Static Conflict Analysis of Transaction Programs

by

Connie Zhang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2000

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Transaction programs are comprised of read and write operations issued against the database. In a shared database system, one transaction program conflicts with another if it reads or writes data that another transaction program has written. This thesis presents a semi-automatic technique for pairwise static conflict analysis of embedded transaction programs. The analysis predicts whether a given pair of programs will conflict when executed against the database.

There are several potential applications of this technique, the most obvious being transaction concurrency control in systems where it is not necessary to support arbitrary, dynamic queries and updates. By analyzing transactions in such systems before the transactions are run, it is possible to reduce or eliminate the need for locking or other dynamic concurrency control schemes.

Acknowledgment

I would like first to acknowledge the guidance given me by Dr. Ken Salem. In his role as supervisor, he continuously advised and aided me. It is my extreme good fortune to have studied under Ken, who is a wonderful person and an amazing professor. I am sincerely thankful for his patience and support.

My readers also deserve special commendation for their diligence in identifying shortcomings in the thesis and in suggesting solutions. Thank you very much, Professors David Toman and Grant Weddell.

Additional thanks are due to the DEMO lab members, Mr. Tim Snider and Dr. Ian Davis, for their co-operation and patience in dealing with my questions and requests.

I had a marvelous time as a graduate student in the Database Research Group and the Computer Science department. Extra thanks must go to Professor George Labahn for his help and encouragement. As well, I wish to thank David, Grant, Tim, Ian, Professor Frank Tompa, and all my friends in the Department for the many stimulating discussions that I will dearly miss.

Financial support from Communications and Information Technology Ontario (CITO), Ontario Graduate Scholarship (OGS), the Department of Computer Science and the University of Waterloo is gratefully acknowledged.

Finally, deepest thanks to my mum who is also my good friend and my best role model. Last, but not least, I want to thank Jeff for everything — what would I have done without you.

Connie Zhang
Department of Computer Science
University of Waterloo
September 2000

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A *transaction program* is a program with operations that access and possibly update various objects in a shared database [1]. One transaction program conflicts with another if it reads or writes data that the other has written. If the sets of objects to be read or updated by the program could be described, then these sets could be compared to similar sets from another transaction program. The transaction programs may conflict if there exist common elements in the sets,

The main contribution of this thesis is a semi-automatic technique for pairwise static conflict analysis of transaction programs. The analysis predicts whether a given pair of programs will conflict when executed against the database.

The thesis presents a procedure for transforming a transaction program into a read query that describes all of the objects on which the transaction program might depend, and a write query that describes all of the objects the same transaction program might insert, delete or modify. Next, the thesis describes a method of analyzing conflicts between two transaction programs by intersecting the read and write queries from one transaction program with the read and write queries from the other. By using a tool developed by the Design Environment for Memory-resident Object-oriented databases (DEMO) project at the University of Waterloo, the DEMO query optimizer (Appendix C), it can then be determined whether the intersection of the sets is empty. A non-empty intersection implies a potential conflict between the two input transaction programs. If the input programs contain parameters, then the result returned from the DEMO query optimizer can be analyzed to determine conditions under which the intersection of the transaction access sets may not be empty.

The thesis is organized as follows. Chapter 2 provides an overview of the technique: what steps are involved in the conflict analysis, and how the steps fit together. Chapter 3 describes the steps in more detail: what algorithms are used, what input each step expects, and what output it produces. Chapter 5 contains a number of examples that illustrate how the technique is applied. The basic concept in the thesis is reminiscent of *predicate locks* used for concurrency control [2]. Chapter 6 describes this and other related work

and compares it to the approach presented in the thesis. There are several potential applications of the technique, the most obvious being concurrency control. Such applications are also discussed in Chapter 6.

Finally, Chapter 7 sums up the results of this work, and notes possible avenues for future work.

Chapter 2

Overview

The goal of the semi-automatic process described in this thesis is to take a pair of transaction programs and analyze them for potential run-time conflicts. That is, the process determines whether the changes made to the database by the execution of one program might affect the execution of the other program.

Transaction programs may be parameterized. Values are bound to the parameters each time the program is executed. The conflict analysis process occurs before the transaction programs are executed. Therefore, parameter values are not known at the time of the analysis. For this reason, the result of the analysis is also parameterized. Specifically, the result is a conflict predicate written in terms of the parameters of the transaction programs.

A *transaction* is the execution of a transaction program with a particular set of values bound to its parameters. Two transactions can be checked for potential conflicts by substituting their parameter values into the parameterized conflict predicate for the corresponding pair of transaction programs. A conflict predicate that evaluates to *TRUE* indicates that the transactions may conflict. A conflict predicate that evaluates to *FALSE* indicates that conflict will not occur.

The conflict analysis is based on sets of queries (called *read selects* and *write selects*) that characterize the behaviour of the transaction programs. Given the read and write selects for two transaction programs, the analysis process generates a conflict predicate for that program pair. Figure 2.1 depicts in more detail the steps involved in the analysis of a pair of transaction programs, T_1 and T_2.

Initially, the input read and write selects are combined to form a read query and a write query for each transaction program. In the conflict query generation stage, first, the read query Q_{R1} of the transaction program T_1, and the write query Q_{W2} of the transaction program T_2 are intersected to create an intersection query Q_{RW}. Similarly, the write query Q_{W1} for T_1 and the read query Q_{R2} for T_2 are intersected to create an intersection query Q_{WR}. A final query $Q_{conflict}$ merges the result the two intersection queries. The result of $Q_{conflict}$ includes all of the objects on which T_1 and T_2 could conflict.

$Q_{conflict}$ is then passed to the DEMO optimizer (Appendix C) in the EQG

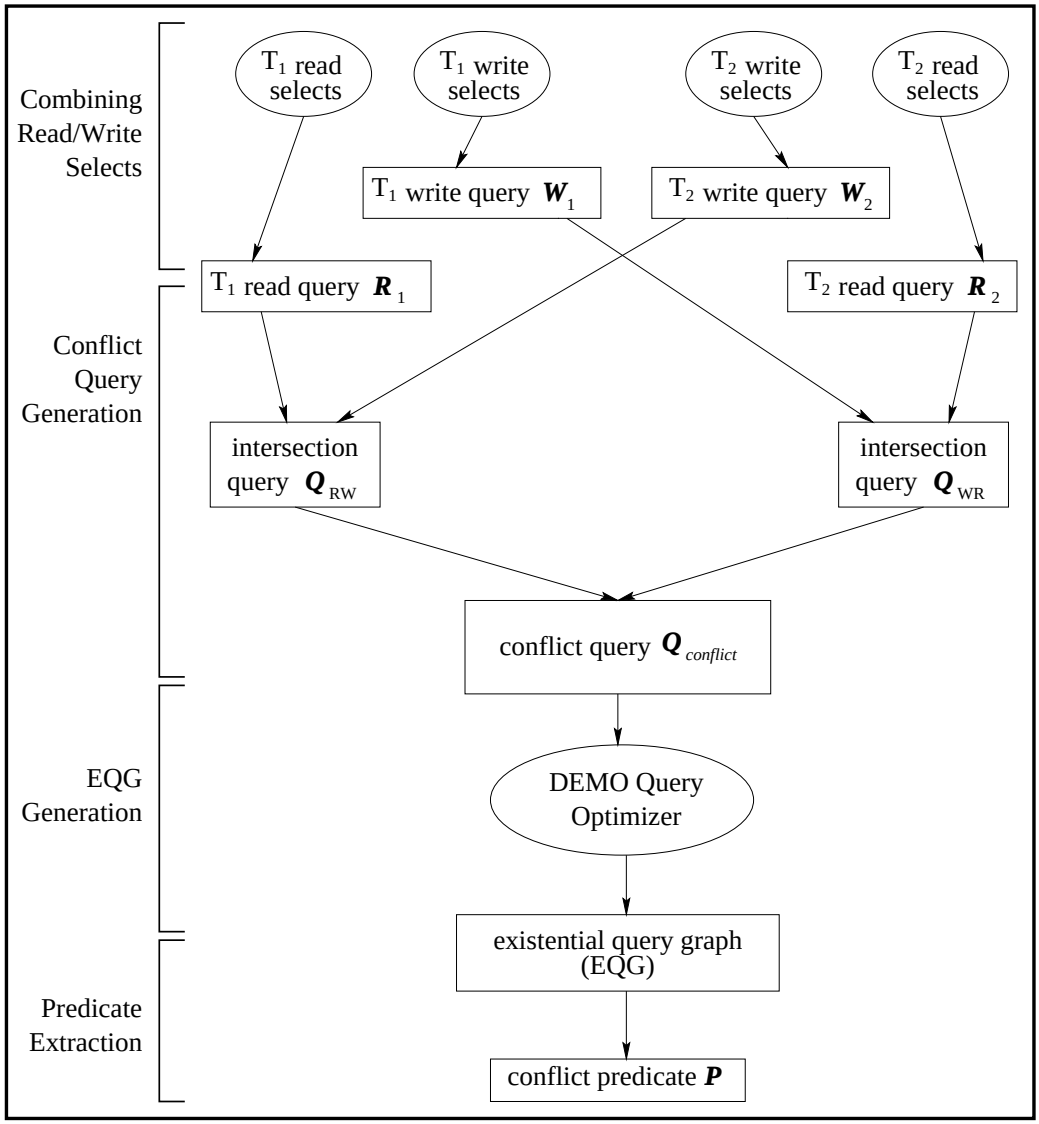


Figure 2.1: Pairwise Transaction Analysis

generation step. The optimizer produces a representation of certain properties of objects in $Q_{conflict}$. From this, it is possible to extract a conflict predicate. The predicate extraction step analyzes the result from the optimizer, and constructs a conflict predicate. The predicate will evaluate to *FALSE* for a particular set of parameter bindings only if $Q_{conflict}$ is known to be empty under those bindings for all possible databases. In other words, it evaluates to *FALSE* only if the two transactions do not conflict.

Chapter 3

Description of Techniques

Chapter 2 presented an overview of the steps involved in finding conflicts in transaction programs. In this chapter, the details of each step will be presented. A sample employee database is defined in Appendix B. The query examples used in this and subsequent chapters are issued against this sample database.

3.1 Modeling Transaction Programs

3.1.1 Definitions and Notation

Transaction programs are assumed to operate against an object database. The database has an associated schema, which defines the types of objects that may be present in the database. The schema may also define additional database constraints. A database D is a set of objects that is consistent with the schema. The set of possible databases that conform to the database schema will be denoted by $\mathcal{D}$. The universe of possible database objects will be denoted by $\mathcal{U}$.

Definition 3.1.1 (Database Delta)
A delta Δ for database D is a set of objects to be inserted into D (denoted by Δ^+) and a set of objects to be deleted from D (denoted by Δ^-) such that all of the following conditions hold:

- $\Delta^+ \in 2^{\mathcal{U}-D}$
- $\Delta^- \in 2^D$
- $((D \cup \Delta^+) - \Delta^-) \in \mathcal{D}$

A database delta can be used to transform one database into another. If Δ is a delta for D, the notation $D + \Delta$ will be used to denote the database obtained by applying Δ to D, i.e., $D + \Delta = ((D \cup \Delta^+) - \Delta^-)$.

A partial order $<$ is defined over the set of possible deltas for database D as follows. Given two deltas Δ_1 and Δ_2 for D, $\Delta_1 < \Delta_2$ if and only if $\Delta_1^+ \subseteq \Delta_2^+$ and $\Delta_1^- \subseteq \Delta_2^-$ and $\Delta_1 \neq \Delta_2$.

A transaction program is a program that can be executed to perform a database transaction. The execution of a transaction program occurs as a sequence of discrete operations. Each operation may change the internal state of the program, and may modify the database. The execution of the operations of a program is controlled by a scheduler, which is not modeled here. The scheduler determines when the executing program performs its next operation. Note that the series of operations that result from the execution of a transaction program may depend on the contents of the database at the points in time at which the operations are initiated by the scheduler.

Definition 3.1.2 (Transaction Program)

A transaction program T consists of the following seven elements:

S_T: *a set of possible internal program states.*

P_T: *a set of formal program parameters. Each parameter consists of a name and an associated value domain. The set of possible distinct assignments of domain values to the parameters of a program will be denoted by $\mathcal{P}_T$ and $T(p)$ will represent a transaction resulting from the execution of T with parameter values $p \in \mathcal{P}_T$.*

$initial_T : \mathcal{P}_T \rightarrow S_T$: *a function that determines the initial state of the program given a set of parameter values.*

$next_T : S_T \times \mathcal{D} \rightarrow S_T$: *a function that determines the new internal state into which T will move as a result of an operation performed from a given internal state and a given database.*

$insert_T : S_T \times \mathcal{D} \rightarrow 2^{\mathcal{U}-\mathcal{D}}$: *a function that determines the objects inserted into the database as a result of an operation performed from a given internal state and a given database.*

$delete_T : S_T \times \mathcal{D} \rightarrow 2^{\mathcal{U}}$: *a function that determines the objects deleted from the database as a result of an operation performed from a given internal state and a given database.*

For every state $s \in S_T$ and every database $D \in \mathcal{D}$, the functions $insert_T(s, D)$ and $delete_T(s, D)$ are required to define a database delta for database D, with $insert_T(s, D)$ defining Δ^+ and $delete_T(s, D)$ defining Δ^-. The notation $nextDB(s, D)$ will be used to refer to the state $D + \Delta$, i.e., the new database that results from the execution of an operation of T from state s and database D.

The program-identifying subscripts used in Definition 3.1.2 will be dropped when it is clear from the context which transaction program is being referred to.

For a given set of program parameter values $p \in P$, there may be some states in S that the program will never enter. Those states that transaction $T(p)$ may enter are said to be *reachable* by $T(p)$.

Definition 3.1.3 (Reachable States)
For any transaction $T(p)$ with parameter values $p \in \mathcal{P}_T$, the set $reachable(p) \subseteq S$ is defined (recursively) as follows:

- *The state $initial(p)$ is in $reachable(p)$*
- *A state s' is in $reachable(p)$ if there exists a state s in $reachable(p)$ and a database instance D such that $next(s, D) = s'$.*

3.1.2 Read Selects and Write Selects

The conflict analysis procedure assumes that the behaviour of each transaction program has been characterized by two sets of parameterized queries. One set characterizes the database objects read by the program, the other characterizes the objects inserted or deleted by the program. The elements of the first set are called *read selects*. The elements of the second are called *write selects*.

The read selects for T are denoted by $\mathcal{R}_T$ and the write selects for T are denoted by $\mathcal{W}_T$. Like T itself, the read and write selects for T are parameterized using P_T. If Q is a query in $\mathcal{R}_T$ or in $\mathcal{W}_T$, then $Q(p, D)$ represents

the result of evaluating Q using parameter values $p \in \mathcal{P}_T$ and database D. As was the case for transaction programs, the program-identifying subscripts will be dropped from $\mathcal{R}_T$ and $\mathcal{W}_T$ when the program is clear from context.

The read and write selects characterize the behaviour of transaction programs in the sense that the selects have certain properties that relate them to the execution of the program. Specifically, for each T, the write selects are assumed to have the *write subsumption property* and the union of the read and write selects is expected to have the *read subsumption property* with respect to T.

Definition 3.1.4 (Transaction-Disturbing Delta)
A database delta Δ for state D is said to disturb transaction $T(p)$ (in state D) if there exists a reachable program state $s \in S_T$ such that at least one of the following conditions holds

- $next(s, D) \neq next(s, D + \Delta)$
- $insert(s, D) \neq insert(s, D + \Delta)$
- $delete(s, D) \neq delete(s, D + \Delta)$

A delta Δ for state D is a *minimal disturbing delta* for $T(p)$ in D if it disturbs $T(p)$ and there is no delta Δ_2 for D such that Δ_2 also disturbs $T(p)$ and $\Delta_2 < \Delta$.

Definition 3.1.5 (Read Subsumption Property)

Let T be a transaction program and let $\mathcal{Q}$ be a set of queries, where each query is parameterized by P_T. $\mathcal{Q}$ is said to have the read subsumption property with respect to T if both of the following conditions hold for all databases D in $\mathcal{D}$, all parameter value assignments $p \in P_T$, all program states in $s \in reachable(p)$, and all minimal disturbing deltas Δ for $T(p)$ in state D:

- $x \in \Delta^- \Rightarrow (\exists Q \in \mathcal{Q} : x \in Q(p, D))$
- $x \in \Delta^+ \Rightarrow (\exists Q \in \mathcal{Q} : x \in Q(p, D + \Delta)$

The read subsumption property states that if some change to the database would cause the behaviour of $T(p)$ to change, then the changing database objects are returned by some query in the query set. In other words, the query set $\mathcal{Q}$ identifies that portion of the database to which T's behaviour is sensitive.

Definition 3.1.6 (Write Subsumption Property)

Let T be a transaction program and let $\mathcal{Q}$ be a set of queries. $\mathcal{Q}$ has the write subsumption property with respect to T if the following conditions hold for all databases $D \in \mathcal{D}$, all parameter value assignments $p \in P_T$, all states $s \in reachable(p)$, and all database deltas Δ (from D) such that $\Delta^+ \subseteq insert_T(s, D)$ and $\Delta^- \subseteq delete_T(s, D)$:

- $x \in \Delta^- \Rightarrow (\exists Q \in \mathcal{Q} : x \in Q(p, D))$

- $x \in \Delta^+ \Rightarrow (\exists Q \in \mathcal{Q} : x \in Q(p, D + \Delta))$

The write subsumption property implies that if T inserts an object into the database, then at least one of the queries in $\mathcal{Q}$ would retrieve that object if it was evaluated after the insertion. Similarly, if T deletes an object from the database, at least one of the queries in $\mathcal{Q}$ would retrieve that object if it was evaluated prior to the deletion.

3.2 Pairwise Transaction Analysis

Figure 2.1 on page 6 illustrated the steps involved in pairwise transaction analysis. This section describes those steps in more detail.

3.2.1 Combining Selects

For each transaction program, the read selects are combined to produce a single read query, which is used as input to the pairwise analysis process. The read query for program T is the union of T's read selects. Similarly, the union of the T's write selects becomes the write query for T.

OQL and SQL both have a "union" operator. However, queries can be combined using the union operator only if the query result tuples are type-compatible. In general, the various queries in $\mathcal{R}_\mathcal{T}$ and $\mathcal{W}_\mathcal{T}$ will not be union

compatible. Therefore, the OQL/SQL built-in union operator cannot be used to combine them.

Figure 3.1 illustrates how a union query can be created from two input queries. The procedure can be expanded to generalize the creation of the union of more than two queries.

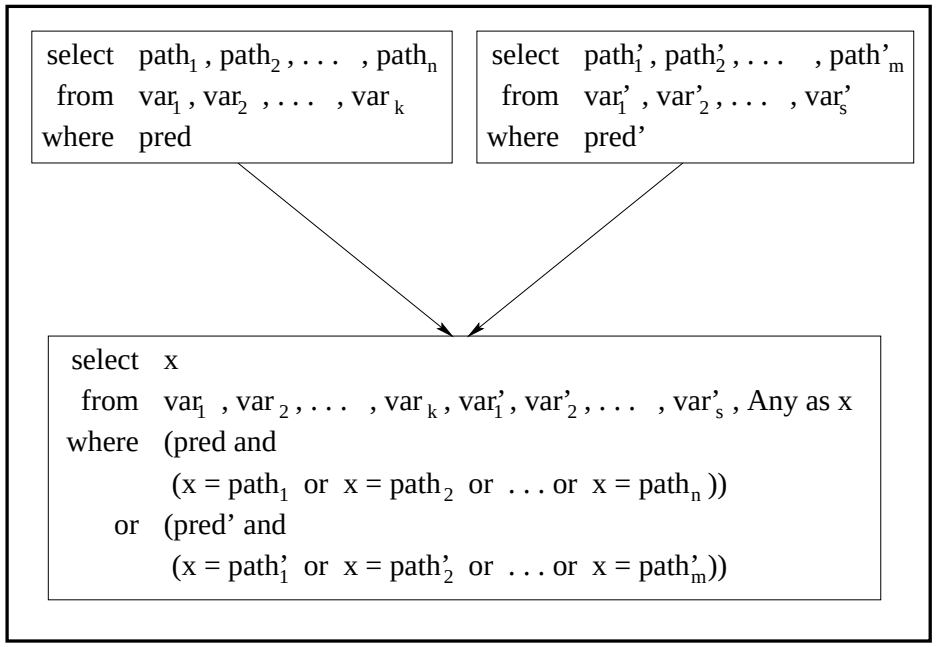


Figure 3.1: Creating the Union of Two Queries

The result of the union query represents the set of objects that exist in either one of the result sets of the input queries. The objects in the result set of the union query have type *Any*. *Any* is explained in Appendix A. It is a supertype of every type. The DEMO optimizer is capable of reasoning about

queries involving Any.

If the set of write selects has the write subsumption property, so will the write query. Similarly, if the set of read selects has the read subsumption property, so will the read query. This follows directly from the definitions of the subsumption properties and the construction of the queries.

Conflict Query Generation

Transactions conflict if they perform operations that do not commute.

Definition 3.2.1 (Transaction Conflict)
Suppose that $T_1(p_1)$ *and* $T_2(p_2)$ *are transactions.* $T_2(p_2)$ *conflicts with* $T_1(p_1)$ *if there exists a database* D*, and program states* $s_1 \in reachable_1(p_1)$ *and* $s_2 \in reachable_2(p_2)$ *such that either*

- $next_1(s_1, D) \neq next_1(s_1, nextDB_2(s_2, D))$*, or*
- $nextDB_1(s_1, D) \neq nextDB_1(s_1, nextDB_2(s_2, D))$

That is, $T_2(p_2)$ conflicts with $T_1(p_2)$ if database insertions or deletions made by $T_2(p_2)$ can cause the behaviour of $T_1(p_1)$ to change. $T_1(p_1)$ and $T_2(p_2)$ are said to be conflicting transactions if $T_1(p_1)$ conflicts with $T_2(p_2)$ or $T_2(p_2)$ conflicts with $T_1(p_1)$ or both.

Conflict queries are used to identify transactions that conflict. Conflict queries are generated from the read and write queries of two transaction program in two steps. The first step intersects the read query R_1 of the transaction program T_1 and the write query W_2 of the transaction program T_2 to produce Q_{RW}. The second step intersects the write query W_1 of T_1 and the read query R_2 of T_2 to produce Q_{WR}.

Figure 3.2 illustrates how the intersection queries are created. The procedure is similar to the procedure used earlier to define the union of queries that may be type-incompatible.

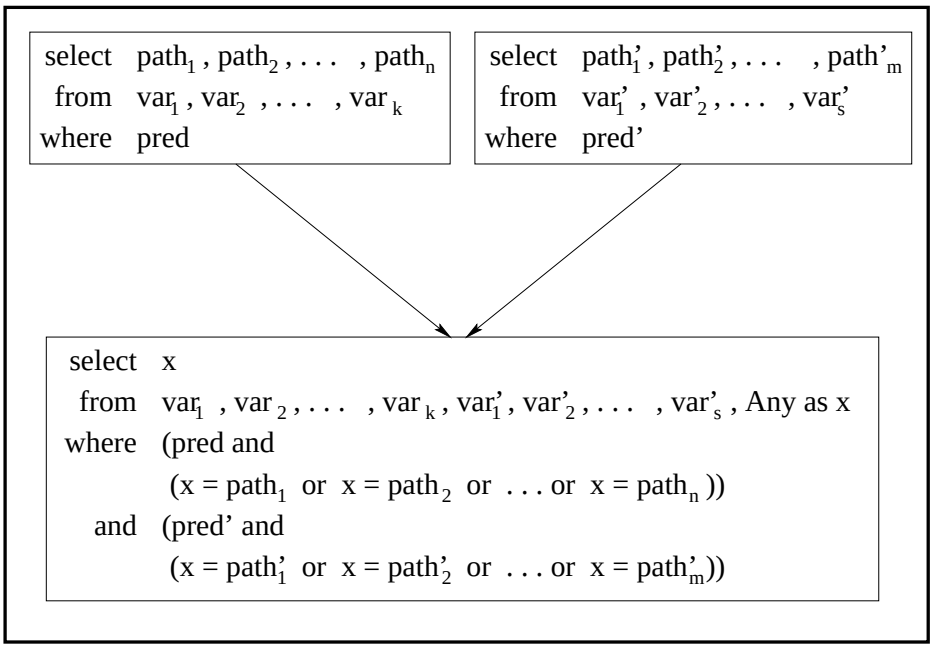


Figure 3.2: Creating the Intersection of Two Queries

Lemma 3.2.1 *Let Q_{RW} be the read/write query produced from the read query for program T_1 and the write query for program T_2. If $Q_{RW}(p_1, p_2, D) = \emptyset$ for all databases D in $\mathcal{D}$, then $T_2(p_2)$ does not conflict with $T_1(p_1)$.*

Proof: Suppose that, for all databases D, $Q_{RW}(p_1, p_2, D) = \emptyset$ and that $T_2(p_2)$ conflicts with $T_1(p_1)$. By Definition 3.2.1 there is a database D and reachable program states s_1 (for $T_1(p_1)$) and s_2 (for $T_2(p_2)$) for which the database delta performed by $T_2(p_2)$ affects the behaviour of $T_1(p_1)$. Let Δ represent $T_2(p_2)$'s $T_1(p_1)$-disturbing database delta from D. Let Δ_{min} be any minimal $T_1(p_1)$-disturbing delta from D such that $\Delta_{min} \leq \Delta$. Since T_1's read query (Q_{R1}) has the read subsumption property, it must be the case that $\Delta^-_{min} \subseteq Q_{R1}(p_1, D)$ and that $\Delta^+_{min} \subseteq Q_{R1}(p_1, D + \Delta_{min})$. Since T_2's write query (Q_{W2}) has the write subsumption property, it must be the case that $\Delta^-_{min} \subseteq Q_{W2}(p_2, D)$ and that $\Delta^+_{min} \subseteq Q_{W2}(p_2, D + \Delta_{min})$. This implies that the intersection of Q_{R1} and Q_{W2} is non-empty either in state D or in state $D + \Delta_{min}$. By construction, Q_{RW} is the intersection of Q_{R1} and Q_{W2} so it is also non-empty in at least one database state, a contradiction.□

Finally, a single query is produced to capture all objects on which the two transaction programs could conflict. This is achieved by taking the union of Q_{RW} and Q_{WR} to get a final query $Q_{conflict}$. Since the intersection queries may not be union-compatible, the method of Figure 3.1 (page 16) is used to define their union.

Theorem 3.2.1 *Suppose that $Q_{conflict}$ is the conflict query generated for transaction programs T_1 and T_2. For all parameter assignments $p_1 \in \mathcal{P}_1$ and $p_2 \in \mathcal{P}_\in$, $T_1(p_1)$ and $T_2(p_2)$ do not conflict if $Q_{conflict}(p_1, p_2, D) = \emptyset$ for all $D \in \mathcal{D}$.*

Proof: This follows immediately from Lemma 3.2.1 and the construction of $Q_{conflict}$.□

EQG Generation

To eventually generate a conflict predicate, it is necessary to determine conditions under which $Q_{conflict}$ will be empty. To achieve this goal, the conflict query is first submitted to the DEMO query optimizer (Appendix C) along with the database schema. The optimizer represents the query internally as an existential query graph (EQG, [3], Appendix C). It applies previously defined inference rules to the graph. If successful, the optimizer returns the modified existential graph as well as an evaluation plan for $Q_{conflict}$. As the optimizer is only allowed to run for a fixed time, it may fail to produce the result in this time.

Predicate Extraction

The existential graph and the evaluation plan for $Q_{conflict}$ are then analyzed to produce a *conflict predicate* C. A conflict predicate is a predicate (in terms of the transaction parameters) which is *FALSE* only if the result of the conflict query is known to be empty for all possible databases.

If the DEMO optimizer fails to finish in the allotted time, the query result is conservatively assumed to be non-empty, i.e., it is assumed that there might exist databases and parameter values for which the two transaction programs conflict. The conflict predicate C is defined to be *TRUE*.

If the DEMO optimizer can infer that the conflict query result must always be empty, then it will return an empty query evaluation plan. In this case, it can be concluded that the two transaction programs do not have common objects upon which they perform conflicting operations. The conflict predicate C is defined to be *FALSE*.

If an evaluation plan for the conflict query is successfully generated and is non-empty, then the two input transaction programs may potentially conflict. In this scenario, the predicate extraction step analyzes the EQG returned by the optimizer and attempts to extract a conflict predicate from it. As noted in Appendix C, the EQG returned by DEMO describes a conjunctive formula

of the form

$$\forall Q | Q \text{is a query result} : (\exists (v_1, \ldots v_n) | (v_1 = Q) \wedge (cond_2) \wedge \ldots \wedge (cond_m))$$

The analysis examines the graph for conjuncts that relate query parameters to other query parameters, or to constants. The conflict predicate C is then defined to be the conjunction of any such conjuncts found by the analysis. (If none are found, then C is defined to be *TRUE*.) Clearly, if such a predicate is *FALSE* then the result of the conflict query must be empty, as required.

The predicate extraction step searches for particular structural patterns in the EQG which correspond to specific kinds of conjuncts. Five specific patterns are considered. Table 3.1 describes the notation that will be used to present these structural patterns. Note that the parameters of the conflict query appear attributes of the query node in the EQG..

Notation	Denotes
V, V_i	nodes V, V_i
S_i	the i^{th} sheet of control
$V \in S_i$	node V in the i^{th} sheet of control
$arc(V_1, p, V_2)$	an arc labelled p that originates from node V_1 and points to node V_2
$loi(V_1, V_2)$	a line of identity arc connecting V_1 and V_2
$\|V\|$	the literal or constant range stored in node V
$\tau(V)$	the type of the object stored in node V

Table 3.1: Notations Representing Elements in an EQG

The five patterns are:

1. a node that has several parameter arcs pointing towards it:

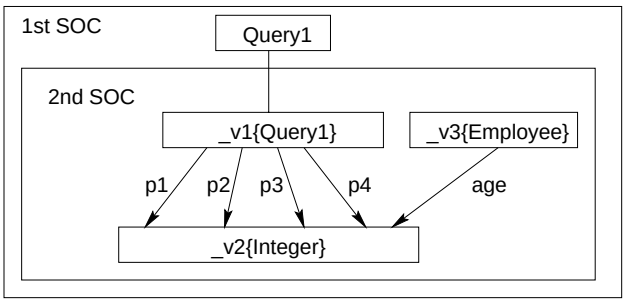
   $\exists\ V,\ V_1,\ V_2,\ loi(V, V_1),\ arc(V_1, p_1, V_2),\ \ldots,\ arc(V_1, p_n, V_2)$

   s.t. $\tau(V) =$ "query," $V \in S_1$, and $V_1, V_2 \in S_2$,

   This corresponds to a conjunct of the form

   $p_1 = p_2 = \ldots = p_n$.

   Here is an example:



   This graph indicates that every "Query1" object has four parameters: "p1," "p2," "p3," and"p4." For every Query1 object, there exists an "employee" object with an attribute "age" whose value matches the query parameters. In this case, the extracted conjunct is $p1 = p2 = p3 = p4$.

2. a node represents a literal, and has parameter arcs pointing towards it:

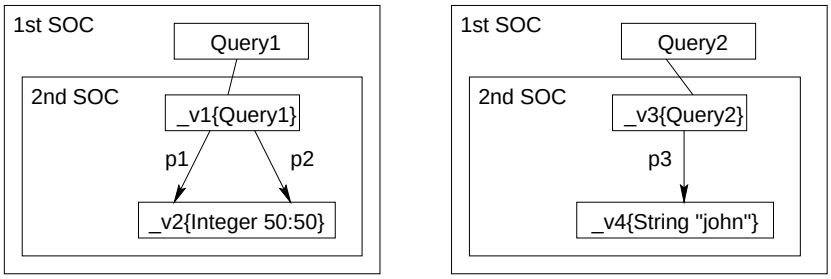
   $\exists\ V,\ V_1,\ V_2,\ loi(V, V_1),\ arc(V_1, p_1, V_2),\ \ldots,\ arc(V_1, p_n, V_2)$

s.t. $\tau(V)$ = "query," $V \in S_1$, $V_1, V_2 \in S_2$, and $|V_2|$ is a literal,

This corresponds to a conjunct of the form

$$p_1 = p_2 = \ldots = p_n = |V_2|.$$

In these two existential graphs:



the first existential graph indicates that $p1 = p2 = 50$. The second graph indicates that $p3 = $'john'.

3. a node represents a constant range, and has a parameter arc pointing towards it:

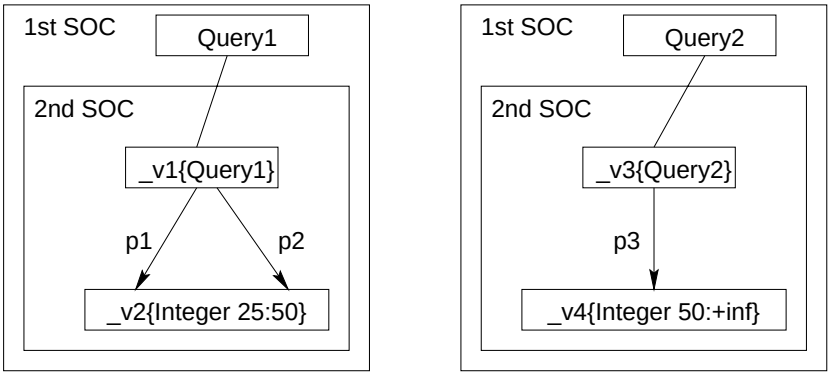
   $\exists\ V, V_1, V_2, loi(V, V_1), arc(V_1, p_1, V_2)$

   s.t. $\tau(V)$ = "query," $V \in S_1$, $V_1, V_2 \in S_2$, and $|V_2|$ is a range,

   This corresponds to a conjunct of the form

   $$|V_2|_{min} \leq p_1 \leq |V_2|_{max}$$

   Here are two examples of such existential graphs:



The first graph indicates that

$$21 \leq p1 \leq 50 \wedge 21 \leq p2 \leq 50$$

In addition, because of the first pattern, the graph also indicates that $p_1 = p_2$. The second existential graph implies that

$$\text{p3} \geq 50.$$

4. two nodes that have parameter arc pointing to them are involved in a "$<$" or a "$\leq$" relationship:

   $\exists\ V, V_1, V_2, V_3, V_4, loi(V, V_1), arc(V_1, p_1, V_2), arc(V_1, p_2, V_3),$

   $arc(V_4, A_1, V_2)$, and $arc(V_4, A_2, V_3)$

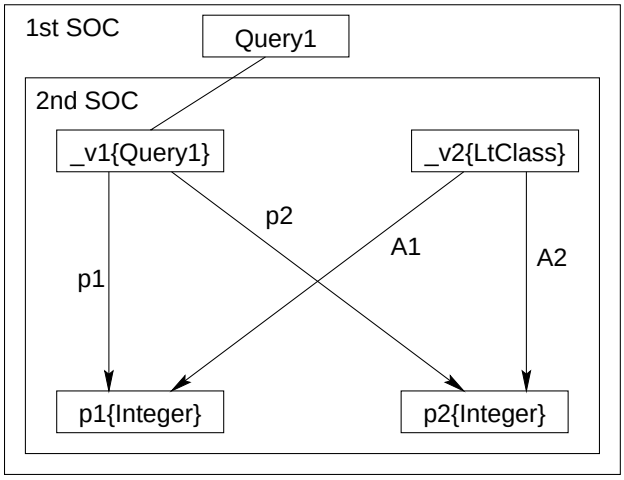
   s.t. $\tau(V) =$ "query," $\tau(V_4) =$ "$<$-class" or "$\leq$-class,"

   $V \in S_1, V_i \in S_2, i = 1, 2, 3, 4$

   This corresponds to a conjunct of the form:

   $p_1 < p_2$ or $p_1 \leq p_2$.

"$A1$" and "$A2$" are key words built into the DEMO system. They represent, respectively, the first and second arguments of a mathematical comparison such as "<," ">," etc. In the DEMO query optimizer, only "<" and "$\leq$" comparisons are used. (">"and "$\geq$" comparisons can be made using "<" and "$\leq$" operators.) Hence, when a directed arc labelled p_1 and a second directed arc labelled $A1$ (or $A2$) both point to node V_2, it follows that p_1 is involved in a "<" or a "$\leq$" comparison. In this example, the graph indicates that $p1 < p2$:



5. a node has a parameter arc pointing towards it; the parameter arc is involved in an identity relation or the negation of an identity relation.

   $\exists\ V,\ V_1,\ V_2,\ V_3,\ loi(V, V_1),\ loi(V_2, V_3)$, and $arc(V_1, p_1, V_2)$

   s.t. $\tau(V) =$ "query," $V \in S_1$, $V_1, V_2 \in S_2$, and $V_3 \in S_i$, $i \geq 2$,

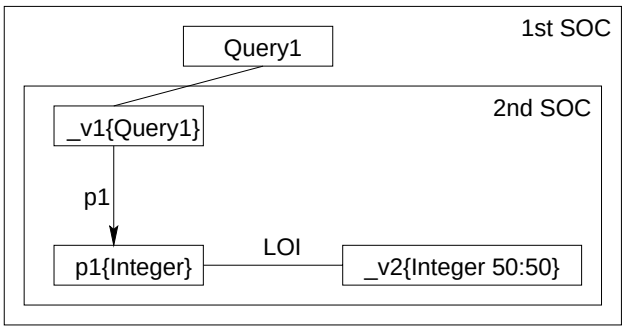
   and $|V_3|$ is a literal.

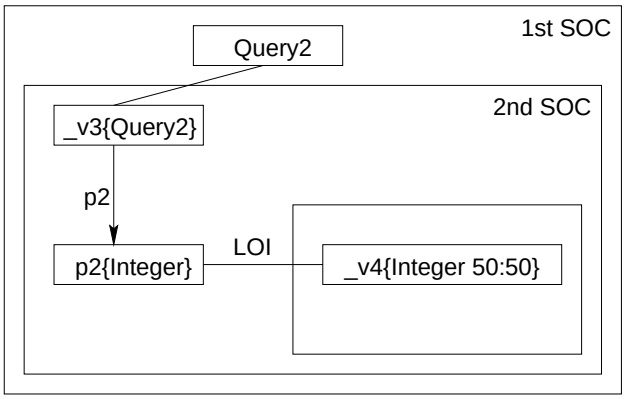
This corresponds to conjuncts of the form

$$p_1 = |V_3| \text{ or } p_1 \neq |V_3|$$

Every sheet of control in an existential graph introduces a new nested scope for additional nodes and graphical elements, and constitutes a negation. If the number of sheets nested between V_2 and V_3 is even, then the even number of negation operations will produce a positive operation: $p_1 = |V_3|$. If the number of sheets nested between V_2 and V_3 is odd, then a negation operator must be inserted: $p_1 \neq |V_3|$.

Here are two examples:





The first graph indicates that $p1 = 50$. In the second graph, the line of identity crosses from one sheet of control into another which implies a negation relation. This indicates that $p1 \neq 50$.

The predicate extraction step will scan an input existential graph, and search for patterns in the graph that match the descriptions of any of the above five cases. Whenever a matching pattern is found, the predicate implied by the pattern is inserted into a predicate set. Finally, after the entire existential query graph has been scanned, all of the predicate expressions in the resulting predicate set are "AND-ed" together to form a conjunctive expression. This expression represents the conflict predicate, C, that is, the predicate condition under which the result of the intersection query is non-empty.

It must be pointed out that C could be simplified, and some simplification attempts have been made in the subprocedures that perform the analysis of EQG's. For example, nested negations are simplified:

$$\underbrace{not(not(\ldots(not}_{\text{n times}}(E))\ldots)) \Longrightarrow \begin{cases} E & \text{if n is even} \\ not(E) & \text{if n is odd} \end{cases}$$

However, currently, the simplification is incomplete. For example, if an existential graph indicates that "p1" and "p2" are equivalent, both (p1 = p2) and (p2 = p1) will appear as conjuncts in the conflict predicate C, even though one of them would suffice.

3.3 Implementation

OQL query parsing and manipulation routines are required for a number of the procedures described in this chapter and the next, such as the transformation of insert, delete and update statements into corresponding select statements, the "pump-and-explode" method, the query union procedure, and the query intersection procedure. The DEMO canonicalizer and its OQL parser are used as tools to assist in implementing these procedures.

The DEMO canonicalizer transforms OQL queries into a canonical form (described in more detail in Appendix C). The main advantage of the canonical form is that a canonicalized query can be easily mapped into an existential query graph. The existential graph of a query captures the query and all of the information associated with it, such as schema information, attribute constraints, subclass definitions, property definitions, path equations, functional dependencies, etc. (See Section 3.2.1 on page 21 for more details.) In terms of query processing, the canonical form also offers a uniform, predefined structure into which all OQL queries can be converted. Having a uniform structure for queries simplifies parsing and manipulation of queries.

The canonicalizer is part of DEMO's OQL parser. The parser also contains other data structures used to represent OQL queries. The algorithms and procedures described in this thesis make extensive use of the data structures and methods defined and implemented by the parser.

Chapter 4

Generating Read and Write Selects

As described in Chapter 3, the analysis of a pair of transaction programs requires, as input, sets of read and write selects for each program. In general, these queries must be generated manually by inspection of the transaction programs. However, for some restricted classes of programs it may be possible to automate the extraction of read and write selects from the transaction programs themselves. This section illustrates some of the issues involved by considering how to extract read and write selects for a restricted class of transaction programs that interact with the database using embedded SQL.

Figure 4.1 illustrates in more detail the steps involved in the process. In the

operation extraction stage, a given transaction program T is scanned and embedded SQL statements are extracted from the program. The output is a set of read operations and a set of write operations that describe the database operations performed by the transaction program. Next, a set of read selects and a set of write selects are produced from the read and write operation sets for the given program T.

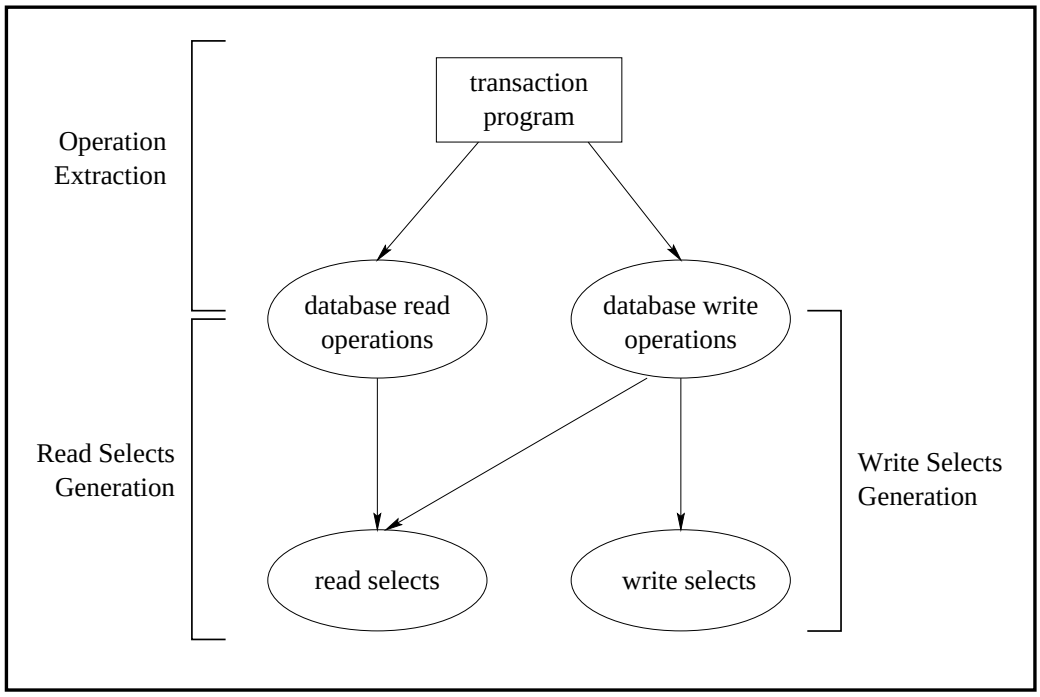


Figure 4.1: Generating Read and Write Selects

4.1 Generating Read and Write Operations

The operation extraction steps involve scanning a given transaction program to extract embedded database operations. These operations are assumed to be extended SQL select, insert, delete and update statements of the restricted forms described in Appendix A. The statements may involve host program variables, but such variables must be input parameters of the transaction program. Furthermore, it is assumed that the program does not change the values of any such parameters prior to their use in any embedded statement. It would be possible for the generation process to support more general relationships between database operation parameters and program parameters. However, such support would require control-flow and data-flow analysis of the transaction program, which is beyond the scope of the thesis.

4.2 Generating Read and Write Selects

In this section, the database read and write operations are further processed and transformed into a set of read select statements and a set of write select statements which can be used as input to the pairwise conflict analysis.

4.2.1 Generating Read Selects

The goal of these steps is to produce a set of read operations (the read selects). The read selects should have the read subsumption property defined in Chapter 3.

Intuitively, the read selects are expected to capture those database objects that, if changed, could affect the set of database objects read, inserted, deleted, or updated by an operation. For example, the following database read operation

```
select E.salary
  from employee as E
 where E.age > 21
```

returns the salary attributes of the employee objects. The age attributes of the employee objects are not returned by the database operation. However, the "where" clause of the select statement ensures that only the salary values of employees over the age of 21 are included in the result of the operation. That is, objects that are not actually returned to the host application by the read operation can nonetheless affect what it reads. Therefore, the read select generated for a database query is a query that returns all of the database objects on which the original query depends.

The generation of read selects involves two sub-procedures. First, the "pumping procedure" adds all of the path expressions from the "where" clause to

the "select" list of the read operation. Figure 4.2 explains the procedure in detail. The pumping procedure is necessary because when given a select statement that represents a database read operation, any change to an object appearing in the "where" clause may affect which objects the operation reads.

The second sub-procedure is the "exploding procedure". For each path expression in the "select" list, the exploding procedure adds all of the prefixes of that path expression to the "select" list. Figure 4.3 explains the procedure in detail.

In the object-oriented database model described in Appendix A, a path expression P refers to a particular object X. The path expression $P = p_1.p_2.p_3...p_{n-1}.p_n$ also specifies the objects through which X is reached. Any change to one of these intermediate objects may cause the path expression to refer to a different object. The exploding procedure ensures that these objects are included in the result sets of the read selects.

```
// pump
//   input: database read operation
//   output: a new "select" list
List pump(Select readOp)
{
  // The list is initialized to contain the original
  // "select" list of the database read operation.
  List sList = readOp.getSelectList();

  // Call the actual recursive pumping procedure.
  return pumpWhereClause(readOp, sList);
}

// pumpWhereClause
//   input: the select statement and the input "select" list
//   output: the new "select" list containing all of the
//           path expressions in the "where" clause
List pumpWhereClause(Select sel, List sList)
{
  // Retrieve the path expressions in the "where" clause
  List pathExpList = sel.getPathExpsInWhere();

  // Add each path expression to the output "select" list
  for (each path expression, p, in pathExpList) {
    sList.insert(p);
  }

  // Go through subqueries recursively.
  for (each subquery select statement, q, in "sel") {
    sList = pumpWhereClause(q, sList);
  }

  return sList;
}
```

Figure 4.2: Generating Read Selects: "Pumping"

```
// explode
//   input: a "select" list (of objects)
//   output: the exploded "select" list
List explode(List sList)
{
  List newList = sList;
  for (each object path expression, pathExp, in "sList") {
    newList = explodePath(pathExp, newList);
  }
  return newList;
}

// explodePath
//   input: the path expression and the existing
//          "select" list
//   output: the exploded "select" list
List explodePath(PathExpression pathExp, List sList)
{
  List newList = sList;

  // If "pathExp" can be parsed into [HEAD].[tail],
  // where pathExp=p(1).p(2)...p(n),
  // HEAD=p(1).p(2)...p(n-1), and tail=p(n)
  if ("pathExp" has format [HEAD].[tail]) {
    newList = explodePath(pathExp.getHEAD(), newList);
  }

  if ("pathExp" does not exist in "newList" already) {
    // Add the path expression into the list
    newList.insert(pathExp);
  }

  return newList;
}
```

Figure 4.3: Generating Read Selects: "Exploding"

Here is an example. Given the following database read operation:

```
select E.salary
  from employee as E
 where E.age > 21
   and E.dept.name = "CS"
```

The pumping procedure transforms it into:

```
select E.salary, E.age, E.dept.name
  from employee as E
 where E.age > 21
   and E.dept.name = "CS"
```

and the exploding procedure transforms the above statement into:

```
select E, E.salary, E.age, E.dept, E.dept.name
  from employee as E
 where E.age > 21
   and E.dept.name = "CS"
```

This query returns all of the database objects on which the original query depends.

Similarly, a database *write* operation (represented by an insert, delete or update statement) produces a read select that describes all of the database objects that might affect the set of objects inserted, deleted, or updated by the statement. The method used to achieve this goal is summarized in Table 4.1.

Database	Write Operation	Read	Select
insert into	obj_1		
	$attr_1$, $attr_2$, ...		
values	$value_1$, $value_2$, ...		
delete	obj_1	select	*list*
where	*pred*	from	obj_1
		where	*pred*
update	obj_1	select	*list*
set	$attr_1 = expr_1$,	from	obj_1
	$attr_2 = expr_2$,	where	*pred*
	...		
where	*pred*		

Table 4.1: Generating Read Selects from Database Write Operations

The format of the insert statement indicates that the objects being added into the database have values that are explicitly specified in the "values" list. Since the object to be inserted depends only on the values list and not on the database, the insert statement does not result in a read select.

For the database write operation represented by a delete statement, the set of objects to be deleted depends on the "where" clause. As a result, the read select produced from "delete" has the same "where" clause as the "delete." The "select" list of the read select is then created using the pump-and-explode method (Figure 4.2 and Figure 4.3).

For example, the delete operation

```
delete employee as E
 where E.age <= 21
```

will be transformed into the read select

```
select E.age, E
  from employee as E
 where E.age <= 21
```

An update statement can be viewed as a delete operation followed by an insert operation. The set of deleted objects includes all of the objects whose attribute values satisfy the predicate in the "where" clause of the update statement. The inserted objects have attribute values are determined by the expressions ($expr_i$) in the "set" clause of the update statement. Since the set of objects to delete depends on the objects appearing in the "where" clause of the update statement, the read select produced from "update" has the same "where" clause. The final select list is then generated using pump-and-explode.

The "set" clause allows the new attribute values of the object being updated to depend on the objects appearing in the expressions on the right hand side of the equations. Any change to these objects may affect the set of new objects. Therefore, these objects must be included in the "select" list of the read select for "update."

Here is a simple example. The following update operation:

```
update employee as E
   set E.salary = E.salary + 100,
       E.dept.name = "CS"
 where E.age > 21
```

will produce the following read select:

```
select E, E.age, E.salary
  from employee as E
 where E.age > 21
```

The path `E.age` is placed in the select list when the "where" clause is pumped. The path `E.salary` comes from the "set" expression in the original query. Finally, the path `E` is generated when `E.age` and `E.salary` are exploded.

4.2.2 Generating Write Selects

The goal of the steps here is to produce a set of write selects that have the write subsumption property. Table 4.2 summarizes the method used.

The "values" list of the insert statement explicitly specifies the attribute values of the objects being added into database D. The write select will thus have a "where" clause that is a conjunction of equalities. For example, an insert statement such as the following:

Database	Write Operation	Write	Select
insert into	obj_1	select	obj_1
	$attr_1$, $attr_2$, ...	from	obj_1
values	$value_1$, $value_2$, ...	where	$attr_1 = value_1$
		and	$attr_2 = value_2$
		and	...
delete	obj_1	select	obj_1
where	$pred$	from	obj_1
		where	$pred$
update	obj_1	select	obj_1
set	$attr_1 = expr_1$,	from	obj_1
	$attr_2 = expr_2$,	where	$pred'$
	...		
where	$pred$		

Table 4.2: Generating Write Selects from Database Write Operations

```
insert into employee as E
            E.salary, E.age
     values 1000, 40, "CS"
```

will produce the following write select:

```
select E
  from employee as E
 where E.salary = 1000
   and E.age = 40
```

For the delete statement, the "where" clause of the delete statement describes the deletion condition. The "where" clause of the write select will have exactly the same condition. For example, if given a database write operation

```
delete employee as E
 where E.age <= 21
```

then using Table 4.2, it will generate the write select

```
select E
  from employee as E
 where E.age <= 21
```

The update statement is viewed as a delete operation followed by an insert operation. The deleted objects are those whose attribute values satisfy the predicate *pred* in the "where" clause of the update statement. If none of the attributes that appear in the "where" clause are modified by the "set" clause, then the deleted objects will also satisfy *pred*. In this case, *pred* is used as the predicate for the write select. Otherwise, the inserted objects may not satisfy *pred*. In this case, the simple, conservative approach of using *TRUE* as the write select's "where" predicate is taken.

Here are two examples. First, using Table 4.2, the update operation

```
update employee as E
   set E.salary = E.salary + 100
 where E.age > 21
```

will produce the following write select

```
select E
  from employee as E
 where E.age > 21
```

On the other hand, if the update operation is

```
update employee as E
   set E.salary = E.salary + 100
 where E.salary > 1000
```

then the write select becomes

```
select E
  from employee as E
```

Note that it would have been possible to infer that the write select set could be limited to include only those employees with salaries greater than 1000, since the update statement increases salaries. However, this would require interpretation of the expressions in the set clause.

Chapter 5

Examples

This chapter contains three examples that illustrate how the semi-automatic technique can be applied to analyze conflicts in transaction programs. Each example shows the results of the individual steps of the technique and the final result.

The input in Example 1 consists of two basic transaction programs. One of them is comprised of a simple read operation that conflicts with a simple write operation in the other transaction program. In Example 2, the two transaction programs perform non-conflicting database operations. Example 3 demonstrates the analysis of transaction programs whose database operations contain parameters.

An implementation decision has been made to convert intermediate queries

in the semi-automatic process into DEMO canonical forms. (The decision is justified in Section 3.3 on page 29 and in Appendix C). The use of the DEMO query canonical forms simplifies query processing, but canonicalized queries are not (nor are they meant to be) easy for humans to read. The queries in the examples below have been kept in SQL-like forms to improve readability.

5.1 Example 1

The input in Example 1 consists of two database transaction programs. The first program, T_1, reads all of the employee objects from the sample database (Appendix B), and the second program, T_2, sets the salary attribute of every employee object to "100." The steps in Figure 4.1, "Generating Read and Write Selects," are applied first to T_1, and then to T_2.

In the process described in Chapter 4, "Generating Read and Write Selects," the read and write operations in T_1 are first manually separated and rewritten. T_1 can be described by the database read operation:

```
select E
  from employee as E.
```

This read operation is transformed into a read select statement using the "pump-and-explode" method:

```
select E
  from employee as E
```

Due to the simplicity of the input select statement, the "pump-and-explode" method does not actually change the query. Since T_1 does not contain any write operations, at the end of generating read and write selects, the output read query Q_{R1} is the same as the above read select, and the output write query Q_{W1} is null.

The same steps are now applied to T_2. T_2 can be described by a database write operation:

```
update employee as E
   set E.salary = 100
```

Using the third rule in Table 4.1, this write operation is transformed into a read select:

```
select E
  from employee as E
```

and a write select using the third rule in Table 4.2:

```
select E
  from employee as E
```

These are the corresponding read query (Q_{R2}) and write query (Q_{W2}) for T_2.

Now that the read and write queries for both T_1 and T_2 have been produced, the process proceeds to pairwise transaction analysis. First, Q_{R1} of T_1 is intersected with Q_{W2} of T_2. Using the method described in Figure 3.2, the resulting query Q_{RW} is:

```
select x
  from employee as E, employee as E2, Any as x
 where (x = E)
   and (x = E2)
```

Next, Q_{W1} is intersected with Q_{R2} to generate Q_{WR}. Since Q_{W1} is null, Q_{WR} is also null. Thus, the conflict query generation step which produces the union of the queries Q_{RW} and Q_{WR} is simplified. The final query $Q_{conflict}$ is the same as Q_{RW}. The result of $Q_{conflict}$ contains all of the objects on which T_1 and T_2 might perform conflicting operations.

In the EQG generation step, $Q_{conflict}$ and the database schema defining $Q_{conflict}$ are submitted to the DEMO query optimizer. The optimizer applies previously defined inference rules to $Q_{conflict}$, and successfully generates a query evaluation plan and an existential graph. Since there are no parameters in T_1 or T_2, the existential graph does not contain any patterns involving parameters. Therefore, the conflict predicate P is set to $TRUE$ and returned as the result of the conflict analysis. This means that, as expected, T_1 and T_2 might perform conflicting operations.

5.2 Example 2

In this example, T_1 deletes all of the employee objects whose salary attribute values are greater than 100, and T_2 reads the name and age attributes of an employee object if the salary attribute value of the object is equal to 100.

Using the syntax for database operations defined in Appendix A, T_1 can be described by the database read operation:

```
delete employee as E
 where E.salary > 100
```

This write operation is transformed into a read select using the second rule in Table 4.1:

```
select E, E.salary
  from employee as E
 where E.salary > 100
```

and a write select using the second rule in Table 4.2:

```
select E
  from employee as E
 where E.salary > 100
```

These are Q_{R1} and Q_{W1} for T_1 at the end of generating read and write selects.

Next, T_2 goes through the same steps. The actions in T_2 can be described as the following read operation:

```
select E.name, E.age
  from employee as E
 where E.salary = 100
```

Using the "pump-and-explode" method, the above read operation is transformed into a read select:

```
select E, E.name, E.age, E.salary
  from employee as E
 where E.salary = 100
```

This read select becomes Q_{R2}. Q_{W2} is null because T_2 does not modify the database.

In the conflict query generation stage of the pairwise transaction analysis, since Q_{W2} is null, the only intersection query that is not null is Q_{WR} which results from intersecting Q_{W1} with Q_{R2}. Using the method described in Figure 3.2, Q_{WR} becomes:

```
select x
  from employee as E, employee as E2, Any as x
 where ((E.salary > 100) and
        (x = E))
   and ((E2.salary = 100) and
        (x = E2 or x = E2.name or
         x = E2.age or x = E2.salary))
```

This is the final query $Q_{conflict}$.

In the EQG generation step, $Q_{conflict}$ and the database schema defining $Q_{conflict}$ are submitted to the DEMO query optimizer. The optimizer applies previously defined inference rules to $Q_{conflict}$, and cannot generate a query evaluation plan because of the unsatisfiable condition in the "where" clause of $Q_{conflict}$:

```
(employee.salary > 100) and (employee.salary = 100)
```

Therefore, T_1 and T_2 do not perform conflicting operations, and the predicate extraction step returns the conflict predicate *FALSE*.

5.3 Example 3

The transaction programs in Example 3 are similar to those in Example 2. The only difference is, in this example, instead of deleting all of the employee objects whose salary attribute values are greater than 100, T_1 deletes all of the employee objects whose salary attribute values are greater than "p1," where "p1" is a parameter whose value is unknown at the time of the analysis. Similarly, T_2 reads the name and age attributes of an employee object if the salary attribute value of the object is equal to an unknown value represented by the parameter "p2."

Therefore, T_1 can be described by a database write operation as:

```
delete employee as E
 where E.salary > :p1
```

and T_2 by a database read operations as:

```
select E.name, E.age
  from employee as E
 where E.salary = :p2
```

The steps to generate read and write selects are applied to T_1 and T_2 much like in Example 2. At the end of the process, Q_{R1} is created as

```
select E, E.salary
  from employee as E
 where E.salary > :p1
```

and Q_{W1} as

```
select E
  from employee as E
 where E.salary > :p1
```

Q_{R2} of T_2 is

```
select E, E.name, E.age, E.salary
  from employee as E
 where E.salary = :p2
```

and Q_{W2} is null.

Similarly, $Q_{conflict}$ is generated as

```
select x
  from employee as E, employee as E2, Any as x
 where ((E.salary > :p1) and
        (x = E))
   and ((E2.salary = :p2) and
        (x = E2 or x = E2.name or
         x = E2.age or x = E2.salary))
```

The result of $Q_{conflict}$ contains all of the objects on which T_1 and T_2 might perform conflicting operations.

In the EQG generation step, $Q_{conflict}$ and the database schema defining $Q_{conflict}$ are submitted to the DEMO query optimizer. The optimizer applies inference rules to $Q_{conflict}$, and successfully generates a query evaluation plan and an existential graph for $Q_{conflict}$. The complete graph for $Q_{conflict}$ is very large. Figure 5.1 displays a segment of the graph that contains the relevant patterns.

The graph segment matches the pattern described in case 4 of the EQG analysis, (see Section 3, page 25). The resulting conflict predicate is

p1 < p2.

When T_1 and T_2 are executed, the actual values for "p1" and "p2" will be substituted into the expression to determine whether conflicts can occur. If

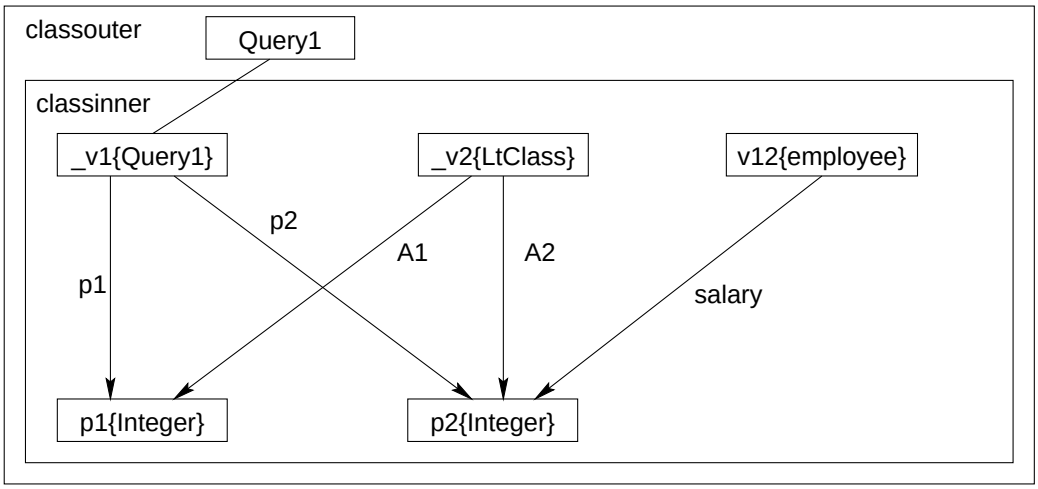


Figure 5.1: Example 3: Segment of Existential Graph

the value of "p1" is less than that of "p2," then (p1 < p2) evaluates to *TRUE*, which means that T_1 and T_2 may perform conflicting operations. On the other hand, if for the given values of "p1" and "p2," (p1 < p2) evaluates to *FALSE*, then T_1 and T_2 do not perform conflicting operations.

Chapter 6

Applications and Related Work

There are several possible applications of the technique described in the thesis. Three particular areas of application will be discussed: transaction management, active databases and the processing of standing queries.

6.1 Application 1: Transaction Management

The concept of transactions originated in the 1950's [4, 5, 1]. Since then, research in the area has resulted in a conceptual transaction system structure shown in Figure 6.1 [4].

The *transaction manager* manages the execution of transactions that access data stored in a local site. The *scheduler* interprets all calls as sequences

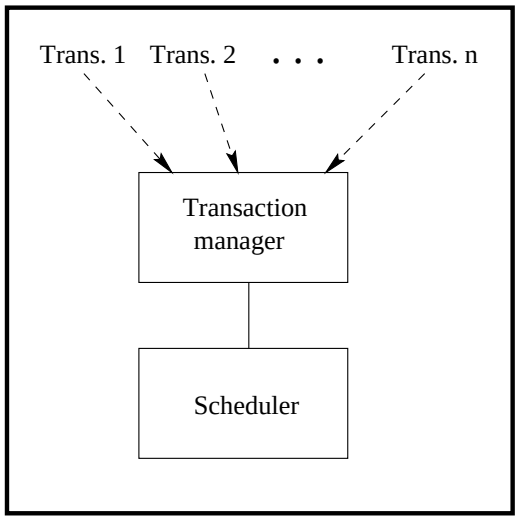


Figure 6.1: Layering of a Transaction System

of reads and writes, and assures a *serializable* schedule. In such a system, transactions usually run concurrently to improve performance, but appear to have been executed sequentially.

The concurrency control problem is the problem of ensuring serializability by controlling the interleaving of read and write operations of conflicting transactions. Concurrency control schemes are typically designed to support dynamically created and executed queries and updates against disk resident databases. However, for certain types of systems, such as embedded control systems, it is not necessary to support arbitrary, dynamic queries and updates. Transaction types in these systems are known in advance, and stay relatively static. By analyzing such transactions before they run, it is pos-

sible to reduce or eliminate the need for locking or other dynamic controls, and thus optimize transaction concurrency control.

The basic idea of the thesis comes from [2], in which Eswaran et al. suggested using predicate locks to ensure consistency. Instead of locking an individual object in a database, predicates are used to specify a logical subset of the database. In general, a transaction, T, could request a lock:

$$< t, [slock|xlock], predicate >.$$

Two predicate locks $< t, mode, p >$, $< t', mode', p' >$ are compatible if [6]:

- $t = t'$ (a transaction does not conflict with itself), or
- both modes are SHARE (shared locks do not conflict), or
- the predicate (p AND p') is not satisfiable (no object can satisfy both predicates).

In this algorithm, transactions dynamically request locks on sets of objects. Each object set is described by a predicate. One possible implementation of a predicate lock system is as follows [6]: when a transaction requests a predicate lock, the system compares the lock request with the other granted predicate locks. If the lock request is compatible with all of the granted requests, it is added to the granted list and granted immediately. Otherwise, the lock request is added to a waiting list. As soon as a transaction ends, its

predicate locks are removed from the granted list. Afterwards, the requests in the waiting list will be re-evaluated against the requests in the granted list. Each predicate compatible with the new granted group of predicates is added to the granted list and granted.

There are a number of differences between the use of predicates in the above implementation of a predicate locking system and the conflict analysis technique described in this thesis. First, the predicate definitions in predicate locks contain no parameters because the parameter values are known at run-time. The cost of comparing predicates is paid at run-time. In embedded systems where transactions are known in advance, conflict expressions for pairs of transactions can be defined statically (before transaction run-time), stored in the system, and evaluated at run-time. Therefore, the expressions generated by the technique of the thesis allows parameterized queries and produces a predicate (in the parameters) as its result.

A second difference is the granularity of coverage of the predicates. The analysis required by predicate locking is performed each time a transaction program performs a database operation. Each predicate defines the footprint of a single operation. The conflict analysis described in the thesis is intended to detect conflicts between complete transactions. In addition, Chapter 4 addresses in more detail on which objects database read and write operations (represented by SQL queries) depend, and how to extract those objects from the SQL queries in an object database. The predicate locking scheme assumes

the existence of a predicate that specifies the logical subset of objects in a transaction request.

Because its predicates cover individual operations, predicate locking may involve multiple predicate tests during the lifetime of a transaction. An advantage of this is that it is not necessary to have a complete description of what the transaction will need to read/write before it starts running. A disadvantage is that a transaction might have to abort if it asks for some part of the database that it cannot have. This cannot happen with PAPRICCA.

The conflict analysis in this thesis, when combined with the algorithm PAPRICCA [7], results in a concurrency control algorithm. In PAPRICCA, a transaction undergoes a single concurrency control test before it starts. Once it passes this test, it is free to run without any further concurrency control. This is possible because the transaction read/write sets essentially pre-declare all parts of the database that the transaction will need. The algorithms proposed in this thesis can be used to generate conflict predicates required to perform the concurrency control test in PAPRICCA.

The issue of recovery arises alongside concurrency control as recovery is needed to guarantee atomicity and durability when failures occur. Raj Rathee's thesis [8] presented a recovery algorithm (*Fuzzy_PBL*) that takes advantage of known transaction data access patterns to reduce recovery related overhead. Fuzzy_PBL takes checkpoints, and performs a combination of transaction level logging and page level logging. Transaction logging is

very efficient but can only be used if the transaction being logged does not interfere with an ongoing checkpoint. The static conflict analysis presented here can be used to generate a predicate conflict expression that will evaluate to *FALSE* only if the transaction does not interfere with a checkpoint (described by a transaction program).

6.2 Application 2: Active Databases

Another possible application area is active database systems [9, 10, 11]. An active database system consists of a (passive) database and a set of active rules. Typically, a rule consists of an event, a condition, and an action. An event is usually an insertion, deletion, or update into a table. A condition is similar to a query, except that the query may be able to refer to the inserted or deleted tuples, as well as the database. An action is a transaction (a program) that may make changes to the database. When triggered by an event, an active database performs an action according to the predefined rule conditions. For example, a rule (often called a trigger) might say: If a tuple is modified in the Emp table (the event), and if the "age" field of the tuple becomes greater than 55 (the condition), then add a new tuple to some other table (the action).

Suppose that there exists a pre-defined update transaction that updates the Emp table tuples whose "age" field values are less than x. Using the technique

presented in this thesis, it is possible to create a predicate expression that evaluates to *FALSE* if there is no overlap between the tuples updated by the transaction (t in Emp such that $t.age < x$) and the tuples satisfying the rule condition (t in Emp such that $t.age > 55$). In this particular case, the predicate expression is ($x > 55$). Before the update transaction is processed, x is bound to an actual value, and the predicate expression is evaluated. If the expression evaluates to *FALSE* (i.e. there is no overlap), then the update transaction need not cause the particular trigger to be activated.

The analysis described in this thesis can create a predicate expression which describes the overlap between the update transaction and the rule conditions of a trigger. The predicate expression provides a way to optimize the evaluation of a rule condition. That is, if the predicate expression is *FALSE*, the evaluation of the rule condition can be omitted, but if the expression is *TRUE*, the rule condition must be evaluated.

6.3 Application 3: Standing Queries

Similar to an active database, a standing query performs an action when it is triggered by an event [12]. A common scenario using standing queries is the "pub-sub" model. In it, a client subscribes to a set of things described by a query, and the server publishes material. The client wants to know if any new publication is relevant with respect to his subscription. Here, the event

is the arrival of a new publication. The standing query defines the client subscription. The action will be to notify the client of a newly available relevant publication.

If server publications can be constructed as pre-defined transactions, then the conflict analysis algorithm of this thesis can be employed to produce a predicate expression that evaluates to *FALSE* only if there is no overlap between such a transaction and the standing query. When the server publishes new material using one of the pre-defined transactions, the predicate expression will be tested. In the case of the result being *FALSE*, the corresponding standing query need not be re-evaluated.

One other similar topic is the efficient maintenance of materialized views [13]. There, the result of a query is stored (materialized), and the question becomes whether an update to the database will cause the materialized result to change. The static analysis can be used to discover the overlap between the stored result (described by a query) and the update action (described by a transaction).

6.4 Predicate Satisfiability

The problem of generating a conflict predicate from $Q_{conflict}$ is closely related to the predicate satisfiability problem. The predicate satisfiability problem is known to be undecidable in general. It is the problem of determining whether

there exists a variable assignment that will cause a given predicate to evaluate to *TRUE*. For a more restricted version of the general case, studies ([14], [15]) have considered the satisfiability problem of conjunctive predicates of the form (X *op* C), or (X *op* Y), or (X *op* Y + C), where X and Y are attributes, C is a constant and $op \in \{<, \leq, =, \neq, >, \geq\}$. This type of satisfiability problem is also NP-complete. However, for the same problem in the real domain [14], or in the integer domain [15] where $\neq$ is removed from the set of operators, there exist polynomial algorithms to solve the problem.

By imposing resource bounds on the computation, the DEMO optimizer avoids spending a very long time trying to generate a conflict predicate. If the computation is taking too long, the DEMO optimizer simply gives up. That is, DEMO may fail to recognize an empty $Q_{conflict}$ (a conflict query with an unsatisfiable set of predicates) as such.

Chapter 7

Summary and Future Work

This thesis describes a semi-automatic process that takes a pair of transaction programs and analyzes them for potential run-time conflicts. Transaction programs may be parameterized. The conflict analysis process occurs before the transaction programs are executed. At this time, parameter values are not known. For this reason, the result of the analysis is a conflict predicate written in terms of the parameters of the transaction programs. The DEMO optimizer is the main tool used during the analysis process to generate the conflict predicate.

Values will be bound to the parameters each time the program is executed. Two transactions can be checked for potential conflicts by substituting their parameter values into the parameterized conflict predicate for the corresponding pair of transaction programs. A conflict predicate that evaluates to *TRUE*

indicates that the transactions may conflict. A conflict predicate that evaluates to *FALSE* indicates that conflict will not occur.

The conflict analysis technique described in this thesis is suited for use in main memory database systems [16] where high transaction throughput is required, or embedded control systems where transactions are known in advance. Chapter 6 describes several potential applications of the technique, such as transactional recovery (the Fuzzy_PBL algorithm [8]) and concurrency control (the PAPRICCA algorithm [7]).

In order to simplify the conflict analysis, the input transaction programs need to be processed manually first to create database read and write operations that conform to the the syntax of Table A.1 (Appendix A and Chapter 4). Possible future studies could look into automating the extraction of database read and write operations from transaction programs written in a common embedded query language (for example, embedded SQL in C). This will improve the usability of the conflict analysis process.

A closer look at the steps described in Chapter 3 points out that the technique is conservative. Improvements can be made on the existing steps to increase concurrency. For instance, it is possible to interpret the expressions in the set clause of an update statement representing a database write operation (see the end of Chapter 4). The result can be used to narrow the sets described by the write selects, which may reduce the size of the conflict set at the pairwise analysis stage. A smaller conflict set can potentially increase transaction

concurrency.

In Chapter 3 on page 28, it was mentioned that the final conflict predicate expression can be further simplified. Converting a predicate to a "simpler" form will reduce the time required to evaluate the predicate against a set of parameter values. Simplifying predicates will involve looking into the problem of predicate equivalence.

Appendix A

Database Model

Because DEMO, the main tool required for implementation, provides an object-oriented database model, this thesis also assumes an object-oriented database model. The model follows the object model defined in DEMO. DEMO in turn follows the ODMG'93 standard for object-oriented databases [17]. The main elements of this object database model are:

- A *database* stores objects, enabling them to be shared by multiple users and applications.
- The basic database modeling primitive is the *object*. Objects are categorized by their *types*.
- The type of an object defines a set of *properties*. These properties can

be *attributes* of the object itself or *relationships* between the object and one or more other objects.

- An attribute of an object has a name, a type and a value. The value of the attribute is an object of the type of the attribute.
- A database is based on a *schema*; The database contains instances of the types defined by its schema.
- The types of a schema form a *lattice*, which defines subtype/supertype relationships. Specifically to DEMO, the top of the lattice is a type named "Any."

The relational data model can be seen as a special case of the object data model, where relation schemas are type definitions, tuples are objects, and the fields of a tuple are its attributes.

The database read and write operations of transactions programs are created in the operation extraction steps detailed in Chapter 4, "Generating Read and Write Selects." These operations are described using the SQL-like select, insert, delete and update statements shown in Table A.1. The select statements of Table A.1 are also used to describe the transaction program read and write operations.

The variables $path_i$ in the "select" list of the select statement are object path expressions that originate from the variables listed in the "from" list. The

Syntax		Comment
select from where	$path_1$, $path_2$, ... obj_1, obj_2, ... $pred$	representing a read operation to the database
insert into values	obj_1 $attr_1$, $attr_2$, ... $value_1$, $value_2$, ...	representing an insert operation to the database
delete where	obj_1 $pred$	representing a delete operation to the database
update set where	obj_1 $attr_1 = expr_1$, $attr_2 = expr_2$, ... $pred$	representing an update operation to the database

Table A.1: Syntax of Database Read and Write Operations

variables obj_i in the queries represent sets of objects upon which the database operations are performed. The variables $attr_i$ in the queries represent properties of obj_i.

The variables $pred$ in the "where" clauses of the queries represent query predicates. Predicates are restricted to those that are accepted by the DEMO OQL parser and canonicalizer [18]. Currently, these SQL operators are allowed in $pred$: "exists," comparison operators (=, >=, >, <=, <, <>), and boolean operators (AND, OR, NOT). These operators are not allowed: "between ... and," "like," "in," "all," and arithmetic operators (+, -, *, /, mod).

Only those select statements whose syntax is recognized by the parser and the canonicalizer are allowed. Presently, for instance, queries that employ set operators (union, intersect, except) are not accepted by the DEMO OQL canonicalizer. Such queries are not included in Table A.1.

The variables $value_i$ in the insert statement represent new values that the attributes $attr_i$ of object obj_1 will have. $value_i$ can be literals (i.e., constant integers, floats, characters and strings), or references to other objects. The variables $expr_i$ in the update statement allow simple mathematical functions of the form

```
<attribute> <op> {<constant> | <attribute>}
```

The variable <attribute> represents properties of the object obj_1. The variable <constant> represents literals. The variable <op> represents a mathematical operator that is any of $+$, $-$, $/$ and *mod*. Hence, the "set" list in the update statement has the form

```
<attribute> = <attribute> <op> {<constant> | <attribute>}
```

where the variable <attribute> on the left-hand side may symbolize the same object attribute as one or both of the variables <attribute> on the right-hand side.

The syntax for the insert statement also requires that the attribute list of the insert statement not be null. This is because, during the steps in generating

read and write selects, database schema information is *not* incorporated into the process, and queries are manipulated purely syntactically for efficiency and simplicity. As a result, during an insertion, all values of all attributes within an object must be explicitly specified.

The database read and write operations in Table A.1 may contain transaction program parameters (but no other program variables). Parameters may occur inside the select, insert, delete and update statements wherever literals are allowed.

Appendix B

Sample *Employee* Database

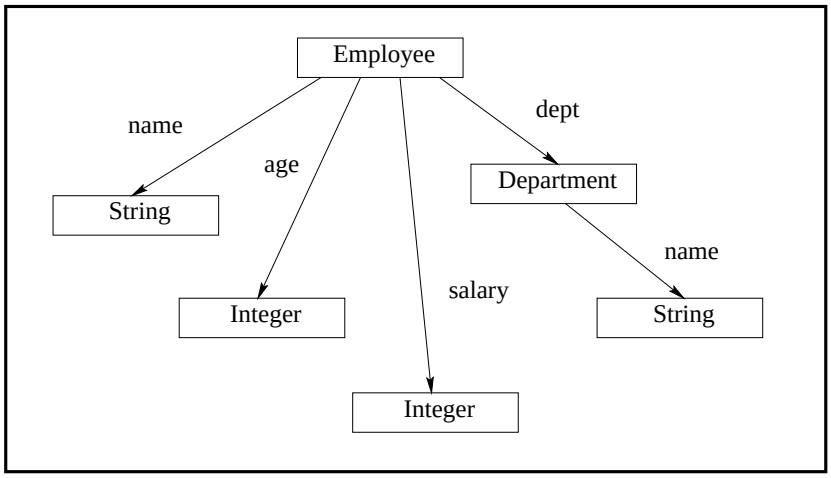


Figure B.1: E-R Diagram of the *Employee* Database

Figure B.1 contains an Entity-Relationship diagram that represents, at a conceptual level, the sample *Employee* database.

Database schemas are passed into the DEMO optimizer using UDR (Universal Data Representation) declarations. UDR is a fine-grained language that was designed to capture both conceptual and internal views of main-memory databases [3]. The following UDR statements define the *Employee* database.

```
# ----------------------
# the Employee class
# ----------------------

(:class Employee
  (:prop name
    (:type String:40)
  )
  (:prop age
    (:type Integer)
  )
  (:prop salary
    (:type Integer)
  )
  (:prop dept
    (:type Department)
  )
  (:prop eIndex
    (:type Eindex)
  )
  (:eq [] [eIndex iEmployee])
)
(:stored name age salary dept)
(:class Eindex
  (:prop iEmployee
    (:type Employee)
  )
  (:pfd {[iEmployee]} {[]})
)
(:index Eindex)
```

```
# ----------------------
# the Department class
# ----------------------

(:class Department
  (:prop name
    (:type String:10)
  )
  (:prop dIndex
    (:type Dindex)
  )
  (:eq [] [dIndex iDepartment])
)
(:stored name)
(:class Dindex
  (:prop iDepartment
    (:type Department)
  )
  (:pfd {[iDepartment]} {[]})
)
(:index Dindex)
```

Appendix C

The DEMO System

The Design Environment for Memory-resident Object-oriented databases (DEMO) project explores how to adapt object-oriented database technology to manage control data for embedded control applications.

The semi-automatic process in this thesis employs two major components of DEMO: 1) the query canonicalizer; and 2) the query optimizer. The canonicalizer transforms queries into a canonical form. The optimizer maps a canonicalized query into an existential query graph (EQG). The semi-automatic process described in this thesis then scans the EQG to construct a conflict predicate.

C.1 Query Canonical Form

The DEMO *Canonical Form* is based on the syntax of OQL queries, but stretches the OQL semantics by allowing the quantification of an infinite class (e.g. *Any*, *Integer*, *String*, etc.) in a "from" list [3]. Informally, the form has the following format

```
select   ...
from     ...
where    exists (
             select   *
             from     ...
             where    boolExpr_1 and ...and boolExpr_n)
```

where $boolExpr_i$ can be one of

- $v_m = v_n$
- $v_m = v_n.a$
- not exists (
  select *
  from ...
  where $boolExpr_{n+1}$ and ...and $boolExpr_{n+k}$)

In the "from" lists of the canonical form, an object type could be "Any," which means **unknown** in DEMO, or "Any_P," which means a parameter of unknown type. Later, by using its knowledge of the database schema, the DEMO optimizer will attempt to deduce the actual type of the objects to form a semantically correct OQL query.

A list of transformation rules have been developed to map an OQL query to a canonical form. (For more information on the rules, see [3].) The transformation preserves the semantics of the input query.

Here is a simple example that illustrates the conversion of an OQL query to the DEMO canonical form. The query below

```
select E
  from Employee as E
```

is canonicalized into

```
select v1 AS z1
  from Employee as E, Any as v1
 where exists
         select *
            from Any as v2, Any as v3
           where (v1 = v2)
             and (v2 = v3)
             and (v3 = E)
```

The canonical form offers a uniform structure into which OQL queries can be converted. The main advantage of having an OQL query in the canonical form is that a canonicalized query can be easily mapped into an existential query graph.

C.2 Existential Query Graph (EQG)

DEMO uses existential graphs as a graphical syntax to capture schema declarations, data constraints, and queries ([3]). The DEMO query optimizer applies previously defined inference rules to generate an EQG as a semantical representation of an input query in canonical form.

The nodes in an EQG represent query variables, and the labels of the nodes indicate variable types. A *labelled arc* with an arrow pointing from node V to node V' indicates that V has the named property of type V'. Every *sheet of control*, illustrated by a rectangle, introduces a new nested scope for additional nodes and graphical elements, and constitutes a *negation*. An undirected arc connecting a pair of nodes, called a *line of identity*, asserts that the two nodes denote the same individual. Each query has a corresponding type, and a query variable (node in the graph) representing the result of the query. The parameters of a query are attributes of the query node in the graph.

For example, when defined by the *Employee* database schema, the query

```
SampleQuery:
    select Employee.salary as sal
      from Employee
     where Employee.salary > 50
       and Employee.salary = :p1
```

generates the EQG in Figure C.1.

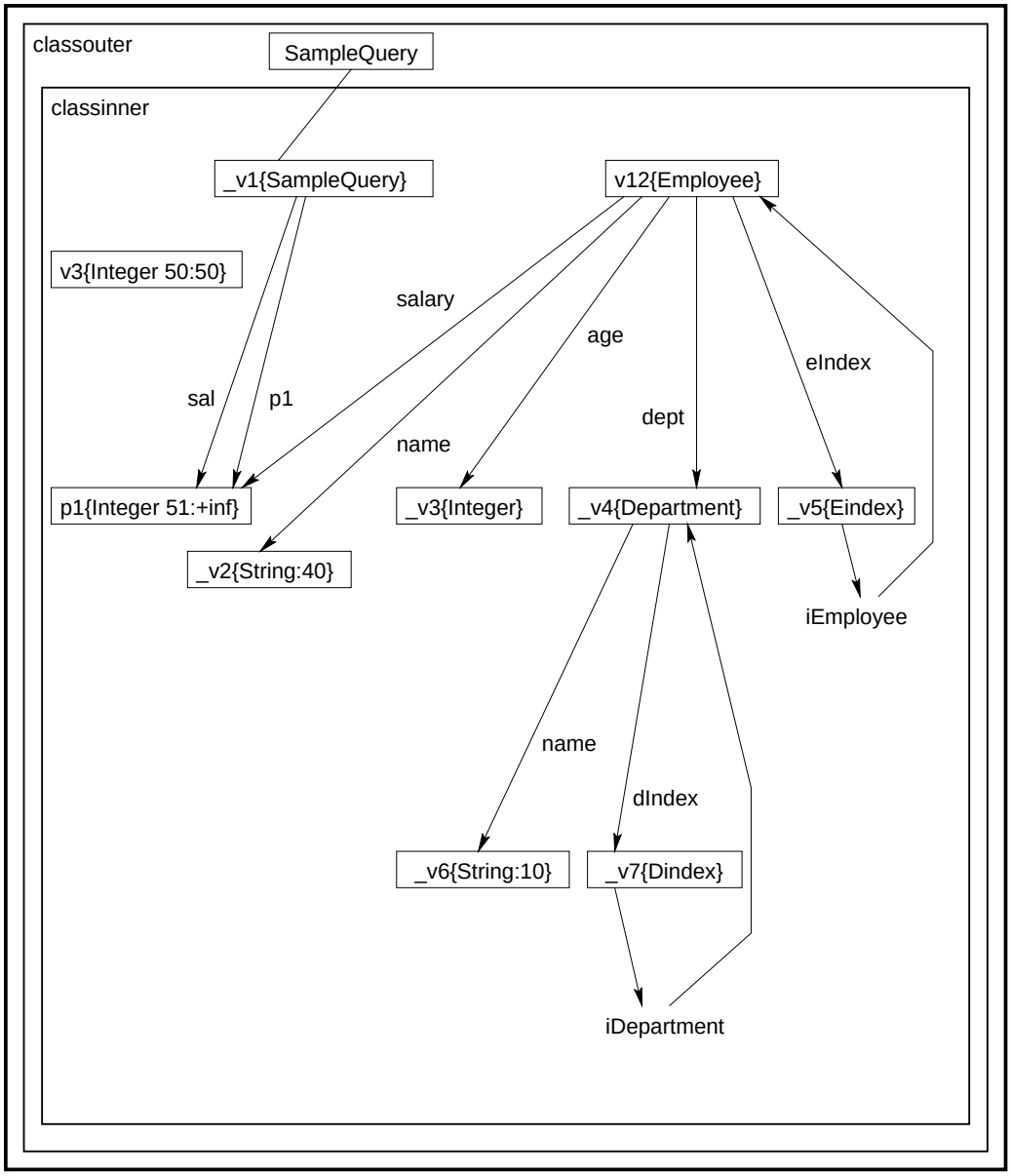


Figure C.1: A Sample Existential Graph

Figure C.1 indicates that every "SampleQuery" object has an attribute "sal" (the query result) and a parameter "p1." When several labelled arcs have arrows pointing from nodes V_1, V_2, ..., V_n to the same node V', then the nodes V_i all share the named property of type V'. In this instance, it is depicted that for every query object (i.e., for every query result), there exists an "employee" object with an attribute "salary" whose value matches not only the query result "sal" but also the query parameter "p1."

The EQG generated by DEMO for a query Q is a graphical representation of a conjunctive formula of the form:

$$\forall Q | Q \text{isaqueryresult} : (\exists (v_1, \ldots v_n) | (v_1 = Q) \wedge (cond_2) \wedge \ldots \wedge (cond_m))$$

where the conjuncts $cond_i$ are conditions involving variables v_i and Q.

For details on existential graphs, reference [3], [19] and [20].

Bibliography

[1] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*, chapter 1, What Is a Transaction Processing System?, pages 5–6. Morgan Kaufmann Publishers, Inc., 1993.

[2] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, 1976.

[3] Petrus Kai Chung Chan. Optimizing oql on legacy main-memory data structures with existential graphs. Master's thesis, University of Waterloo, 1997.

[4] Patrick O'Neil. *Databases - Principles, Programming, Performance*, chapter 9, Update Transactions, pages 672–673. Morgan Kaufmann Publishers, Inc., 1994.

[5] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*, chapter 9, Update Transactions, pages 439–442. The McGraw-Hill Companies, Inc., 3 edition, 1997.

[6] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*, chapter 7, Isolation Concepts, pages 404–405. Morgan Kaufmann Publishers, Inc., 1993.

[7] Peter Shawn Yang. PAPRICCA: A Pre-Analyzing Predicate-Based Concurrency Control Algorithm. Master's thesis, University of Waterloo, 1998.

[8] Raj K. Rathee. Using Advance Knowledge of Transactions to Provide Efficient Recovery Management. Master's thesis, University of Waterloo, 1997.

[9] M. Berndtsson and J. Hansson. Issues in Active Real-Time Databases. In *Proceedings of the 1st International Workshop on Active and Real-Time Database Systems*, Workshops in Computing, pages 142–157. Springer, 1995.

[10] Dale Skeen. Real-Time Queries in the Enterprise. *BYTE*, pages 47–48, Feb 1998.

[11] S. Chakravarthy. Early Active Databases: A Capsule Summary. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):1008–1011, 1995.

[12] S. Chawathe, S. Abiteboul, and J. Widom. Representing and querying changes in semistructured data. Technical report, Stanford University, 1997.

[13] Jose A. Blakeley and Frank Wm. Tompa Per-Ake Larson. Efficiently Updating Materialized Views. *SIGMOD Conference*, pages 61–71, 1986.

[14] Sha Guo, Wei Sun, and Mark A. Weiss. On Satisfiability, Equivalence, and Implication Problems Involving Conjunctive Queries in Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, 8(4):604–616, 1996.

[15] Daniel J. Rosenkrantz and Harry B. Hunt. Processing Conjunctive Predicates and Queries. In *Proceedings of the 6th International Conference on Very Large Databases*, pages 64–72, 1980.

[16] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[17] R.G.G. Cattell, editor. *The Object Database Standard: ODMG - 93*, chapter 4, Object Query Language, pages 53–85. Morgan Kaufmann Publishers, Inc., 1996.

[18] DEMO Project Team, University of Waterloo. *DEMO OQL Parser and Canonicalizer Manual*, 1999.

[19] D.D. Roberts. *The Existential Graphs of Charles S. Peirce*. Mouton, The Hague, 1973.

[20] M. Gyssnes, J. Paredaens, and D. Van Gucht. A graph-oriented object database model for databases and end-user interfaces. *Proceedings SIGMOD'90*, pages 24–33, 1990.