NOTE TO USERS

This reproduction is the best copy available.

UMI®

CONCURRENT TRANSACTION LOGIC WITH PRIORITY AND TIMING CONSTRAINTS

by

JIWEN GE
B.Eng
Hangzhou, ZheJiang, P.R.China, 1989

A thesis

presented to Ryerson University

in partial fulfillment of the

requirement for the degree of

Master of Applied Science

in the Program of

Electrical and Computer Engineering

Toronto, Ontario, Canada, 2004

©JIWEN GE 2004

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

UMI Number: EC52923

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI®

UMI Microform EC52923
Copyright 2009 by ProQuest LLC.
All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest LLC
789 E. Eisenhower Parkway
PO Box 1346
Ann Arbor, MI 48106-1346

Borrower's Page

Ryerson University requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Name	Signature	Address	Date

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CONCURRENT TRANSACTION LOGIC WITH PRIORITY AND TIMING CONSTRAINTS

Master of Applied Science 2004

JIWEN GE

Electrical and Computer Engineering

Ryerson University

Abstract

Concurrent Transaction Logic ($\mathcal{CTR}$) is a deductive language for programming database transaction applications that integrates queries, updates, and transaction composition in a complete logical framework. The language supports all the properties of classical transactions and the properties found in some new transaction models, e.g., sub-transactions, transaction rollback, and concurrent transactions.

The contributions of this thesis are twofold. First, it extends $\mathcal{CTR}$ to account for timing-event-based prioritized concurrent systems in which transactions may have priority and timing constraints. This extension of $\mathcal{CTR}$, here called $\mathcal{TP}$-$\mathcal{CTR}$, provides a high-level logic programming framework for specifying and simulating executions of timed transactions and trigger-events commonly present in real-time concurrent applications. Second, it describes a Prolog implementation of $\mathcal{TP}$-$\mathcal{CTR}$. The implemented $\mathcal{TP}$-$\mathcal{CTR}$ prototype supports the translation from $\mathcal{TP}$-$\mathcal{CTR}$ to $\mathcal{CTR}$. Underlying this protocol, we use a simplified Rate-Monotonic algorithm [10] to schedule the execution of constraint concurrent transactions and built-in timing predicates to handle transaction time-relations.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Acknowledgment

I appreciate the support and encouragement from Dr. Marcus Vinicius Santos , my supervisor. His expertise in Transaction Logic and Logic programming was instrumental for this work.

I would like to thank all the committee members for their participation in my thesis defense.

I would like to thank the Electrical and Computer Engineering Department at Ryerson University for the financial support of my study and research.

I would like to thank all staff of the Electrical and Computer Engineering at Ryerson University for their kind help.

I wish to convey the warmest thanks to my family for their endless support.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Contents

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Figures

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Tables

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 1

Introduction

THIS thesis introduces concurrent transaction logic with priority and timing constraints($\mathcal{TP}$-$\mathcal{CTR}$), an extension of Concurrent Transaction Logic ($\mathcal{CTR}$). $\mathcal{TP}$-$\mathcal{CTR}$ is designed to provide a high-level logic programming framework for specifying and simulating the control logic of time-related concurrent systems.

Concurrent Transaction Logic is a recently proposed deductive database language, introduced by Bonner and Kifer in [1]. It is based on Transaction Logic ($\mathcal{TR}$) [3], also known as *sequential* Transaction Logic. $\mathcal{CTR}$ extends $\mathcal{TR}$ with connectives for modeling the concurrent execution of complex processes, in the sense that it integrates concurrency, communication, and database updates in a complete logic framework.

In $\mathcal{CTR}$, concurrency is accomplished by interleaving the execution of concurrent transactions. Such mechanism is implemented in the $\mathcal{CTR}$ prototype in terms of a pure Round-Robin algorithm [10]. Hence, in $\mathcal{CTR}$, the underlying scheduling algorithm for the selection of next executed transaction component from a concurrent transaction execution does not follow any explicit priority-based scheme.

As for sequential transactions, $\mathcal{CTR}$ uses the sequential conjunction operator ($\otimes$) to denote the order of transaction execution, e.g., $a \otimes b \otimes c$ means: first executes transaction a, then transaction b, then transaction c. Exception handling excluded, in $\mathcal{CTR}$, there is no other mechanism one can use to force a particular execution sequence of transactions. Moreover, $\mathcal{CTR}$ does not provide any timing constraint type predicate which is necessary

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

for embedding scheduled events in a database transaction application system, e.g., a real-time database application in which some events can trigger expected transactions at specified times.

The study on the use of logics in real-time systems has been the subject of substantial research in the past and nowadays. In [4], Bellini and Mattolini reviewed a selection of the most representative temporal logics designed for real-time systems. For the specification of a real-time system behaviour, the time behaviour of a system is naturally described with constraints on event occurrences in [5]. Chen and Tsai present a modification of a pure temporal logic in [6], describing the system behaviours in terms of absolute timing of events as well as their relative ordering which can tell when the state actually occur. Torp and Jensen even assigned data with time properties in [7] .

As well, Ulusoy and Sivasankaran studied the use of logics to specify priority properties in real-time systems [8, 9] respectively. More specifically they focus on the design of scheduling algorithms to improve the protocol efficiency and maximize the number of transactions satisfying their real-time constraints.

However, these works focus either on priority constraints or on timing constraints. They do not combine these two types of constraints in one logic framework. We deem this greatly limits their application as formalism for specifying timing-related systems. In our approach, on the other hand, we are able to handle both types of constraints.

To be able to provide a formalism for specifying and simulating timing-event-based systems with explicit priority, in this thesis we extend $\mathcal{CTR}$'s inference system by defining a priority-constraint-based inference system, which allows us to formally execute formulas using a SLD-style refutation mechanism. Such inference system is implemented in the interpreter in terms of a simplified Rate-Monotonic Scheduling algorithm instead of a pure Round Robin scheduling algorithm. Our logical framework also introduces timing constraint formulas for specifying timing properties commonly found in real-time application domains. This improved timing-event-based concurrent transaction logic we call $\mathcal{TP}$-$\mathcal{CTR}$. Like $\mathcal{CTR}$, $\mathcal{TP}$-$\mathcal{CTR}$ is a language for programming database transac-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

tions and applications. $\mathcal{TP}$-$\mathcal{CTR}$'s timing and priority constraints enable programmers simulate real-time features in $\mathcal{TP}$-$\mathcal{CTR}$, e.g., time-event-driven and interrupt. The time-event-driven feature can be simulated by the $\mathcal{TP}$-$\mathcal{CTR}$'s timing constraints. Also, the key feature of real-time software system, interrupt, can be created by combining these two constraints: timing constraints and priority constraints. These features extend the use of logic programming languages to timing-event-based real-time database transaction applications.

A lot of real-time simulation platforms, current in use or under research, use procedure languages to design the real-time system, such as c in VxWorks [11] and RT-Linux [12]. It is unquestionable that the solutions designed by procedural languages usually provide high performance for real-time applications. But this is not always the case. To some real-time systems, e.g., real-time database applications, real-time systems with complicated control logic, simulations designed with procedural languages appear to be difficult and timing-consuming jobs. This is because procedural languages focus on computation result. But complicated logic include numerous computation results corresponding to the large amount of composite scenarios of pre-conditions. In such cases, simulating with procedural language is timing-consuming and low efficient. On the other hand, logic programming shows its advantages on this problem since logic programming focuses on the nature of control logic and explores every possible solution of it. As a type of logic programming, $\mathcal{TP}$-$\mathcal{CTR}$ not only has the natural advantages of logic programming to handle complicated control logic problems, but also includes real-time functionalities. By combining priority and timing constraints, and the high-level logic programming framework provided by $\mathcal{CTR}$-based approaches in a complete logic framework, $\mathcal{TP}$-$\mathcal{CTR}$ provides a possible better solution to handle these real-time applications with complicated control logic over procedural language c.

Along with the introduction of $\mathcal{TP}$-$\mathcal{CTR}$, the thesis also presents an implementation of $\mathcal{TP}$-$\mathcal{CTR}$ in Prolog, based on the $\mathcal{CTR}$ prototype. It is assumed that the reader is already familiar with Prolog. The prototype runs on the XSB prolog interpreter [15, 16],

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

currently the most efficient deductive database system.

Thesis Outline

The thesis is organized as follows:

- Chapter 2 briefly presents the essentials of $\mathcal{CTR}$'s syntax and semantics. Those readers more acquainted with $\mathcal{CTR}$ could skip this chapter.
- Chapter 3 describes a system view of the $\mathcal{CTR}$ prototype software, and gives a detailed description of how program execution and scheduling take place in the prototype.
- Chapter 4 introduces $\mathcal{TP}$-$\mathcal{CTR}$, its priority and timing constraints syntax, informal semantics and inference engine, and shows how the inference engine is used to execute constraint concurrent transaction Horn rules.
- Chapter 5 presents the developed $\mathcal{TP}$-$\mathcal{CTR}$ prototype, focusing on the differences between the $\mathcal{TP}$-$\mathcal{CTR}$ prototype and the $\mathcal{CTR}$ prototype.
- Chapter 6 provides $\mathcal{TP}$-$\mathcal{CTR}$ program examples on two different application areas, namely, time-based database transaction, and real-time control system.
- Chapter 7 concludes the work and elaborates on possible improvements.

Besides, the thesis includes one appendix, which presents a tutorial on how to use the software application developed in this thesis.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 2

An introduction to $\mathcal{CTR}$

CONCURRENT Transaction Logic is a deductive database language for programming database transactions and applications. The language, an extension of Transaction Logic [3], integrates concurrency, communication, and database updates in a complete logic framework. This chapter outlines the language using the terminology of deductive databases. Details are available in [1, 2].

2.1 Syntax

The syntax of Concurrent Transaction Logic is similar to that of first order logic, except that it extends first-order logic with three new logical connectives: $\otimes$, called *sequential conjunction*; $\mid$, called *concurrent conjunction*; and a modality of isolation, $\odot$, for specifying *atomic* actions that 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 resulting logical formulas are called *transaction formulas*.

In certain important situations, $\mathcal{CTR}$ has an elegant, top-down, SLD-style proof procedure that can be expressed within $\mathcal{CTR}$ itself. The definition below, lists the conditions that characterizes these situations. The subset of $\mathcal{CTR}$ that satisfies these conditions is called the *Horn fragment* of $\mathcal{CTR}$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Definition 1 (The Horn fragment of $\mathcal{CTR}$) The syntax of the Horn fragment of $\mathcal{CTR}$ is defined recursively as follows:

- An atomic formula is an expression of the form $p(t_1, ..., t_n)$, where p is a predicate symbol, and $t_1, ..., t_n$ are terms.
- A *concurrent sequential goal* is in any formula of the form:
  - An atomic formula; or
  - $a_1 \otimes ... \otimes a_i$; or
  - $a_1 \mid ... \mid a_i$; or
  - $\odot\, a_1$

  where each a_i is a concurrent sequential goal, and $i \geq 0$.
- If a is a concurrent sequential goal and t is an atomic formula, then $t \leftarrow a$, is a *concurrent Horn rule.*
- A *transaction base* is a set of concurrent Horn rules.

□

If a and b are transaction formulas, then informally:

- $a \otimes b$ means: first execute a, then execute b.
- $a \mid b$ means: execute a and b concurrently.
- $\odot a$ means: execute a "atomically", i.e., without interleaving with other transactions.
- $t \leftarrow a$ means: to execute t is sufficient to execute a.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.2 Database States and Elementary Updates

In $\mathcal{CTR}$, a pair of oracles, called *state data oracle* and *state transition oracle*, specify elementary database operations. The state data oracle specifies a set of database state queries, and the state transition oracle specifies a set of database elementary updates. These oracles are not fixed because any pair of oracles can be *plugged into* a $\mathcal{CTR}$ theory. For ease of reference, below we present the definition of these oracles, introduced in [3].

Definition 2 (state data oracle) A *state transition oracle* O^d, is a mapping from sets of state identifiers to sets of first-order formulas. □

Intuitively, if $\mathbf{D}_i$ is a state identifier, then $O^d(\mathbf{D}_i)$ is the set of formulas considered to be all the truths known about the state $\mathbf{D}_i$. In practice, it is not necessary to materialize all these truths. Because, given a logical formula ϕ and a state identifier $\mathbf{D}_i$, the proof theory for $\mathcal{CTR}$ only needs to know whether $\phi \in O^d(\mathbf{D}_i)$. Thus, to do inference in $\mathcal{CTR}$, an enumeration of $O^d(\mathbf{D}_i)$ is all that is needed.

Definition 3 (state transition oracle) A *state transition oracle* O^t, is a mapping from pairs of state identifiers to sets of ground atomic formulas. These ground atoms are referred to as *elementary transitions.* □

Intuitively, if $\mathbf{D}_1$ and $\mathbf{D}_2$ are two state identifiers, and $b \in O^t(\mathbf{D}_1, \mathbf{D}_2)$, then b is the set of elementary updates that change state $\mathbf{D}_1$ into state $\mathbf{D}_2$. An elementary update can thus be non-deterministic, since for each update, the transition oracle defines a binary relation on states. In practice, this relation does not have to be materialized. Instead, for a given update u, and a given state $\mathbf{D}_1$, the proof theory of $\mathcal{CTR}$ only needs an enumeration of the possible successor states, $\mathbf{D}_2$.

2.2.1 The Relational Oracle

The examples in this thesis use the notion of relational databases, in which a state is a set of tuples, and elementary transactions consist of the insertion and deletion of individual

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

tuples from the database.

In [1, 2, 3], Bonner and Kifer represent relational databases in the usual way as sets of ground atomic formulas. Moreover, they use two predicates, *ins* and *del*, to insert and delete atoms from the database. The definition of *Relational Oracles* formalizes this idea.

Definition 4 (Relational Oracles) : A state D is a set of ground atomic formulas. The data oracle simply returns all these formulas. Thus $O^d(\mathbf{D}) = \mathbf{D}$.

Moreover, for each p in D, the transition oracle defines two new predicates, $ins(p)$ and $del(p)$, representing the insertion and deletion of single atom p in D respectively as follows:

$$ins(p) \in O^t(\mathbf{D}_1, \mathbf{D}_2) \ \ \textit{iff} \ \ \mathbf{D}_2 = \mathbf{D}_1 + \{p\}$$

$$del(p) \in O^t(\mathbf{D}_1, \mathbf{D}_2) \ \ \textit{iff} \ \ \mathbf{D}_2 = \mathbf{D}_1 - \{p\}$$

□

It should be noted, however, that $\mathcal{CTR}$ is restricted neither to relational databases, nor to update operations based on single tuple. For instance, databases could be deductive, object-oriented, disjunctive, or a collection of scientific objects, such as matrices or DNA sequences. Likewise, database operations could include SQL-style bulk updates, or the insertion and deletion of rules, or complex scientific calculations, such as the Fourier transformation and matrix inversion. In $\mathcal{CTR}$, the set of states is determined by the data oracle. Changing the oracles can change the set of states, and thus the set of semantic structures. This is one way in which different oracles give rise to different versions of $\mathcal{CTR}$. In this thesis, we used the relational oracle for the simplicity.

2.3 Examples of $\mathcal{CTR}$ Formulas

This section gives some examples to illustrate $\mathcal{CTR}$'s syntax and informal semantics. The full description of $\mathcal{CTR}$ semantics is available in [1, 3]. We start with simple examples of sequential goals and concurrent sequential goals.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Sequential conjunction and database update predicates: The formula below illustrates the sequential conjunction ($\otimes$) combined with the elementary update predicates *del* and *ins*:

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

"Delete $r(a)$ from the database, and then insert $s(a)$".

Concurrent sequential goal: The formula below illustrates the use of the concurrent and sequential conjunctions, $\mid$ and $\otimes$, respectively, in the specification of concurrent transactions:

$$(\phi_1 \otimes \phi_2) \mid (\varphi_1 \otimes \varphi_2)$$

"Execute concurrently the transactions $\phi_1 \otimes \phi_2$ and $\varphi_1 \otimes \varphi_2$. To execute $\phi_1 \otimes \phi_2$, first do ϕ_1 then ϕ_2, and similarly for $\varphi_1 \otimes \varphi_2$".

Horn rules: Like classical logic, $\mathcal{CTR}$ has a Horn-like fragment with both a procedural and a declarative semantics. The formula:

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

defines a subroutine with name q and parameter X. Given the parameter value a, $q(a)$ commits if the atom $r(a)$ exists in the databases before the updates execute and assuming $r(X)$ is a updatable database tuple.

In the following examples, we show how $\mathcal{CTR}$ can be used to combine elementary operations into complex transactions.

Example 1 **(Database transactions for operating an online store)** Assume a relation *inventory(Name, Amt)* represents the amount of available goods in a store's inventory, where *Name* is the goods' name and *Amt* is an integer representing the amount available. To simplify, we ignore the price and other factors. The rules below define three transactions in the transaction base:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$
\begin{aligned}
process\_order(CustomInfo, Name, Amt) \leftarrow \quad & \\
& \odot (\ inventory(Name, OldAmt) \\
& \otimes OldAmt \geq Amt \\
& \otimes NewAmt\ is\ OldAmt - Amt \\
& \otimes change\_inventory(Name, OldAmt, NewAmt) \\
& \otimes ins(record(CustomInfo, Name, Amt)))
\end{aligned}
$$

$$
\begin{aligned}
supply(SupplierInfo, Name, Amt) \leftarrow \quad & \\
& \odot (\ not\ inventory(Name, OldAmt) \\
& \otimes ins(inventory(Name, Amt)) \\
& \otimes ins(supplier(SupplierInfo, Name, Amt))) \\
supply(SupplierInfo, Name, Amt) \leftarrow \quad & \\
& \odot (\ inventory(Name, OldAmt) \\
& \otimes NewAmt\ is\ OldAmt\ +\ Amt \\
& \otimes change\_inventory(Name, OldAmt, NewAmt) \\
& \otimes ins(supplier(SupplierInfo, Name, Amt)))
\end{aligned}
$$

$$
\begin{aligned}
change\_inventory(Name, OldAmt, NewAmt) \leftarrow \quad & \\
& \odot (\ del(inventory(Name, OldAmt)) \\
& \otimes ins(inventory(Name, NewAmt)))
\end{aligned}
$$

The first rule specifies: to sell an amount *Amt* of goods *Name* to a customer *Customer-Info*, first check how many goods are available in the store's inventory database. If the available amount, *OldAmt*, is no less than the requested amount, *Amt*, then deduct *Amt* from the available amount *OldAmt*, and update the latest inventory amount, and then record this transaction in the database; otherwise the *process_order* transaction should fail since the available amount of goods can not meet the demand of the order. The second rule specifies the supplying transaction: to supply an amount *Amt* of goods into the inventory, if the goods' name are new to the inventory, add a new inventory record with the goods' name and amount, and then add another supplying record to the supplier's account; Otherwise, if the goods have already a record in the inventory, add the amount into that inventory record, and add another record to the supplier's account in the same way. The last rule is in charge of changing the amount of a goods in the inventory record: first delete the old record with the *OldAmt*, then insert a new record with *Name* and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

NewAmt.

Due to the $\odot$ operator, *process_order*, *supply*, and *change_inventory* are all executed "atomically". That is , they execute either entirely or not at all. The transaction below specifies the concurrent execution of *supply* and *processing_order*:

$$supply(supplier1, tape1, Amt1) \mid processing\_order(customer1, tape1, Amt2)$$

That is, an amount *Amt*1 of goods *tape*1 is supplied by *supplier*1 while a customer places an order for *tape*1. Because this is intended to be a transaction, if one sub-transaction fails, then both sub-transactions are rolled back whatever the other sub-transaction succeeds or fails.

Notice that, the $\mathcal{CTR}$ program behaves correctly while an equivalent Prolog program could not: updates in Prolog are not logical. If the execution fails after an update is performed, the update cannot be undone. So although execution in Prolog can be backtracked, the database does not roll back to its initial state if updates are involved. Thus transaction fail in a Prolog program will lead to database inconsistency. □

The above example illustrates the combination of concurrency and updates that $\mathcal{CTR}$ supports. The next example shows how $\mathcal{CTR}$ also supports communication, where two processes synchronize themselves by exchanging messages via communication signals.

Example 2 **(Synchronization between two elevator control processes)** Assume *stopRequest_control* and *moving_control* are two processes specified to control the stop request control system and the moving control system of an elevator model, respectively. Here, our goal is to illustrate the communication between the two concurrent processes. Hence, we simplify our model to enable one round processing of these two control processes, without considering the loop condition.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$
\begin{aligned}
stopRequest\_control \leftarrow\ & task_{11} \\
& \otimes send(requestSignal, FloorNum) \\
& \otimes receive(stopSignal, Flr) \\
& \otimes task_{12}
\end{aligned}
$$

$$
\begin{aligned}
moving\_control \leftarrow\ & task_{21} \\
& \otimes receive(requestSignal, FloorNum) \\
& \otimes moveTo(FloorNum) \\
& \otimes send(stopSignal, Flr) \\
& \otimes task_{22}
\end{aligned}
$$

where $task_{11}$ and $task_{21}$ are initialization tasks of $stopRrequest\_control$ and $moving\_control$, respectively; $task_{12}$ and $task_{22}$ are the post-handling tasks of *stopRequest_control* and *moving_control*, respectively.

The next transaction specifies the concurrent execution of *stopRequest_control* and *moving_control*:

$stopRequest\_control \mid moving\_control$

During execution, *stopRequest_control* and *moving_control* can communicate and synchronize the execution of their tasks effectively by sending and receiving messages along channel *requestSignal* and *stopSignal*. Note: $send(Ch, Msg)$ and $receive(Ch, Msg)$ are two communication predicates, which are used to send and receive a message Msg along a channel Ch, as shown in [1]. $moveTo$ cannot start moving the elevator to a specific floor until $task_{11}$ finishes and a $FloorNum$ message is received along the *requestSignal* channel from the stop request control process. Likewise, *stopRequest_control* process cannot continue until $moveTo$ finishes and a stop message Flr is received along the *stopSignal* channel. In such a way, the two processes communicate with each other and synchronize their execution steps.

□

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.4 $\mathcal{CTR}$'s Inference System

This section first introduces the SLD-style resolution of the $\mathcal{CTR}$ inference system. This inference system is used to formally execute transactions. In Chapter 4, we will introduce an extension of this inference system, which allow us to formally execute transactions involving constraints.

SLD-style resolution

The inference system manipulates expressions called *sequents*, which have the form

$$\mathbf{P}, \mathbf{D} \vdash (\exists)\, \phi$$

where $\mathbf{P}$ is a program, $\mathbf{D}$ is any legal database state, and ϕ is a concurrent sequential goal. The informal meaning of such a sequent is that, based on program $\mathbf{P}$ the formula $(\exists)\, \phi$ can be proved from state $\mathbf{D}$.

Let the concurrent sequential goal clause be the expression

$$\begin{array}{c} \leftarrow G_0 \\ \text{where } G_0 \text{ is the sequent } \mathbf{P}, \mathbf{D}_1 \vdash (\exists)\, \phi \end{array} \tag{2.1}$$

A SLD-style refutation of $\leftarrow G_0$ is a sequence of goal clauses $\leftarrow G_0 \cdots \leftarrow G_n$ where G_n is the *empty clause*, *i.e.*, the sequent $\mathbf{P}, \mathbf{D}_n \vdash ()$, where $\mathbf{D}_n$ is a database state, and $()$ denotes the empty formula. This sequent is an axiom of the inference system, and this axiom states that the empty formula is true on any database state. Each $\leftarrow G_{i+1}$ is obtained from $\leftarrow G_i$ by using the inference system later presented in this section.

Before presenting $\mathcal{CTR}$'s inference system, we deem relevant to also present the notion of *hot components* introduced in [1]. Summarily, hot components of a transaction ϕ, denoted $hot(\phi)$, is the set of transactions ready for execution in ϕ. Formally:

Definition 5 (Hot components) Let ϕ be a concurrent sequential goal. Its set of hot components, $hot(\phi)$, is defined recursively as follows:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- $hot(()) = \{\ \}$, where ( ) denotes the empty goal;
- $hot(b) = \{b\}$, if b is an atomic formula;
- $hot(\psi_1 \otimes \cdots \otimes \psi_n) = hot(\psi_1)$;
- $hot(\psi_1 \mid \cdots \mid \psi_n) = hot(\psi_1) \cup \cdots \cup hot(\psi_n)$;
- $hot(\odot\psi) = \{\odot\psi\}$

□

Definition 6 (CTR's Inference system) The inference system consists of one axiom and four inference rules.

Axiom: $\mathbf{P}, \mathbf{D} \vdash ()$, for any state $\mathbf{D}$

Inference rules: In rules 1-4, σ is a substitution, ψ and ψ' are concurrent sequential goals, and $hot(\psi) = a$.

1. *Applying rule definitions:* Suppose $b \leftarrow \beta$ is a rule in $\mathbf{P}$ whose variables have been renamed so that the rule shares no variables with ψ. If a and b unify with mgu σ, then
$$\frac{\mathbf{P}, \mathbf{D} \vdash (\exists)\, \psi'\sigma}{\mathbf{P}, \mathbf{D} \vdash (\exists)\, \psi}$$
where ψ' is obtained from ψ by replacing an element of a by β.

2. *Querying the database:* If $\mathcal{O}^d(\mathbf{D}_i) \models^c (\exists)a\sigma$, and $a\sigma$ and $\psi'\sigma$ share no variables, then
$$\frac{\mathbf{P}, \mathbf{D} \vdash (\exists)\, \psi'\sigma}{\mathbf{P}, \mathbf{D} \vdash (\exists)\, \psi}$$
where ψ' is obtained from ψ by deleting an element of a.

3. *Executing elementary updates:* If $\mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2) \models^c (\exists)a\sigma$, and $a\sigma$ and ψ' share no variables, then

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$\frac{\mathbf{P}, \mathbf{D}_2 \vdash (\exists)\, \psi'\sigma}{\mathbf{P}, \mathbf{D}_1 \vdash (\exists)\, \psi}$$

where ψ' is obtained from ψ by deleting an element of a.

4. *Executing atomic transactions:* If $\odot\alpha$ is the hot component in ψ, then

$$\frac{\mathbf{P}, \mathbf{D} \vdash (\exists)\, (\alpha \otimes \psi')}{\mathbf{P}, \mathbf{D} \vdash (\exists)\, \psi}$$

where ψ' is obtained from ψ by deleting an element of $\odot\alpha$.

□

Based on these inference rules, one can prove the execution sequence of both sequential executions and concurrent executions. Let us consider the following program.

$$\begin{array}{l} p \leftarrow a_1 \mid a_2 \\ a_1 \leftarrow ins(a) \\ a_2 \leftarrow ins(b) \end{array} \tag{2.2}$$

The deduction of the transaction p is illustrated in Table 2.1.

Table 2.1: A deduction for program (2.2)

Sequents	Inference rule	Hot components
$\mathbf{P}, \{\} \vdash p$	1	{p}
$\mathbf{P}, \{\} \vdash a_1 \mid a_2$	1	$\{a_1, a_2\}$
$\mathbf{P}, \{\} \vdash ins(a) \mid a_2$	3	$\{ins(a), a_2\}$
$\mathbf{P}, \{a\} \vdash a_2$	1	$\{a_2\}$
$\mathbf{P}, \{a\} \vdash ins(b)$	3	$\{ins(b)\}$
$\mathbf{P}, \{a, b\} \vdash ()$	*Axiom*	$\{\,\}$

Each sequent in the table is derived from the one below by an inference rule. The deduction succeeds because the bottom-most sequent is an axiom. Carried out top-down, the deduction corresponds to an execution of the transaction p in which atoms a and b are inserted into the empty database in the order, first a, then b.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 3

The $\mathcal{CTR}$ Prototype

THE $\mathcal{CTR}$ prototype [13, 17] is an implementation of the Horn fragment of $\mathcal{CTR}$ and $\mathcal{CTR}$'s inference system.

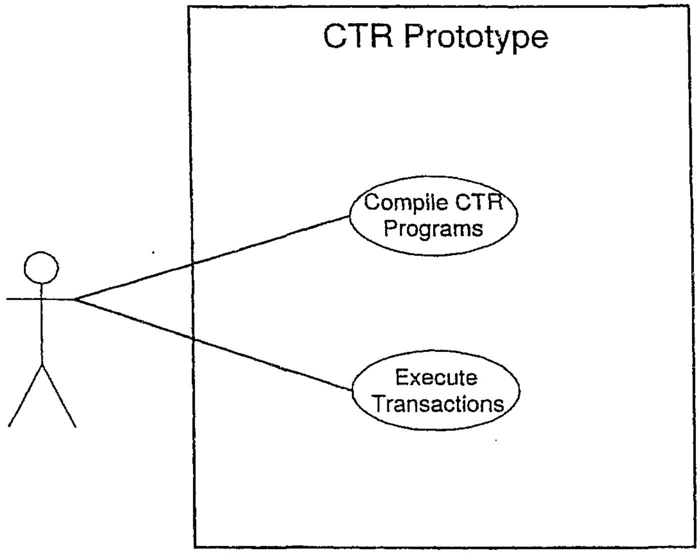
To facilitate the understanding of the $\mathcal{TP}$-$\mathcal{CTR}$ prototype later introduced in Chapter 5, in this chapter we outline the $\mathcal{CTR}$ prototype by presenting a system view of the software application available in [17]. This chapter is organized as follows. First it presents a system description of the $\mathcal{CTR}$ prototype, describing its major components. Then it shows how the prototype handles sequential execution, concurrent execution, and database updates and queries.

3.1 System Description of the $\mathcal{CTR}$ Prototype

Figure 3.1 shows a *use case* diagram of the $\mathcal{CTR}$ prototype. As such, it shows the system functionalities seen from the user's viewpoint. These functionalities are two: to *compile* a $\mathcal{CTR}$ program into an internal program; and to *execute* transactions based on the translated internal program.

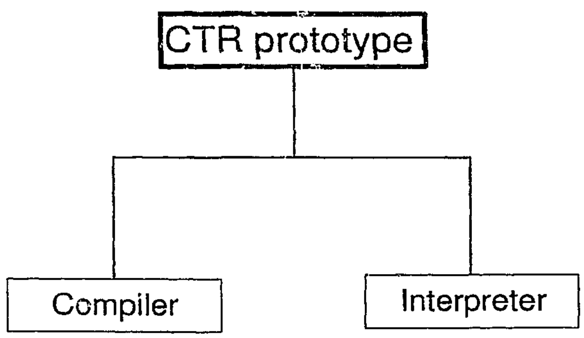
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Figure 3.1: Use case diagram for the $\mathcal{CTR}$ system



To handle these functions, the $\mathcal{CTR}$ prototype system can be seen as consisting of a compiler and an interpreter, as shown in Figure 3.2 .

Figure 3.2: The $\mathcal{CTR}$ prototype



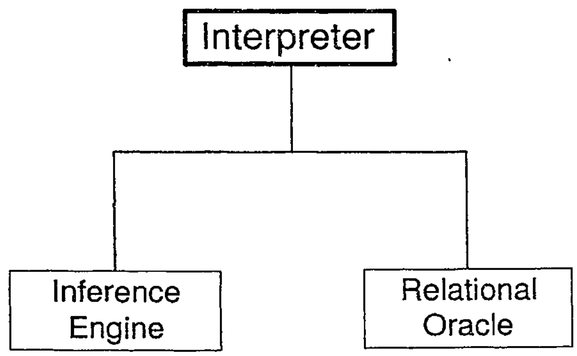
The role of the $\mathcal{CTR}$ compiler introduced in [17] is to translate a $\mathcal{CTR}$ program into an internal format recognizable to the $\mathcal{CTR}$ interpreter. The $\mathcal{CTR}$ interpreter is an

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

implementation of the $\mathcal{CTR}$ inference engine, which is used to prove transactions based on the relational oracle and the translated $\mathcal{CTR}$ programs.

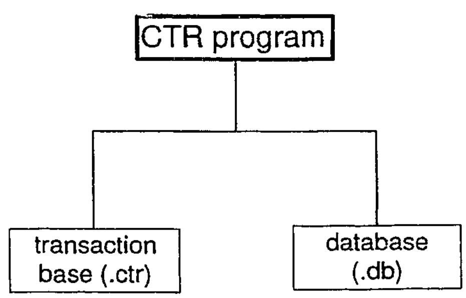
Figure 3.3 shows the two most important components of the $\mathcal{CTR}$ interpreter: the inference engine and the relational oracle.

Figure 3.3: The $\mathcal{CTR}$ Interpreter



In the $\mathcal{CTR}$ prototype, a $\mathcal{CTR}$ program consists of a *transaction base* file and a *database* file, as shown in figure 3.4.

Figure 3.4: The $\mathcal{CTR}$ program files



Transaction-base files have the extension *.ctr* and database files have the extension *.db*. A transaction base is a program consisting of a set of concurrent Horn rules. Table

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.1 illustrates the notations used in the $\mathcal{CTR}$ prototype and the corresponding notations used in $\mathcal{CTR}$.

Table 3.1: Corresponding notations used in the $\mathcal{CTR}$ prototype and $\mathcal{CTR}$

$\mathcal{CTR}$ Prototype	$\mathcal{CTR}$	Meaning
:-	$\leftarrow$	*Logical implication*
*	$\otimes$	*Sequential conjunction*
#	$\mid$	*Concurrent conjunction*
o	$\odot$	*Modality of isolation*

A $\mathcal{CTR}$ prototype database is a Prolog rulebase, consisting of *database rules* and *database tuple declarations.* A database rule is a normal Prolog rule, which has one of the following forms:

- an atomic formula; or
- ϕ :- α, where ϕ is an atomic formula and α is a clause.

There are two main reasons for putting rules in a $\mathcal{CTR}$ database:

1. to update them (only those rules in the $\mathcal{CTR}$ database can be changed at runtime).
2. to express queries with negation-as-failure.

To illustrate, below we show an excerpt of a $\mathcal{CTR}$ database:

finished :- not(unfinished).
unfinished :- a(X).
$a(100).\ a(99).\ \cdots\ a(1).\ a(0).$

The first and second rules specify a query with negation as failure. The atomic formulas $a(\_)$ specify which atoms are true in the initial database state. To update a database atom during execution, it must be declared as an updatable database tuple. *Database tuple declarations* are of the form

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

updatable R/n.

For instance, considering the database formulas above, the database tuple declaration below specifies that the tuple a with arity one, is updatable.

updatable a/1.

The example below illustrates the functionality of *declarations* and *ground atomic formulas.*

Example 3 (A simple book order application) The $\mathcal{CTR}$ program below exemplifies a simple book order system. The book inventory is represented by the tuple $book/1$ in the database.

- The transaction-base file *book.ctr*:
  $order(X)$:- $o(book(X) * del(book(X))) * writeln(succeeded)$.
  $order(X)$:- $writeln(failed)$.
- The database file *book.db*:
  updatable $book/1$.
  $book(1)$.
  $book(2)$.

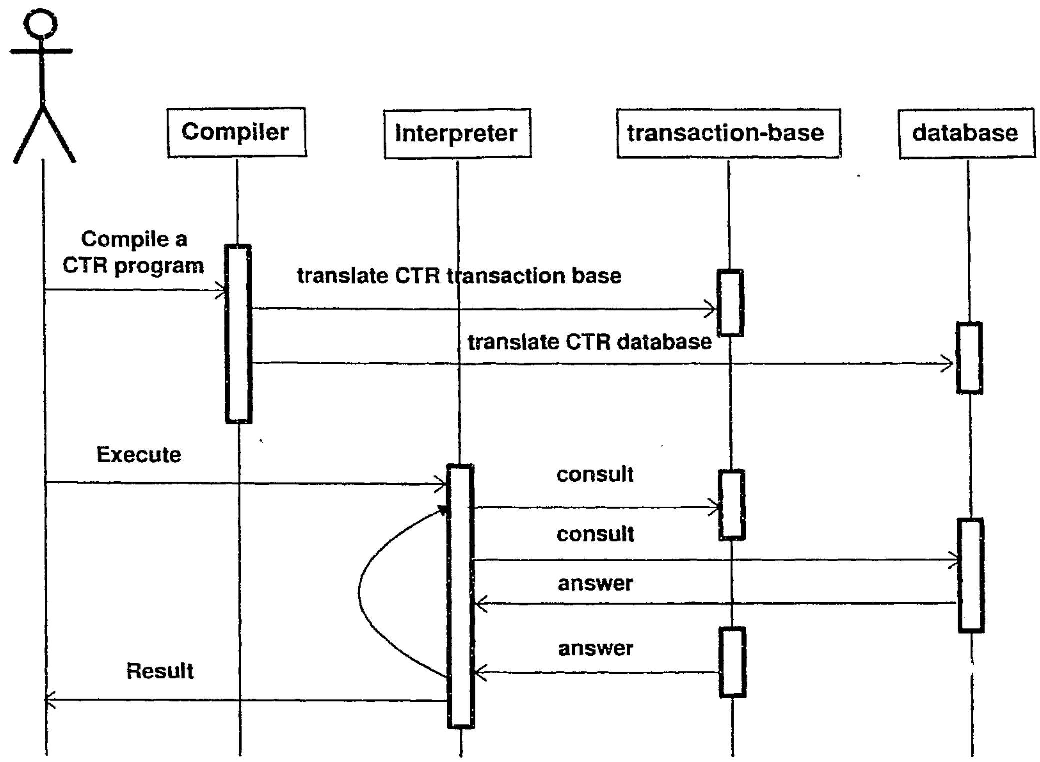
The transaction base specifies how orders of books are placed. The database defines an updatable tuple $book/1$, and includes two initial instantiations of the tuple $book$, $book(1)$ and $book(2)$. □

Figure 3.5 shows a use case scenario describing how the user interacts with the $\mathcal{CTR}$ prototype to compile and interpret a $\mathcal{CTR}$ program:

- The user compiles a $\mathcal{CTR}$ program.
  - The $\mathcal{CTR}$ prototype compiler translates the Horn rules in the $\mathcal{CTR}$ program's transaction base.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Figure 3.5: $\mathcal{CTR}$ prototype use case scenario



- The $\mathcal{CTR}$ prototype compiler translates the rules in the $\mathcal{CTR}$ program's database.

- When the user executes a transaction, the $\mathcal{CTR}$ prototype interpreter executes the transaction with inference engine. In each resolution step, the $\mathcal{CTR}$ prototype interpreter communicates with the transaction base and the database according to the executed transaction component in the following ways:
  - Consulting the transaction base.
  - Consulting the database.
  - Updating the database.
  - The $\mathcal{CTR}$ interpreter outputs the result to the user.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.2 Executing $\mathcal{CTR}$ Prototype Formulas

The $\mathcal{CTR}$ prototype interpreter uses a *Round-Robin scheduling algorithm* to schedule multi-processes in concurrent transaction executions. This algorithm assumes that each transaction process in a concurrent execution has the same priority level. Thus, in the $\mathcal{CTR}$ prototype, these concurrent processes have the same opportunity to execute.

In the $\mathcal{CTR}$ prototype, the *hot component set* introduced in Chapter 2 is implemented as a Prolog list. The transaction in the head of the list is selected for execution. If this transaction is defined as a rule in the transaction base, then the transaction formula in the body of the rule replaces the transaction, and then is inserted at the end of the list. When a transaction is selected, the $\mathcal{CTR}$ prototype interpreter executes one element of the transaction, and then feeds the remaining elements of the transaction back into the end of the list. This is consistent with the $\mathcal{CTR}$ inference system presented in Chapter 2. In this way, the concurrent sequential transactions in the queue are able to execute in equal turn.

In the rest of this section, we analyze sequential executions and concurrent executions in the $\mathcal{CTR}$ prototype respectively.

3.2.1 Sequential Execution

Sequential execution is the simpler type of execution, where transactions are connected with sequential conjunction operators. These transactions are executed in sequential order from left to right.

When executing a sequential conjunction, such as $p_1 * p_2 * ... * p_n$, the $\mathcal{CTR}$ prototype interpreter starts the execution from p_1, then p_2, and repeats until p_n is completed. If any given transaction p_i fails, the whole transaction will roll back.

Also, any transaction p_i could have been defined by rules. For instance, p_1 could have been defined as another sequential conjunction, i.e., p_i :- $a * b$. During execution, the $\mathcal{CTR}$ prototype interpreter would decompose any composite operation, e.g., replacing p_1

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

with $a * b$.

3.2.2 Concurrent Execution

The prototype simulates concurrency by interleaving the execution of concurrent processes with a Round-Robin scheduling algorithm. Consider the concurrent transaction execution:

$task$:- $a_1 * a_2 * ... * a_n \# b_1 * b_2 * b_3 * ... * b_n$.

To run this transaction, first, the Round-Robin algorithm picks up the first element from the first concurrent sub-transaction, a_1, and executes it, then followed by the first element from the other concurrent sub-transaction, b_1. After that, the Round-robin algorithm continues to pick up elements from the two concurrent sub-transactions in the same fashion and execute them until all the elements in the two sub-transactions are executed. The resulting execution sequence is: $a_1, b_1, a_2, b_2, ..., a_n, b_n$.

This applies for transactions consisting of more than two concurrent transactions, e.g.,

$task$:- $a_1 * a_2 \# b_1 * b_2 \# c_1 * c_2$.

The resulting execution sequence is $a_1, b_1, c_1, a_2, b_2, c_2$. And for transactions defined as concurrent rules, e.g.,

$task$:- $a_1 * \alpha \# \beta * b_3$.

α :- $a_2 * a_3$.

β :- $b_1 * b_2$.

After replacing transactions α and β with their respective definitions, the resulting execution sequence is :

$task$:- $a_1 * a_2 * a_3 \# b_1 * b_2 * b_3$.

In this way, a composite transaction can be decomposed to an execution sequence consisting of elementary operations.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.3 State Updates and Queries

In $\mathcal{CTR}$, the semantics of database states queries and updates are defined by a pair of relational oracles, called, state data oracle and state transition oracle. The $\mathcal{CTR}$ prototype uses the following predicates to implement the state queries and updates defined in the oracles.

- $db(p)$ determines whether atom p is true in the current database state.
- $ins(p)$ inserts atom p in the database.
- $del(p)$ deletes atom p in the database.

To be able to preserve the database consistency when a transaction that updates the database fails, the $\mathcal{CTR}$ prototype uses a backtracking mechanism for undoing updates, i.e., to rollback the failed execution and enable re-execution. It performs a *relative commit* of the updates, as opposed to an *absolute commit*, which commits only after the entire transaction succeeds. Considering the following transaction:

$$ins(a) * db(b) * ins(c)$$

Assume atom a and c are not in the initial database. The first update $ins(a)$ inserts the atom a into the database. Then, the second operation queries the database. If the atom b exists in the database, then the third operation, $ins(c)$, will take place and thus the three atoms are all in the final database. However, if the atom b is not in the database, then the transaction execution fails, and the update $ins(a)$ is undone. In this case, although the update performed by $ins(a)$ has been already executed, the database is able to restore to its initial state and the database consistency is thus guaranteed. The undo is possible because the commit was not absolute, but was relative to the overall transaction.

To undo updates in a failed transaction, the $\mathcal{CTR}$ prototype actually introduces two underlying atom tags to represent the existence of an updatable atom p, namely, *inserted*(p) and *deleted*(p).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The atom p is true in the database if *inserted*(p) is true and *deleted*(p) is not true in the database. The atom p is not true in the database if *deleted*(p) is true or *inserted*(p) is not true.

By implementing these two atom tags, the technique in [17] leads to transaction rollback or partial rollback, and re-execution when a failure occurs, and thus prevents the Prolog database from being corrupted by backtracking through updates.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 4

$\mathcal{TP}$-$\mathcal{CTR}$

ALTHOUGH $\mathcal{CTR}$ is a logic for specifying and executing database processes involving queries and updates concurrently as well as serially, $\mathcal{CTR}$ does not support explicit priority constraints in its logic framework. In $\mathcal{CTR}$, the *programmer* [1] uses the sequential and concurrent connectives to specify the ordering of a transaction execution. In a concurrent conjunction, there is no preference over which transaction will be selected for execution. Hence, $\mathcal{CTR}$'s orientation is towards logic *eventuality* rather than real-time *immediacy*. This particular feature of the logic has limited its application extended to real-time domains. Also, $\mathcal{CTR}$ can not handle timing-related applications due to lack of timing constraints, e.g., a timing relation to specify when to execute a process.

To overcome this limitation and extend concurrent transaction logic to the real-time domain, we extend the $\mathcal{CTR}$ logic framework to a so-called timing-event-based prioritized concurrent transaction logic ($\mathcal{TP}$-$\mathcal{CTR}$), which provides a high-level formalism for specifying priority constraints and timing constraints in timing-event-based transaction applications. Summarily, $\mathcal{TP}$-$\mathcal{CTR}$ extends $\mathcal{CTR}$ in the sense that it introduces *constraints* in concurrent Horn rules, *a translation mechanism* to translate such rules, and *an inference system* to handle prioritized transactions.

In this chapter, we describe the $\mathcal{TP}$-$\mathcal{CTR}$'s syntax, informal semantics, the inference system and the interpretation of constraint concurrent Horn rules.

[1] Like $\mathcal{CTR}$, $\mathcal{TP}$-$\mathcal{CTR}$ is a logic for programming transactions. Hence in this thesis, *users* and *programmers* are considered synonyms

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.1 Syntax of $\mathcal{TP}$-$\mathcal{CTR}$ Priority and Timing Constraints

$\mathcal{CTR}$ does not include priority constraints in its logic framework. In $\mathcal{CTR}$, the programmer uses the sequential and concurrent connectives to specify the transaction execution order in programs. For instance, in a concurrent conjunction, all the concurrent processes have the same opportunity to be selected for execution. This feature of the logic has limited its application in real-time domains.

$\mathcal{TP}$-$\mathcal{CTR}$, an extension of $\mathcal{CTR}$, is designed to provide a high-level framework for specifying priority constraints and timing constraints in concurrent transaction logic systems. To allow the specification of priority constraints and timing constraints that may be used to trigger or interrupt other transactions, we use *constraint concurrent Horn rules.* In $\mathcal{TP}$-$\mathcal{CTR}$, $b \leftarrow \phi : \psi$ is a constraint concurrent Horn rule if $b \leftarrow \phi$ is a concurrent Horn rule and ψ is a *constraint formula.* A $\mathcal{TP}$-$\mathcal{CTR}$ *program* is composed of a set of constraint concurrent Horn rules. The definition of the concurrent Horn rule $b \leftarrow \phi$ here is the same as that in $\mathcal{CTR}$, as shown in definition 1. Below is the definition of the constraint formula ψ.

***Definition 7* (Constraint formula)** A Constraint formula is any formula of the form:

- $priority(\alpha, p)$, where α is an atomic formula in ϕ and p is an integer number, representing the priority level of the atomic formula α.
- $timeElapsed(t) \rightarrow triggerEvent(\alpha)$, where t is a positive integer number, and α is an atom in ϕ.
- $\psi_1 \wedge \cdots \wedge \psi_k$, where each ψ_i is a constraint formula, and $K \geq 0$.

$\square$

With the above extended syntax definition, we can use *constraint concurrent Horn rules* to specify priority and timing constraints, which may be used to trigger or interrupt other transactions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

When a constraint concurrent Horn rule has no constraint formula, it becomes a concurrent Horn rule. A $\mathcal{TP}$-$\mathcal{CTR}$ *program* consists of both a set of concurrent Horn rules and a set of constraint concurrent Horn rules.

4.2 Informal Semantics of $\mathcal{TP}$-$\mathcal{CTR}$

Next we introduce the informal semantics of constraint concurrent Horn rules from the users' viewpoint.

Priority constraints

In $\mathcal{TP}$-$\mathcal{CTR}$, priority constraints specify the execution priority of transactions occurring in a concurrent goal formula. For instance, the constraint concurrent Horn rule below specifies the execution priorities of the concurrent transactions p, q, and r.

$$s \leftarrow p \mid q \mid r : priority(p, 2) \wedge priority(q, 1) \wedge priority(r, 1) \tag{4.1}$$

> "To execute s, execute p, q, and r concurrently, observing that p has higher priority than q and r, and q has the same priority as r."

Notice that the second argument of the priority predicate represents the execution priority level of the predicate occurring in its first argument. The bigger the number representing the priority level, the higher its execution priority is. Priority levels ranges from 1 to 100. The default priority level of a transaction predicate is 1, i.e., if a constraint concurrent Horn rule does not specify the priority of a transaction predicate explicitly, then it is given the lowest execution priority.

But then one might ask: why use the complicated formula above when one can more succinctly write the equivalent and more straightforward formula below?

$$s \leftarrow p \otimes (q \mid r)$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This is because priority constraints like the ones in Expression (4.1) are useful specially when combined with timing constraints. The examples in the rest of this section illustrate the point.

Timing Constraints

Timing properties are usually specified by the timing relations among events. In [5], the timing relations refer to terms of time that specify constraints among events. We are interested in these timing properties and adopt some of them into our system . Below we illustrate some examples involving timing constraints.

Let b, c, and d be transactions. The rule below specifies a timing constraint for b:

$$a \leftarrow b \mid c \mid d : timeElapsed(100) \rightarrow triggerEvent(b) \tag{4.2}$$

The rule above specifies the following: to execute a, execute b, c, and d concurrently, but delaying the execution of b 100 seconds. That is, for the first 100 seconds, only c and d should run concurrently, and then after 100 seconds, transaction b should be triggered, and thus added to the concurrent execution.

The example below illustrates how priority and timing constraints can be combined to provide interesting real-time *interrupt* behavior.

Let again b, c, and d denote transactions. The constraint concurrent Horn rule below, specifies timing and priority constraints for transaction b.

$$a \leftarrow b \mid c \mid d : timeElapsed(100) \rightarrow triggerEvent(b) \wedge priority(b, 2)$$

The priority constraint specifies that b has execution priority over c and d. Notice that no constraint is specified for c or d, i.e., they run as a flat concurrency (with the default priority level 1). The timing constraint specifies that the execution of b should start 100 seconds after a is activated.

Let's assume that c and d represent complex transactions whose respective execution times last for more than 200 seconds time. Given these assumptions, since b has high

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

execution priority, the activation of b should interrupt the execution of c and d. That is to say, after 100 seconds, b starts execution, and c and d are suspended because of their lower priority level. Until b finishes its execution, the execution of c and d can be resumed. This behavior of concurrent transaction execution, created by the combined use of timing and priority constraints, illustrates the interrupt functionality. This is what often happens in a real-time domain scenarios, and the reason why we introduce the priority constraint into the extended system.

Some comments on the use of constraints in $\mathcal{TP}$-$\mathcal{CTR}$: one may have noticed that priority constraints are not suitable for all transactions in a concurrent sequential goal. For example, suppose a, b, and c are transactions, and r is defined by the rule below:

$$r \leftarrow a \otimes b \otimes c$$

It does not make sense to specify a priority for the transactions occurring in the body of the rule above, since the semantics of the $\otimes$ connective specifies the execution order of the goal $a \otimes b \otimes c$. Priority constraints only make sense when they refer to transactions occurring in a concurrent conjunction, since it can change the execution sequence of the transactions involved in the conjunction.

4.3 Inference System

The inference system introduced here differs from the inference system introduced in $\mathcal{CTR}$ in the sense that in $\mathcal{CTR}$, there is no criteria for selecting which transaction in a concurrent goal will be executed. This means in $\mathcal{CTR}$, the execution order of concurrent transactions is non-deterministic. In $\mathcal{TP}$-$\mathcal{CTR}$, on the other hand, a simplified Rate Monotonic scheduling algorithm based on the priority level of a predicate, is employed as the selection criteria. Thus, the execution order of concurrent transactions with different priority levels is deterministic.

The next definition formalizes this idea. It defines the transaction to be executed amongst the candidates for execution. We refer to it as the "hottest" component of a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

concurrent sequential goal.

Definition 8 (Hottest component) Let ϕ be a concurrent sequential goal. Its hottest component, denoted $hottest(\phi)$, is defined recursively as follows:

- $hottest(()) = \{\}$, where $()$ is the empty goal;
- $hottest(b) = \{b\}$, if b is a atomic formula;
- $hottest(\psi_1 \otimes \cdots \otimes \psi_k) = hottest(\psi_1)$;
- $hottest(\psi_1 \mid \psi_2 \mid \cdots \mid \psi_k) = \begin{cases} hottest(\psi_1) & \text{, if } pLevel(hottest(\psi_1)) \geq \\ & \quad pLevel(hottest(\psi_2 \mid \cdots \mid \psi_k)) \\ hottest(\psi_2 \mid \cdots \mid \psi_k) & \text{, otherwise} \end{cases}$,
  where $pLevel(\phi)$ denotes the priority level of ϕ.
- $hottest(\odot\psi) = \odot\psi$.

□

Since the compiler adds an extra first parameter to all prioritized predicates and encodes in this parameter the priority level of the predicate, one can obtain the priority level of a sequential goal as follows:

- $pLevel(p(priority(l), \cdots)) = l$, if $p(\cdots)$ is an atomic formula;
- $pLevel(\psi_1 \otimes \cdots \otimes \psi_k) = pLevel(\psi_1)$

Like $\mathcal{CTR}$, $\mathcal{TP}$-$\mathcal{CTR}$ also uses the SLD-style resolution introduced in Chapter 2, the only difference between them being the inference rules and the notion of hot component used in $\mathcal{CTR}$, and hottest component used in $\mathcal{TP}$-$\mathcal{CTR}$. In $\mathcal{CTR}$, hot component is a set representing the sub-transactions ready for execution. In $\mathcal{TP}$-$\mathcal{CTR}$, hottest component is the first highest priority-level component of the $\mathcal{CTR}$ hot component set.

Axiom: $\mathbf{P}, \mathbf{D} \vdash ()$, for any state $\mathbf{D}$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Inference rules: In rules 1-3, σ is a substitution, ψ and ψ' are concurrent sequential goals, and $hottest(\psi) = a$.

1. *Applying rule definitions:* Suppose $b \leftarrow \beta$ is a rule in $\mathbf{P}$ whose variables have been renamed so that the rule shares no variables with ψ. If a and b unify with mgu σ, then

$$\frac{\mathbf{P}, \mathbf{D} \vdash (\exists)\, \psi'\sigma}{\mathbf{P}, \mathbf{D} \vdash (\exists)\, \psi}$$

where ψ' is obtained from ψ by replacing a by β.

2. *Querying the database:* If $\mathcal{O}^d(\mathbf{D}_i) \models^c (\exists) a\sigma$, and $a\sigma$ and $\psi'\sigma$ share no variables, then

$$\frac{\mathbf{P}, \mathbf{D} \vdash (\exists)\, \psi'\sigma}{\mathbf{P}, \mathbf{D} \vdash (\exists)\, \psi}$$

where ψ' is obtained from ψ by deleting a.

3. *Executing elementary updates:* If $\mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2) \models^c (\exists) a\sigma$, and $a\sigma$ and ψ' share no variables, then

$$\frac{\mathbf{P}, \mathbf{D}_2 \vdash (\exists)\, \psi'\sigma}{\mathbf{P}, \mathbf{D}_1 \vdash (\exists)\, \psi}$$

where ψ' is obtained from ψ by deleting a.

4. *Executing atomic transactions:* If $\odot\alpha$ is the hottest component in ψ, then

$$\frac{\mathbf{P}, \mathbf{D} \vdash (\exists)\, (\alpha \otimes \psi')}{\mathbf{P}, \mathbf{D} \vdash (\exists)\, \psi}$$

where ψ' is obtained from ψ by deleting $\odot\alpha$.

Each inference rule consists of two sequents, and has the following interpretation: if the upper sequent (G_{i+1}) can be inferred, then the lower sequent (G_i) can also be inferred.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.4 Interpreting Constraint Concurrent Horn Rules

Based on the constraint formalism presented in the above sections, here we initially present how the $\mathcal{TP}$-$\mathcal{CTR}$ interpreter translates constraint concurrent Horn rules into concurrent Horn rules. Then we show how to formally execute prioritized constraint concurrent Horn rules using the SLD-style resolution procedure. Finally we illustrate how the $\mathcal{TP}$-$\mathcal{CTR}$ compiler translates the concurrent Horn rules with timing constraints.

4.4.1 Compiling Priority Constraints

To illustrate how the translation takes place, we use the following example: let the program below define transactions s, p, and q.

$$\begin{array}{l} s \leftarrow p \mid q : priority(p, 2) \\ p \leftarrow ins(r(a)) \\ q \leftarrow ins(r(b)) \end{array} \tag{4.3}$$

Notice that the first rule specifies: to execute s, execute p and q concurrently. Its constraints specify: in the concurrent sequential goal $p \mid q$, p has higher priority than q. The second and third rules specify: to execute p, insert the atom $r(a)$ in the database; to execute q, insert the atom $r(b)$ in the database.

In essence, the compilation consists of translating the priority predicates occurring in the constraint formula into function terms. These function terms are added as an extra argument to the respective predicates in the head and body of the concurrent Horn rules.

Below we present the result of the translation of program (4.3).

$$\begin{array}{l} s \leftarrow p(priority(2)) \mid q(priority(1)) \\ p(priority(2)) \leftarrow ins(r(a)) \\ q(priority(1)) \leftarrow ins(r(b)) \end{array} \tag{4.4}$$

Notice that the compiler assigned the default priority level 1 to transaction q.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.4.2 Using the Inference System to Execute Transactions

With the $\mathcal{TP}$-$\mathcal{CTR}$ four inference rules, we can deduce the transaction s defined in program (4.3) as in the Table 4.1. Notice in program (4.3) and on Table 4.1 that if no priority constraints is used in the specification of s, then there are two possible sequence of database states when s executes: one in which $r(a)$ is inserted first and then $r(b)$ is inserted, and the other in which $r(b)$ is inserted first and then $r(b)$. However, because of the priority constraints bound to transaction p and q, they force the transaction s to run in a definite sequence, which causes $r(a)$ to be inserted before $r(b)$ is inserted in the database.

Table 4.1: A deduction for program (4.3)

Sequents	Inference rule	Hottest component
$\mathbf{P}, \{\} \vdash s$	1	s
$\mathbf{P}, \{\} \vdash p(priority(2)) \mid q(priority(1))$	1	$p(priority(2))$
$\mathbf{P}, \{\} \vdash ins(r(a)) \mid q(priority(1))$	3	$ins(r(a))$
$\mathbf{P}, \{r(a)\} \vdash q(priority(1))$	1	$q(priority(1))$
$\mathbf{P}, \{r(a)\} \vdash ins(r(b))$	3	$ins(r(b))$
$\mathbf{P}, \{r(a), r(b)\} \vdash ()$	*Axiom*	$\{\}$

4.4.3 Compiling Timing Constraints

The program below specifies a timing constraint for transaction b.

$$\begin{array}{l} a \leftarrow b \mid c \mid d : timeElapsed(100) \rightarrow triggerEvent(b) \\ b \leftarrow ins(r(e)) \\ c \leftarrow ins(r(f)) \\ d \leftarrow ins(r(g)) \end{array}$$

The $\mathcal{TP}$-$\mathcal{CTR}$ interpreter translates it into the following program

$$\begin{array}{l} a \leftarrow b \mid c \mid d \\ b \leftarrow timeElapsed(100) \otimes ins(r(e)) \\ c \leftarrow ins(r(f)) \\ d \leftarrow ins(r(g)) \end{array} \qquad (4.5)$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

where $timeElapsed(t)$ is a built-in predicate, which is satisfied when t seconds have elapsed since the execution of concurrent rule a has started. This predicate is interpreted by the $\mathcal{TP}$-$\mathcal{CTR}$ prototype interpreter in a specific way in the chapter 5. Here, we present the logical definition of the predicate *timeElapsed* below:

$$
\begin{aligned}
timeElapsed(DelayTime) \leftarrow \quad & \\
& startTime(InitialTime) \\
& \otimes now(PresentTime) \\
& \otimes (PresentTime \geq InitialTime + DelayTime) \\
timeElapsed(DelayTime) \leftarrow \quad & \\
& startTime(InitialTime) \\
& \otimes now(PresentTime) \\
& \otimes (PresentTime < InitialTime + DelayTime) \\
& \otimes timeElapsed(DelayTime)
\end{aligned}
$$

In the above definition, you can notice that if pre-condition *(PresentTime $\geq$ InitialTime + DelayTime)* fails, then *timeElapsed(DelayTime)* stays in a sort of while-loop until the condition (*PresentTime $\geq$ InitialTime + DelayTime*) is satisfied, and the corresponding transaction is then triggered.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 5

The $\mathcal{TP}$-$\mathcal{CTR}$ Prototype

THE $\mathcal{TP}$-$\mathcal{CTR}$ prototype is an implementation of the Horn fragment of $\mathcal{TP}$-$\mathcal{CTR}$ and the inference system, both introduced in Chapter 4. In Chapter 3 we have presented the outline of the $\mathcal{CTR}$ prototype. Since the $\mathcal{TP}$-$\mathcal{CTR}$ prototype is an extension of the $\mathcal{CTR}$ prototype, here we focus on their differences.

As the $\mathcal{CTR}$ prototype shown in Figure 3.2, the $\mathcal{TP}$-$\mathcal{CTR}$ prototype also has two major components: the compiler and the interpreter. Both have been extended to handle the timing and priority constraints introduced in $\mathcal{TP}$-$\mathcal{CTR}$. In this chapter, we first present how the $\mathcal{TP}$-$\mathcal{CTR}$ prototype compiler translates a $\mathcal{TP}$-$\mathcal{CTR}$ program. Then we show how the $\mathcal{TP}$-$\mathcal{CTR}$ prototype interpreter works.

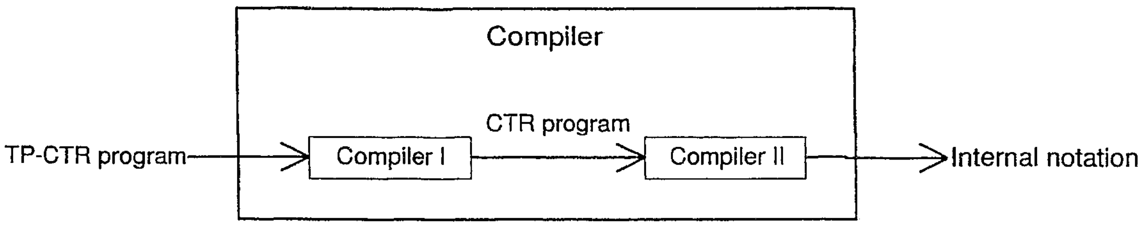
5.1 The $\mathcal{TP}$-$\mathcal{CTR}$ Prototype Compiler

Before executing a $\mathcal{TP}$-$\mathcal{CTR}$ program, a user first has to compile the transaction base and database files. In a $\mathcal{TP}$-$\mathcal{CTR}$ program, priority constraints and timing constraints are, in fact, a syntax sugar, which we call *constraint concurrent Horn rules.* Unlike the $\mathcal{CTR}$ prototype, the $\mathcal{TP}$-$\mathcal{CTR}$ prototype compiler first translates these constraint concurrent Horn formulas into $\mathcal{CTR}$ recognizable formulas. Figure 5.1 shows how this process takes place.

This translation consists of two stages, carried on by compiler I and compiler II,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Figure 5.1: The $\mathcal{TP}$-$\mathcal{CTR}$ compiler.



respectively, as illustrated in Figure 5.1. Compiler I translates the constraint concurrent Horn rules in a $\mathcal{TP}$-$\mathcal{CTR}$ program into an intermediate $\mathcal{CTR}$ program. Then compiler II translates this intermediate program into an internal notation format.

The reason why the $\mathcal{TP}$-$\mathcal{CTR}$ prototype compiler translates a $\mathcal{TP}$-$\mathcal{CTR}$ program into a $\mathcal{CTR}$ recognizable program instead of using $\mathcal{CTR}$ recognizable program at the beginning is because the translated $\mathcal{CTR}$ recognizable syntax looks clumsy, unlike a typical neat Prolog-like syntax. For example, assigning every transaction a priority level and distributing timing constraints to the triggered transactions. To make formulas neat and clear to users, the $\mathcal{TP}$-$\mathcal{CTR}$ prototype puts all these constraints in the syntax sugar format and leaves the compiler to do the translation. Moreover, compiler I takes care of this part of the translation, only. Hence, introducing more complicated timing constraints and priority constraints into the system would require only extending compiler I. This improves the flexibility of the $\mathcal{TP}$-$\mathcal{CTR}$ prototype compiler.

The example below illustrates the idea of compiler I. Suppose we have the following rules in the transaction base:

$task$:- $initTime * taskA : timeElapsed(10) \longrightarrow triggerEvent(taskA).$

$taskA$:- $monitor('taskA\ complete').$

Compiler I creates a temporary transaction-base file, which is added an extension *.temp* to the end of the old transaction-base file. Below are the corresponding translated rules:

$task$:- $initTime * taskA.$

$taskA$:- $timeElapsed(10) * monitor('taskA\ complete').$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

If a rule in the original transaction base does not have a constraint, the translated rule is the same as the original one.

The compiling process carried on by compiler II is more elaborate, and thus deserves a more detailed analysis of the processes and notations used within. The next two sections provide this analysis.

5.1.1 Transaction Base Internal Syntax

Like a $\mathcal{CTR}$ program, a $\mathcal{TP}$-$\mathcal{CTR}$ program consists of a transaction-base and a database too. We saw in Section 5.1 that ultimately a $\mathcal{TP}$-$\mathcal{CTR}$ program is translated into an internal notation format. This subsection introduces the internal notation format.

The transaction base of a $\mathcal{TP}$-$\mathcal{CTR}$ program has two kinds of components: atomic formulas and rules. Atomic formulas do not change after translation.

On the other hand, constraint concurrent Horn rules in transaction-base are in the form of α :- β : γ, where α is an *atomic formula*, β is a *transaction formula*, and γ is a *constraint formula*, which can include both priority constraints and timing constraints. The head α is translated into the internal notation $trans(\alpha)$. Transaction formula notations involving sequential conjunction and concurrent conjunction, are translated into internal notations as follows:

- Sequential conjunction: $seq([a_1, a_2, ..., a_n])$ is the internal representation of transaction formula $a_1 * a_2 * ... * a_n$.
- Concurrent conjunction: $conc([a_1, a_2, ..., a_n])$ is the internal representation of the transaction formula $a_1 \# a_2 \# ... \# a_n$.
- Modality operator: $isolate(\theta)$ is the internal representation of $o(\theta)$.

Constraint formula notations involving priority and timing constraints are translated as follows:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Priority constraint: $a(priority(Level))$ is the internal representation of the transaction a with the priority constraint $priority(Level, a)$.

- Timing constraint:
$$seq([timeElapsed(a, Time), or([seq([isolate(seq([event(a, Time), now(X),$$
$$db(start\_time(Y)), X >= Time + Y, del(event(a, Time))]))$$
$$, b]), seq([event(a, Time), a])])])$$
is the internal representation of the rule body of a :- b, where a is triggered by the timing constraint $timeElapsed(Time) \rightarrow triggerEvent(a)$. Note: the predicate *timeElapsed(Time)* in the translated intermediate file is replaced by this internal notations with the $\mathcal{TP}$-$\mathcal{CTR}$ prototype interpreter recognizable built-in predicate $timeElapsed(a, Time)$.

The next examples illustrate these translation methods of internal syntax.

Example 4 (Translating sequential and concurrent goals) Suppose a goal connected with sequential and concurrent conjunctions is as the form of:

p :- $a * b \# c$.

Like the $\mathcal{CTR}$ compiler, $\mathcal{TP}$-$\mathcal{CTR}$ compiler translates it into the following internal syntax:

$trans(p)$:- $conc([seq([a, b]), c])$. □

If a rule has priority constraints, it is translated as shown in the example below:

Example 5 (Translating priority constraints) Suppose in a transaction-base, there is the following simple priority constraint concurrent Horn rule:

p :- $a \# b$: $priority(a, 8)$.

which has the following intermediate and internal representations during and after compilation:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

p :- $a(priority(8))$ # b. - Intermediate syntax, i.e., equivalent $\mathcal{CTR}$ rule
$trans(p)$:- $conc([a(priority(8)), b])$. - $\mathcal{TP}$-$\mathcal{CTR}$ internal syntax

□

The following example shows the original, intermediate and internal syntax in the case of a timing constraint concurrent Horn rule.

Example 6 (Translation of timing constraint rules) The rules below specify the concurrent execution transactions a and b. For the sake of easy understanding, the two transactions are just assigned simple tasks. The transaction formula a is scheduled to execute ten seconds after the execution begins.

p :- $initTime\ *\ (a\#b)\ : timeElapsed(10) \longrightarrow triggerEvent(a)$.

a :- $monitor('transaction\ a\ completed')$.

b :- $monitor('transaction\ b\ completed')$.

As the first step, the $\mathcal{TP}$-$\mathcal{CTR}$ compiler I translates the $\mathcal{TP}$-$\mathcal{CTR}$ program above into the following $\mathcal{CTR}$ program:

p :- $initTime\ *\ (a\#b)$.

a :- $timeElapsed(10)\ *\ monitor('transaction\ a\ completed')$.

b :- $monitor('transaction\ b\ completed')$.

Then the $\mathcal{TP}$-$\mathcal{CTR}$ compiler II translates the $\mathcal{CTR}$ program into an internal notation.

$trans(p)$:- $seq([initTime,\ conc([a,\ b])])$.

$trans(a)$:- $seq([timeElapsed(a, 10), or([seq([isolate(seq([event(a, 10), now(X),$
$db(start_time(Y)), X >= 10 + Y, del(event(a, 10))])),$
$monitor('transaction\ a\ completed')]), seq([event(a, 10), a])])])$.

$trans(b)$:- $monitor('transaction\ b\ completed')$.

□

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

When a $\mathcal{TP}$-$\mathcal{CTR}$ program combines timing constraints and priority constraints, it is translated in the way as shown in the example below.

Example 7 (Translating priority and timing constraint) Suppose we add a constraint to the rule p shown in Example 6, resulting in the following rule:

$$p \text{ :- } initTime \ * \ (a\#b) \ : timeElapsed(10) \rightarrow triggerEvent(a)$$
$$\wedge\ priority(a, 8).$$

First, the $\mathcal{TP}$-$\mathcal{CTR}$ compiler I translates them into the following $\mathcal{CTR}$ program:

p :- $initTime \ * \ (a(priority(8))\#b)$.

$a(priority(Z)) \ :- \ timeElapsed(10) \ * \ monitor('transaction\ a\ completed')$.

b :- $monitor('transaction\ b\ completed')$.

Then the $\mathcal{TP}$-$\mathcal{CTR}$ compiler II translates the $\mathcal{CTR}$ program above into the following internal notation:

$trans(p)$:- $seq([initTime,\ conc([a(priority(8)),\ b])])$.

$trans(a(priority(Z)))$:- $seq([timeElapsed(a(priority(Z)), 10), or([seq([isolate(seq([$
$event(a(priority(Z)), 10), now(X), db(start\_time(Y)), X >= 10 + Y,$
$del(event(a(priority(Z)), 10))])), monitor('transactionacompleted')])$
$, seq([event(a(priority(Z)), 10), a(priority(Z))])])])$.

$trans(b)$:- $monitor('transaction\ b\ completed')$.

□

By recursively applying the translation rules above, the $\mathcal{TP}$-$\mathcal{CTR}$ compiler translates a transaction-base file into a $\mathcal{TP}$-$\mathcal{CTR}$ transaction-base object file.

5.1.2 The $\mathcal{TP}$-$\mathcal{CTR}$ Prototype Database

The syntax and semantics of the $\mathcal{TP}$-$\mathcal{CTR}$ prototype database is the same as in the $\mathcal{CTR}$ prototype database. The only difference is that internally, the system uses a couple of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

other system predicates. In the $\mathcal{TP}$-$\mathcal{CTR}$ prototype databases, there are three system-required database tuple declarations in $\mathcal{TP}$-$\mathcal{CTR}$ prototype databases, namely:

- *updatable event/2.* - the declaration of timing-event-to-be-triggered list
- *updatable priorityList*/1. - the declaration of concurrent transaction priority list
- *updatable start_time*/1. - the declaration of execution start time

5.2 The $\mathcal{TP}$-$\mathcal{CTR}$ Prototype Interpreter

The $\mathcal{TP}$-$\mathcal{CTR}$ prototype interpreter consists of inference engine and relational oracle, as the $\mathcal{CTR}$ prototype interpreter shown in Figure 3.3. Although both the $\mathcal{CTR}$ prototype and the $\mathcal{TP}$-$\mathcal{CTR}$ prototype use the same relational oracle, the $\mathcal{TP}$-$\mathcal{CTR}$ prototype interpreter use a different inference engine. Compared to the $\mathcal{CTR}$ inference engine, it implements the notion of *hottest component* instead of *hot component*, as shown in Chapter 4. During execution, the $\mathcal{TP}$-$\mathcal{CTR}$ inference engine picks up the *hottest* transaction component, i.e., the first highest-priority-level *element* in the *hot component* queue.

The interpreter is the key module of the $\mathcal{TP}$-$\mathcal{CTR}$ prototype. The user interacts with the prototype by presenting transaction goals to be executed, *execute*(*Goal*). The interpreter then picks up the *hottest component* of the goal for execution. If the program in the transaction base does not include any constraints, the priority level of all transactions is considered as the default lowest priority level. In this case, the notion of *hottest component* introduced in $\mathcal{TP}$-$\mathcal{CTR}$ system is analogous to the notion of *hot component* introduced in $\mathcal{CTR}$ system because the underlying simplified Rate-Monotonic algorithm handles the non-priority-constraint goals in the same way as that of the Round-Robin algorithm.

The inference engine of $\mathcal{TP}$-$\mathcal{CTR}$ uses the underlying scheduling algorithm to pick up the *hottest component* in a concurrent transaction execution. It also interprets the priority and timing constraints and drives the execution of the transactions. The rest of this

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ection is organized as follows: Next we introduce the adopted underlying scheduling algorithm - *simplified Rate Monotonic Algorithm*($SRMA$)[10]. Then we present how priority constraints are interpreted, and how the $SRMA$ schedules the concurrent transactions. Finally, we present how the $\mathcal{TP}$-$\mathcal{CTR}$ timing constraints are interpreted.

5.2.1 Underlying Scheduling Algorithm in the $\mathcal{TP}$-$\mathcal{CTR}$ Interpreter

Before introducing the underlying scheduling algorithm in the $\mathcal{TP}$-$\mathcal{CTR}$ interpreter, we first give a brief introduction of the Round-Robin scheduling algorithm used in the $\mathcal{CTR}$ interpreter.

Round-Robin Scheduling Algorithm

The Round-Robin algorithm is one of the oldest, simplest and most widely used scheduling algorithms, designed especially for time-sharing systems. It assigns each concurrent process an unit of time in a one-by-one sequence and feeds the process that has used up its share of time back to the end of the sequence. In $\mathcal{CTR}$, Bonner made a small adjustment: using one step transaction execution instead of one unit of cpu time, to simplify transaction execution in a concurrent sequence. Thus, the Round Robin scheduling algorithm in the $\mathcal{CTR}$ system assigns one execution step time to the selected transaction, which is in the head of the concurrent transaction sequence. In $\mathcal{CTR}$, one execution step means one inference step. In this way, the $\mathcal{CTR}$ scheduler picks up the first hot component, executes one inference step, feeds the unfinished transaction component back to the end of the sequence, and releases CPU resource to the scheduler in the mean time.

The Round-Robin scheduling algorithm is a simple and effective algorithm for a concurrent transaction system without priority demands. However, in reality, many concurrent transaction systems need to assign priority level to some special transactions. For example, in a classified-customer financial system, a transaction request from a higher level customer always has preference over that from a lower level customer. When faced with such requirements, the Round-Robin scheduling algorithm shows its application limita-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

tion. In order to handle such types of applications, the $\mathcal{TP}$-$\mathcal{CTR}$ system uses a simplified version of *Rate monotonic algorithm.*

Simplified Rate-Monotonic Scheduling Algorithm

In the SRMA, each concurrent transaction is assigned a fixed priority by user or by default. Assuming task a and task b are concurrent execution transactions, during execution, the SRMA schedules tasks a and b in the following way:

- Task a can not be executed before task b finished, if task b has a higher priority level than task a.
- Task a has an equal chance to execute as task b, if task a and b have the same priority level.

In the design of $\mathcal{TP}$-$\mathcal{CTR}$, our main goal is to verify the logic of time-event-based prioritized transaction systems, while introducing features of real-time systems, i.e., priority constraints and timing constraints. However, we do not include a great variety of real-time features at current stage. More precisely, we ignore an important feature of real-time systems, *deadline*, which improves the efficiency of concurrent systems by giving up some tasks running over *deadlines*. Moreover, the fixed priorities of the system are assigned by users when writing $\mathcal{TP}$-$\mathcal{CTR}$ programs. These adjustments simplify the Rate Monotonic algorithm in our system by ignoring the transaction execution period assessment and corresponding automatic priority assignment, which is not important in a timing-event-based transaction verification system.

5.2.2 Interpreting and Scheduling Priority Constraints

Section 5.1 shows how the $\mathcal{TP}$-$\mathcal{CTR}$ compiler translates a concurrent Horn rule with priority constraints. This subsection presents how the $\mathcal{TP}$-$\mathcal{CTR}$ prototype interpreter interprets priority constraints represented in internal notations.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

After the translation by the $\mathcal{TP}$-$\mathcal{CTR}$ prototype compiler, a priority constraint is dded as the first parameter of the corresponding transaction, as shown in the example 5. Then the $\mathcal{TP}$-$\mathcal{CTR}$ prototype interpreter uses a SRMA to schedule constraint concurrent Horn rules. Each concurrent transaction in a $\mathcal{TP}$-$\mathcal{CTR}$ program has a priority level, which is used to determine their execution priority. To find the first highest priority level transaction from a concurrent sequence, the SRMA needs to know both the priority level of a transaction and the highest priority level of the concurrent transaction sequence. In the $\mathcal{TP}$-$\mathcal{CTR}$ prototype, lists are used to store these variants.

- The list *priorityList* stores all priority levels of the concurrent transactions in a priority descending order. For example, *priorityList* $[3, 2, 2]$ means there are three transactions in the concurrent sequence, and their priority levels are 3, 2 and 2, respectively.

- During interpretation, a transaction's priority level is indicated by the last element of the transaction list, e.g., $[seq([a, b]), 2]$ means the $seq([a, b])$ has priority level 2. The priority level will not disappear before this sequential transaction execution finishes.

The example below illustrates how priority constraints are interpreted by the $\mathcal{TP}$-$\mathcal{CTR}$ interpreter according to the simplified RMA.

Example 8 (Interpreting a $\mathcal{TP}$-$\mathcal{CTR}$ program) Consider the following transaction base:

$$\begin{array}{lll} task & :- & taskA\#taskB\#taskC \\ & & : priority(taskA, 1) \wedge priority(taskB, 2) \wedge priority(taskC, 2). \\ taskA & :- & ins(in(a)). \\ taskB & :- & ins(in(b)). \\ taskC & :- & ins(in(c)). \end{array}$$

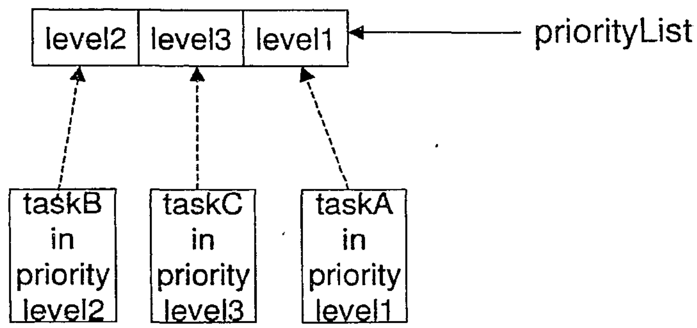
When interpreting the concurrent transaction *task*, the sub-transaction *taskA*, *taskB* and *taskC* are converted to the following *conc* list in the $\mathcal{TP}$-$\mathcal{CTR}$ prototype interpreter:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

conc([[ins(in(a)),1],[ins(in(b)),2],[ins(in(c)),2]])

where the respective priority level of each execution transaction is the last element of their list. The list *priorityList* is used to store the three transaction priority levels, as shown in Figure 5.2:

Figure 5.2: The list *priorityList* in $\mathcal{TP}$-$\mathcal{CTR}$ interpreter.



where *level*1 is the taskA's priority level 1, *level*2 is the taskB's priority level 2, and *level*3 is the taskC's priority level 2. The elements of the *priorityList* sort in an descending order, from higher priority level to lower priority level. □

During interpretation, the SRMA uses the following steps to determine the *hottest component*:

1. Picks up the first element of the execution transaction list
2. Gets the priority level of the first element from the execution transaction list
3. Gets the highest priority level from the *priorityList*(the first element)
4. If the current element's priority level is not the highest priority level, puts it in the end of the execution transaction list, then returns to the first step, and repeats steps 1 to 4.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5. Otherwise, the element is the *hottest component* and the interpreter should execute one step of this element.

   - If this step is the last step of the belonged transaction execution, deletes the first element of *priorityList*. Then returns to the step 1 if there is still transactions in the execution queue; or ends execution if no other transaction is left in the execution sequence.
   - Otherwise, returns the remaining of this element to the end of the execution transaction list, and then returns to the step 1.

In such a way, the $\mathcal{TP}$-$\mathcal{CTR}$ interpreter implements the inference system using a SRMA to formally execute $\mathcal{TP}$-$\mathcal{CTR}$ programs.

5.2.3 Interpreting Timing Constraints

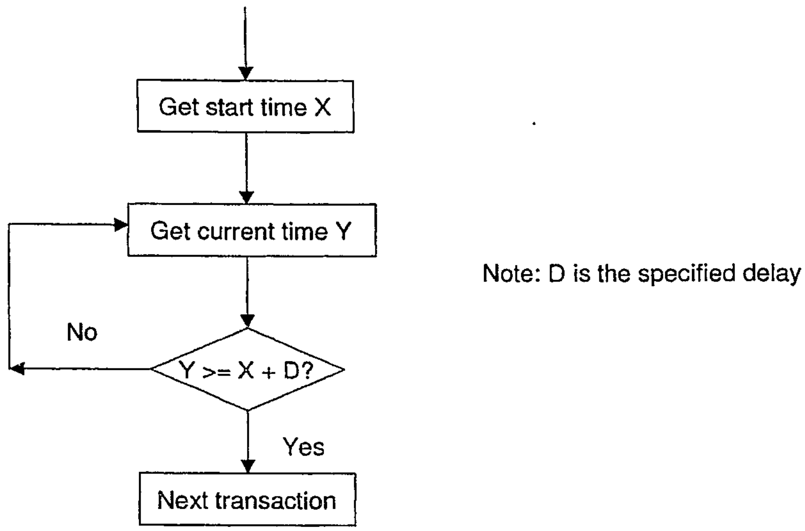
At current stage, the $\mathcal{TP}$-$\mathcal{CTR}$ prototype has two timing constraints: *timeElapsed*/1 and *delay*/1. *timeElapsed* is used to set an absolute timing relation, where a transaction is set to be triggered at an absolute time instant. *delay* is used to set a relative timing relation, where a transaction is set to start a period of time after the end of another transaction.

Actually, the timing constraints *timeElapsed* and *delay* are similar from a logic viewpoint. The only difference is the start time instant. Figure 5.3 illustrates the underlying logic for these two timing relations.

Both timing constraints are handled using such logic within the $\mathcal{TP}$-$\mathcal{CTR}$ interpreter. The timing constraint *timeElapsed* uses the *start_time*(X) to retrieve the start time, which stores the start time instant of the whole execution. In $\mathcal{TP}$-$\mathcal{CTR}$, we use timing-event-based mechanism for this timing constraint: satisfaction of the constraint precondition is used to trigger another transaction event. The $\mathcal{TP}$-$\mathcal{CTR}$ interpreter uses an atom *event* to specify the triggered transaction. After the timing condition is satisfied and the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Figure 5.3: Timing logic handled in the $\mathcal{TP}$-$\mathcal{CTR}$ interpreter.



specified transaction is triggered, the atom *event* corresponding to this transaction is deleted from the database system.

The other timing constraint of the current $\mathcal{TP}$-$\mathcal{CTR}$ prototype, *delay*, is used to define a delay relationship between two transactions. Thus, its start time is not the start time of the execution, but the start time of itself. When the specified delay time goes, the delay loop will end and it will go to the second transaction execution.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 6

Time-based Prioritized $\mathcal{TP}$-$\mathcal{CTR}$ Program Examples

To illustrate how $\mathcal{TP}$-$\mathcal{CTR}$ can be used to specify priority and timing constraints in timing-event-based prioritized transaction system, this chapter presents two examples in different application areas. More specifically, the first example regards financial transaction. The second example regards real-time logic control system.

6.1 A Time-based Financial Transaction Application

Example 9 is an extension of the *financial transaction application* presented in [1], where Bonner and Kifer illustrate how $\mathcal{CTR}$ can be used to specify the atomicity of financial transactions. Here we show how timing features and priority features in $\mathcal{TP}$-$\mathcal{CTR}$ is used to schedule tasks in a long-run financial transaction application.

Example 9 **(Specifying time-based prioritized financial transactions)** For the sake of understanding, first we briefly introduce the rules used in Bonner and Kifer's financial transaction example in [1].

$$\begin{aligned} transfer(Amt, Acct1, Acct2) \leftarrow \odot\, (&withdraw(Amt, Acct1) \\ &\otimes deposit(Amt, Acct2)) \end{aligned}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$withdraw(Amt, Acct) \leftarrow balance(Acct, Bal)$$
$$\otimes\, Bal \geq Amt$$
$$\otimes\, change\_balance(Acct, Bal, Bal - Amt)$$

$$deposit(Amt, Acct) \leftarrow balance(Acct, Bal)$$
$$\otimes\, change\_balance(Acct, Bal, Bal + Amt)$$

$$change\_blance(Acct, Bal1, Bal2) \leftarrow del(balance(Acct, Bal1))$$
$$\otimes\, ins(balance(Acct, Bal2))$$

The first predicate, *transfer*, specifies how a money transfer transaction is accomplished, i.e., by withdrawing an amount *Amt* from one bank account *Acct*1 and then *depositing* the *Amt* into another bank account *Acct*2. The next two predicates, *withdraw* and *deposit*, define the *withdraw* transaction and the *deposit* transaction, respectively. Both of them use the predicate *change_balance* to update an account's balance via the built-in elementary updates *del(balance(Acct, Bal)* and *ins(balance(Acct, Bal)*.

Based on the above basic transaction rule definitions, the $\mathcal{TP}$-$\mathcal{CTR}$ program below simulates a long-run time-based financial transaction scenario: to a source account, higher priority level transfer requests can always be answered even when normal transfer transactions are executing. In the program, a timed prioritized transfer transaction is assigned with a higher execution priority. Another transfer transaction process, assigned with the default priority level, runs recursively every two seconds until the remaining amount in the account is not enough for another transfer.

$$transfer\_process \leftarrow transfer\_queue(Fee1, Client, Broker)$$
$$\mid transfer\_urgent(Fee2, Client, Urgent\_Acct)$$
$$: timeElapsed(100) \rightarrow tirggerEvent(transfer\_urgent$$
$$(Fee2, Client, Urgent\_Acct))$$
$$\wedge priority(transfer\_urgent$$
$$(Fee2, Client, Urgent\_Acct), 2)$$

$$transfer\_queue(Amt, Acct1, Acct2) \leftarrow balance(Acct1, Bal)$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$\otimes\ (Bal \geq Amt)$$
$$\otimes\ transfer(Amt, Acct1, Acct2)$$
$$\otimes\ delay(2)$$
$$\otimes\ transfer\_queue(Amt, Acct1, Acct2)$$

$$transfer\_queue(Amt, Acct1, Acct2) \leftarrow monitor('transfer\ ends')$$

$$transafer\_urgent(Amt, Acct1, Acct2) \leftarrow transfer(Amt, Acct1, Acct2)$$

The first constraint concurrent Horn rule specifies the simulation process. It consists of two predicates, $transfer\_queue(Fee1, Client, Broker)$ and *transafer_ urgent(Fee2, Client, Urgent_Acct)*, which runs concurrently. The constraint formula specifies a timing event for *transafer_urgent*, which has priority in this concurrent execution and is scheduled to take place 100 seconds after the simulation starts. The transaction *transfer_queue* implements a long-run series of bank account transfer transactions, in which an amount $Fee1$ is transferred from one account *Client* to another account *Broker*, and the transfer process is called recursively every two seconds until the balance of *Client* is smaller than the amount $Fee1$. In this example, we adjust the initial amount of the account *Client* and the transferred amount $Fee1$ to ensure the transaction $transfer\_urgent$ is triggered during the execution of the transaction $transfer\_queue$.

The result shows that the specified prioritized event takes place right at 100 seconds, and the transaction $transafer\_urgent$ is triggered and then interrupts the normal transfer process by becoming the *hottest component* of the concurrent conjunction. As expected, the amount $Fee2$ is thus transferred from the *Client*'s account to the urgent account *Urgent_Acct* with priority. Only after this prioritized transfer transaction is completed, the interrupted normal transfer process can then resume its normal execution.

□

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6.2 A Simplified Elevator Logic Control Application

Besides the area of database transaction applications, $\mathcal{TP}$-$\mathcal{CTR}$ can also be used to simulate real-time control systems. Working with priority constraints, a *timeElapsed* timing-event-based constraint can create an *interrupt* functionality, which is scheduled to take place at specified time. The *interrupt* is one key feature of a real-time control system. Example 10 presents how to use $\mathcal{TP}$-$\mathcal{CTR}$ to simulate the *interrupt* in an elevator logic control system.

Example 10 (Simulating a simplified elevator controller) For the sake of understanding, below we present the assumptions for a simplified elevator controller model and basic control logic rules:

- The elevator is for a ten-floor building with one stop button on each floor. When the button is pushed, a stop request is sent to the controller.
- The elevator cage takes about 3 seconds to go up or down one floor.
- Every floor has a location sensor to indicate the elevator's position.
- In any exceptional case, the elevator should take *stop* action immediately.

Based on these assumptions, the program below specifies the simplified control logic for the elevator controller:

$$\begin{aligned}
&simulate \leftarrow moving_control(stop)\\
&\qquad \mid user_request \mid accident\\
&\qquad : timeElapsed(31) \rightarrow triggerEvent(accident)\\
&\qquad \wedge\, priority(accident, 2)\\
&user_request \leftarrow req(5) \mid req(8) \mid req(2)\\
&\qquad : timeElapsed(10) \rightarrow triggerEvent(req(5))\\
&\qquad \wedge\, timeElapsed(30) \rightarrow triggerEvent(req(8))\\
&\qquad \wedge\, timeElapsed(50) \rightarrow triggerEvent(req(2))\\
&req(Level) \leftarrow ins(stop_req(Level))
\end{aligned}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$
\begin{aligned}
moving\_control(Status) \leftarrow\ & Status = stop \\
& \otimes\ stop\_req(Level) \\
& \otimes\ moveto(Level) \\
& \otimes\ del(stop\_req(Level)) \\
& \otimes\ moving\_control(stop) \\
moving\_control(Status) \leftarrow\ & moving\_control(stop)
\end{aligned}
$$

$$
\begin{aligned}
moveto(Level) \leftarrow\ & accSignal \\
& \otimes\ stop\_handle \\
moveto(Level) \leftarrow\ & not\ accSignal \\
& \otimes\ getLocation(X) \\
& \otimes\ X <> Level \\
& \otimes\ moveto(Level) \\
moveto(Level) \leftarrow\ & not\ accSignal \\
& \otimes\ getLocation(X) \\
& \otimes\ X = Level \\
& \otimes\ stop\_handle
\end{aligned}
$$

$$
accident \leftarrow ins(accSignal)
$$

where basic atomic formulas and built-in predicates in the specification have the following definition:

- The atomic formula *stop_req(Level)* is used to store the stop requests to be answered.
- *stop_handle* is a built-in predicate to stop the elevator.
- *getLocation(X)* is a built-in predicate to retrieve the current elevator location.
- The atom *accSignal* is used to denote an urgent exceptional event.

In the program, the first constraint concurrent Horn rule specifies the entire simulation process. It consists of three concurrent transactions, *moving_control*, *user_request* and *accident*. The constraint formula specifies an exceptional event: an *accident* which is scheduled to take place at 31 seconds after the simulation starts. The predicates *moving_control* and *user_request* implement the normal elevator control logic. The predicate *user_request* simulates use case scenario: three stop requests are scheduled to take

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

place at specified times. The predicate *moving_control* is used to control the elevator: moving to a floor according to the stop requests. The predicate *moveto*(*Level*) performs the elevator moving action, and monitors any exceptional accident as well.

When the specified accident takes place at the 31 seconds time instant, because of its priority, the predicate *accident* works as an interrupt and notifies the system at once by inserting the atom *accSignal* into the database. In our case, the elevator is still moving from 5th floor to 8th floor at that time. The predicate *moveto*(*Level*) receives this signal, then correspondingly executes the predicate *stop_handle* to stop the elevator at once. □

This example demonstrates that $\mathcal{TP}$-$\mathcal{CTR}$ can be used to simulate some intrinsically real-time phenomena, e.g., the interrupt effect and timing relations. By elaborately combining priority constraints and timing constraints in different ways, we can verify different elevator logic control cases with the $\mathcal{TP}$-$\mathcal{CTR}$ program.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 7

Conclusions and Future Works

IN the previous chapters, we presented an extension of Concurrent Transaction Logic, a formalism originally designed to handle state changes in deductive databases. We extended $\mathcal{CTR}$ by defining a logic framework in which the user can not only specify concurrent transaction processes as logic programs, but also define priority and timing constraints on these processes. This increases the flexibility and power of the language. Users are no longer forced to schedule the transaction order only by the sequential conjunction and concurrent conjunction. The execution order of concurrent transactions can also be specified or changed by extra ways: assigning timing constraints and priority constraints to these transactions. The interrupt effect created by combining the event-driven feature of the timing constraint *timeElapsed* with priority constraints schedules transactions in a more flexible and powerful way, and thus opens the logic to a new range of advanced application, e.g., real-time domain application, long-run time-related application, digital circuit design with clock-driven and continuous-simulation of the designed digital circuit, etc.

To allow the formal execution and thus the simulation of such programs, we introduced an inference system that is able to handle priority constraints with the underlying simplified Rate-Monotonic scheduling algorithm. The formal execution of programs works as SLD-style refutation mechanism. By this way, our approach provides a high-level framework for specifying and executing transaction logic programs involving priority and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

timing properties. Also, we have shown how logic programming techniques can be used to implement real-time database application domains.

To make theory meet practice, we have implemented our $\mathcal{TP}$-$\mathcal{CTR}$ prototype in XSB Prolog and have tested the examples presented in this thesis.

There are still some issues left open, especially real-time issues. Below we elaborate on them.

- *Sophisticated Timing Constraints.* The timing properties in our timing constraint are quite straight-forward at present stage. It only handles the timing property of a transaction *when to be triggered*, and a delay relation between two transactions. However, in reality, timing properties of a real-time system are much diverse, such as described in [4, 5, 6]. Some of them relate to the *deadline* of a transaction as well as the initial time instant. Adopting these ideas into the $\mathcal{TP}$-$\mathcal{CTR}$ timing constraint will be valuable to expand its real-time application domain.

- *Underlying scheduling algorithm.* The current $\mathcal{TP}$-$\mathcal{CTR}$ possible prototype uses a simplified Rate Monotonic Scheduling algorithm to schedule concurrent transactions. In real-time domain, this algorithm is straight-forward and may be inefficient in some cases. It should be possible to adopt a more sophisticated scheduling algorithms, to improve the performance of the $\mathcal{TP}$-$\mathcal{CTR}$ prototype in real-time domains.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Appendix A

$\mathcal{TP}$-$\mathcal{CTR}$ Tutorial

A.1 $\mathcal{TP}$-$\mathcal{CTR}$ Prototype File System

The $\mathcal{TP}$-$\mathcal{CTR}$ prototype in this thesis is extended on the base of Concurrent Transaction Logic with Recovery prototype available in [17]. It consists of the following models:

- *ctr.P* - the basic $\mathcal{TP}$-$\mathcal{CTR}$ interpreter
- *parser.P* - a parser for $\mathcal{TP}$-$\mathcal{CTR}$ rules
- *updates.P* - the code for back-trackable updates
- *load.P* - startup routine that loads the prototype modules
- *upload.P* - a module including rules to load a $\mathcal{TP}$-$\mathcal{CTR}$ transaction-base and a database into XSB system.
- *timer.P* - including $\mathcal{TP}$-$\mathcal{CTR}$ system rules regarding timing constraints

A.2 Getting Started

The $\mathcal{TP}$-$\mathcal{CTR}$ prototype was implemented in XSB Prolog, and consists of the modules introduced above. To run the implementation, at query prompt, the *load* module must first be consulted, then invoke the predicate *ctr_init*. The predicate *ctr_init* is defined

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

in the module to load all necessary files into XSB Prolog and initialize the necessary parameters. Below we illustrate how the user interacts with the system.

$xsb -i

XSB Version 2.5 (Okocim) of March 11, 2002

[i686-pc-linux-gnu; mode: optimal; engine: slg-wam; gc: indirection; scheduling: local]

| ? – [*load*].

[load loaded]

yes

| ? – *ctr_init.*

[ctr loaded] - *basic $\mathcal{TP}$-$\mathcal{CTR}$ interpreter*

[updates loaded] - *code for back-trackable updates*

[parser loaded] - *parser for prototype rules*

[upload loaded] - *compilers for loading and translating rules*

[timer loaded] - *embedded timing constraints*

[scrptutl loaded] - *necessary XSB system module for system timing predicates*

yes

A.3 $\mathcal{TP}$-$\mathcal{CTR}$ Prototype Commands and Program Files

Like $\mathcal{CTR}$ programs, $\mathcal{TP}$-$\mathcal{CTR}$ programs are also stored in the transaction-base and database with filename extensions *.ctr* and *.db*, respectively. Although in some cases the database file maybe empty, it is still needed in our prototype.

A.3.1 Compiling Commands in $\mathcal{TP}$-$\mathcal{CTR}$

The $\mathcal{TP}$-$\mathcal{CTR}$ programs are compiled with the following three commands:

- *ctr_comp(program_name).* - to compile both the transaction-base and database;

- *comp_trans(program_name).* - to compile the transaction-base only;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- *comp_db(program_name)*. - to compile the database only.

where program_name is the filename of the $\mathcal{TP}$-$\mathcal{CTR}$ program without extension. After compiled, a $\mathcal{TP}$-$\mathcal{CTR}$ transaction-base file generates two files:

program_name.ctr.temp

program_name.ctr.o

where *program_name.ctr.temp* is an intermediate compiled file, in which the timing constraints and priority constraints have been translated into a recognizable $\mathcal{CTR}$ program. The *program_name.ctr.o* file is the final transaction object file via the underlying $\mathcal{TP}$-$\mathcal{CTR}$ compiler.

Like the $\mathcal{CTR}$ database object file, a $\mathcal{TP}$-$\mathcal{CTR}$ compiler creates a $\mathcal{TP}$-$\mathcal{CTR}$ database object file with the name:

program_name.db.o

where the extension *.db.o* denotes the file to be a database object file.

A.3.2 Execution Command

To execute a transaction in the $\mathcal{TP}$-$\mathcal{CTR}$ program after compilation, the user uses the following command:

execute(transaction_name).

where *transaction_name* is the head of one of the rules in the transaction-base file.

A.4 The $\mathcal{TP}$-$\mathcal{CTR}$ Prototype Syntax

A.4.1 Transaction Rules

The constraint concurrent Horn rules consists of three parts: a rule head, a rule body, and a constraint body.

Head :- Body : Constraint Body.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

or

Head :- Body.

or

Head.

where

- *Head* is an atomic formula;
- *Body* is a sequence of transaction actions (or queries) connected by any of the following conjunction operators:
  - sequential conjunction($*$),
  - concurrent conjunction($\#$),
  - isolation(o).
- *Constraint Body* is a sequence of the following constraints connected by conjunction($\wedge$):
  - timing constraint $timeElapsed(X) <- triggerEvent(Transaction\_name)$,
  - priority constraint $priority(Transaction\_name, Priority\_Level)$

A.4.2 Database Rules

A database is any Prolog rule or atom in the forms of format:

Head :- Body.

or

Head.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

..5 Built-in $\mathcal{TP}$-$\mathcal{CTR}$ Predicates

The prototype has some built-in predicates, which are used in the transaction-base or database. These predicates should not be re-defined by the user.

A.5.1 Database Declaration

Database atoms represent tuples in a relation. Like table in relational database, a relation should be declared before it can be accessed. The statement below declares a database relation $Name$ with arity $NArgs$:

updatable Name/Nargs.

By default the following basic system database declarations and tuples are always declared for any $\mathcal{TP}$-$\mathcal{CTR}$ program:

updatable event/2	- timing driven event list
updatable start_time/1	- start time instant of an execution
updatable priorityList/1	- a current executed transaction priority list in descending order
priorityList([ ])	- the initial empty list

A.5.2 Built-in Predicates in Transaction-base

For the declared database atoms, there are three built-in predicates for accessing them:

- $db(p(x))$ and $empty(p(x))$ - query atom $p(x)$ in the database
- $ins(p(x))$ - inserts atom $p(x)$ into the database
- $del(p(x))$ - deletes atom $p(x)$ from the database

At present stage, there are the following built-in constraint predicates:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- $timeElpased(X)$ - a timing constraint based on an absolute timing instant
- $delay(X)$ - a timing constraint based on a relative timing instant
- $priority(Transaction\_name, priority\_level)$ - priority constraint

Besides, in the $\mathcal{TP}$-$\mathcal{CTR}$ prototype we have kept the useful $\mathcal{CTR}$ *monitor* command below, which can be used to trace a transaction execution.

$monitor(Task)$.

where Task is a name or a term. Note that $monitor(Task)$ does not execute $Task$, it only displays messages. When the monitor executes, it displays one of the messages:

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

A.6 Programming Examples

The following examples illustrate some simple $\mathcal{TP}$-$\mathcal{CTR}$ programs and their execution. They are executed in XSB Prolog, with the loaded $\mathcal{TP}$-$\mathcal{CTR}$ prototype.

Example A.1 Concurrent Processes with Priority Constraints

The following program executes three concurrent non-interacting transactions. The three transactions have different priority levels and has two tasks respective. The program name is *interleave*. Below are the contents of the database file and transaction file.

Database file *interleave.db*:

updatable event/2.
updatable priorityList/1.
priorityList([]).

Transaction Base file *interleave.ctr*:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$parent \text{ :- } taskA\#taskB\#taskC \ : \ priority(taskA, 2) \wedge$$
$$priority(taskB, 2) \wedge$$
$$priority(taskC, 1).$$

*taskA :- monitor(taskA1) * monitor(taskA2).*

*taskB :- monitor(taskB1) * monitor(taskB2).*

*taskC :- monitor(taskC1) * monitor(taskC2).*

Compiling the program and executing a transaction:

| ?- ctr_comp(interleave).	- *compile the program*
yes	
| ?- execute(parent).	- *execute the parent goal*
completed taskA1	- *complete the first task of taskA*
completed taskB1	- *complete the first task of taskB*
completed taskA2	- *complete the second task of taskA*
completed taskB2	- *complete the second task of taskB*
completed taskC1	- *complete the first task of taskC*
completed taskC2	- *complete the second task of taskC*
yes	

Example A.2 Interrupt in Concurrent Processes

This example shows how the interrupt is created by combining a timing constraint and a priority constraint in a long-run program. The *initial* process is timed to start at 1 sec after the execution start, and the three concurrent sub-processes of the process *parent* have the same priority level. The subprocess of the *initial* process, *interrupt*, has higher priority level in the concurrent execution.

Database file *interrupt.db*:

updatable event/2.

updatable start_time/1.

updatable a/1.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

updatable priorityList/1.

priorityList([]).

finished :- not(unfinished).

unfinished :- a(X).

$a(100).\ a(99).\ \cdots\ a(1).\ a(0).$

Transaction Base file *interrupt.ctr*

$parent$:- $initTime * (long\_run\_transaction\#initial\#short\_transaction)$

$: timeElapsed(1) -> triggerEvent(initial)$

$\wedge priority(long\_run\_transaction, 2)$

$\wedge priority(initial, 2)$

$\wedge priority(short\_transaction, 2).$

$long\_run\_transaction$:-$o(del(a(X)) * monitor(X)) * long\_run\_transaction.$

$long\_run\_transaction$:- $finished.$

$initial$:- $monitor(interrupt\_init\_task) * interrupt : priority(d, 5).$

$interrupt$:- $monitor(interrupt\_task\_1) * monitor(interrupt\_task\_2).$

$short\_transaction$:- $monitor(short\_transaciotn\_task\_1)$

$*monitor(short\_transaction\_task\_2)$

$*monitor(short\_transaction\_task\_3)$

$*monitor(short\_transaction\_task\_4).$

Compiling the program and executing a transaction:

| ?- ctr_comp(interrupt).	- *compile the program*
yes	
| ?- execute(parent).	- *execute the parent goal*
completed 100	- *shows one round of long_run_transaction is completed*
completed short_transaction_task_1 completed short_transaction_task_2	- *shows one round of short_transaction is completed*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
ompleted 99
ompleted short_transaction_task_3
completed short_transaction_task_4
completed 98
     .
     .
     .
completed 75
completed interrupt_init_task      - shows the initial task of the interrupt is
completed 74                         completed
completed interrupt_task_1         - shows one task of the interrupt is completed
completed interrupt_task_2
completed 73
     .
     .
     .
completed 0
yes
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Bibliography

[1] Anthony J. Bonner and Michael Kifer, "Concurrency and Communication in Transaction Logic",*Proceedings of the Joint International Conference and Symposium on Logic Programming(JICSLP)*, Sep. 2-6 1996, Pages 142-156. MIT Press.

[2] Anthony J. Bonner, "Concurrency 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] Anthony J. Bonner and Michael Kifer, "Transaction Logic programming (or a logic of declarative and procedural knowledge)",*Technical Report CSRI-323, University of Toronto*, November 1995. http://www.cs.toronto.edu/ bonner/transaction-logic.html.

[4] P. Bellini, R. Mattolini, and P. Nesi, "Temporal Logics for real-Time System Specification," *ACM Computing Surveys*, Vol.32, No.1, March 2000.

[5] B. Dasarathy, "Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them", *IEEE Trans. Software Engineering*, Vol. SE-11, No. 1, January 1985, PP.80-86.

[6] Horng-Yuan Chen and Jeffrey J.P. Tsai and Yaodong Bi, "An Event-Based Real-Time Logic to Specify the Behavior and Timing Properties of Real-Time Systems," *IEEE*, 0-8186-2300-4/91 (c)1991.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[7] Kristian Torp, Christian S. Jensen, Richard T. Snodgrass, "Effective timestamping in databases," *The VLDB Journal*, (c)Spring-Verlag 2000.

[8] Ögur Ulusoy, Geneva G. Belford, "Concurrency Control in Real-Time Database Systems," *ACM*, 089791-472-4/92/0002/0181, 1992.

[9] Rajendran M. Sivasankaran, John A. Stankovic, Don Towsley, Bhaskar Purimetla, Krithi Ramamritham, "Priority assignment in real-time active databases," *The VLDB Journal* , (c) Spring-Verlag 1996.

[10] Wayne Wolf, "Computer as Components: Principles of Embedded Computing System Design", *Morgan Kaufman Publishers*, 2001, ISBN 1-55860-541-X.

[11] Neugass H., Espin G., Nunoe H., Thomas R., Wilner D., "VxWorks: an interactive development environment and real-time kernel for Gmicro," *TRON Project Symposium*,1991. Proceedings., Eighth , 21-27 Nov. 1991 Page(s): 196 -207

[12] Terrasa, A.; Garcia-Fornes, A , "Real-time synchronization between hard and soft tasks in RT-Linux" *Real-Time Computing Systems and Applications*, 1999. RTCSA '99. Sixth International Conference on , 13-15 Dec. 1999 Page(s): 434 -441

[13] Amalia F. Sleghel, "An Optimizing Interpreter for Concurrent Transaction Logic" *Thesis for degree of Master of Science in University of Toronto*, 2000.

[14] Jiwen Ge, Marcus Vinicius Santos, "Using logic programming techniques to handle timing and priority constraints in real-time systems" *Seventh World Conference On Integrated Design &Process Technology* IDPT Vol.2 2003, ISSN No. 1090-9389, Page(s): 800-806

[15] Konstantinos Sagonas, Terrance Swift, David S. Warren, Juliana Freire, Prasad Rao, Baoqiu Cui, Ernie Johnson, "The XSB System Version 2.5 Volume I: Programmer's Manual" *SUNY at Stony Brook*, 2002.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[16] Konstantinos Sagonas, Terrance Swift, David S. Warren, Juliana Freire, Prasad Rao, Baoqiu Cui, Ernie Johnson, "The XSB System Version 2.5 Volume II: Programmer's Manual" *SUNY at Stony Brook*, 2002.

[17] Anthony Bonner, "*CTR* prototype with recovery", *ftp://ftp.cs.toronto.edu/cs/ftp /pub/bonner/papers/transaction.logic/prototype/recoverable/*.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.