An Optimizing Interpreter for Concurrent Transaction Logic

by

Amalia F. Sleghel

An Optimizing Interpreter for Concurrent Transaction Logic

for the degree of Master of Science, 2000

by

Amalia F. Sleghel

Department of Computer Science

University of Toronto

Abstract

Concurrent Transaction Logic is a new deductive database language that integrates queries, updates, and transaction composition in a simple logical framework. The language supports all the properties of classical transactions, and also the properties found in many new transaction models, such as nested transactions, concurrency within individual transaction, and fine-grained control over abort and rollback.

This thesis describes a Prolog implementation of Concurrent Transaction Logic. The prototype supports functionality in several areas including database queries, backtrackable updates, transaction definition, execution schedules, serial and concurrent execution of transactions. Two optimizations have been developed with this implementation: backtracking elimination for non-interacting processes, and query compilation. We illustrate the language and the implementation through a series of examples. The implementation is freely available on the Web.

Contents

List of Figures

Chapter 1

Introduction

The classical model of database transactions has been widely successful in applications like banking, airline reservations, and inventory control. In these traditional applications, transactions perform only simple operations on small amounts of simply-structured data. Unfortunately, the classical model is inappropriate for new database applications involving distributed systems, complex data structures, and cooperation between multiple users or multiple concurrent processes. Examples include CAD, office automation, collaborative work, manufacturing control, and workflow management. To address these new issues, Bonner and Kifer [3] developed a new deductive database language, *Concurrent Transaction Logic* (or CTR), which is an extension of *Transaction Logic* (or TR) [1].

1.1 Concurrent Transaction Logic

Even though the updates are an important feature of any database programming language, they are not accounted for by the classical Horn semantics of deductive databases. TR is an extension of the first-order classical logic that provides a logical account of state changes in databases and logic programs. It introduces an operator, *serial conjunction*, that allows a user to specify a linear order on a set of transactions. The new language,

CTR, extends TR with connectives for modeling the concurrent execution of complex processes. Concurrency is a new feature that increases the flexibility, performance and power of the language. In CTR, concurrent processes execute in an interleaved fashion and can communicate via the database and synchronize themselves.

Unlike other deductive database languages, CTR is the only deductive database language that integrates concurrency, communication, and database updates in a completely logical framework, including a natural model theory and a simple proof theory. The language supports all the properties of classical transactions, and also properties found in many new transaction models, such as nested transactions, concurrency within individual transaction, cooperation between concurrent activities, a separation between atomicity and isolation, and fine-grained control over abort and rollback [4].

This thesis presents an implementation of CTR in Prolog. It is assumed that the reader is already familiar with Prolog. The prototype runs on the XSB Prolog interpreter [12], currently the most efficient deductive database system; however it can also run on other Prolog interpreters, such as Quintus Prolog and Binary Prolog, with some simple adjustments. Our implementation consists of Prolog programs that define CTR connectives, backtrackable updates, and implementation-specific commands and directives. Along with the basic implementation, the thesis describes the optimizations available with this prototype: an optimization for the non-interacting processes, and a form of query compilation.

Our prototype is built on top of a simple, non-optimized interpreter, including code for backtrackable updates [2]. Source code, documentation and a description of this simple interpreter is available through the CTR web page (*www.cs.toronto.edu/~bonner/ctr.html*). The implementation includes a translator that converts CTR syntax into internal syntax. Backtracking elimination is the first optimization that can be applied to CTR programs. It implements an algorithm based on a dependency graph, which determines interactions in a CTR program. The optimization increases program execution efficiency through the elimination of needless backtracking as described in Chapter 6. The second optimization implements a query compilation[1] algorithm that replaces certain CTR formulas with equivalent Prolog terms. Since the Prolog terms are executed more efficiently, the optimized programs have better performance.

A CTR program consists of a set of transaction rules and a set of database rules and tuples, stored in a transaction base and in a database, respectively. During compilation, the prototype translates the transaction base and database into Prolog terms, and loads them into the Prolog database, so they can be accessed during program execution.

The thesis is organized as follows:

- Chapter 2 provides an overview of *Concurrent Transaction Logic*, CTR formal syntax, elementary operations, and CTR programs.

- Chapter 3 describes the prototype syntax and its relation to the formal syntax. It gives a detailed description of program execution and scheduling in the prototype, including the handling of failures, deadlocks and rollback.

- Chapter 4 presents commands for compiling, optimizing, and executing CTR programs using the prototype, along with the files created.

- Chapter 5 describes the implementation. It gives a description of the prototype modules, internal syntax, main features of the implementation, and the interpreter role.

- Chapter 6 provides an overview of the backtracking elimination, which is the first optimization available with the prototype. It presents the dependency graph, the algorithm used to determine non-interaction, and its implementation.

- Chapter 7 presents the second optimization, the query compilation, the algorithm implemented, and its applicability.

[1] More details on query compilation are provided in Chapter 7.

The thesis includes a Tutorial in the Appendix, which contains examples of CTR programs and commands presented in XSB sessions. We also developed a Concurrent Transaction Logic web page[2] . It presents the language, examples of CTR programs and XSB terminal sessions, and the tutorial from the Appendix. The web page allows the user to view and download the source code, and also offers the option to download the entire implementation as a compressed tar file. There are links to related papers and documentation with more information on deductive database languages.

1.2 Related Work

A comprehensive comparison of CTR and other formalisms can be found in [4, 1, 3, 7]. In this section, we reproduce and paraphrase the most salient parts.

Formalisms for Concurrency: Numerous formalisms have been developed to allow concurrently executing processes to interact and communicate, and some (such as process algebras) allow new processes to be created recursively at runtime. They include, various process algebras [35, 9, 31, 26, 27, 36], many kinds of Petri net [41, 29, 30, 23, 33], as well as temporal logic [15, 16], state charts [10], concurrent transition systems [11], and concurrent logic programming [13, 14]. However, unlike CTR, most of these formalisms are not database or logic-programming languages.

"CTR has many of the features of process algebras. These include concurrent access to shared resources, communication between sequential processes, and the ability to isolate (or hide) the inner workings of a group of processes from the outside world. Like all process algebras, CTR is compositional, so processes can be defined recursively in terms of subprocesses. It is therefore possible to specify multi-level processes [39], even when the number of levels is determined at runtime. However, unlike process algebras, CTR

[2]The web page URL is *www.cs.toronto.edu/~bonner/ctr.html*

also provides high-level support for database functions. These include declarative queries, bulk updates, views, and serializability [5, 3]. CTR also has many features of advanced transaction models, including sub-transaction hierarchies, relaxed $ACID$ requirements, and fine-grained control over abort and rollback [6]. This integration of process modeling and database functionality is reflected in the formal semantics of CTR, which is based on both database states and events, while the semantics of process algebras is based entirely on events. " [4].

There has been extensive research on concurrency in databases, and logic programming. Some of the earliest attempts at adding concurrency to logic programs were $PARLOG$, *Concurrent Prolog*, GHC, and related languages [13]. In these languages, once the flow of control commits to a certain action, this action does not backtrack. This is quite unlike the programming style promoted by CTR, in which updates can be undone.

There are also some similarities between CTR and existing concurrent logic programming (CLP) languages [13, 14]. However, "CTR is the only deductive database language that integrates concurrency, communication, and database updates in a completely logical framework. Such integration presents interesting new possibilities for the programmer. For instance, concurrent processes can now communicate via the database; that is, one process can read what another process has written. This form of communication leads to a programming style that is very different from that of CLP languages."

"In CLP languages, concurrent processes communicate via shared variables and unification. This kind of communication is orthogonal to communication via the database. Both are possible in CTR. Implementations of CTR may therefore adopt many of the techniques of shared-variable communication developed for CLP." [3]. The main difference is that most process modeling in CLP is purely operational, and is not accounted for by the classical Horn semantics of logic programs. This is especially true for CLP programs with updates, for which the classical semantics is inadequate. In contrast,

CTR has a purely logical semantics with transactional capabilities, which are completely missing from other logic programming languages.

Based on the situation calculus [24], Reiter et al. have developed a new language, called $CONGOLOG$ [17], which includes primitives for combining actions into more complex ones. This development addresses some of the problems that motivated CTR. However, there are numerous differences between CTR and $CONGOLOG$. A key difference is that $CONGOLOG$ is fundamentally incompatible with the logic-programming paradigm. In particular, Horn logic programs cannot be combined with $CONGOLOG$ programs.

Logic-Based Formalisms for Updates: Since these formalisms are without concurrency, they are more related to TR [1] than to CTR [3].

Prolog supplies primitive operators for doing updates via the database, such as *assert* and *retract*. "Unfortunately, these update operators have no logical semantics, and each time a programmer uses them, he moves further away from declarative programming. Moreover, *Prolog* programs using these operators are often awkward and the most difficult to understand, debug and maintain. These problems are all exacerbated by concurrency. " [3].

In the area of declarative languages for database updates, "Winslett [32] did fundational work by supplying the semantic definition for the result of updating a logical theory. Grahne, Katsuno, and Mendelzon [18, 19, 20] axiomatized various theories of state transition and studied tractable cases of *elementary state transitions*. The TR approach to state transitions is inspired by these results" [1].

"Manchanda and Warren [37] introduce *Dynamic Prolog*, which is a logic system where update transactions do not leave a residue in the database when a transaction fails. Like TR, this logic can update views, and transactions can be non-deterministic. However, the proof theory is impractical for carrying out updates, since one must know the final database state before inference begins." [1].

Chapter 2

Overview of Concurrent Transaction Logic

Concurrent Transaction Logic is a deductive database language for programming database transactions and applications. The language, an extension of Transaction Logic[1], provides a variety of features, including concurrency and communication, queries and updates, and support for traditional and advanced transactions. This chapter outlines the language using the terminology of deductive databases. Details are available in[2], [3].

2.1 Syntax

Concurrent Transaction Logic is based on classical logic, increasing its capabilities by introducing three new logical connectives: $\otimes$, called *sequential composition*, $|$, called *concurrent composition*, and a modality of isolation, $\odot$, for specifying transactions. A transaction executes atomically and in isolation, *i.e.*, it does not communicate or interact with other programs. These operators are used to specify queries and to combine simple transactions into complex ones.

The formal *CTR* syntax is defined recursively as follows:

- An atomic formula in first-order predicate logic is a *goal.*
- If $a_1, ..., a_n$ are goals, then so are $a_1 \otimes ... \otimes a_n$, $a_1 \mid ... \mid a_n$, and $\odot a_1$.
- If a is a goal and t is an atom, then $t \leftarrow a$, is a rule.
- A finite set of rules is a *rulebase.*
- A rulebase together with a goal is a *program.*

If a and b are goals, then the CTR formulas can be interpreted informally as follows:

- $a \otimes b$ - "*First execute a, then execute b, and commit iff both a and b commit.*"
- $a \mid b$ - "*Execute a and b concurrently, and commit iff both a and b commit.*"
- $\odot a$ - "*Execute a as a transaction and commit iff a commits.*"
- $t \leftarrow a$ - "*An execution of a is also an execution of t, where t commits if a commits.*"
- $t \leftarrow a \vee b$ - "*An execution of a or b is also an execution of t, where t commits if either a or b commits.*"

Disjunction is a convenience that adds no new power to the language[1], since $t \leftarrow a \vee b$ is equivalent to two rules:

$$t \leftarrow a$$
$$t \leftarrow b.$$

There is a close relationship between the modality of isolation and the traditional notion of serializability [38]. In particular, the goal $\odot a_1 \mid \odot a_2 \mid ... \mid \odot a_n$ executes the programs $a_1, a_2, ..., a_n$ serializably.

[1]See Appendix for an example of disjunction.

2.2 Elementary Operations

CTR combines elementary database operations into complex transaction programs. Formally, the elementary database operations are specified by a pair of oracles:

- the *data oracle*, O^d, which is a mapping from states to sets of first-order formulas, specifying a *static* semantics. Intuitively, if D is a database state, then $O^d(D)$ is the set of formulas that are true of the state.

- the *transition oracle*, O^t, which is a mapping from pairs of states to sets of ground atomic formulas, specifying a *dynamic* semantics. Intuitively, if D_1 and D_2 are database states, then $b \in O^t(D_1, D_2)$ means that b is an elementary update that changes state D_1 into state D_2.

As an example, one can easily define oracles for relational databases. First, for each p:

$$p\,(x) \in O^d(D) \;\; \texttt{iff} \;\; p\,(x) \in D$$
$$empty.p \in O^d(D) \;\; \texttt{iff} \;\; p\,(x) \notin D \;\; \texttt{for all x.}$$

Second, the relational transition oracle defines two new predicates, *ins.p* and *del.p*, for each base predicate symbol p, representing the insertion and deletion of single atoms. In particular,

$$ins.p\,(\overline{x}) \in O^t(D_1, D_2) \;\; \texttt{iff} \;\; p\,(x) \notin D_1 \;\; \texttt{and} \;\; D_2 = D_1 \cup \{p\,(\overline{x})\}\texttt{, and}$$
$$del.p\,(\overline{x}) \in O^t(D_1, D_2) \;\; \texttt{iff} \;\; p\,(x) \in D_1 \;\; \texttt{and} \;\; D_2 = D_1 - \{p\,(\overline{x})\}\texttt{.}$$

Intuitively, these operations have the following interpretation:

- $p(x)$ - *"Commit iff p(x) is in the database."*

- *empty.p* - *"Commit iff the database contains no atoms of the form p(x)."*

- *ins.p*(x) - *"Insert atom p(x) into the database, and commit."*

- *del.p*(x) - *"Delete atom p(x) from the database, and commit."*

2.3 Examples

This section gives examples of Concurrent Transaction Logic programs, taken from [3], [4]. We start with simple examples of goals to illustrate the syntax. The goal:

$$del.r(a) \otimes ins.s(a)$$

is a program that deletes $r(a)$ from the database, and then inserts $s(a)$. Since both components of the program always commit, the program will also always commit. The rule:

$$q(X) \leftarrow r(X) \otimes del.r(X) \otimes ins.s(X)$$

defines a subroutine with name q and parameter X. Given the parameter value a, $q(a)$ commits if $r(a)$ is in the database before the updates are invoked. The subroutine deletes $r(a)$ and inserts $s(a)$ into the database. If $r(a)$ is not in the database, then the execution of q fails and the database contents does not change.

The next example illustrates the use of subroutines and shows how CTR can be used to combine elementary operations into complex transactions.

Example 2.1: Financial Transactions

Suppose the balance of a bank account is represented by relation $balance(Acct, Amt)$. The rulebase below defines four transactions:

- *change_balance(Acct,*Bal_1*,*Bal_2*)*, to change the balance of an account from Bal_1 to Bal_2
- *withdraw(Amt,Acct)*, to withdraw an amount from an account
- *deposit(Amt,Acct)*, to deposit an amount into an account
- *transfer(Amt,*$Acct_1$*,*$Acct_2$*)*, to transfer an amount from one account to another.

The *transaction base* for these transactions contains the following rules:

$$transfer(Amt, Acct_1, Acct_2) \leftarrow \odot\,[withdraw(Amt, Acct_1) \otimes deposit(Amt, Acct_2)]$$
$$withdraw(Amt, Acct) \leftarrow$$
$$balance(Acct, Bal) \otimes Bal \geq Amt \otimes change\_balance(Acct, Bal, Bal - Amt)$$
$$deposit(Amt, Acct) \leftarrow balance(Acct, Bal) \otimes change\_balance(Acct, Bal, Bal + Amt)$$
$$change\_balance(Acct, Bal_1, Bal_2) \leftarrow del.balance(Acct, Bal_1) \otimes ins.balance(Acct, Bal_2)$$

Since the rules use serial conjunction operators only, they are evaluated from left to right. In the first rule, the transfer of an amount, Amt, from $Acct_1$ to $Acct_2$ consists of two steps: first withdraw Amt from $Acct_1$, and, if it succeeds, deposit the same amount, into $Acct_2$. According to the second rule, to withdraw Amt from an account, $Acct$, the balance of the account is retrieved by the query $balance\,(Acct, Bal)$ and, then the test $Bal \geq Amt$ compares the balance with the amount to ensure that the account will not be overdrawn. If both operations succeed, the account balance is updated from Bal to $Bal - Amt$. In a similar fashion, to deposit Amt in $Acct$, the balance of the account is first retrieved and, if the query executes successfully, then the balance is updated from Bal to $Bal + Amt$. The last rule uses two elementary updates, $del.balance$ and $ins.balance$, to change the balance of an account from Bal_1 to Bal_2.

Since the *transfer* is executed as a transaction (due to the $\odot$ operator), more than one transfer can be executed concurrently without any ill side effects. For example, consider the following concurrent transfers:

$$?- transfer\,(Fee, Client, Broker) \mid transfer\,(Cost, Client, Seller)$$

If one of the transfers succeeds and the other fails, then CTR behaves correctly, rolling back the entire transaction. In the same situation, Prolog would undo only the failed update, thus leaving the database in an inconsistent state. $\square$

In the above example, the two transfers are executed in isolation. More generally, though, CTR processes can communicate and synchronize by exchanging messages along communication channels, as shown in the next example.

Example 2.2: Synchronization

Suppose that two processes, $processA$ and $processB$, which invoke three tasks each, are defined by the following transaction base:

$processA \leftarrow taskA_1 \otimes ins.start\,(ch_1, B_2) \otimes taskA_2 \otimes start\,(ch_2, A_3) \otimes taskA_3$

$processB \leftarrow taskB_1 \otimes start\,(ch_1, B_2) \otimes taskB_2 \otimes ins.start\,(ch_2, A_3) \otimes taskB_3$

The next transaction executes them concurrently :

$$? - processA \mid processB$$

During execution, the processes communicate and synchronize themselves by effectively transmitting messages through channels ch_1 and ch_2. Tasks $taskA_1$ and $taskB_1$ execute concurrently, however $taskB_2$ cannot start unless a message from $processA$ is received through channel ch_1 to signal that task $taskA_1$ has been completed. The transaction $ins.start\,(ch_1, B_2)$ places the message B_2 in channel ch_1. The same technique is used to communicate that the task $taskB_2$ has been successfully completed and the task $taskA_3$ is allowed to start. In CTR, communication channels are implemented as a binary database relation, $start$. Intuitively, $start\,(ch, Msg)$, means that channel ch contains message Msg. Sending a message along a channel amounts to inserting a tuple in the $start$ relation, checking for a message amounts to retrieving a tuple, and receiving a message amounts to deleting a tuple. □

The declarative semantics and the smooth integration of process specification with database access make production workflow a natural application of CTR, as illustrated in the next example. In general, a production workflow involves different work items and activities organized into factory-like production lines. More on the application of CTR to workflow management can be found in [4].

The workflow example that follows involves three interacting workflows: two of them process items that will be combined and processed by the third workflow. Assuming that the workflows are activities taking place in a genome laboratory, one workflow might process long DNA samples while the second might process short DNA samples. Next, the overlapping pairs of short and long samples are processed by a third workflow to determine the exact nature of their overlap.

Example 2.3: Asynchronous Workflow Network

The transaction base for this example consists of two sets of rules. The first set simulates the execution of two workflows, $workflow_1$ and $workflow_2$. Each work item processed by a workflow has a record associated with it that is stored, initially, in the "in basket", represented by database relation $item_i$. After the item has been processed by one of the two workflows, it is stored in the "out basket", represented by database relation $basket_i$.

$$
\begin{aligned}
&produce_i \leftarrow simulate_i \otimes ins.finished_i \\
&simulate_i \leftarrow getItem_i\,(W) \otimes [simulate_i \mid (workflow_i\,(W) \otimes putItem_i\,(W))] \\
&simulate_i \leftarrow item_i.empty \\
&getItem_i\,(W) \leftarrow \odot\,[item_i\,(W) \otimes del.item_i\,(W)] \\
&putItem_i\,(W) \leftarrow ins.basket_i\,(W)
\end{aligned}
$$

The first rule defines the transaction $produce_i$. It invokes one of the workflows and inserts the atom $finished_i$ when it is successfully completed. The second rule defines the actual simulation by deleting an item from the "in basket", spawning a new workflow instance to process it, and inserting the processed item in the "out basket". The predicate $simulate_i$ is recursive and allows concurrent executions of multiple instances of the workflow. The recursion is completed when there are no more items left in the "in basket", as specified by the third rule. The remaining rules define the transaction $getItem_i$ and $putItem_i$ that update the database relations.

The third workflow, $workflow_3$, selects items from two input database relations, $basket_1$ and $basket_2$, processes them together, and terminates when one of the input baskets is empty. The second set of rules specifies the transactions used to simulate $workflow_3$:

$$consume \leftarrow simulate_3 \otimes ins.finished_3$$
$$simulate_3 \leftarrow selectItems\,(I, J) \otimes [simulate_3 \mid workflow_3\,(I, J)]$$
$$simulate_3 \leftarrow finished_1 \otimes basket_1.empty$$
$$simulate_3 \leftarrow finished_2 \otimes basket_2.empty$$
$$selectItems\,(I, J) \leftarrow$$
$$\odot\,[basket_1\,(I) \otimes basket_2\,(J) \otimes suitable\,(I, J) \otimes del.basket_1\,(I) \otimes del.basket_2\,(J)]$$

The $workflow_3$ simulation is called by the transaction $consume$ in the first rule, which also updates the database inserting the atom $finished_3$. The simulation is defined recursively in the second rule: first, items are selected and deleted from the "in baskets", then a new instance of $workflow_3$ is spawned to process the items. As specified in the third and fourth rule, the recursion is stopped when there are no more items in one of the "in baskets" and the workflow that used it as "out basket" has completed as well. The last rule selects pairs of items to be processed by $workflow_3$, and updates the "in baskets" by deleting items from the database.

Finally, the transaction $work$ invokes $workflow_1$, $workflow_2$, and $workflow_3$, and executes them concurrently:

$$work \leftarrow produce_1 \mid produce_2 \mid consume$$

Initially, $workflow_1$ and $workflow_2$ execute concurrently creating items for processing by $workflow_3$. As soon as the first items meeting the selection criteria for $workflow_3$ processing are created, it starts executing. When there are no more items that can be processed by $workflow_3$, the $work$ transaction completes. □

Chapter 3

Syntax and Behavior from User's Viewpoint

The *CTR* prototype described in this thesis uses a syntax that is slightly different from the formal syntax of *CTR* (as presented in Chapter 2). This chapter outlines the prototype syntax and its relation to the formal syntax. It also gives a detailed description of program execution and scheduling in the prototype, including the handling of failures, deadlocks and rollback. The *CTR* implementation offers the capability to monitor program execution with a *monitor* command that is presented later in the chapter, along with the special processing of the Prolog terms.

3.1 User Syntax

A *CTR* program consists of a set of rules that have a similar syntax to Prolog rules, with the exception of symbols that define operators. The implementation stores the rules in two different storage areas: the *transaction base* and the *database*. This section specifies the syntax for each storage area, assuming that the reader is already familiar with Prolog syntax.

3.1.1 Transaction Base

A transaction base is a set of transaction rules, where a *transaction rule* is defined as one of the following:

- an atomic formula
- $\alpha : - \beta$, where α is an atomic formula and β is a *transaction formula.* The symbol $: -$ separates the head and the body of the rule, replacing the symbol $\leftarrow$ from the formal theory.

In this prototype, a *transaction formula* can have one of the following forms:

- an atomic formula
- if θ and ψ are transaction formulas, so is $\theta \star \psi$. The infix operator, $\star$, is called *sequential composition,* and corresponds to the symbol $\otimes$ from the *CTR* formal theory.
- if θ and ψ are transaction formulas, so is $\theta \# \psi$. The symbol # is the infix operator *concurrent composition,* and corresponds to the symbol $|$ from the *CTR* formal theory.
- if θ is a transaction formulas, so is $o\theta$. The prefix operator, o, represents the *isolate* operator, and corresponds to the symbol $\odot$ defined in the *CTR* formal theory.
- if θ is a transaction formulas, so is $o1\theta$. The prefix operator $o1$ is a special case of the *isolate* operator called *isolate one,* implemented to optimize program execution[1].
- if θ and ψ are transaction formulas, so is $\theta \setminus\!/ \ \psi$. The symbol $\setminus\!/$ is the infix operator *disjunction,* and corresponds to the symbol, $\vee$, from the *CTR* formal theory.

[1] Described in detail in Chapter 6.

As an example, the following two expressions represent the same rule written in the two different syntaxes:

- $q(X) \leftarrow r(X) \otimes p(X) \mid s(X)$ - syntax of the formal theory
- $q(X) :- r(X) \star p(X) \# s(X)$ - syntax of the prototype.

3.1.2 Database

In the CTR prototype, a database is a Prolog rulebase. In particular, it can store both *database rules* and *database tuples*. A *database rule* is defined as one of the following:

- an atomic formula
- $\alpha :- \beta.$, where α is an atomic formula and β is a *query formula.*

A *query formula* is defined as:

- an atomic formula
- α , β , where α and β are *query formula*s.

For instance, the following is a database rule:

$$t(X,Y) :- \ p(X), r(X,Y).$$

Database tuples may be updatable or non-updatable (the default). A user can declare that the tuple in a relation R/n (*i.e.*, relation R with arity n) is updatable, by including the following statement in the database:

$$\textit{updatable} \;\; R/n.$$

In this case, the tuples in R can be updated by CTR programs, and the updates will be undone during backtracking. For instance, the relation *balance* with two arguments, *account* and *amount*, can be declared as updatable using the statement:

$updatable\ \ balance/2.$

Tuples themselves are represented as atomic formulas, in the deductive database tradition. For example, the following two atomic formulas represent two tuples of the *balance* relation:

$balance(account1, 100).$

$balance(account2, 20).$

3.1.3 Elementary Updates

The CTR prototype provides the elementary operations presented in Section 2.2 using the following user syntax:

- $ins\,(p\,(x))$ - insert atom $p\,(x)$ into the database
- $del\,(p\,(x))$ - delete atom $p\,(x)$ from the database
- $empty\,(p\,(x))$ - the database contains no atoms of the form $p\,(x)$

The following two expressions represent the same rule written in the two syntaxes:

- $p\,(X) \leftarrow empty.r\,(X) \otimes ins.q\,(X)\,.$ - syntax of the formal theory
- $p\,(X) :- empty\,(r\,(X)) \star ins\,(q\,(X))\,.$ - syntax of the prototype.

The prototype provides the elementary updates as *strong updates*, *i.e.*, $del\,(b)$ fails if b is not initially in the database, and $ins\,(b)$ fails if b is initially in the database. This is in contrast to the *weak updates* usually presented in the formal theory, where $del.b$ and $ins.b$ always succeed, regardless of whether b is initially in the database or not [4].

3.2 Execution and Scheduling

CTR programs can execute sequentially and concurrently. This section describes the simplest case of both kinds of execution, *i.e.*, in the absence of communication, deadlock, and backtracking. The examples of this section assume that all elementary operations immediately succeed. Execution with failures will be described in the next section. Since database rules do not use CTR operators, their execution is determined by Prolog, and is not included in this section.

3.2.1 Sequential Execution

Sequential execution is the simplest type of execution, and occurs when the set of goals in a rule are separated by *sequential composition* operators. In this case, the goals are called *sequential goals* and are executed in sequential order from left to right.

Suppose that the following transaction rule is submitted for execution:

$$q :- \ p_1 \star p_2 \star \ldots \star p_n.$$

where p_i are all elementary operations. First, p_1 is started, executed and completed successfully, then p_2 is started, executed and completed. This repeats until the last goal is completed and, assuming that none of the goals fails, the entire transaction succeeds. Of course, each of the p_i may themselves be defined by rules. For example, when p_2 is defined as:

$$p_2 :- \ a \star b.$$

the execution of q will return to the next goal in the rule body, p_3, after a and b have been executed (in that order).

3.2.2 Concurrent Execution

Program execution is *concurrent* when the goals are separated by *concurrent composition* operators, in which case they are called *concurrent goals.* The prototype is a serial program that simulates concurrency by interleaving the execution of concurrent processes, in a *round-robin* fashion. This is easily illustrated with a simple example. Consider the following goal, consisting of two processes executed concurrently:

$$a_1 \star a_2 \star a_3 \star \ldots \star a_n \mid b_1 \star b_2 \star b_3 \star \ldots \star b_n$$

where all a_i and b_i are elementary operations. It should be noted that the operator $\star$ binds more tightly than $|$. Thus, this formula is equivalent to:

$$(a_1 \star a_2 \star a_3 \star \ldots \star a_n) \mid (b_1 \star b_2 \star b_3 \star \ldots \star b_n)$$

The prototype executes this goal as follows: the first operation, a_1, of the first concurrent goal is executed, followed by the first operation of the second concurrent goal, b_1. After the first operations have been completed, the second operations of each concurrent goal, a_2 and b_2, are submitted for execution in that order, and this is repeated until both concurrent goals are fully executed. This results in the following execution sequence:

$$a_1, b_1, a_2, b_2, a_3, b_3, \ldots, a_n, b_n\, .$$

If the goals do not have the same number of operations, then the execution sequence will be:

$$a_1, b_1, a_2, b_2, a_3, b_3, \ldots, a_n, b_n, b_{n+1}, \ldots, b_m$$

where n is the number of operations in the first goal, m in the second, and $n < m$.

Suppose the goal consists of three sequential processes executing concurrently:

$$a_1 \star a_2 \mid b_1 \star b_2 \mid c_1 \star c_2$$

where a_i, b_j, and c_k are all elementary operations. The first elementary operations, a_1, b_1, c_1, are executed sequentially, then the second operations, a_2, b_2, c_2, are executed. Thus, the execution sequence for this example is:

$$a_1, b_1, c_1, a_2, b_2, c_2 \,.$$

The examples given so far illustrate the concurrent execution of goals that have elementary operations only. In the next example, some of the goals are defined by other transaction rules:

$$a_1 \star p \mid q \star b_3.$$
$$p :- \quad a_2 \star a_3.$$
$$q :- \quad b_1 \star b_2.$$

where a_i, b_j are elementary operations. The first operation that executes is a_1, followed by the first elementary operation of the second goal. Since q is defined as $b_1 \star b_2$, b_1 executes next, followed by the second elementary operation from the first goal. The goal p is defined as $a_2 \star a_3$, so the next elementary operation submitted for execution is a_2. The next elementary operation of the second goal, b_2, executes, followed by a_3 and b_3.

Based on the definitions of p, q , the execution of the concurrent processes from this example is equivalent to the execution of:

$$a_1 \star a_2 \star a_3 \mid b_1 \star b_2 \star b_3 \,,$$

resulting in the following execution sequence:

$$a_1, b_1, a_2, b_2, a_3, b_3 \,.$$

The next example illustrates the execution when the concurrent processes are defined recursively. Consider the following recursive definition of p and q:

$$p\,(n) :-\; a\,(n) \star p\,(n-1)\,.$$
$$p\,(0)\,.$$
$$q\,(n) :-\; b\,(n) \star q\,(n-1)\,.$$
$$q\,(0)\,.$$

where a and b are elementary operations that immediately succeed. Suppose the following concurrent goal is submitted for execution:

$$p\,(s) \mid q\,(s)\,.$$

Using the definition of p, q with the argument s:

$$p\,(s) :-\; a\,(s) \star p\,(s-1)\,.$$
$$q\,(s) :-\; b\,(s) \star q\,(s-1)\,.$$

the concurrent goal becomes:

$$a\,(s) \star p\,(s-1) \mid b(s) \star q\,(s-1)\,.$$

After applying the definition $s+1$ times, the initial goal becomes:

$$a\,(s) \star a\,(s-1) \star ... \star a\,(1) \mid b(s) \star b\,(s-1) \star ... \star b\,(1)\,.$$

The prototype does not completely unfold rules before interleaving them, but for simplification, this is a good way to explain how the elementary operations are scheduled. When the concurrent goals are submitted for execution, the first operation from the first goal, $a\,(s)$, is started and completed, followed by the first operation of the second goal, $b\,(s)$. In a similar fashion, the processes continue to interleave until all the operations are executed:

$$a\,(s)\,, b(s), a\,(s-1)\,, b\,(s-1)\,, ..., a\,(1)\,, b\,(1)\,.$$

Since the elementary operations used in the previous examples immediately succeed, the execution of the programs presented in this section is always successful. Sequential execution submits the elementary operations one after another, while concurrent execution uses round-robin scheduling. However, there are sometimes instances when elementary operations do not succeed immediately. The prototype handles these situations using a backtracking mechanism that is presented in more detail next.

3.3 Failure, Deadlock, Abort, and Rollback

This section explains the CTR prototype behavior when eventually no process can continue execution. In the case of a single process, it is called *failure*. In the case of multiple processes, this is called *deadlock*. Based on the type of operations executed before the failure or deadlock occurs, each of these may lead to *rollback* or *partial rollback* and re-execution. As in the previous section, both sequential and concurrent executions are considered and presented separately.

The prototype has a backtracking mechanism for updates, not just for queries as in Prolog. Thus, database consistency is preserved even when a transaction that updated the database fails. During process execution, the prototype performs a *relative commit* of the updates, as opposed to an *absolute commit*, which takes place only after the entire transaction succeeds.

3.3.1 Sequential Execution

A sequential transaction executes goals one after another, and it succeeds if and only if all its individual goals succeed. It performs like an atomic operation that executes to completion or not at all.

The *relative* versus *absolute commit* of database updates is illustrated by the next example. Consider the following sequential transaction:

$$p :- \; ins(a) \star b \star c \star del(d)\,.$$

and assume that a and c are not in the database initially, but b and d are. The first update, $ins(a)$, inserts atom a into the database, and performs a *relative commit*. Next, the second operation, b, queries the database and succeeds, followed by the third operation, c, that fails since c is not in the database. At this point, the entire transaction fails, and the update to the database is undone. Even though the update performed by $ins(a)$ has been already committed, the database can be restored to its initial state. The undo is possible because the commit was not absolute, but was relative to the overall transaction.

The next example shows that a *local abort* does not always cause a *global abort*, and explains transaction execution when one of its goals acts as a *choice point*. Suppose that a sequential transaction is defined by the following rules:

$$q :- \; del(a_1) \star p.$$
$$p :- \; del(a_2).$$
$$p :- \; ins(a_2).$$

where a_2 is not in the database initially, and p is a process defined by the second and third rules.

Recall that the insert and delete operations are *strong updates*, *i.e.*, $del\,(b)$ fails if b is not initially in the database, and $ins\,(b)$ fails if b is initially in the database. During the execution of transaction q, the update $del(a_1)$ succeeds assuming atom a_1 is already in the database. Then p is invoked, and one of the updates, $del(a_2)$ or $ins(a_2)$, is chosen non-deterministically and submitted for execution. Since a_2 is not in the database, the update $del(a_2)$ fails, and p is re-executed using the third rule. If $ins(a_2)$ also fails[2], then there is no re-execution for p that would succeed. This failure triggers the abort of the entire transaction, so all the updates are undone, and the rollback is a *total rollback*. However, if $ins(a_2)$ succeeds, then the transaction has a successful execution.

[2]Both $ins.a_2$ and $del.a_2$ could fail if relation r was not declared as *updatable*, as described in Section 3.1.2

Note that, the abort of $del(a_2)$ is a *local abort*, and it does not always cause the abort of q, which is a *global abort*. Since p has more than one possible execution, it acts as a *choice point* and a *save point* within q, so any failure within p has only a local effect and need not cause an abort of q.

3.3.2 Concurrent Execution

As in the sequential case, the execution of a concurrent goal succeeds if and only if all the individual processes succeed. However, unlike sequential execution, where processes execute one after another, a concurrent execution interleaves the executions of several processes. During execution, the process that executes first is chosen non-deterministically, followed by other processes without a pre-determined execution order. Even though a particular interleaving fails, the transaction need not abort, since another interleaving might succeed. A concurrent transaction is unsuccessful if and only if all possible interleavings fail.

Example 3.1 The execution of a concurrent goal when a elementary operation fails is illustrated by the following simple example. Suppose a concurrent goal invokes two elementary operations:

$$\varphi \mid \psi$$

During execution, there are two possible interleavings for the operations, with the following execution schedules:

$$\varphi, \psi \qquad\qquad \psi, \varphi$$

Suppose that the elementary operation ψ fails, and the interleaving φ, ψ is executed first. The failure of this execution sequence does not cause the failure of the goal. After the updates to the database are undone, the second interleaving, that is ψ, φ, is submitted for execution. In this instance, the failure of ψ causes the the failure of the second execution sequence. Since both interleavings fail, the goal execution fails, and all the database updates are undone. □

Intuitively, if each a_i is an elementary operation, then $a_1 \mid a_2 \mid a_3 \mid ... \mid a_n$ has $n!$ execution schedules.

Example 3.2 The next example describes the execution of a concurrent goal when a process is forced to fail. Consider the goal:

$$a \mid b \mid fail$$

where a, b, c are elementary operations that always succeed, and $fail$ is the built-in Prolog predicate that causes the goal to fail always. In the absence of any optimization, all possible execution schedules are considered before a *global abort* takes place. Since the goal has 3 elementary operations, there is a total of 6 (or 3!) distinct execution schedules:

$$\begin{array}{lll} a, b, fail & a, fail, b & b, a, fail \\ b, fail, a & fail, a, b & fail, b, a\,. \end{array}$$

□

Example 3.3 The CTR prototype can use the database for communication between processes and synchronization, as shown in the following example. Suppose that a concurrent goal is defined as follows:

$$\begin{array}{l} p \mid q. \\ p :- \ a_1 \star a_2 \star a_3 \star ins.b_4 \star a_4 \star a_5 \star a_6. \\ q :- \ b_4 \star b_5 \star b_6. \end{array}$$

and assume that b_4 is not in the database initially. The sequential processes are executed concurrently by interleaving their elementary operations. Since the second process starts by querying the database for b_4, it only succeeds after the atom b_4 is inserted into the database. However, this is done by the first process after the execution of a_1, a_2, and

a_3. Thus, all the execution schedules that query the database for the atom b_4 before it is inserted into the database fail. Assuming that, with the exception of b_4, all a_i and b_i immediately succeed, one of the successful execution schedules for the goal is:

$$a_1, a_2, a_3, ins.b_4, b_4, a_4, b_5, a_5, b_6, a_6\,.$$

Thus, even though the goal invokes two processes that have many possible interleavings, the processes succeed only if they are executed in a specific sequence. In effect, the insertion of b_4 into the database triggers a synchronization message from one process to the other, enabling the user to control the execution sequence. □

3.4 Prolog Terms, *monitor*

The CTR prototype accepts Prolog terms as part of goal definitions. Their processing is explained in this section. The prototype also provides an easy way to trace the execution of CTR programs using a special command called *monitor* .

3.4.1 Prolog Terms

A Prolog term is treated as an elementary operation by the CTR prototype, *i.e.*, its execution is not interleaved with that of other processes.

Both database and transaction base areas can invoke Prolog terms in rule definitions. However, the database should contain Prolog rules only, as illustrated by the next example. Suppose that the following rules represent the contents of a database:

$$p(X,Y,Z) :- \; q(X), r(Y,Z), s(Y,Z).$$
$$r(Y,Z) :- \; t(Y), s(Z).$$
$$r(Y,Z).$$

Since they contain Prolog terms only, they are executed as any other Prolog rules, in top-down, left-to-right fashion.

The Prolog terms may also be part of a transaction base, in Prolog rules or in combination with CTR terms. Simple examples of Prolog terms used in transaction base rules are:

$read\,(X)$ `- reads next term from the input stream`

$write\,(Y)$ `- writes term` Y `to the current output stream`

In the following example:

$$p\,(X) : -functor\,(g\,(a, b)\,, F, M) \star q\,(M)\,.$$

the Prolog term $functor$ is used in a transaction base rule to determine the arity of g.

The rule in the next example has both CTR and Prolog terms and may be included in a transaction base:

$$s(U, V) : - \;\; t(U) \star (r(V), p(U, V)) \star q(U).$$

The CTR rule is executed using the sequential scheduling explained in Section 3.3.1. However, the term $(r(V), p(U, V))$ is a Prolog term that is evaluated and has its execution determined by Prolog.

3.4.2 The *monitor* Command

The user can include the *monitor* command in rules in the transaction base to trace various tasks during program execution. This command is a CTR specific command, with the following syntax:

$$monitor(Task).$$

where $Task$ is a name. When the *monitor* command executes, it displays one of the messages:

- *commiting* $Task$ - when $monitor(Task)$ is executed
- *undoing* $Task$ - when $monitor(Task)$ is rolled back.

Note that $monitor(Task)$ does not execute $Task$, it only displays messages. The next example shows the messages that are displayed during the execution of a goal with three sequential processes. Suppose that a program transaction base contains the following rule:

$$p :- \ monitor(a_1) \star monitor(a_2) \star monitor(a_3).$$

The execution of p generates the following messages:

commiting a_1

commiting a_2

commiting a_3

An artificial way to force a goal to abort is to execute the predicate $fail$ as the last process of a goal . The next example illustrates *monitor* command behavior when a goal fails:

$$q :- \ monitor(a_1) \star monitor(a_2) \star monitor(a_3) \star fail.$$

During the goal execution, the *monitor* command displays the following messages:

commiting a_1

commiting a_2

commiting a_3

undoing a_3

undoing a_2

undoing a_1

The above messages confirm that the prototype performed as expected, *i.e.*, that backtracking takes place after the processes are forced to fail. Since the rule invokes a sequential goal, only one execution schedule is possible for q, which fails and is undone.

The *monitor* command can also be used in conjunction with concurrent goals. The next example invokes the command to trace the execution of a concurrent goal during processes synchronization:

$$
\begin{aligned}
& p \mid q. \\
& p :- \ \ monitor(a_1) \star monitor(a_2) \star monitor(a_3) \star ins\,(b_4) \\
& \qquad \star monitor(a_4) \star monitor(a_5) \star monitor(a_6). \\
& q :- \ \ b_4 \star monitor(b_4) \star monitor(b_5) \star monitor(b_6).
\end{aligned}
$$

When the concurrent goal $p \mid q$ executes, the following messages are displayed:

commiting a_1

commiting a_2

commiting a_3

commiting b_4

commiting a_4

commiting b_5

commiting a_5

commiting b_6

commiting a_6

confirming that the database can be successfully used to synchronize processes. While executing p and q, the two processes run concurrently, without interacting with each other. Process q cannot start until b_4 is inserted into the database. However, b_4 is not inserted unless p executes $ins\,(b_4)$, which will only happen after it executes the first three tasks. In this way, p is synchronized with q. □

Chapter 4

Compiling, Optimizing, and Executing a CTR Program

To execute a CTR program, the user has to first compile the source code from both transaction-base and database files. The compilation process creates object files that the prototype consults during goal execution. After this compilation, the user has the option to optimize the programs, generating optimized CTR object files. This chapter describes the files used during compilation and optimization, along with the commands that generate them. All commands for the CTR prototype are executed from the XSB interpreter[1].

4.1 Standard CTR Files

This section gives the format for the files necessary to create and run CTR programs. As previously stated, the CTR prototype requires the transaction-base and database rules to be stored in separate files. The file name for the transaction base has the following format:

[1] For details, see Appendix.

program_name`.ctr`

where *program_name* is user-assigned name for the program and *.ctr* is the standard file extension that the prototype uses to identify a transaction-base storage file.

For each *CTR* program, a transaction-base file and a database file are required. If there are no database rules and/or declarations, then an empty database file should be created. The database file name format is:

program_name`.db`

where *program_name* is the user assigned name for the program and *.db* is the extension for database storage files. For a *CTR* program, the transaction-base and database file must have the same *program_name*.

The transaction base compilation generates a transaction-base object file with the name:

program_name`.ctr.o`

where *program_name* is the same as the source program name and with the extension *ctr.o* identifying the file as a transaction-base object file.

The database object file created by compilation has the name:

program_name`.db.o`

where *program_name* is the source program name and *db.o* is the extension used for database object files.

4.2 Optimized *CTR* Files

The files presented next are created and used by the prototype only when the programs are optimized. There are two types of optimization, which are presented in more detail in Chapter 6 and 7.

The first optimization creates an object transaction file with the name:

program_name`.opt1.o`

where *program_name* is the same as the *CTR* program name and the extension *opt1.o* identifies the file as an optimized transaction-base file.

The second optimization creates new transaction and database object files:

program_name`.opt2.o`

where *program_name* is the *CTR* source program name and the extension *opt2.o* identifies the file as an optimized transaction-base file. The optimized database file generated has the name format:

program_name`.opt2.db`

where *program_name* is the *CTR* program name and *db2.o* is the standard extension for an optimized database file.

4.3 *CTR* Compilation Commands

This section presents the commands that can be used to compile *CTR* programs, along with some simple examples.

The *CTR* programs are compiled using two commands with the following format:

- comp_trans(*program_name*). - to compile the transaction base
- comp_db(*program_name*). - to compile the database

where *program_name* is the name that the user associates to the the transaction-base and database files.

For example, to compile the following *CTR* program files created by the user:

- $workflow.ctr$ - transaction-base file
- $workflow.db$ - database file

the commands used are:

```
comp_trans(workflow).
comp_db(workflow).
```

In this particular case, the compilation process creates the following object files:

- $workflow.ctr.o$ - transaction-base object file
- $workflow.db.o$ - database object file.

When a goal executes, if no optimization is performed, the prototype uses the goal definitions from the object files.

4.4 CTR Optimization Commands

As previously stated, there are two types of optimization that the user can apply to the object files to improve CTR program performance. The first type of optimization eliminates backtracking for some processes that are executed in isolation. This is achieved by replacing the *isolate* operator, o, with *isolate one*, $o1$, under special conditions[2]. The second type of optimization improves program efficiency by replacing CTR terms with Prolog terms whenever possible[3].

Since the *isolate* operators can only be used in the transaction-base (and not the database), the first optimization modifies the transaction-base object file only. The format for the optimization command is:

[2]More details on this type of otimization can be found in Chapter 6.
[3]This optimization is explained in Chapter 7.

`opt1_trans(`*program_name*`).`

where *program_name* is the user-assigned program name.

After the compilation and first optimization, a second type of optimization can be performed. This optimization translates CTR query formulas into the corresponding Prolog terms, so it is a *query compilation*. It affects both the transaction base and the database, and is invoked using the command:

`opt2_trans(`*program_name*`).`

where *program_name* is the name of the program to be optimized.

The following example optimizes the following object files:

- $workflow.ctr.o$ - transaction-base object file
- $workflow.db.o$ - database object file.

The first optimization is performed using the command:

`opt1_trans(`$workflow$`).`

that creates the file:

$workflow.opt1.o$ - transaction-base optimized object file.

Next, the second optimization can be performed:

`opt2_trans(`$workflow$`).`

generating the following optimized files:

- $workflow.opt2.o$ - transaction-base optimized object file
- $workflow.opt2.db$ - database optimized object file.

After program optimization is performed, the prototype uses the goal definitions from the optimized object files.

4.5 Summary Example

The source program $workflow$ can be compiled and optimized using the following CTR commands, which are executed from the XSB interpretor:

$comp\_trans\,(workflow)$.

$comp\_db\,(workflow)$.

$opt1\_trans\,(workflow)$.

$opt2\_trans\,(workflow)$.

Before these commands are executed, the user has to create the source programs and store them in the input files:

$workflow.ctr$

$workflow.db$

The following object files are produced by the commands:

$workflow.ctr.o$

$workflow.db.o$

$workflow.opt1.o$

$workflow.opt2.o$

$workflow.opt2.db$

Chapter 5

Basic Interpreter

This chapter presents the prototype internal syntax, that is the object files syntax, and implementation. The object files consist of Prolog code that is generated by the compilation of the source files, which consist of CTR code.

The prototype implementation consists of seven Prolog modules[1]:

- *ctr.pl* - the basic CTR interpreter
- *parser.pl* - a parser for CTR rules
- *updates.pl* - the code for backtrackable updates
- *load.pl* - startup routine that loads the prototype modules
- *upload.pl* - a module that loads a CTR transaction base and a database into XSB
- *optim1.pl* - an optimization that eliminates unnecessary backtracking
- *optim2.pl* - optimization code for special query compilation

The user must start the XSB interpreter, and compile these modules inside XSB[2].

[1]The CTR interpreter and the backtrackable updates module were developed by professor Anthony Bonner, and are the starting point for our implementation.

[2]The Appendix gives more details on the prototype startup.

5.1 CTR Internal Syntax

This section presents the internal syntax, describes more clearly the relationship to Prolog syntax, and the relationship between the internal and user syntax[3].

5.1.1 Transaction Base

During compilation, the prototype translates CTR *transaction rules* into internal syntax as follows:

- a CTR atomic formula is a Prolog atom, so its user and internal syntax are identical
- a rule $\alpha : - \beta$, where α is an atomic formula and β is a *transaction formula,* is translated into $\alpha : - \delta$, where δ is the translation of the *transaction formula* β.

The user syntax of t*ransaction formulas*[4] is translated into internal syntax using the following rules:

- $a_1 \star a_2 \star ... \star a_n$ has the internal representation $seq\left([a_1, a_2, ..., a_n]\right)$
- $a_1 \# a_2 \# ... \# a_n$ has the internal representation $conc\left([a_1, a_2, ..., a_n]\right)$
- $o\left(\theta\right)$ has the internal representation $isolate\left(\theta\right)$
- $o1\left(\theta\right)$ has the internal representation $isolate1\left(\theta\right)$

The examples presented next illustrate the relationship between user and internal syntax. Here is a simple example that shows the user syntax and internal syntax for a rule with a sequential goal:

- $p : - a \star b.$ - user syntax

[3]For details on user syntax see Section 3.1.
[4]The *transaction formulas* are presented in Subsection 3.1.1.

- $p :- seq([a, b]).$ - internal syntax

The following example gives the user and internal syntax for a rule with isolated and concurrent goals:

- $q :- a_1 \# o(b) \# a_2 \# a_3.$ - user syntax
- $q :- conc([a_1, isolate(b), a_2, a_3]).$ - internal syntax

When a CTR rule consists of a combination of sequential and concurrent goals, it is translated as shown in the following example:

- $p :- a \star b \# c.$ - user syntax
- $p :- conc([seq([a, b]), c]).$ - internal syntax

It should be noted that the *concurrent composition* operator has the highest precedence, followed by the *sequential composition*, and then the is*olate* operator. However, the user can override the precedence of the CTR operators by using square brackets. If square brackets are used in the previous example, then the rule internal syntax becomes:

- $p :- a \star [b \# c].$ - user syntax
- $p :- seq([a, conc([b, c])]).$ - internal syntax

The compilation of transaction-base files creates the CTR transaction-base object files by applying recursively the translation rules. More examples of source program compilation are provided in Appendix.

5.1.2 Database

Since the CTR database contains *database rules* and *database tuples* are written using Prolog code[5], the user syntax and internal syntax are identical. The rules and tuples

[5] More details on the *database rules* and *tuples* syntax can be found in Subsection 3.1.2.

are copied into the CTR object database file, are loaded into Prolog database, and are invoked during programs execution.

The *updatable* statement is used to declare the updatable *database tuples*[6]. During compilation, these statements are translated using the reserved predicate *modif*. In general, a database statement with the user syntax:

$$updatable\ \ R/n.$$

has the internal syntax:

$$modif\,(R\,(X_1, X_2, ..., X_n))\,.$$

As a summary, the prototype translates the database user syntax into internal syntax as follows:

- the CTR *database tuples* and *database rules* have the same user and internal syntax;
- each *updatable* statement is replaced by an atomic formula of the form $modif\ (X)$.

The following example shows both the user and internal syntax for an updatable relation named $source$:

- $updatable\ source/1.$ - user syntax
- $modif\ (source\,(X_1)).$ - internal syntax

After compilation, the Prolog rules from the transaction-base and database object files can be replaced with more efficient rules using the optimization process.

[6]For more details, see Section 3.1.2.

5.2 Backtrackable Updates

This section describes one of the main features of the CTR prototype: backtrackable updates. As opposed to the update operations provided by Prolog, the CTR prototype undoes updates during backtracking (since CTR treats backtracking as transaction rollback).

There are various techniques for implementing backtrackable updates in Prolog. A standard technique is described next, and is performed using the following lines of Prolog code:

$$del\,(p) : -retract\,(p)\,, onbacktracking\,(p)\,.$$
$$onbacktracking(p)\,.$$
$$onbacktracking(p) : -assert\,(p)\,, !, fail.$$

When a tuple is in the database and the predicate *del* is invoked to delete it, the built-in Prolog predicate *retract* is called first to delete the tuple. Next, the predicate *onbacktracking* is executed successfully, using its first definition, and the deletion succeeds. However, if the transaction invoking *del*, fails, then the second definition of *onbacktracking* is executed during backtracking. In this particular case, the Prolog built-in predicate *assert* inserts the tuple back into the database, so the database is in its original state even after a transaction fails. The *cut* $fail$ combination, or !, $fail$, forces the parent goal *del* to fail.

In the following example, a transaction that updates the database is executed, and the results of the backtrackable update operation, *del*, are compared to the ones obtained using the built-in Prolog predicate *retract*. The *backtrack* program compiled in the next XSB session contains the standard backtracking code presented above. In both cases the transaction executed fails, and the comparison shows that the database updates are undone properly only when backtracking is performed.

```
dvp.cs> xsb
XSB Version 1.7.1 (7/10/97)
[Solaris, optimal mode]
| ?- compile(backtrack).          % backtrack program is compiled
[Compiling ./backtrack]
% Specialising partially instantiated calls to onbacktracking/1
[backtrack compiled, cpu time used: 0.3400 seconds]
yes
| ?- reconsult(backtrack).        % backtrack program is loaded
[backtrack loaded]
yes
| ?- assert(p).                   % p is inserted into the Prolog database
yes
| ?- p.                           % query to verify that p is in the Prolog
yes                               % database
| ?- del(p),fail.                 % p is deleted using strong delete
no                                % the transaction fails
| ?- p.                           % p is still in the Prolog database
yes
| ?- retract(p),fail.             % p is deleted using the Prolog built-in
no                                % predicate and the transaction fails
| ?- p.                           % p is not in the Prolog database
no
```

First, atom p is inserted into the database, then the transaction:

```
del(p),fail.
```

is executed. Since p is in the database, the operation $del(p)$ succeeds, the atom p

is deleted from the database. To force backtracking the Prolog built-in predicate *fail* is executed next, and the transaction fails anyway. During backtracking the database updates are undone, and the database is restored to its state before transaction execution. The only update operation executed is the deletion of p from the database, and the subsequent query shows that p is still in the database after the transaction fails.

The XSB session continues with the execution of the following Prolog command:

```
retract(p),fail.
```

This shows the difference between CTR updates (*ins* and *del*) and Prolog updates (*assert* and *retract*). It performs the same operations; however, the deletion of p from the database is done using the Prolog built-in predicate $retract\,(p)$. Exactly as in the previous transaction, it fails; however, the backtracking does *not* undo the update, thus leaving the database in an inconsistent state. The execution of this transaction shows that Prolog executions are *not* transactional.

It is not hard to undo updates during backtracking. Unfortunately, if this is done in a straightforward way, it can corrupt the database. In particular, in some Prologs, the standard backtracking code does not always work properly, because the system can be corrupted if updates are performed during backtracking. The following example shows one of these situations, assuming that the tuples $p\,(a)$, $p\,(b)$, and $p\,(c)$ are initially in the Prolog database. When the following command, that uses the predicate *del* defined in the standard backtracking code described earlier, executes inside a Prolog session, some Prolog interpreters either return errors or do not produce the correct results:

```
| ?- p(X), del(p(X)), fail.
```

The correct execution of the command should delete one of the tuples $p\,(a)$, $p\,(b)$, or $p\,(c)$ from the database then fail and undo the deletion. However, as a result of the corruption of the hash table, some of the Prolog interpreters execute the transaction incorrectly.

An example is XSB Version 1.7.1., where the command execution produces an infinite loop[7].

Because of such problems, a backtracking mechanism that correctly undoes updates is required. The CTR prototype provides a solution to these issues in the module *updates.pl*. The module implements a technique that prevents the Prolog database from being corrupted when backtracking through updates. The result is that updates performed in CTR are correctly undone during backtracking. The prototype described in this thesis is built on top of these update operations. More details on the implementation of these backtrackable updates can be found in [38].

5.3 Interpreter

The interpreter is the starting point for the prototype and is the central module of this CTR implementation. It provides the definition for the *exec* predicate, which is used to submit goals for execution. The predicate has one argument and can be invoked during an XSB session by typing:

$$exec\,(Goal)\,.$$

where *Goal* is the name of the goal to be executed. The *Goal* definition has to be part of a CTR program that was compiled, therefore it is already loaded into the Prolog database.

When a formula is submitted for execution, the prototype uses the interpreter to determine the *hot components* and to execute them recursively one at a time. The *hot components* are the goals from a formula that are ready for execution at a given time[8].

Since sequential goals are submitted for execution from left to right, the formula from the following example:

[7] More recent versions of XSB avoid this problem by providing backtrackable updates as special built-in predicates.

[8] More details on *hot components* can be found in [3] Section 4.1.

$$p \star q \star r.$$

has as *hot component* the goal p.

Considering the concurrent formula given in the next example:

$$a \mid (b \star c) \mid d.$$

the set of *hot components* is composed of the goals a, b, and c.

The interpreter implementation determines the set of *hot components* based on the internal syntax of the formulas. Its source code is in the *ctr.pl* module, and it assumes that the backtrackable updates logic presented in Section 5.2 is already implemented.

Chapter 6

Optimization of Non-Interacting Processes

This chapter describes the first optimization supported by the prototype: backtracking elimination. It provides information on *isolate* and *isolate*1 operators, the algorithm implemented to eliminate backtracking, and numerous examples. This optimizer assumes that the database contains no rules. The optimization is applicable to CTR programs which execute processes in isolation, and it replaces the operator *isolate*, or o, with *isolate*1, or $o1$, using criteria defined in Section 6.2.

6.1 *Isolate* and *Isolate1* Operators

The role of the isolate operators, o and $o1$, is to restrict communication between processes that execute concurrently. For example, when a transaction executes three processes concurrently :

$$\alpha \# \beta \# \gamma.$$

the prototype allows them to communicate or interact during execution, since each process can read what the others write. The database thus acts as the medium of communication.

The communication between process β and all the other processes can be restricted using the *isolate* operator, so the transaction becomes:

$$\alpha \# (o\beta) \# \gamma.$$

Here α and γ can communicate with each other, but not with β, since it executes in isolation.

When all concurrent processes execute in isolation, they execute as a set of serializable transactions, as in the following example:

$$(o\gamma_1) \# (o\gamma_2) .$$

which executes as one of the following:

$$\gamma_1 \star \gamma_2 . \qquad \gamma_2 \star \gamma_1 .$$

Concurrent processes may occur *within* an *isolate* operator, as in:

$$o (\alpha\#\beta\#\gamma) .$$

The optimization described in this chapter focuses on such situations. In particular, if the concurrent processes (*eg*, α, β, γ) do not interact, then o can be replaced by $o1$.

In general, during the execution of a concurrent program, the CTR interpreter tries all possible interleavings of the processes until one succeeds[1]. In some instances, however all interleavings have the same result and backtracking is not necessary, as shown in the example below. This is the case when the operations in concurrent processes commute [34]. In such cases, CTR gives the user the ability to say that only one interleaving should be tried. This is achieved by replacing the operator o, with the operator o_1, which

[1] Section 3.3.2 describes concurrent execution in more detail.

increases the efficiency of execution by eliminating the search through other equivalent interleavings.

To see this, consider the example of Financial Transactions in Example 2.1 in which the predicate $transfer\,(Amt, Act_1, Act_2)$ transfers an amount, Amt, from account Act_1 to account Act_2. Recall that the transfer fails if there is not enough money in Act_1 initially. The following transaction executes three transfers concurrently, to transfer \$100 from account $acct1$ into each of three other accounts:

$$\begin{aligned} &o(transfer\,(100, acct1, acct2) \,\#\, transfer\,(100, acct1, acct3) \qquad\qquad (*)\\ &\qquad \#\, transfer\,(100, acct1, acct4)). \end{aligned}$$

The $isolate$ operator is used to execute the transaction atomically, *i.e.*, if one $transfer$ fails, then all the others must be rolled back.

If the balance in account $acct_1$ is less than \$300, then one of the transfers fails, regardless of the order in which the three transfers execute. There is therefore no point in trying more than one execution order. In this example, the user can replace o by $o1$, using his knowledge that $transfer$ transactions commute:

$$\begin{aligned} &o1(transfer\,(Amt, acct1, acct2) \,\#\, transfer\,(Amt, acct1, acct3)\\ &\qquad \#\, transfer\,(Amt, acct1, acct4)). \end{aligned}$$

The advantage of the modified transaction is that, by using $o1$, it prevents needless backtracking during execution. Without this modification, if any of the $transfer$ transactions fail during execution, then the concurrent transaction would backtrack and another interleaving would be executed until all possible interleavings are tried. Since the concurrent $transfers$ do not interact, all such interleavings will produce the same result, *i.e.*, they will all fail. Hence, backtracking is not necessary and the operator $o1$ can be used.

For instance, consider two elementary operations, α, β. Suppose they are executed in isolation, as in the following simple example:

$o(\alpha \# \beta)$.

In this case, α and β can interact only with each other, and not with their environment (*i.e.*, other processes). Thus, since α and β are elementary, there are only two possible interleavings, α, β and β, α. The prototype will first execute one of these interleavings, say, α, β. If this is not successful, then α and β are backtracked, any updates are undone, and the other interleaving, β, α, is executed.

In contrast, when the user specifies the $isolate1$ operator:

$o1(\alpha \# \beta)$.

only one of the two interleavings is executed to determine the outcome of the transaction, so backtracking is prevented. It is sufficient to try just one interleaving if α and β commute, *i.e.*, if they do not interact.

6.2 Detecting Non-Interaction

As previously mentioned, backtracking is eliminated by replacing the $isolate$ operator with $isolate1$. In general, this replacement can be done when concurrent processes within the $isolate$ operator do not interact. The problem is to determine when processes interact and when they do not. This can be done manually by the user, as in the $transfer$ example above, and in some cases, it can be done automatically, as described in this section.

Consider a transaction that invokes a number of concurrent processes in isolation. The optimization replaces $o\varphi$ by $o_1\varphi$ in either of the following cases:

- φ executes only queries, or only inserts, or only deletes (which do not change the outcome of a transaction for different interleavings).

- the concurrent processes in φ invoke a combination of updates and queries, but each process accesses different predicates (so the processes do not interact).

The prototype detects such interactions by examining the rules in the transaction base using an algorithm based on the *dependency graph* of a CTR program, defined below. For simplicity, we assume that the database contains only atomic formulas. Without this assumption, things are more complex. For example, suppose that the transaction base contains the rule:

$$i :-\ o\,(b\#ins\,(c))\,.$$

and the database contains the rule:

$$b :-\ c\,.$$

Because of the database rule, the order in which b and $ins\,(c)$ execute is important. However, to determine this, the prototype would have to examine the database as well as the transaction base. To avoid this complication, we assume that the database contains only atomic formulas, and no rules.

6.3 Dependency Graph

The optimization implements an algorithm that uses a *dependency graph* to detect the interaction between processes. This subsection defines *dependency graphs*, and explains the process used to build a graph, with an example that shows the role of *dependency graphs* in optimization.

Given a CTR program, the following are *nodes* in the *dependency graph*:

- all predicates that are heads of rules
- all CTR elementary operations invoked in the rules

Given two nodes in the *dependency graph*, X and Y, the graph has an *edge* from X to Y, if for some rules:

- X is the predicate in the head of the rule
- Y is a predicate or an elementary operation in the rule body.

Example 6.1 Consider the following transaction base:

$$i \ :- \ o(a\#\,(ins\,(g) \star del\,(e)))\,.$$
$$a \ :- \ b\#c.$$
$$b \ :- \ d \star e.$$
$$c \ :- \ g \star f.$$

where d, e, f, g are updatable predicates.

The nodes of the *dependency graph* are:

- predicates that are heads of rules:

$i \qquad a \qquad b \qquad c$

- all elementary operations, which are either queries or updates:

$ins\,(g) \qquad del\,(e) \qquad d \qquad e \qquad g \qquad f$

The graph itself is shown is in Figure 6.1.

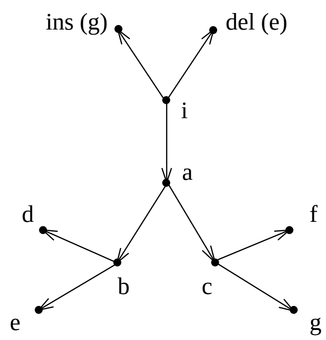

Figure 6.1: Dependency Graph 1

The *dependency graph* for this example has the following edges:

$\langle i, a\rangle$ $\langle i, ins\,(g)\rangle$ $\langle i, del\,(e)\rangle$ $\langle a, b\rangle$ $\langle a, c\rangle$

$\langle b, d\rangle$ $\langle b, e\rangle$ $\langle c, g\rangle$ $\langle c, f\rangle$

6.4 Interactions

The optimization uses the *dependency graph* defined in the previous section to determine the interaction between processes. The predicates invoked by a CTR program can be included in one of the categories defined in this section, which are used in the optimization algorithm.

An *updatable predicate* is one that is:

- stored in the database, and
- can be updated during the CTR programs execution

As described in Section 3.1.2, an *updatable predicate* has to be defined in the database using the statement:

updatable R/n.

where R is the predicate name and n its arity.

Definition 6.1 [Query Set] Given a predicate p, the *query set of* p, or $qs\,(p)$, is the set of all updatable predicates reachable from node p in the *dependency graph.*

The *query set of a formula* φ, or $qs\,(\varphi)$, is the union of the *query sets* of all predicates in φ.

Consider the rules defined in Example 6.1:

$$i \;:-\; o(a\#\,(ins\,(g) \star del\,(e)))\,.$$

$$a \;:-\; b\#c.$$

$$b \;:-\; d \star e.$$

$c \ :- \ g \star f.$

the following query sets are associated to the predicates and formulas from the example:

$$qs\,(a) \ = \ \{d, e, f, g\}$$

$$qs\,(b) \ = \ \{d, e\}$$

$$qs\,(\varphi) \ = \ \{\} \ \ , \ \texttt{where} \ \ \varphi \ = \ ins\,(g) \star del\,(e)$$

Definition 6.2 [Insert Set] An *updatable predicate, p*, invoked in a CTR formula as the argument of an *insert* operation, such as $ins.p\,(\overline{x})$, is an *insert predicate.* Given the predicate r, the *insert set of* r, or $is\,(r)$, is the set of all *insert predicates* reachable from node r in the *dependency graph.*

The *insert set of a formula* φ, or $is\,(\varphi)$, is the union of the *insert sets* of all predicates in φ.

Given the rules from Example 6.1, the following are *insert sets*:

$$is\,(a) \ = \ \{\}$$

$$is\,(i) \ = \ \{g\}$$

$$is\,(\varphi) \ = \ \{g\} \ \ , \ \texttt{where} \ \ \varphi \ = \ ins\,(g) \star del\,(e)$$

Definition 6.3 [Delete Set] An *updatable predicate, p*, invoked in a CTR formula as the argument of a *delete* operation, such as $del.p\,(\overline{x})$, is a *delete predicate.* Given the predicate r, the *delete set of* r, or $ds\,(r)$, is the set of all *delete predicates* reachable from node r in the *dependency graph.*

The *delete set of a formula* φ, or $ds\,(\varphi)$, is the union of the *delete sets* of all predicates in φ.

As an example, some of the delete sets for Example 6.1 are:

$$ds\,(a) \ = \ \{\}$$

$$ds\,(i) \ = \ \{e\}$$

$$ds\,(\varphi) \ = \ \{e\} \ \ , \ \texttt{where} \ \ \varphi \ = \ ins\,(g) \star del\,(e)$$

The optimizer determines the interaction between processes based on these three sets. For instance, two concurrent processes interact if the intersection between the *query set of one process* and the *insert set of the other process* is not an empty set.

In the example, since the intersection between $qs\,(a)$ and $is\,(\varphi)$ is $\{g\}$, processes a and φ would interact if executed concurrently.

All the combinations of the sets which cause interaction can be summarized in the following table:

	query set	*insert set*	*delete set*
query set		*	*
insert set	*		*
delete set	*	*	

Table 6.1 Interaction Table

In the table, the '*' symbol denotes the set types causing interaction. For example, two concurrent processes interact if the intersection between the *insert set of the formula* for a process and the *delete set of the formula* for the other process is not empty. This is represented by a '*' symbol in the *insert set* row and *delete set* column of the table. The following definition is based on Table 6.1.

Definition 6.4 [Non-Interaction] Two processes, ψ_i and ψ_j, *do not interact* if all of the following sets are empty:

$$
\begin{array}{lll}
qs\,(\psi_i) \cap is\,(\psi_j) & is\,(\psi_i) \cap qs\,(\psi_j) & ds\,(\psi_i) \cap qs\,(\psi_j) \\
qs\,(\psi_i) \cap ds\,(\psi_j) & is\,(\psi_i) \cap ds\,(\psi_j) & ds\,(\psi_i) \cap is\,(\psi_j)
\end{array}
$$

for $i \neq j$.

Definition 6.5 [Optimizable Formula] The formula $o\,(\psi_1\#\psi_2\#...\#\psi_n)$ is *optimizable* if ψ_i and ψ_j do not interact for all $i \neq j$.

Given a CTR formula, the optimizer replaces every *optimizable sub-formula*, $o(\varphi)$, by $o1(\varphi)$.

6.5 Examples

The following examples demonstrate the optimization algorithm when two or more concurrent processes are executed within the scope of an isolation operator.

Example 6.2 In this example, the optimization is performed on an isolated transaction without updates. Consider the transaction:

$$j \ :- \ o(p\#q\#r\#s)\,.$$

where p, q, r, and s are *updatable predicates*, *i.e.*, they are queries. The *dependency graph* for this example is:

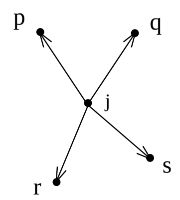

Figure 6.2: Dependency Graph 2

The *insert sets* and the *delete sets* for all the concurrent processes are empty, so the transaction invokes only non-interacting processes. Thus the execution of various interleavings does not change the outcome of the transaction, so the *isolate* operator can be replaced with *isolate*1, and the transaction becomes:

$$j \ :- \ o1(p\#q\#r\#s)\,.$$

Example 6.3 The isolated transaction in this example contains updates and a query which interact, so it cannot be optimized. Given the rule:

$$k \ :- \ o(del\,(a_1)\,\#ins\,(a_2)\,\#ins\,(a_1)\,\#a_1)\,.$$

where a_1 and a_2 are *updatable predicates*. Here is the *dependency graph* associated to the rule:

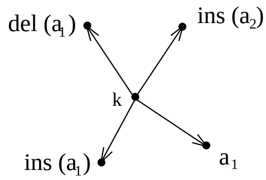

Figure 6.3: Dependency Graph 3

Consider the following formulas with the corresponding *sets:*

$ds\,(\varphi)$ $=\{a_1\}$, where $\varphi = del\,(a_1)$

$is\,(\psi)$ $=\{a_1\}$, where $\psi = ins\,(a_1)$

$ds\,(\varphi) \cap is\,(\psi) = \{a_1\}$

Since the intersection of the *insert* and *delete set* for the concurrent processes $del\,(a_1)$ and $ins\,(a_1)$ is not empty, the processes interact. In this case, the *isolate* operator cannot be replaced by *isolate*1. □

Example 6.4 There are instances when a rule has more than one *isolate* operator, and only one can be optimized. Given the rule definition:

$$a\ :-\ o(ins\,(s)\,\#del\,(v))\ \star\ o(ins\,(p)\,\#del\,(p)\,\#ins\,(q))\,.$$

with the updatable predicates: p, q, s, v. Figure 6.4 shows the *dependency graph* for this example.

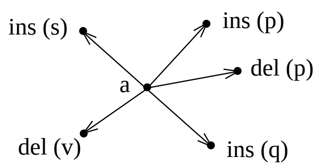

Figure 6.4: Dependency Graph 4.

The rule contains two isolated processes executed sequentially, where only the first *isolate* can be replaced with *isolate*1 during optimization. The second isolated process performs two elementary operations that interact, $ins\,(p)$ and $del\,(p)$, so different interleavings of this process produce different results, and it can not be optimized. Since there is more than one isolate operator, the optimization algorithm attempts to optimize each isolate operator separately. The first isolate has only non-interacting operations, however the second has a path from $ins\,(p)$ to $del\,(p)$ showing that the operations interact and cannot be optimized. The optimization replaces the above rule with the following:

$$a\ :-\ o1(ins\,(s)\,\#del\,(v))\ \star\ o(ins\,(p)\,\#del\,(p)\,\#ins\,(q))\,.$$

To see how the optimization process is applied to a rule with multiple levels of isolation, suppose that the rule defined above is modified as follows:

$$a\ :-\ o(o(ins\,(s)\,\#del\,(v))\ \#\ o(ins\,(p)\,\#del\,(p)\,\#ins\,(q))).$$

Backtracking elimination starts at the lowest level, so the *isolate* in $o(ins\,(s)\,\#del\,(v))$ can be replaced by *isolate*1, however the *isolate* operator in $o(ins\,(p)\,\#del\,(p)\,\#ins\,(q))$ cannot be replaced, since $ins\,(p)$ and $del\,(p)$ do not commute. The *isolate* operator at the front of the transaction can also be replaced, since the processes:

$$o1(ins\,(s)\,\#del\,(v)) \qquad\qquad o(ins\,(p)\,\#del\,(p)\,\#ins\,(q))$$

access different predicates, and therefore do not interact. Thus, the optimized rule is:

$$a\ :-\ o1(o1(ins\,(s)\,\#del\,(v))\ \#\ o(ins\,(p)\,\#del\,(p)\,\#ins\,(q))).$$

□

The examples presented so far explain the optimization for isolated transactions executing elementary operations. In practice, however many transactions invoke predicates defined in separate rules. In this case, the optimizer checks the interaction between the concurrent processes by analyzing the elementary operations performed by the other rules, as shown in the examples described next.

Example 6.5 An example that explains the optimization process for two concurrent predicates executed in isolation is presented here. Consider the following rules:

$$j \; :- \; o(p\#a)\,.$$
$$p \; :- \; ins\,(q)\,\#del\,(e)\,\#a\#ins\,(v) \star t.$$
$$a \; :- \; ins\,(b) \star del\,(c) \star d \star e.$$
$$d \; :- \; del\,(m)\,.$$

where q, e, v, t, b, c, m are *updatable predicates*, as defined in Section 6.4[2].

The optimization is applied to the definition of predicate j from the first rule. The following *dependency graph* and *sets* are determined for this example:

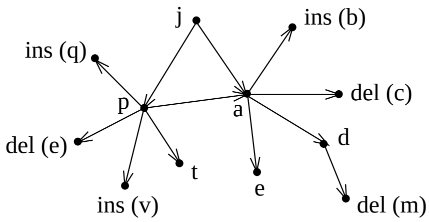

Figure 6.5: Dependency Graph 5

$$ds\,(p) \;=\; \{e, c, m\}$$
$$qs\,(a) \;=\; \{e\}$$
$$ds\,(p) \;\cap\; qs\,(a) \;=\; \{e\}\,.$$

Since both the *query set* and the *delete set* for processes p and a include e, there is interaction between the processes. The execution of different interleavings for p and a produces different results, so the optimization can not change the *isolate* operator, and the backtracking is not eliminated. □

Example 6.6 Backtracking can be eliminated by optimizing the set of rules presented in this example. Consider the following CTR program:

[2]Note that, in CTR, the operator $\star$ binds tighter than $\#$, so the second rule executes as:

$$p \; :- \; ins\,(q)\,\#del\,(e)\,\#a\#\,(ins\,(v) \star t)\,.$$

$$i \;:-\; o(p\#d)\,.$$

$$p \;:-\; ins\,(q)\,\#del\,(e)\,\#a\#ins\,(v) \star t.$$

$$a \;:-\; ins\,(b) \star del\,(c) \star d \star e.$$

$$d \;:-\; del\,(m)\,.$$

where q, e, v, t, b, c, m are *updatable predicates*. Since the first rule executes the concurrent processes p and d in isolation, the optimization is applicable.

The sets corresponding to the rules in this example are:

$$qs\,(p) \;=\; \{e,t\} \qquad is\,(p) \;=\; \{q,b,v\} \qquad ds\,(p) \;=\; \{e,c,m\}$$

$$qs\,(d) \;=\; \{\} \qquad is\,(d) \;=\; \{\} \qquad ds\,(d) \;=\; \{m\}$$

$$ds\,(p) \;\cap\; ds\,(d) \;=\; \{m\}\,.$$

The *dependency graph* for the program is:

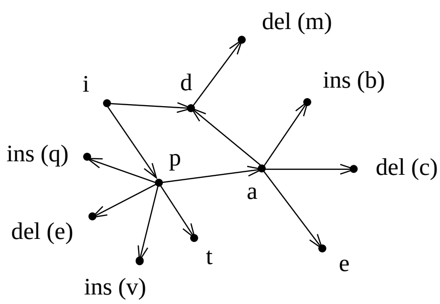

Figure 6.6: Dependency Graph 6

Since d has both the *query set* and *insert set* empty, their intersection with any of the sets associated to the definition of p is an empty set. The only intersection of the sets that is non-empty is the one between the *delete sets* of p and d. According to Table 6.1, two *delete sets* do not cause interaction, so the definition of predicate i can be optimized and *isolate* is changed to *isolate*1.

After the optimization is performed, the rule definition for i becomes:

$$i \;:-\; o1(p\#d)\,.$$

and there are no other changes to the rule definitions from this example. □

6.6 Implementation Outline

The prototype reads in the object transaction-base file, and parses each rule. If the rule can be optimized, then the optimized rule is written to the optimized object file; otherwise, the rule is copied to the file unchanged.

Given a CTR program that consists of a set of rules, first the *dependency graph* is created, using the relation *edge*, *i.e.*, for each edge from X to Y, the optimization algorithm inserts a tuple:

$$edge\,(X, Y)$$

into the Prolog database.

After all the edges are inserted, the predicate $reachable\,(P, Q)$ is defined as the transitive closure of the relation *edge*. It succeeds if there is a path between the nodes P and Q in the *dependency graph*. For concurrent processes that execute within the scope of an isolation operator, the implementation uses the predicate *reachable* to determine the *query*, *insert*, and *delete sets* associated to a process. At present, we assume that the transaction base is non-recursive, so that there are no loops in the *dependency graph*. This simplifies the computation of the transitive closure.

Finally, the predicate *interact* is defined to perform the intersection between the sets associated to the isolated concurrent processes. The predicate $interact\,(\varphi, \psi)$ is false if φ and ψ do not interact as defined in Section 6.4. Otherwise, $interact\,(\varphi, \psi)$ is true.

Chapter 7

Query Compilation

The second optimization available with the prototype replaces certain CTR formulas with equivalent Prolog terms. The only formulas that can be translated are those representing queries, so the optimization is a form of query compilation. This section defines the criteria used for optimization, and gives examples to illustrate.

7.1 Query Compilation Process

After CTR programs are compiled and backtracking is eliminated, the programs can be further optimized using a query compilation process. By translating some CTR formulas into Prolog terms, the optimized programs have a better performance, since the CTR interpreter is bypassed.

During the query compilation process, a CTR formula that represents a query can be translated into a Prolog term, based on the following criteria:

- if a term in a rule body executes only queries in isolation, then the CTR term is replaced by the corresponding Prolog term;
- if the rule body consists of a single isolation operator containing only queries, then the entire rule is translated into a Prolog rule. During rule translation the

optimizer uses CTR-generated database predicates by adding the suffix *_ctr_db* to the predicates invoked by the rule. In this case, the translated rule is placed in the program database (which is loaded into the Prolog database during execution).

Note that, since the prototype executes Prolog terms in isolation, this optimization transforms isolated queries into isolated queries.

CTR detects queries using the *dependency graph,* described in Chapter 6. The query compilation uses the graph to create the *query*, *delete* and *insert sets* for each predicate in the transaction base. The algorithm attempts to optimize only those terms in which both the *insert* and *delete sets* are empty.

The benefits of query compilation are further explained in the next section, which gives several examples.

7.2 Examples

The programs in the following examples provide more details on the query compilation process.

Example 7.1 This simple example presents the optimization performed for a CTR program, when a term in a rule body executes only queries in isolation. It describes the selection criteria that is applied, and the corresponding translation. Consider the following rule with the *dependency graph* in Figure 7.1:

$$h \ :- \ o(a \star b) \,\#\, (ins\,(c) \star del\,(d))\,.$$

where a, b, c, d are updatable predicates.

Using the *dependency graph* from Figure 7.1, the algorithm determines the sets associated to the predicates. In this example, the predicates executed in isolation, a and b, have the *insert* and *delete sets* empty, so the term can be optimized.

The term:

$o\,(a \star b)$

represents the sequential execution of two queries in isolation, so it can be translated into the following Prolog term:

(a, b)

based on the first criteria used for optimization.

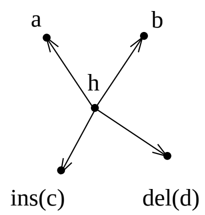

Figure 7.1: Dependency Graph 7

The query compilation creates the new rule:

$h \;:-\; (a, b)\,\#\,(ins\,(c) \star del\,(d))\,.$

which is placed in the optimized transaction-base file. □

Example 7.2 The next example illustrates the second selection criteria, when an entire rule is optimized. Consider a CTR program with the following rule definitions:

$a \;:-\; o\;(b)\,.$

$b \;:-\; c.$

$c \;:-\; d.$

where d is an updatable predicate. Here is the *dependency graph* for this example:

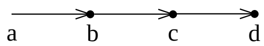

Figure 7.2: Dependency Graph 8

The *dependency graph* shows that the isolated process b has the *insert* and *delete sets* empty, so the rule is optimized. The first rule executes a query in isolation, so the optimizer translates it using a CTR generated database predicate:

$$a \; :- \; b\_ctr\_db. \quad (1)$$
$$b \; :- \; c. \quad (2)$$
$$c \; :- \; d. \quad (3)$$

For the optimization to be transparent to the user, rules (2) and (3) must be kept in the transaction base, so these transactions can be executed independently.

In addition to the rule translation that takes place in the transaction base of the CTR program, the following new database rules are inserted into the *program database* and loaded into the Prolog database:

- the CTR definitions for the optimized predicates are translated using the new predicate names:

$$b\_ctr\_db \; :- \; c\_ctr\_db. \qquad corresponds \; to \; (2)$$
$$c\_ctr\_db \; :- \; d\_ctr\_db. \qquad corresponds \; to \; (3)$$

- the following rules ensure that the new predicate definitions include the Prolog rules and tuples from the database:

$$b\_ctr\_db \; :- \; b.$$
$$c\_ctr\_db \; :- \; c.$$
$$d\_ctr\_db \; :- \; d.$$

For example, these rules are needed if the database contains:

$$b \; :- \; e.$$
$$c \; :- \; f, g.$$

where e, f, and g are updatable predicates.

After optimization, to execute predicate a, the prototype executes the optimized rule (1) using the Prolog rules from the database object file. Since these rules are loaded into the Prolog database, the CTR interpreter is bypassed and the program has better performance. □

Example 7.3 The following example describes another case where an entire rule can be optimized. Consider the set of rules, with the corresponding *dependency graph* in Figure 7.3:

$p :- o(q)$.

$q :- r \star s.$

with r and s updatable predicates.

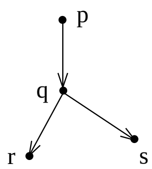

Figure 7.3: Dependency Graph 9

The isolated process q has the *insert* and delete sets empty, so query compilation is applied. The predicate q is defined by the second rule which invokes only queries, so the rule is translated along with all the predicates invoked. The transaction base for the program becomes:

$p :- q\_ctr\_db.$

$q :- r \star s.$

The definition of q is left in the transaction base so the user can execute the transaction defined in the second rule. As a result of optimization, the following rules are added to the program database and loaded into the Prolog database:

$q\_ctr\_db :- r\_ctr\_db,\ s\_ctr\_db.$

$q\_ctr\_db :- q.$

$r\_ctr\_db :- r.$

$s\_ctr\_db :- s.$

The last three rules are created to ensure that the database tuples or rules defining q, r, s are taken into account during execution. □

Example 7.4 As opposed to the previous examples, the program defined next cannot be optimized, even though the first rule invokes a predicate in isolation, similar to the one in the Example 7.3:

$$p \; :- \; o(q)\,.$$
$$q \; :- \; ins\,(r) \star s.$$

where r and s are updatable predicates. The *dependency graph* for this program is:

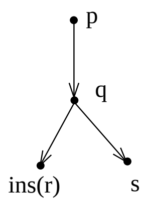

Figure 7.4: Dependency Graph 10

The definition of predicate p executes q in isolation, with q defined by the second rule of the program. The optimization algorithm uses the *dependency graph* to determine the sets associated to q:

$$qs\,(q) \; = \; \{s\}$$
$$is\,(q) \; = \; \{r\} \; \textit{note: insert set is not empty}$$
$$ds\,(q) \; = \; \{\}$$

As previously mentioned, if one of the *insert* or *delete sets* for a predicate is not empty, then the optimization cannot be applied, since the predicate is not a pure query. The definition of q performs an update operation, so it cannot be optimized and both rules remain unchanged after the optimization.

Chapter 8

Conclusions and Future Work

In the previous chapters we described our implementation of Concurrent Transaction Logic (CTR) in XSB Prolog. CTR is a logic programming language that provides a seamless integration of process modeling (concurrency and communication) and data modeling (queries and updates), including transactional capabilities (abort and rollback). To efficiently implement this integrated system, new optimization techniques are required. This thesis takes a first step in this direction by implementing two such optimizations: backtracking elimination and query compilation. Backtracking elimination is an optimization for non-interacting processes, which implements an algorithm based on a *dependency graph.* The second optimization implements a query compilation algorithm that replaces certain CTR formulas with equivalent Prolog terms. The prototype developed consists of a set of Prolog programs that define CTR connectives, backtrackable updates, and implementation-specific commands and directives.

Examples of CTR programs and XSB terminal sessions are available in the Appendix. The CTR web page[1] provides additional information on the implementation and offers the option to view and download the entire implementation. There are links to related work on deductive database language and other research papers.

[1]The web page URL is *www.cs.toronto.edu/˜bonner/ctr.html*

Since the main objective of this thesis is the optimization of CTR programs, two directions for the future work extend the work done in this thesis.

- *Detecting Non-Interacting Processes.* Our automation of backtracking elimination is quite conservative. The goal is to extend this optimization to detect a wider range of non-interacting processes. Even though detecting non-interaction is undecidable, special cases are decidable. In this thesis we look for processes that access different relations. However, even if the processes access the same relations, they may still not interact (*i.e.*, if they access different tuples, or perform increment operations, which commute). Such interactions can be detected at run-time as in classical concurrency control theory [34].

- *Program Compilation.* The query compilation presented in this thesis replaces certain CTR formulas with equivalent Prolog terms. It should be possible to extend this idea to a wider class of CTR formulas, including formulas with updates. By compiling CTR rules into Prolog rules we can avoid the overhead of running the CTR interpreter. This increases performance because compiled programs run faster than interpreted programs.

In addition, the implementation can be extended in other ways. An important example is *crash recovery.*

- *Crash Recovery.* A new direction for future work is to implement an extension of the current prototype to include crash recovery for long-running activities such as workflow [4, 8]. This is different from crash recovery in classical transaction management, which entirely undoes a failed transaction. For long-running activities [34], we do not want to undo possibly entire hours or days of work [8], so the run-time system must record both the program state and database state. A CTR implementation with crash recovery would allow a program to continue from where it left off.

Appendix A

CTR Tutorial

The tutorial in this appendix presents an overview of the prototype commands, the transaction base and database rules syntax, and built-in CTR predicates. It gives programming examples and XSB [12] terminal sessions. This tutorial can also be found on the CTR web page[1].

A.1 Getting Started

The CTR prototype was implemented on the XSB Prolog interpreter, and consists of a number of modules. To run the implementation, the *load* module must first be consulted. It defines the predicate *init*, which compiles and loads all the modules. The following XSB session shows the output when loading the CTR prototype in XSB Prolog:

```
$xsb
XSB Version 1.7.1 (7/10/97)
[Solaris, optimal mode]
| ?- [load].
[load loaded]
```

[1] The web page URL is *www.cs.toronto.edu/˜bonner/ctr.html.*

```
yes
| ?- init.
[ctr loaded]     - basic CTR interpreter
[updates loaded] - code for backtrackable updates
[parser loaded]  - parser for prototype rules
[upload loaded]  - translates and loads rules
[optim1 loaded]  - optimizer that eliminates unnecessary backtracking
[optim2 loaded]  - optimization code for special query compilation
yes
```

A.2 Prototype Commands

All commands for the *CTR* prototype are executed from the *XSB* interpreter. A *CTR* program consists of the transaction rules and database rules, which are stored in separate files:

- *program_name*.ctr - transaction base file
- *program_name*.db - database file

For each program, the *program_name* has to be the same for both transaction and database files. Both transaction base and database files must be created, even if one of them is empty.

A.2.1 Compilation Commands

A *CTR* program with the name *program_name* is compiled using two commands with the format:

- comp_trans(*program_name)* - to compile the transaction base

- comp_db(*program_name)* - to compile the database
- ctr_comp(*program_name)* - to compile and optimize a *CTR* program. It was created for convenience, by invoking all the compilation and optimization commands.

The compilation creates the object files:

- *program_name*.ctr.o - transaction-base object file
- *program_name*.db.o - database object file

The transaction base rules are parsed and compiled into a set of Prolog rules. The new rules are stored in the transaction-base object file and loaded into the Prolog database. Since the database rules are Prolog rules, they are copied to the database object file and loaded into the Prolog database.

A.2.2 Optimization Commands

After a *CTR* program has been compiled, the first optimization—backtracking elimination—can be invoked using the command:

`opt1_trans(`*`program_name`*`)`

It modifies the transaction-base object file, and creates the optimized file:

`program_name``.opt1.o`

Query compilation, the second optimization, can then be performed on *CTR* programs, with the following command:

`opt2_trans(`*`program_name`*`)`

The output files generated by query compilation are:

- *program_name*.opt2.o - transaction-base optimized object file
- *program_name*.opt2.db - database optimized object file

A.2.3 Execution Command

A CTR goal defined in the transaction base file is submitted for execution using the command $exec$, with the format:

$$exec\,(Goal)\,.$$

A.3 Prototype Syntax

A.3.1 CTR Rules

The transaction base rules, or CTR rules, have a rule head and a rule body with the syntax:

```
Head :- Body.
or
Head.
```

where Head is an atomic formula, and Body is a sequence of queries and updates connected by any of the following operators: *serial conjunction* ($\star$), *concurrent conjunction* ($\#$), *isolation* (o or $o1$), and *disjunction* ($\backslash/$).

The following is an example of a CTR rule from a transaction base file:

$$process\,(I) : -task\,(I,1) \star task\,(I,2) \star task\,(I,3)\,.$$

that executes three tasks sequentially.

The next rule uses logical disjunction to choose between two tasks:

$$process\,(I) : -\;\; task\,(I,1)\backslash/task\,(I,2)\,.$$

A transaction executes a task in isolation in the following rule:

$$process\,(I) : -\;\; task\,(I,1)\,\#task\,(I,2)\,\#o\,(task\,(I,3))\,.$$

A.3.2 Database Rules

A database rule is any Prolog rule or atomic formula. Prolog programs are treated as elementary operations, and are executed in isolation. In practice, the interpreter can handle *any* Prolog rule in the database, including rules having non-logical operators. Of course, the logical semantics of CTR applies only if the database contains purely logical rules.

A.4 Built-in CTR Predicates

The prototype defines some built-in predicates, which can be invoked in the database or the transaction base. These predicates should not be redefined by the user.

A.4.1 Database Declaration

Each database tuple manipulated by a CTR program is part of a relation. For a relation to be updatable, the database must contain a declaration of the form:

$$updatable\ \ Name/Narg.$$

This declares that a relation *Name* with arity *Narg* is updatable. Without this declaration, the prototype will not allow a CTR program to update a relation.

A.4.2 Transaction Base Predicates

For the database relations declared by an *updatable* statement, the following update commands are available in CTR:

- $ins(p\,(x))$ - inserts atom $p\,(x)$ into the database
- $del(p\,(x))$ - deletes atom $p\,(x)$ from the database

During commands execution, a task can be traced by including the *monitor* command in the transaction base rules. It has the syntax:

$$monitor\,(Task)\,.$$

and displays messages when *monitor* executes or rolls back, as shown in the examples below.

A.5 Programming Examples

The following examples present some simple CTR programs and their execution. They are executed using XSB Prolog, with the CTR prototype already loaded. The CTR web page has links to all the transaction base and database files specified in this section.

Example A.1 Concurrent Processes

The following transaction program executes two non-interacting processes concurrently, where each process is a sequence of three tasks. The prototype simulates concurrency by interleaving the execution of $process\,(a)$ and $process\,(b)$, in a *round-robin* fashion[2]. The program name is $interleave2$, and the contents of the corresponding transaction base and database files are as follows.

<u>Database file *interleave2.db:*</u> (empty file)

<u>Transaction Base file *interleave2.ctr:*</u>

$$parent : -process\,(a)\,\#process\,(b)\,.$$
$$process\,(I) : -task\,(I,1) \star task\,(I,2) \star task\,(I,3)\,.$$
$$task\,(I,J) : -\ \ monitor\,(task\,(I,J))\,.$$

[2]More details are available in Subsection 3.2.2.

```
| ?- ctr_comp(interleave2).        - the program is compiled
yes                                  and optimized
| ?- exec(parent).                 - execution of parent goal
committing task(a,1)               - first task of process(a) commits
committing task(b,1)               - first task of process(b) commits
committing task(a,2)               - second task of process(a) commits
committing task(b,2)               - second task of process(b) commits
committing task(a,3)               - third task of process(a) commits
committing task(b,3)               - third task of process(b) commits
yes
```

Example A.2 Concurrent Processes with Isolation

This example is a modified version of the previous program, which illustrates the role of the *isolate* operator, o. As mentioned in Section 6.1, isolation is used to restrict communication between processes that executes concurrently.

Note that $process\,(b)$ is executed in isolation. In the sample execution, $process\,(a)$ begins executing, is interrupted by $process\,(b)$, which executes completely, after which, $process\,(a)$ continues executing.

Database file *interleave2a.db:* (empty file)

Transaction Base file *interleave2a.ctr:*

$$parent : -process\,(a)\,\#o\,(process\,(b))\,.$$
$$process\,(I) : -task\,(I,1) \star task\,(I,2) \star task\,(I,3)\,.$$
$$task\,(I,J) : -monitor\,(task\,(I,J))\,.$$

```
| ?- ctr_comp(interleave2a).       - the program is compiled
yes                                  and optimized
```

```
| ?- exec(parent).            - execution of parent goal
committing task(a,1)          - process(a) starts executing
committing task(b,1)          - complete execution of
committing task(b,2)            process(b) in isolation
committing task(b,3)
committing task(a,2)          - process(a) continues its
committing task(a,3)            execution.
yes
```

Example A.3 Logical Disjunction

CTR executes a disjunctive goal by executing one of its disjuncts, chosen non-deterministically. The following program uses logical disjunction to choose between two sequential processes, which execute three sequential tasks each. In the sample execution, the first sequential process is executed, is forced to fail, and is undone. The second process is then executed, is forced to fail, and is undone.

Database file *disjunction.db:* (empty file)

Transaction Base file *disjunction.ctr:*

$$\begin{aligned} &choose : -\,(monitor\,(a_1) \star monitor\,(a_2) \star monitor\,(a_3)) \\ &\qquad \backslash/ (monitor\,(b_1) \star monitor\,(b_2) \star monitor\,(b_3))\,. \end{aligned}$$

```
| ?- ctr_comp(disjunction).   - the program is compiled
yes                             and optimized
| ?- exec(choose).            - execution of choose goal
committing a1                 - display from monitor
committing a2                   showing that the ai are
committing a3                   committed
```

```
yes
```

Note the use of $fail$ in the next XSB session, to show that backtracking takes place when an execution is unsuccessful. It forces both processes to be tried.

```
?- exec(choose),fail.             - choose will be forced to fail
committing a1                     - the ai are committed
committing a2
committing a3
undoing a3                        - the ai are rolled back
undoing a2
undoing a1
committing b1                     - the bi are committed
committing b2
committing b3
undoing b3                        - the bi are rolled back
undoing b2
undoing b1
no                                - all execution schedules have
                                    failed
```

Example A.4 Synchronizing Processes

In the following CTR program, the $parent$ transaction forces two interacting processes to execute concurrently while synchronizing themselves. The program starts by executing tasks from both $process\,(a)$ and $process\,(b)$. After executing $task\,(b_2)$, $process\,(b)$ waits until $process\,(a)$ inserts the atom $start\,(b_3)$ into the database. At this point, $process\,(b)$ continues, and $process\,(a)$ waits until $process\,(b)$ inserts $start\,(a_7)$ into the database. At this point, both processes continue to execute tasks.

Database file *synchro.db:*

$$updatable \;\; start/1.$$

Transaction Base file *synchro.ctr:*

$$parent : -process\,(a)\,\#process\,(b)\,.$$

$$\begin{aligned} process\,(a) : -&task\,(a_1) \star task\,(a_2) \star task\,(a_3) \star task\,(a_4) \\ &\star task\,(a_5) \star task\,(a_6) \star ins\,(start\,(b_3)) \\ &\star start\,(a_7) \star task\,(a_7) \star task\,(a_8)\,. \end{aligned}$$

$$\begin{aligned} process\,(b) : -&task\,(b_1) \star task\,(b_2) \star start\,(b_3) \\ &\star task\,(b_3) \star task\,(b_4) \star task\,(b_5) \star task\,(b_6) \\ &\star ins\,(start\,(a_7)) \star task\,(b_7) \star task\,(b_8)\,. \end{aligned}$$

$$task\,(I) : -monitor\,(task\,(I))\,.$$

Note that the database file declares the predicate *start* as *updatable*, so it can be updated by the rules in the transaction base file. Since the database file is not empty, it is compiled in the following XSB session.

```
| ?- ctr_comp(synchro).              - the program is compiled
yes                                    and optimized
| ?- exec(parent).                   - execution of parent goal
committing task(a1)                  - process(a) alternates with
committing task(b1)                    process(b)
committing task(a2)
committing task(b2)
committing task(a3)                  - process(b) waits for start(b3)
committing task(a4)                    to be inserted in database
committing task(a5)
```

```
committing task(a6)
committing task(b3)              - process(a) waits for start(a7)
committing task(b4)                to be inserted in database
committing task(b5)
committing task(b6)
committing task(b7)
committing task(a7)              - process(a) alternates with
committing task(b8)                process(b) again
committing task(a8)
yes
```

Bibliography

[1] A.J. Bonner and M. Kifer. Transaction Logic Programming. Technical Report CSRI-323, University of Toronto, November 1995.

[2] A.J. Bonner. Concurrent Transaction Logic. Presented at the *Dagstuhl Seminar on Transaction Workflows*, July 15-19 1996, International Conference and Research Center for Computer Science, Schloss Dagstuhl, Wadern, Germany.

[3] A.J. Bonner and M. Kifer. Concurrency and Communication in Transaction Logic. In *Joint Int'l Conference and Symposium on Logic Programming*, pages 142-156, Bonn, Germany, September 1996. MIT Press.

[4] A.J. Bonner. Workflow, Transactions, and Datalog. Accepted to the *ACM Symposium on the Principles of Database Systems (PODS)*, 1999, Philadelphia, PA.5 Barry Wilkinson. *Digital System Design*. Prentice Hall 1987.

[5] A.J. Bonner and M. Kifer. A logic for programming database transactions. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 5, pages 117–166. Kluwer Academic Publishers, March 1998.

[6] A.J. Bonner. Transaction Datalog: a compositional language for transaction programming. In *Proceedings of the International Workshop on Database Programming Languages*, number 1369 in LNCS, pages 373–395. Springer-Verlag, 1998. Workshop

held in Estes Park, Colorado, August 1997. Long version available at http://www.cs.toronto.edu/~bonner/papers.html#transaction-logic.

[7] A.J. Bonner and M. Kifer. The State of Change: A Survey. In B. Freitag, H. Decker, M. Kifer and A. Voronkov, editors, *Transactions and Change in Logic Databases*, volume 1472 of Lecture Notes in Computer Science, pages 1-36. Springer-Verlag, Berlin, 1998.

[8] A.K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications.* Morgan-Kaufmann, San Mateo, CA, 1992.

[9] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, Englewood Cliffs, NJ, 1985.

[10] D. Harel. "StateCharts: A Visual Formalism for Complex Systems". *Science of Computer Programming*, 8:231–274, 1987.

[11] D. Harel, O. Kupferman, and M.Y. Vardi. On the complexity of verifying concurrent transition systems. Technical Report CS97-01, Faculty of Mathematical Sciences, The Weizmann Institute of Science, Rehovot, Israel, 1997.

[12] David S. Warren. Konstantinos F. Sagonas, Juliana Freire, Prasad Rao, and Terrance Swift, *The XSB Programmer's Manual Version 1.7.1*, SUNY at Stony Brook, 1997.

[13] E. Shapiro. A family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3), 1989.

[14] E. Shapiro, editor. *Concurrent Prolog*, volume 1. MIT press, Cambridge, MA, 1987.

[15] E.A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*, pages 997–1072. Elsevier, 1990.

[16] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 244–263, 1986.

[17] G. De Giacomo, Y. Lesperance, and H.J. Levesque. ConGolog, a concurrent programming language based on the situation calculus: language and implementation. Submitted, February, 1999.

[18] G. Grahne. Updates and counterfactuals. In *Proceedings of the International Conference on Knowledge Representation and Reasoning (KR)*, pages 269–276, Boston, Mass., April 1991.

[19] G. Grahne and A.O. Mendelzon. Updates and subjunctive queries. Technical Report KRR-TR- 91-4, CSRI, University of Toronto, July 1991.

[20] H. Katsuno and A.O. Mendelzon. On the difference between updating a knowledge base and revising it. In *Proceedings of the International Conference on Knowledge Representation and Reasoning (KR)*, pages 387–394, Boston, Mass., April 1991.

[21] Henry F. Korth and Abraham Silberschatz. *Database System Concepts*. McGraw-Hill, 1991.

[22] Ivan Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 1991.

[23] J. Desel. Free Choice Petri Nets. *Cambridge University Press*, 1995.

[24] J. McCarthy. Situations, actions, and clausal laws, memo 2. Stanford Artificial Intelligence Project, 1963.

[25] J. W. Lody. *Foundations of Logic Programming*. Springer-Verlag, 1987. Foundations of Logic

[26] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.

[27] J.C.M. Baeten and W.P. Weijland. Process Algebra. *Cambridge University Press*, 1990.

[28] J.D. Ullman. *Principles of Database and Knowledge Base Systems*. Computer Science Press, 1989.

[29] K. Jensen. *Colored Petri Nets*. Springer-Verlag, Berlin, 1996.

[30] K. Jensen and G. Rozenberg, editors. *High Level Petri Nets*. Springer-Verlag, Berlin, 1991.

[31] M. Hennessy. *An Algebraic Theory of Processes*. MIT Press, Cambridge, MA, 1988.

[32] M. Winslett. A model based approach to updating databases with incomplete information. *ACM Transactions on Database Systems*, 13(2):167–196, 1988.

[33] N. Adam, V. Atluri, and W. Huang. Modeling and analysis of workflows using Petri nets. *Journal of Intelligent Information Systems*, 10(2):131–158, March 1998.

[34] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Databases*. Addison-Wesley, 1987.

[35] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[36] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, September 1992.

[37] S. Manchanda and D.S. Warren. A logicŋbased language for database updates. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 363–394. Morgan-Kaufmann, Los Altos, CA, 1988.

[38] S.Y.K. Hung. *Implementation and Performance of Transaction Logic in Prolog*, 1996, Master Thesis, University of Toronto.

[39] U. Dayal and Q. Chen. From database programming to business process programming. In *Proceedings of the International Workshop on Database Programming Languages*, Gubbio, Umbria, Italy, September 1995. Springer-Verlag.

[40] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, 1994.

[41] W. Reisig. Petri Nets: An Introduction. Springer-Verlag, Berlin, 1985.