Section 1, our approach overcomes the limitations of the lack of formal semantics in graphical approaches and unmanageable complexity. In addition, we also provide a formal mechanism to verify complex transactions. It shares common features with Architecture Description Languages, which mostly focus on conceptual designs instead of real implementations. A more recent work (Glasser et al., 2004) stresses on aspects of executability and interoperability with the .NET language and we have realized that connecting with successful software systems will offer visible and realistic benefits to present E-Commerce systems. Although this is not the focus of this paper, a discussion has been provided and a concrete implementation is being considered.

BDL was first proposed to describe the concurrent behavior of simple objects or a group of objects (Bertrand and Augeraud, 1999), while complex E-Commerce transaction systems involve workflows which are activity-based. When we try to use the BDL on the E-Commerce systems, this inconsistency forced us to seek for support from the event semantics. We therefore extended the BDL further. In fact, the entities in a multi-organizational environment can be re-packaged as objects and the activities, which then can be considered with event semantics. At least, the event semantics
is obvious at each level of transaction patterns and architectures. Consequently, existing verification tools can be used to verify transaction systems or part of the systems. Furthermore, the extensions to the BDL may lead to new effective verification or testing tools to be developed.

7. CONCLUSIONS AND FUTURE WORK
In this paper, we have described the syntax, semantics and usage of a Behavioural Description Language (BDL) developed for modeling and verifying E-Commerce transaction processes. On the basis of transaction models, we build the BDL notations to express E-Commerce transaction processes. We also put forward the concept of transaction pattern and transaction architecture which are the containers for BDL. The advantages of using BDL in E-Commerce systems are: i) it can model real world, concurrent and complex business transactions, ii) its expressive feature, iii) the model developed by BDL can be verified through existing verification tools and subtle errors may be detected. Consequently, the reliability can be improved in the development of E-Commerce
An Event Algebra Based System for Verifying E-Commerce Transactions

transactions. In other words, we introduced the modeling and verification power into the E-Commerce applications to gain the benefits of flexibility, expressiveness and structure.

The present work addresses issues in automation of transactions in E-Commerce also. We have shown that complex long transactions, which exist in B2B, B2C and C2C can be modeled through the BDL introduced in this paper. Due to the variety and complexity of an E-Commerce transaction system, the development of an effective verification tool is being under consideration. Firstly, a GUI visualization tool is very useful so that a complex transaction can be easily built, understood and accepted. Further, a theorem prover or an equivalent system could be developed so that the BDL notations and transaction architectures can verify the correct operation of the underlying E-Commerce structure. Mapping to the standard Business Process Execution Language (BPEL), which is currently considered as the most important execution language, is another beneficial implementation. The XML-based BPEL provides event, fault and compensation handlers and the reader can refer to BPEL for Web Services (2003) for further reading. We are also interested in whether composite events can be used to give semantic and executable support to high-level transaction architectures. The research on Complete Interaction Sequences (White et al, 2001; White and Almezen, 2000) is also worthy of following up.

ACKNOWLEDGEMENT
We thank Professor Sastry of the University of Akron for his insightful comments, which improved the quality of this paper.

REFERENCES
An Event Algebra Based System for Verifying E-Commerce Transactions


APPENDIX: FEATURES OF TRANSACTION MODELS

A.1 ACID Transactions
As Bernstein et al (2004) pointed out, a transaction is a computation whose execution is constrained by the fundamental properties of atomicity, consistency, independence, and durability (collectively known as the ACID properties (Bernstein et al., 2004)). ACID transaction includes flat transaction, chained transaction and nested transaction and these are briefly described below:

A.1.1 Flat transaction
A flat transaction is one in which one of the operations in a tuple of operations fails. An example is the debit/credit transaction where a debit operation is done but the credit operation crashes. Figure A.1 (zur Muehlen, 2004) shows that, after the transaction begins, it is guaranteed to have a result, namely, success or failure. This makes the system change from one state to another consistent state and, in addition, makes the system more durable. The flat transaction is proven to be highly effective as it has an excellent control over the short and concurrent database access, while not losing strict consistency requirements. Although the flat transaction is not suitable to model complicated business processes, surprisingly, most of the transactions are flat nowadays (Vogel and Rangarao, 1999).

A.1.2 Chained Transaction
A chained transaction is an evolution of the flat model, which addresses the business problem where “save points” are needed during the execution of the transaction. Figure A.2 (zur Muehlen, 2004) demonstrates the important fact that if there is a crash, the transaction which is being committed can be aborted without the need to abort all of the committed jobs. That means that one need not have to go back to the beginning of the transaction – unlike the flat transaction.

A.1.3 Nested Transaction
A nested transaction can have many sub-transactions within the main transaction; it is convenient for modeling more complex business logic. Figure A.3 shows a transaction T with two sub-
transactions $T_1$ and $T_2$, with each having sub-transactions $T_{11}$ & $T_{12}$ and $T_{21}$& $T_{22}$ respectively. The execution of committing $T$ depends on whether all of the sub-transactions have completed. There is a synchronization point before $T$ commits to check the results of the sub-transactions. A unit of transaction like $T_{12}$ can be committed or aborted individually, however, the system will only make $T_{12}$ persist while all its parent transactions $T_1$ and $T$ commit.

From Table A.1 we can see that if a parent transaction aborts, all of its committed children e.g., $T_{21}$ and $T_{22}$ will also be aborted. However, if a sub-transaction aborts, the parent may not be aborted; instead, the parent can perform the following actions:

A. Re-try the sub-transaction and execute at least a minimum re-try number.
B. Prepare an alternative sub-transaction and execute.

Compared with the flat transaction, the nested transaction has two major advantages: i) it can recover from partial failure and ii) the sub-transactions can be executed concurrently.

### A.I.4 Two Phase Commit Protocol for Distributed Transaction

The two-phase commit protocol, usually referred as 2PC, is developed for distributed transaction in which the update resources are located in different places. As an extension of flat ACID transaction,

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Sub Transactions</th>
<th>Alive</th>
<th>Commit List</th>
<th>Abort List</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>$T_1,T_2$</td>
<td>Yes</td>
<td>$T_1,T_{12}$</td>
<td>$T_{11},T_2$</td>
</tr>
<tr>
<td>$T_1$</td>
<td>$T_{11},T_{12}$</td>
<td>Yes</td>
<td>$T_1,T_{12}$</td>
<td>$T_{11}$</td>
</tr>
<tr>
<td>$T_2$</td>
<td>$T_{21},T_{22}$</td>
<td>No (Aborted)</td>
<td>$T_2$</td>
<td>$T_{11}$</td>
</tr>
<tr>
<td>$T_{11}$</td>
<td></td>
<td>No (Aborted)</td>
<td>$T_{12}$</td>
<td></td>
</tr>
<tr>
<td>$T_{12}$</td>
<td></td>
<td>Yes</td>
<td>$T_{12}$</td>
<td></td>
</tr>
<tr>
<td>$T_{21},T_{22}$</td>
<td></td>
<td>No (Parent Aborted)</td>
<td>$T_{21},T_{22}$</td>
<td></td>
</tr>
</tbody>
</table>

Table A.1: Nested Transaction Execution Information List of Figure A.3
2PC has to keep the atomicity property by committing all updates in different resource managers together (Vogel and Rangarao, 1999). The transaction coordinator sends a “prepare-to-commit” message in the first phase to all participating resource managers and requests them to reply. In the second phase, the coordinator first collects the replies. If all of the resource managers are ready to commit, then the coordinator will send the commit request to all resource managers. After sending the replies to the coordinator, all distributed resource managers wait for the commit request and they will commit together.

As shown in Figure A.4 (zur Muehlen, 2004), if one of the resource manager e.g., A fails, the resource manager will abort locally and send a “Not Ready” reply to the coordinator. The coordinator will send “Abort” request to all resource managers which have replied “Ready”.

Figure A.5: The Workflow for the Trip Booking Business Transaction

NB: The black circles and diamonds represent merge and decision-points respectively. As noted in the paper, the black diamond notation ahs been introduced to strengthen the "rollback" features of E-Commerce transactions, viz., a transaction will not be committed until it passes through the established guards.
In summary, the ACID transactions have the features of short duration and strict isolation. However, in the business field, the ACID transactions may have problems to support workflow based transactions. zur Muehlen (2004) expressed these problems using a trip-booking workflow example where it is not possible to apply atomicity when the transaction includes calculating the cost and sending the invoice. At the same time, it is almost impossible to preserve isolation when preparing invoices for the customers (see Figure A.5) (zur Muehlen, 2004).

**BIOGRAPHICAL NOTES**

Renyi Zhao received his Bachelor degree from Harbin Institute of Technology, China, in 1996 and his Master's degree in Information Technology from the University of Newcastle, Australia, in 2004. Currently he is pursuing his PhD at the University of Newcastle, Australia, in the area of event algebra and formal languages for mandatory access control design.

Professor Lakshmi Narasimhan obtained his Bachelor’s degrees in Physics and Electronics Engineering from the University of Madras and Indian Institute of Science respectively. He also obtained his Master’s and PhD degrees respectively from the Madras Institute of Technology and University of Queensland, Australia. He worked at his alma mater in Australia as a lecturer and senior lecturer and was also the Director of the PA3SE (Parallel Algorithms, Architectures And Software Engineering) Research Laboratory. In 1995 he moved to the Australian Defence and Science and Technology Organisation (DSTO) as a Principal Research Scientist and was leading the program on Information Management and Information Fusion. Later he worked as a full professor at the Computer Science Department, University of North Texas at Denton, USA. Currently he is the Professor and Chair in Software Engineering at the University of Newcastle, Australia.

Professor Narasimhan has published over 120 papers in the areas of computer architecture, parallel and distributed computing, software testing, text and audio processing, E-Commerce, Software process, and information management and fusion. His articles have appeared in archival journals such as various IEEE Transactions and IEE Proceedings. Professor Narasimhan was the founding chair of the IEEE Computer Society, Queensland Chapter and initiated the International Conference series ICA3PP (IEEE International Conference on Algorithms and Architectures) and acted as its Inaugurating Chair. Currently his research interests are the areas of large-scale system engineering, testing and visualization, Agent technology and distributed object computing. He is a Senior Member of IEEE, ACM, ACS and IEAust.