LOGICAL FOUNDATIONS OF ACTIVE DATABASES

by

Iluju Kiringa

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Logical Foundations of Active Databases

Iluju Kiringa
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2003

Classical database management systems (DBMSs) have been enhanced over the past fifteen years with the addition of rule-based programming to obtain active DBMSs. Active behavior is mainly characterized by a rule language and an execution model. Execution models go hand in hand with advanced transaction models (ATMs) which relax the so-called ACID (Atomicity-Consistency-Isolation-Durability) properties. Both rule languages and execution models have been proposed in an *ad hoc* way to deal with applications which are not easily implementable using the classical DBMSs. Therefore an open problem in this area is to formally account for active behavior using a uniform formalism.

This thesis gives logical foundations to active databases using the situation calculus, a logic for reasoning about actions. Our approach appeals to theories in which one may refer to all past database states. We give a logical semantics to an ATM by specifying this as a theory of the situation calculus called *basic relational theory*, which is a set of sentences suitable for referring to past database states in the context of database transactions. We express the properties of the ATM as formulas of the same calculus. Such properties are logically entailed by the basic relational theory that captures the ATM. We illustrate our framework by formalizing various transaction models. Next, we introduce *active relational theories*, which extend basic relational theories by capturing typical aspects of active behavior found in existing active DBMSs. We give a formal semantics to various features of active behavior by using active relational theories. We capture the most popular active rules, the Event-Condition-Action (ECA) rules, as programs written in a situation calculus based logic programming language that accounts for parallelism and we also write transaction programs in the same language. We provide an abstract ECA rule interpreter for executing transaction programs that have active relational theories as background axioms. We also classify various execution semantics for ECA rules. Finally, we give a method for implementing active relational theories in Prolog.

Dedication

To Odyssée, Phillip and Claude,

to the memory of my mother, Marie-Josée Modiri, and

to the memory of Professor Raymond Reiter

Acknowledgements

At the end of this long and eventful journey throughout the Ph.D. process, it is with great pleasure and deep sense of recognition that I would like to acknowledge the help and support of many people. First of all, I would like to posthumously express my most grateful thanks to Raymond Reiter. Professor Reiter has been my supervisor five years long until his untimely death on September 16, 2002. He was a very exceptional scientist, and a great AI and database researcher. He made scientific contributions of the highest quality that will last for ever and permanently influence Computer Science. My thesis, that builds on his early work on database theory, is just a small piece of his tremendous influence. I have learned many aspects of CS from him, both from the point of view of content and form. I have also learned from him how an excellent professor has to be, both scientifically and as a human being.

My grateful thanks also go to Hector Levesque, my supervisor. Professor Levesque coached me upon my arrival in Toronto, when Ray was away for six months on sabbatical. He also took over the job of supervisor after Ray's passing. His quick, brief and incisive comments always came to the point, and they helped a lot. Tony Bonner and John Mylopoulos were my database background. Those who could see the flaws with respect to the database content in my work. They did it meticulously. Thank you for that. I remember Ray saying, "Tony will push you to the limits ! John is one of the best mentors around!"; and he was right. Yves Lespérance deserves a great thank for pointing out many subtleties related to the situation calculus. Finally, a special thanks to Alex Borgida, my external examiner, who took his time to make useful suggestions for improving this thesis.

The generosity and love of my wife Odyssée, and my sons Phillip and Claude made my arduous work easier to finish. They endured as much (or even more than) as I did myself. The pain and joys of carrying out my Ph.D. tasks have also been theirs. They deserve a very special thanks for that.

I would also like to thank my numerous friends at UofT for their support, advice, and help in many aspects of life in this great university: Mikhail E. Soutchanski, Alfredo Gabaldon, Denilson Barbosa, Sebastian Sardinas, Marcelo Arenas, Dattraya Kulkarni, Lucia Moura, and Daniel Panario. My thanks also go to Yilan Gu, Eugenia Ternovskaia, Marsha Chechik, Ric Hehner, Yongmei Liu, etc (the list is not exhaustive) for their feedback on many issues. My friends in life were a constant source of encouragement: Paul Balebela, Jean-Claude Beia, Benjamin Bajikijaie, Dr Jean-Pierre Diamani and family, Jan and Maria Ebben, Dr Kadari Mwene Kabyana, Sylvain Kalala, Dr Célestin Kalambayi, Nola Kianza, Father Etienne Lefevere, Cléophas Leke, Jean-Rémi Makana, Father Joseph Mbo Omaw, Dr Grégoire Mbuyi, Father Joseph Mbuyi, Marcellin Mbwa-mboma, Dr Blaise Mpeti, Dr Basile Mpotompendo, Dr Crispin Ngwey and family, Bishop Phillipe Nkiere, Mastaki N'sebuyange, Modeste Nshole, Father Donat Nshole, Dr Mwamba Tshibangu, Damas Umba, Dr Fréderic Wandey, etc (again the list is not exhaustive).

Finally, I warmly thank my parents Dominique Kiringa et the late Marie-José Modiri for their constant encouragement. This thesis is partly dedicated to the memory of my mother. I am sure that her eternal rest will be even more sweetened by my Ph.D.

Contents

List of Figures

Important Definitions and Abbreviations

Chapter 1

Introduction

1.1 Background and Motivation

Active databases unify traditional database technology with rule-based programming to express reactive capabilities. Traditional database management systems (DBMSs) are passive in the sense that only users or application programs can activate definition and manipulation operations on stored data. An important and useful enhancement would be the addition of active behavior to them to obtain active DBMSs (ADBMSs). Here, the system itself performs some definition and manipulation operations automatically, based on a suitable representation of the (re)active behavior of the application domain and the operations performed during a database transaction.

The concept of rule and its execution are essential to ADBMSs. An ADBMS has two major components, a *representational* component called a rule language, and an *executional* component called an execution model. The rule language is used to specify the active behavior of an application. Typical rules used here are the so-called EVENT-CONDITION-ACTION (ECA) rules which are a syntactical construct representing the notion that an action must be performed upon detection of a specified event, provided that some specified condition holds. The execution model is best explained in conjunction with the notion of database transaction. A database transaction is a (sometimes nested or detached) sequence of database update operations such as *insert*, *delete*, and *update*, used to insert tuples into, delete them from, or update them in a database. The execution model comes in three flavors: immediate execution, deferred execution, and detached execution, meaning that the execution of rules is interleaved with the execution of database update operations, is done at the end of transactions, and is done in a separate transaction, respectively.

Although the term "active databases" was coined by M. Morgenstern as early as 1983 ([Mor83]), ADBMSs have attracted the attention of researchers in databases only over the last decade ([JF96], [Pat99]). The perceived need for active behavior in DBMSs stems from some advantages ADBMSs have over their passive counterpart ([WC96], [FT95], [PDW$^+$93]): they (i) perform functions that passive DBMSs can only encode in applications, (ii) are needed to implement a wide range of applications that cannot be implemented using passive DBMSs, and (iii) perform tasks requiring special-purpose subsystems in traditional DBMSs.

General integrity constraints (ICs) and triggers are examples of the new functionalities provided by ADBMSs. Integrity constraints state conditions that must be valid over data stored in the database. Though some limited kind of ICs such as key constraints and referential ICs are supported in traditional DBMSs, ADBMSs support more general and flexible ICs that are executable at any time to initiate compensating actions whenever violations of ICs are detected. Triggers state that some actions are to be invoked upon the detection of certain conditions. In the case of violations of ICs, these actions are, as mentioned above, compensating ones.

Data-intensive expert systems, view materialization, or workflow management systems are some of the applications that are beyond the scope of passive DBMSs ([WC96]). Data-intensive expert systems are expert systems whose data exceeds main memory; they store their excess data in a database or a file system. Since they generally use a rule language and a database, they can be built using an ADBMS as platform.

View materialization deals with the propagation of changes made to relations in a database to views defined on these relations. The reason for doing so is that materialized views must be kept consistent with the relations over which they have been defined. Since changes to base relations of a database can be viewed as triggering events and their use to update views can be defined as actions, they also can be managed using an ADBMS as platform.

A workflow management system (WMS) controls interconnected computational tasks. Such a system monitors the state of managed tasks, invoking other tasks whenever appropriate and passing data between them as needed. Since monitoring and action or task invocation are typical of ADBMSs, the later are suitable platforms for developing WMS applications.

In ADBMSs, many of the new functionalities are encoded in the system kernel. Therefore, no need for special-purpose subsystems arises in them. Tasks such as authorization, statistics gathering, view management, or simple ICs are simply “hardwired” into the system kernel.

Attracted by these obvious advantages of ADBMSs, various vendors of relational DBMSs have incorporated active behavior facilities in their products, and the object-relational SQL standard (SQL3) is supporting active rules as well ([Pat99],[WC96]).

1.2 The Problem: A Formal Account of Active Databases

There is a large body of literature on active databases. In particular, most researchers in the area have worked on prototypes; several active database systems have been developed (see [FT95], [WC96], [PCFW95], and [Pat99] for a catalogue of the most influential systems). Unfortunately, lacking a common theory for the semantics of ECA-rules and being developed independently from each other, they exhibit different behaviors for syntactically similar rules ([FT95]), so that the question of giving a unifying formal foundation to active databases is an important open problem; that is, an investigation of the formal representation and reasoning aspects of rules is an open topic of research. This lack of foundational work on active databases leads, for instance, Ceri and Widom to express the following complaint:

> ... there is no unifying theory underlying active database systems comparable to the theory underlying deductive database systems (...). A formal foundation for active database rule languages would provide a very important step in understanding and characterizing the commonalities and differences across systems ([WC96]).

Without stating it explicitly, the unifying theory that Ceri and Widom have in mind is a restricted form of first order logic, namely DATALOG with negation. The lack of a uniform formal semantics for characterizing active databases and the existence of several ad-hoc, different, and hard to compare operational semantics have also been pointed out in [Zan93], [Wid93], [PCFW95], [FT95], [CC95], [BL96], and [Pat99]. This is the source of some of the technical problems that active databases face such as a clear justification of the correctness of the implementation.

This thesis *aims to provide a logic-based formal account of active databases*. Active rules are a dynamic domain, thus such a formal account should be provided in a framework suitable for modeling changing worlds. For this, we choose the situation calculus (proposed in [MH69], and enhanced by the Cognitive Robotics Group at the University of Toronto as shown in [Rei01]), which is a language for modeling dynamical domains, as the particular logical framework within which to carry out our formalization of active databases. We use this calculus *both* to represent the active behavior and its execution semantics, and to reason about the properties of this behavior. Situation calculus theories used to represent active databases and their execution semantics play a role similar to that of the restricted forms of first order logic that are used in deductive databases.

Work has been done on formalizing *some* aspects of active databases. However, work done with a view towards a unifying theory accounting for most of the aspects of active databases is scarce (see, for instance, [Wid93], [Zan93], [FT95], and [RPS99]). Existing proposals use a variety of modeling formalisms. However, we strongly believe for reasons that will be clear in Chapter 4 that the situation calculus permits much simpler solutions to most of the formalization problems in active databases than those furnished by existing formalisms. One of these reasons is that accounting for database transactions involves representing and reasoning about actions and change from database state to database state, all of which are suitably done in the situation calculus.

Informally, the situation calculus forms a suitable logical framework within which a general theory of actions in a dynamically changing world can be formulated. The basic ontological categories of the calculus are *situations* (histories) and *actions*. Situations can be interpreted as world histories constituted by a list of actions ([Rei96]), whose execution brings the world from one situation to another. The world situations are described by a set of properties called *fluents*, which become true or false when actions are performed. For modeling purposes, one should provide a suitable set of axioms (a theory) which specifies how the world has to change as a result of action executions. In providing such a set of axioms, one has to cope with the so called frame problem, which is the problem of providing a very large number of axioms specifying what remains invariant during changes. In [Rei01], Reiter proposes a sort of action theory, called basic action theory, which is suitable for reasoning about actions and, moreover, embodies a solution to the frame problem.

By providing a formal account of active databases in this thesis, we contribute to a solution to the aforementioned open problem of foundational work, which has been largely neglected until recently (see [FT95], [Wid93], and [Zan93] for a discussion of this problem).

1.3 Challenges

As we argued in the previous section, we use the situation calculus both to account for the representational and the executional components of active databases, and to reason about rules. In achieving our aims, we shall encounter many challenges. Firstly, we shall have to focus on the technicalities of building a logical framework suitable for giving a *uniform* formal account of (ideally all) the relevant aspects of active databases. Secondly, we should use this framework to model the different concepts of advanced database transactions such as nested and hierarchical transactions; furthermore, we should use this framework to model active rules as programs written in a situation calculus-based language and to model the different flavors of execution semantics as suitable closed formulas of the situation calculus. Thirdly, we shall have to (correctly) implement the obtained specifications in order to execute them. Finally, we shall use the framework to capture some aspects the ECA rule standard SQL3.

1.4 Methodology

In [Rei84], Reiter proposes a logical (proof theoretic) formalization of relational databases by formulating the following assumptions that underly relational databases: the closed world assumption (CWA), the unique name assumption (UNA), and the domain closure assumption(DCA). He formalizes relational databases as a set of ground assertions $R(a_1, \ldots, a_n)$ over a function-free first order language $\mathcal{L}$, where R is a n-ary relational symbol in $\mathcal{L}$, and $a_1, \ldots, a_n$ are constant symbols in $\mathcal{L}$, together with a set of axioms stating the above assumptions, Clark's completion (Clark[1978]), and equality. All of these assertions and axioms form a *relational theory*. A query is a statement in $\mathcal{L}$. Finally, an answer to a query is a logical consequence of the database.

Without an explicit reference to [Rei84], [Rei95] formalizes database relational schemas as fluents whose truth value evolves over time as a result of updates formalized as actions in the situation calculus.

Building on [Rei84] and [Rei95], our thesis *aims to account formally for active databases as a further extension of the relational model to accommodate new world knowledge, in this case representational issues associated with rules, their execution semantics, and database updates*. We achieve this goal by using the situation calculus. We think that the situation calculus may be used for such a formal account because of the following advantages: the calculus provides a formalization of primitive transactions; it is a logic for describing dynamical systems; it provides a foundation for a theory of complex actions in which an operational semantics of active systems can be formulated; it expresses knowledge about the world in a declarative way, thus allowing an implementation independent characterization of active systems; and its notion of action can be used to formalize both the action part and the event part of active rules, whereas the condition part of active rules may be formalized using first order logic.

It is clear that several other similar logics could have been used for our formalization. Prominent examples are dynamic logic, event calculus, and transaction logic. We prefer the situation calculus, due to a set of benefits it offers, the most desirable of which are the following: actions are considered first-class citizens of the logic, thus allowing us to remain within the language to reason about them; and the explicit addressing of the frame problem that inevitably occurs in the context of the database updates.

1.5 Contributions

1.5.1 Modeling Database Transactions as Relational Theories

Various advanced transaction models (ATMs) have been proposed to extend the classical flat transactions ([GA95]) by relaxing some of the ACID (Atomicity-Consistency-Isolation-Durability) properties (see [Elm92],[JK97] for a collection of the best examples of these models). The ATMs aim at improving the functionality and the performance of new applications that do not accommodate the ACID properties.

We extend the specification of database updates given in [Rei95] to a general theory of database transactions with ACID properties. We start by extending situation calculus theories of [Rei01] to non-Markovian theories ([Gab00]), which explain change in terms of all past situations and not solely based on the previous situation. Non-Markovian theories are needed, for example, to concisely (i.e., with few axioms) capture the semantics of transaction actions which very often refer to the history of a database. Next, we introduce the building blocks of our specification framework for representing relational database transactions in the situation calculus. After that, we model relational flat databases transactions as second order theories called *basic relational theories*, and define a legal log as one in which all database actions that are possible have indeed been executed. Here, we focus on using basic relational theories to model flat transactions with ACID properties, which turn out to be properties of logs that are legal. We then generalize these relational theories to deal with variations of classical flat transactions, and various ATMs, among which closed and open nested transactions are good examples. We show how to build relational theories corresponding to these various ATMs, to formulate properties of these ATMs in the situation calculus and to prove these properties as logical consequences of the relational theories.

1.5.2 Modeling Active Databases as Active Relational Theories

We extend the framework for modeling ATMs to one for modeling the reactive and execution models of active behaviors. The new theories we introduce here are called *active relational theories*. As active databases are intimately related to transactions, a substantial building block of these new theories is made of relational theories. An active relational theory precisely encompasses a basic relational theory capturing a specific ATM and axioms for typical active database fluents that are induced by the original database fluents of the domain. We use ConGolog, a situation calculus based language for modeling and simulating dynamic domains ([DGLL00]). A set of ECA-rules is expressed as a ConGolog program whose execution models are embedded both into its structure and into the semantics of a special ternary predicate Do which serves as an abstract interpreter for rules.

More specifically, we show how ConGolog programs are used to capture transactional behavior and how to use the semantics of these programs in simulating the transactional behavior with the basic relational theories corresponding to various ATMs as background axioms. Next, we introduce further building blocks that are specific to active databases into the specification framework laid down so far. These building blocks include an event logic, a fragment of the situation calculus used to capture and specify event algebras in logic. Then, we formally define the active relational theories and show some of the properties of event logics. After that, we model various execution models as ConGolog programs. The main result here is a set of classification theorems for the various semantics identified. These theorems say roughly which semantics are equivalent and which are not. Finally, we tackle the important issue of standard (e.g. confluence, termination, etc.) and non-standard properties of a set of rules.

1.5.3 Method for Implementing Basic and Active Relational Theories

Thus far, we have been involved in a theoretical development that ultimately yields active relational theories, which are theories of the situation calculus that capture the dynamic of (relational) active databases in the context of database transactions.

Building on this theoretical development, we extend a method for implementing basic action theories in Prolog presented by Reiter in [Rei01] for active relational theories. The justification for Reiter's method lies in a consequence of a fundamental theorem of logic programming due to K. Clark (See [Llo88] for Clark's result). Suppose that we have a first order theory in definitional form, in the sense that all its axioms are formulas of the form $(\forall \vec{x})P(\vec{x}) \equiv \phi$. Clark's theorem says that, whenever a logic program `P` obtained from a definitional theory $\mathcal{T}$ by taking the if-halves of the sentences $(\forall \vec{x})P(\vec{x}) \equiv \phi$ yields the answer "yes" on a sentence ψ, then ψ is logically entailed by $\mathcal{T}$; also, whenever `P` yields the answer "no" on ψ, then $\neg\psi$ is logically entailed by $\mathcal{T}$. The main long-term goal in providing this method is the use of the theories that we study in this thesis as a conceptual model for ADBMSs.

Finally, we model some important aspects of the SQL3 standard for active rules following the guidelines set in the theoretical part of the thesis.

1.5.4 Summary of the Contributions

The main contributions of this thesis are:

Specifying database transactions:

- semantics of various versions of the classical flat transactions using relational theories;
- semantics of closed and open nested transactions using relational theories;
- proof of properties of ATMs by establishing logical consequences of sentences of the situation calculus capturing these ATMs;
- method for simulating the captured ATMs using (Con)GOLOG;

Specifying the representation and execution of ECA rules:

- specification of event algebras in the situation calculus;
- semantics of the following dimensions of active behavior: event consumption modes, rule priorities, and net effects;
- specification of various execution models in the situation calculus, together with their coupling modes: immediate, deferred, and detached execution models;
- classification theorems for some of the execution models.

Method for implementing the specifications:

- implementation theorem for active relational theories;
- abstract interpreter for transactional programs whose background theories are active relational theories for closed nested transactions;
- semantics of SQL3 in the framework of this thesis.

1.6 Overview of the Thesis

The remainder of this document is organized as follows. The next chapter lays down the necessary background information on ADBMS needed to follow the ideas presented throughout the following chapters. We will give a basic introduction to the main components of an active database, particularly to rule languages and execution models, also called operational semantics. Then, we will focus on reviewing some previous work concerned with a formal account of active databases.

In Chapter 3, we will provide the reader with a short and self-contained introduction to the situation calculus; there, we will insist particularly on the notion of basic action theories and give a second order semantics for it.

Chapter 4 will show how to model relational database transactions as theories of a peculiar sort called basic relational theories. There, we will extend the approach of [Rei95] by modeling traditional database transactions — and not only primitive updates — with ACID properties and more advanced transaction mechanisms used in active database systems.

Chapter 5 is devoted to extending the basic relational theories to model the representational component of active behaviors. The new theories we will introduce in this chapter are called active relational theories. Since active databases are closely related to transactions, basic relational theories will be a substantial building block of active relational theories. The new ingredients will be axioms for capturing the aspects of active behavior related to events.

With active relational theories in hand, we represent the execution models of active behavior in Chapter 6 by compiling a set of given ECA rules into a ConGolog program, a rule program, whose structure is constrained according to that of the given execution model. Then, we give the semantics of the execution of a rule program in connection with the execution of a transaction program modeled in ConGolog.

The reader would most probably be curious about how useful the whole theoretical development in Chapters 2 through 6 is. We will satisfy this curiosity in Chapter 7 by giving a simple implementation method for ADBMSs modeled following the guidelines set in Chapters 4–6. We consider Chapter 7 to be an important step towards a methodology for developing active rule systems.

Finally, Chapter 8 summarizes the thesis, states our main results, and points out some open problems and research challenges for future work.

Chapter 2

Background

In this chapter, we present a brief overview of the background our work is based on. First, we give a basic introduction to active databases, mainly following the presentation in [Pat99]; doing so, we focus particularly on the two main components of an ADBMS which are rule languages and execution models, by depicting their various aspects, called dimensions. We then review previous work dealing with a formal account of active databases and show how it is similar to or diverges from work presented in this thesis.

2.1 Fundamentals of Active Database Management Systems

2.1.1 Basic Definitions

Before we proceed dealing with active databases, we first introduce some definitions of basic relational database concepts, following [AHV95] and [Rei84].

To specify particular structures of the data to be stored in a database, data models have been provided. A *data model* consists of a mathematical notation to describe data, and a technique for manipulating them. The former is known as data definition language (DDL), whereas the later isknown as the data manipulation language (DML). In DMLs, two capabilities are provided: answering queries posed to stored data and updating them. The most important data models to date are the network (based on graphs), relational (based on relations), and object-oriented (also based on graphs) models. When first-order logic is used as a mathematical notation for describing data, we have a *deductive data model*; the corresponding DML is the first-order evaluation of logical formulas.

Without loss of generality, in this document we consider the relational data model introduced by Codd ([Cod70]) or its deductive counterpart (without database views) whenever nothing else is stated explicitly. We present the relational model in this section and defer the treatment of the deductive counterpart to the next section. We assume the existence of two disjoint countably infinite sets `att` of attributes and

$\mathbf{relname}$ of relation names. We also assume a countably infinite set $\mathbf{dom}$ called the domain. There is a mapping $\mathbf{Dom}$ from $\mathbf{dom}$ to $\mathbf{att}$ such that, for each $A \in \mathbf{att}$, $\mathbf{Dom}(A)$ is the underlying domain of A.

Definition 2.1 *A* relation schema *(or, simply, relation) is a relation name $R \in \mathbf{relname}$; it is said to have a finite set of attributes. The arity of a relation schema is the cardinality of the set of its attributes.*

Definition 2.2 *Let R be a relation. A set $M \subseteq \mathbf{att}$ of attributes of R is a* key *iff (1) no two tuples of R have all the attributes in M the same, and (2) no set M' such that $M' \subseteq M$ fulfills property (1).*

Definition 2.3 *A* database schema *is any nonempty, finite set DB of relation schemas; that is $DB = \{R_1[U_1], \ldots, R_n[U_n]\}$, where $U_i \subseteq \mathbf{att}$, and $R_i \in \mathbf{relname}$; $R_i[U_i]$ denotes a relation R_i over a set U_i of attributes. Note that U_i may appear more than once in the list $R_1[U_1], \ldots, R_n[U_n]$.*

Definition 2.4 *A* relation schema instance *(or, simply, relation instance) of a relation schema $R[U]$ is a (possibly empty) finite set of tuples $< \tau_1, \ldots, \tau_n >$, where $\tau_i \in \mathbf{Dom}(A_i)$ and n is the arity of R.*

Definition 2.5 *A* database schema instance *(or, simply, database instance) of a database schema DB is a mapping* **I** *with domain DB, such that* **I**(R) *is a relation instance of R for each $R \in DB$.*

Usually, tuples stored in a relation instance are subject to *integrity constraints* which are restrictions imposed on the database instances to satisfy certain properties. For example, we can require in a database instance about humans that no one is aged more than 120 years, and that no one has two id numbers. There are at least three sorts of ICs described in the literature ([CF97]). The first sort is constituted of built-in ICs, which have a fixed structure and meaning for all application domains. The most important of built-in ICs are: *non null ICs*, which specify attributes that cannot be assigned a null value; *primary key ICs*, which specify sets of attributes that are keys, and *unique attribute ICs*, which specify sets of attributes that uniquely identify each tuple of a relation without being a key.

The second sort of ICs are cardinality constraints, of which *referential IC* is a common special case. Cardinality ICs restrict the number of tuples that can participate in a relationship between relations, and referential IC limits the relationship to a one-to-many relationship, the one relation being called the referencing relation and the many relation the referenced one.

Finally, the third sort of ICs is constituted of the *generic ICs*, which are ICs with a generic structure and meaning, thus permitting the expression of arbitrary restrictions that are dependent of application domains. Generic ICs are usually classified into static ICs involving one single database state and dynamic ICs which involve two database states.

Traditionally, DBMSs support built-in and cardinality ICs. Generic ICs are supported in more advanced DBMSs. A consistent database instance is one that satisfies all the ICs. An integrity constraint

IC is said to be verified on a database instance DB iff DB satisfies it; and IC is enforced on DB iff DB is designed in such a way that IC is verified on DB.

Example 2.1 *Let us consider a stock trading database (adapted from [WC96]). The database schema contains the following relations:*

$$price(\textbf{stockId}, price, time),$$
$$stock(\textbf{stockId}, price, closingprice), and$$
$$customer(\textbf{custId}, balance, stockId)$$

The $price$ relation stores information on stock prices and times that are received from an external source. The $stock$ relation reflects both the actual stock prices and closing prices of the previous day. Finally, for stocks that are monitored for some customers, the $customer$ relation indicates for each customer the stock monitored and that customer's actual balance. The explanation of the attributes is as follows:

$stockId$: *string; identification number of a stock*

$price$: *real; current price of a stock*

$time$: *integer; pricing time*

$closingprice$: *real; closing price of the previous day*

$custId$: *string; identification number of a customer*

$balance$: *real; balance of a customer*

Figure 2.1 gives an instance of the database schema above. Database instances are usually presented in the form of tables. We indicate primary keys in boldface in the database schema. We assume for convenience that all values are not null, and, doing so, we avoid dealing with the difficult problem of representing null values ([Rei86]). A generic IC for this database schema is to require that the customer balances cannot drop below zero; another would be to require that all components of a tuple be of the right type.

Information stored in a DBMS can be retrieved by using queries expressed in an appropriate DML. Queries are expressed in two different kinds of DMLs which are: relational algebra, where queries are expressed as an application of special purpose operators such as union, set difference, Cartesian product, projection, and selection to relations; and relational calculus, where queries are expressed as first order logical formulas that tuples in the intended answer must satisfy. Relational algebra is closely related to Codd's original relational model, whereas the relational calculus is more related to its deductive counterpart.

In addition to querying, DMLs are also used for updating the content of databases. Traditional database update operations are the well-known constructs *insert*, *delete*, and *update*.

price		
ST1	$100	100100:4PM
ST2	$110	10010:9AM
ST3	$50	100100:1PM

stock		
ST1	$100	$100
ST2	$110	$100
ST3	S50	$60

customer		
Sue	$10000	ST1
Zang	$5000	ST2

Figure 2.1: An instance of the stock trading example

A *database management system* is a software system for efficiently creating and manipulating data consistently stored in database instances according to some specific data model. It implements both the DDL and the DML corresponding to the specific data model.

2.1.2 Dimensions of Active Behavior

Many DBMSs have been proclaimed to be active. ADBMSs have often been defined as systems expressing reactive behavior ([DGG95]). However, the authors of these systems do not agree on the precise explanation of the term ”active” since they have proposed system architectures before proposing formal foundations for ADBMSs. This is why [DGG95] proposes some mandatory characteristics that a system should exhibit to be called active. There, it is observed that many existing ADBMSs contain common features that cannot simply be generalized since this would result in meaningless statements. They also point out that a definition of ADBMSs can not simply be abstracted from their intended application domains. Instead, they propose general as well as specific features that characterize ADBMSs, and that we call dimensions of ADBMSs after [PDW$^+$93].

An ADBMS captures the (re)active behavior of application domains. A system embodies reactive behavior if it offers the possibility of automatic actions in response to relevant happenings called *events*. A reactive behavior involves an association of events with actions that should be performed automatically by the system once these events occur; it also involves a way of detecting the occurrences of events; it finally involves a specification of how the system should perform the actions associated with the events that may have occurred.

To be an ADBMS, a DBMS *should* (i) necessarily be a DBMS; (ii) support mechanisms for defining and managing ECA-rules by providing syntactic means for defining events, conditions, and actions; (iii) support rulebase evolution; (iv) have a well-defined execution model capable of detecting event occurrences, evaluating conditions, and executing actions, having a well-defined execution semantics, and incorporating some user defined or predefined conflict resolution mechanism.

An ADBMS may *optionally* contain a representation of information on ECA-rules using their data model. Moreover, it may support a clear programming environment providing a development process and tools like a rule browser, a rule designer, a rulebase analyzer, a debugger, a maintenance facility, a trace mechanism, and some tuning capabilities.

An *active database management system* is a database management system extended with at least the mandatory dimensions of active behavior.

In the remainder of this section, we focus on spelling out details of the mandatory dimensions of ADBMSs. Of the four conditions for being an ADBMS stated above, condition (i) is obvious and condition (iii) is rather of managerial nature, such that we are left with the following main components of an ADBMS:

- a reactive model, also considered a knowledge model ([Pat99]), for defining events and associating them with actions; and

- an execution model for monitoring events and reacting to detected events.

For short, we will refer to these components as those of active databases and we will call them the *active behavior* of the corresponding active database. We will also say later that an active database is characterized by these two components and by some other ones. Below, we review each of the two components in turn. Some managerial aspects of rules sets will still be considered in Chapter 6, where we will introduce them when needed.

COMPONENT	DIMENSIONS / VALUES
EVENT	TYPE: primitive, complex SOURCE: update operation, exception, external, clock, transaction GRANULARITY: tuple- or set-oriented ROLE: mandatory, optional, none
CONDITION	ROLE: mandatory, optional, none CONTEXT: DB_T, $BIND_E$, DB_E, DB_C
ACTION	TYPE: update operation, rollback, information, external, do instead CONTEXT: DB_T, $BIND_E$, $BIND_C$, DB_E, DB_C, DB_A

Figure 2.2: Overview of mandatory dimensions of the knowledge model of ADBMSs

2.1.3 Knowledge Model: ECA-rules

The knowledge model is expressed in ECA-rules. Figure 2.2, in which we restrict a corresponding figure in [Pat99] to relational models, summarizes the mandatory dimensions of the knowledge model. Rules have three parts: event, condition, and action. It must be noted that other dimensions such as rule execution priorities and coupling modes between event detection and condition evaluation may also be incorporated in the syntactic specification of the knowledge model, although they belong in fact to the execution model. Users, applications, or DB administrators define rules, using an extension of the DDL called rule language. Other terms used synonymously for rules in ADBMSs are: ECA-rules, triggers, monitors, alerters, or production rules.

Note that we can tentatively write an ECA-rule in the form

$$EVENT \ \& \ CONDITION \rightarrow ACTION,$$

which informally means: if the event specified by $EVENT$ occurs and the evaluation of the condition part $CONDITION$ yields true, then the system executes the action part specified by $ACTION$.

Other forms of rules exist. In deductive DBMSs, logic programming rules are used to provide a recursive definition of views. A deductive database rule is a Condition-Condition rule of the form

$$CONDITION_1 \rightarrow CONDITION_2,$$

where $CONDITION_1$ is a conjunction of literals and $CONDITION_2$ is an atom. Usually, deductive database rules are written in the form

$$A \leftarrow L_1, \ldots, L_n, \ m \geq 0.$$

However, much of the strategies developed for efficient rule processing in response to queries on recursive views are inadequate for processing active rules. This is why work has been done in extending deductive DBMSs with active behavior, in implementing deductive capabilities using ADBMSs, and in integrating both approaches (See the section on related work).

Another widespread form of rules are expert system rules. They are Condition-Action (CA) rules of the form

$$CONDITION \rightarrow ACTION.$$

Thus they are just a special case of and are therefore subsumed by ECA-rules. However, the lack of "operation-specific behavior" in CA rules have favored ECA-rules in the DB community, although the former are more declarative than the later ([WC96]).

Using an active database rule language, users syntactically specify the desired knowledge model of a database.

Definition 2.6 (Generic syntax of ECA-rules) *An ECA rule is a syntactical construct of the form*

$$\textbf{ON}\ Event\ \textbf{WHEN}\ Condition\ \textbf{THEN}\ Action$$

Each of the parts of a rule is specified using a specific language: an event language for events, a condition language for conditions, and an action language for actions.

Event Languages. What triggers a rule is an event. An event is a "happening of interest" that occurs instantaneously at specific time points; it is to be distinguished from an event occurrence. An event occurrence is a tuple of the form $< pe, eid >$, where pe is a primitive event and eid is an event identifier, respectively. A rule is *triggered* if its event part matches an event occurrence.

Examples of primitive events are data modification operations, data retrieval operations, temporal specifications, and application-defined events. These are *primitive events*. *Composite events*, that is, combination of primitive events or other composite events, are also considered using operators such as logical junctors, sequences, or temporal qualifications.

The problem of specifying composite events in active databases has been recognized as non-trivial ([CM91], [GJS92], [GD94], [Zan95]). Proposals for solving it can be classified in three groups. The first group uses regular expressions and context-free grammars to specify complex composite events (see, e.g., [GJS92]). The second group uses graph-based methods to specify them ([CM91], [GD94]). Finally, the third group use a logic-based approach (for good examples of this approach, see [GJS92], [Zan95]). We briefly examine work in [GJS92] as a representative of the first and the last approaches and do not consider the second approach which uses techniques similar to those used in the first one.

To specify primitive and complex events, Gehani et al. ([GJS92]) use event expressions. An event expression E is defined as a mapping from an event history h to an other event history h' containing only those event occurrences of h at which the events specified in E happen. An empty history is denoted by $null$. Formally, an event expression E is a function $2^h \rightarrow 2^h$, where $E[h] \subseteq h$. $E[h]$ denotes the application of E to h. An event occurrence $e \in h$ is said to satisfy E iff $e \in E[h]$;E is said to take place at e. Any two event occurrences e_1 and e_2 are simultaneous if their eids are identical.

Event expressions contain constructs like $NULL$, which denotes the null event, $E_1 \wedge E_2$, which denotes the conjunction of two events E_1 and E_2, $!E$, which denotes the negation of an event E, etc. Expressions are given a precise semantics; for instance, we have: for any E, $E[null] = null$; $NULL[h] = null$; for a given primitive event pe, $pe[h]$ is the maximal subset of h containing event occurrences of the form $< pe, eid >$; $E_1 \wedge E_2[h] = E_1[h] \cap E_2[h]$; $(!E)[h] = (h - E[h])$; etc.

Gehani *et al.* introduce some more operators as abbreviations to make the specification of composite events easier. They show that event expressions are equivalent to regular expressions

Events in an ADBMS are also classified into tuple-oriented and set-oriented, depending on whether the granularity of the event is a single entity or an entire set thereof. This distinction is inherited by rules.

Events are sometimes classified in physical events on the one side, and in logical events on the other side ([FT95]). A physical event is the physical occurrence of some activity. A logical event is the effect of that activity. A system may support one of these classes of events or both. An advantage of physical events over logical ones is more fine-grained control of the behavior of rules with respect to triggering operations. However, logical events have the advantage of being more declarative.

The source dimension specifies the origin of an event. Possible alternatives for relational databases are: update operations $insert$, $delete$, and $update$; exceptions raised by application programs; points in time; happenings external to the database world (e.g. sensor readings); and transaction commands such as $begin$, $commit$, and $rollback$.

Finally, there is a role dimension specifying whether the event part of ECA rules should always appear (*mandatory*), whether it may be omitted (*optional*), or whether it should not at all appear (*none*). The *none* role means that all rules are CA rules. Usually, most systems adopt the mandatory role.

Condition Languages. Generally, database predicates such as conditions written in a SQL *where* clause, restricted predicates, database queries, and application-defined conditions like procedures are used in condition languages.

Roles for conditions are defined in a way similar to those for events, except that the most widespread role is *optional*. The *none* role means that all rules are EA rules.

The *context* dimension specifies the database state in which the condition is evaluated: DB_T is the database state at the beginning of the current transaction; DB_E is the database state at the moment of event occurrence; and DB_C is the database state at the moment of condition evaluation.

If the language has parameterized events, a mechanism for referencing bindings of events must be included in the condition language; this mechanism determines the *context* value $BIND_E$ of conditions.

Action Languages. Actions involve a *task* to be performed. They include: *update operations*; transactions commands like *rollback* and *commit*; actions to *inform* users of specific database situations of interest; application procedures that may involve *external* calls; and alternative actions to *do instead* of the actual action associated with the event part of the rule.

The *do instead* task is an instance of the event/action link which is the connection between the action execution and the triggering action. From this point of view, three variations of rules are possible. In the first case, called *after* rules, the action execution of the rule is normally performed after the action associated with the triggering events have been performed; in the second case, called *before* rules, the action execution of the rule is performed before the triggering action; and in the third case, called *instead of* rules, the action execution is performed instead of the triggering action.

The *context* dimension specifies the database state in which the action is performed. In addition to the values of the context dimension of conditions, actions have two more values: DB_A and $BIND_A$. The value DB_A is the database state at the moment of action execution evaluation; $BIND_A$ is the mechanism needed to pass bindings from events and conditions to the action. Examples of values for $BIND_A$ are: parameter mechanism passing data returned by a query or a procedure, or passing data satisfying the predicate representing the condition part of the rule.

An important distinction is made between actions that are atomic and those that are interruptible. An action is atomic if the rule that responded to the triggering event cannot be suspended until the action it is executing is finished; it is interruptible if it can be.

Active systems keep track of transition values during run time. A transition is a DB state change during the evaluation of the conditions and the execution of actions. A rule language may include a mechanism for referencing transitions. There are explicit and implicit mechanisms. Explicit ones are the use of parameterization, and special keywords like *inserted*, *deleted*, *updated*, which are commonly called *delta* relations. Further reserved words are *new*, for denoting the new value of the modified item, and *old*, for the old value. Implicit mechanisms follow a general principle: in a triggered rule, references to some item D in the rule's condition or action is made implicitly to a transition value of D.

2.1.4 Execution Model

The execution model specifies the run-time behavior of a set of active rules. Although the details vary from system to system, the general pattern of the execution model is the following: The defined set of rules is monitored by the system for relevant events; then, for any given rule that is triggered by some event and before its action can be executed, its condition is checked, and if true, its action is finally executed. When the action part of a rule is executed, the rule is said to *fire*. A triggered rule whose condition part is evaluated to $true$ is said to be *activated*.

The following is a generic execution model algorithm:

```
while there are triggered rules do
    1. select a triggered rule R
    2. evaluate R's condition
    3. if R's condition is true then execute R's action
```

To obtain a full rule execution semantics, this simple model is completed and extended with additional execution features.

Figure 2.3, which again is a restriction of a corresponding figure in [Pat99] to relational models, summarizes the mandatory dimensions of the execution model. However, before presenting these dimensions, we give some details on how events are monitored.

Event Monitoring. Occurrences of events must be detected by the system in order to trigger rules that have these events specified in their event part. An event detector has as task to check which simple event

DIMENSIONS	VALUES
EVENT/CONDITION COUPLING	immediate, deferred, detached
CONDITION/ACTION COUPLING	immediate, deferred, detached
CONSUMPTION MODE	none, local, global, ...
GRANULARITY	tuple-, set-oriented
NET EFFECT POLICY	yes, no
PRIORITY	none, absolute, relative, dynamic, numerical
SCHEDULING	sequential, concurrent
ERROR HANDLING	backtrack, ignore

Figure 2.3: Overview of mandatory dimensions of the execution model of ADBMSs

occurs or which composite event occurs as a consequence of the occurrence of a primitive event. A naive method of detecting composite events can be described as follows:

```
while there are occurring events do
    1. pick one event $e$ that has occurred
    2. determine which composite events $e$ is part of
    3. For each composite event determined in Step 2,
       check whether all other events that are part of it have already occurred
endwhile
```

The method is hopelessly inefficient, since information about all past events must be maintained and inspected on each new occurrence of a primitive event. More efficient methods exist ([CM91], [GJS92], [GD94]). For example, an incremental event detection technique based on finite automata is described in [GJS92]. The input to the automaton are the primitive events occurrences from the event history. The automaton is fed such occurrences as they happen, in the order of their event identifier. Each event occurrence causes the automaton to change its state. When the automaton enters an accepting state, the composite event implemented by the automaton is recognized as taking place at the primitive event last read, and the corresponding rule is triggered. Given an event expression E, a corresponding automaton is inductively built on the structure of E.

A problem that may occur is that of exponential blow-up of states when constructing cross-products of automata. Such cross-products are needed for example in the construction of automata for conjunctions of event expressions. Another problem, pointed out in [GD94], is a gap that occurs by using two mechanisms, one to define events (regular expressions), an other to detect their occurrences. With this remark, Gatziu recognizes the need of using a unique “mechanism” to bridge the gap between event definition and event detection. In this thesis, we will be bridging even larger gaps than this one, for example the one between knowledge model representation and execution model specification.

Execution Models and Transactions. In [MD89], the notion of coupling modes is introduced to describe the synchronization of rule triggering, condition evaluation, and action execution; it also describes the relationship between rules and transactions. Generally, the concept of “coupling mode” is now used in the former sense.

There are two kinds of coupling modes: the Event-Condition and the Condition-Action coupling modes. They describe the temporal relationship between triggering events and condition evaluation, and between condition evaluation and action execution, respectively. Two possible values are possible for both dimensions: *immediate coupling* and *delayed coupling*. In the former setting, the condition is immediately evaluated upon termination of the triggering events; in the later one, it is delayed until some defined time point. There are two sub-cases of the later setting: *deferred* and *detached*. In deferred mode, conditions are evaluated within the same transaction, whereas in detached mode they are evaluated in a separate transaction. Details of these modes are explained later in details.

The treatment that the event triggering a rule may undergo is called *event consumption mode*. Here, two issues that are relevant are the scope and the time of event consumption. The first issue amounts to the question of how far the processed events retain their triggering capabilities, and the second one amounts to when events are consumed. Three scopes of event consumption are possible: *no* consumption (meaning that processed events remain capable of triggering further rules), *local* consumption (meaning that they no longer can trigger the processed rule), and *global* consumption (meaning that they no longer can trigger any other rule). Two kinds of time consumption are possible: either before condition evaluation (rule consideration time), or after condition evaluation (rule execution time).

The *transition granularity* determines the relationship between event occurrences and rule instantiations; it can be *tuple-oriented* or *set-oriented*. In a tuple-oriented granularity, the action execution is coupled with each single tuple of the DB triggering the rule. In a set-oriented granularity, however, the action execution is performed once for all triggering instances.

The *net effect policy* indicates whether and how the net effects of events should be taken into account rather than their individual occurrence. Such net effects are accumulating only changes that really affect the database; sample policies are: (i) if a record is first updated and then deleted, only the deletion is retained; (ii) if a record is first inserted and then updated, an insertion of the updated record is retained; (iii) if a record is updated many times, the composition of all updates is retained as a single update; (iv) finally, if a record is deleted after being inserted, this amounts to nothing having happened.

It is possible that many rules are triggered at the same time. Thus some kind of conflict resolution mechanism is required to select one rule for execution from the set of triggered rules. The selection may be arbitrary or based on *priorities* assigned to rules. These priorities can be relative if declared between pairs of rules, absolute if assigned to rules as a total order, numerical if they are a mapping from integers to rules, or dynamic if they are dynamically assigned to rules at execution time.

When multiple rules are triggered, a *scheduling mechanism* should be used for rule firing. Rules could be executed sequentially or in parallel. However, while avoiding conflict resolution issues, concurrent execution must include some concurrency control mechanism.

Since rules are processed in the context of DB transactions, the relationship between both should be presented in a more detailed way than done up to this point. Recall that this relationship is established, as stated earlier, through *coupling modes* which prescribe when a rule condition test or action has to be performed or executed.

To define a (classical) transaction, we first distinguish between atomic updates and DB actions enforcing the transaction semantics. An **atomic update** is any one of the expressions $R\_insert(\vec{x})$, $R\_delete(\vec{x})$, and $R\_update(\vec{\imath}, \vec{o})$. Intuitively, $R\_insert(\vec{x})$ ($R\_delete(\vec{x})$) means the insertion (deletion) of the tuple $\vec{x}$ into (from) the relation R, and $R\_update(\vec{\imath}, \vec{o})$ means that some update is performed on the tuple $\vec{\imath}$ of R, yielding the output tuple $\vec{o}$.[1]

A DB action enforcing the transaction semantics is a user or application generated action belonging to the set $\{Begin, Commit, Rollback\}$.

Now, we can define an **atomic transaction** AT as a sequence $[upd_1, \ldots, upd_n]$ of atomic updates, where upd_1 is always $Begin$ and upd_n is either $Commit$ or $Rollback$. For example, we may have the sequence

$$[Begin,\ price\_insert(IBM, 100, 500PM),$$
$$customer\_insert(SMITH, 10000, IBM),$$
$$price\_update((IBM, 100, 5PM), (IBM, 110, 530PM)),\ Rollback],$$

with the obvious meaning. Finally, we define a **transaction** T as a sequence $AT_1, \ldots, AT_n$ of atomic transactions.

The classical transaction theory ([BHG87]) distinguishes between transactions and transaction programs. A transaction, as introduced above, can be viewed as a particular execution (or execution trace) of a transaction program, usually written in an Algol-like language. In the sequel of this section, we deal with transactions, not transaction programs.

[1]This specification of transactions is like the approach described in [FT95]. However, unlike our atomic updates that are primitive modification operations, an atomic update in [FT95] is more generally defined as a first-order macro of the form

$$upd(\vec{\imath}, \vec{o}) \equiv \Phi_{upd(\vec{\imath}, \vec{o})},$$

where $\Phi_{upd(\vec{\imath}, \vec{o})}$ is a first-order formula or a SQL statement specifying the update to be undertaken.

Now the relationship between rule execution and transaction semantics can be enlightened in different scenarios, given the definitions above.

In the first scenario, called *immediate execution*, triggered rules are fired after each atomic update of every atomic transaction. Thus rule execution is interleaved with the execution of atomic updates.

In a second scenario, called *deferred execution*, the condition evaluation and the action execution of triggered rules occur at the end of atomic transactions, within the *same transaction*; this means that the rule execution module is executed each time an atomic transaction has been executed, taking control over the transaction and giving control back to it once it successfully terminates or aborting it otherwise.

An alternative to this scenario is the case where condition evaluation and action execution are processed in two different transactions. Two situations are possible: the separate transaction is a nested transaction of the original transaction (this refinement is found, e.g., in HiPAC ([WC96])); or it is an independent transaction (as an example, see Ode ([WC96])). In both cases, a mechanism for concurrency control that is not part of the rule execution engine is needed.

Properties of Rule Sets. Once a set of rules has been written, termination/non-termination of the rule processing algorithm is one of the major issues to deal with. Informally, a set of rules is guaranteed to terminate if for all user transactions and initial database states, rule processing reaches a state where no further rules are triggered. Non-termination occurs in particular when there is a direct or indirect cycle in the triggering chain. Several methods for handling this issue exist: the burden of the detection is up to the programmer; the system kernel fixes some upper limit for triggering cycles; the rule language includes some restrictions ruling out the possibility of non-termination ([BCW93a], ([BCP95], [AHW95]).

Another property of importance is confluence. A set of rules is confluent if the outcome of its execution is independent of the order of rule consideration.

Finally, determinism is often desirable for rule sets. A set of rules is deterministic if the outcome of its execution always result in a unique database set regardless of the order of execution of rules.

2.1.5 Architectures and Implementations

Many research and some commercial ADBMSs exist, using a variety of architectures: layered, built-in, and compiled ([Wid94], [WC96]) (For a summary of these systems and the semantic choice they have adopted for the various dimensions of active behavior, see [FT95]; for an in depth coverage of some of them, see [WC96] and the 1994 RIDE special issue on active databases).

A layered architecture is composed from two layers: a traditional DBMS, and an active module built on top of it. A built-in architecture is a construction tightly coupling both the passive and active components of the ADBMS; here, there is a single kernel exhibiting both a traditional and an active behavior. Finally, a compiled architecture compiles active rules included in an application into their effects; thus, it requires no run-time event monitoring, nor does it require a run-time rule processing.

Historically, everything began with the project HiPAC ([MD89]), which has been very influential in the research community. The project Starburst, developed at IBM, is an example of a mature relational active database. As representative for an object-oriented projects, Chimera and Ode are well known. As an example of a project related to the deductive database data model, A-RDL can be mentioned. Most of the research projects are implementing a layered architecture (see [WC96] and [Pat99] for a detailed description of these systems).

In implementing ADBMSs, specific issues are involved. The most important ones are: rule management through a command language, concurrency control, crash recovery, authorization, error handling, rule tracing, efficient condition evaluation, rule compilation, and interaction with applications. Some of these issues such as rule management and rule tracing amount to providing powerful rule programming tools. Others are to be included in the system kernel to increase its functionality. Conceptually, they are not specific to ADBMSs; they are desirable of every programming system.

2.2 Work on Database Transactions

2.2.1 ACTA

Chrysanthis and Ramamritham ([Chr91],[CR94]) present a framework called ACTA which allows to specify effects of transactions on objects and on other transactions. Our framework is similar to ACTA. In fact, we use the same building blocks for ATMs as those used in ACTA. However, the reasoning capability of the situation calculus exceeds that of ACTA for the following reasons: (1) the database log is a first class citizen of the situation calculus, and the semantics of all transaction operations – $Commit$, $Rollback$, etc. – are defined with respect to constraints on this log. Nowhere have we seen a quantification over histories in ACTA, so that there is no straitforward way of expressing closed form formulas involving histories in ACTA. (2) Our approach goes far beyond ACTA as it is an implementable specification, thus allowing one to automatically check properties of the specification using an interpreter. To achieve this goal, we prove an implementation theorem that justify a Prolog implementation of the specifications. Finally, (3) although ACTA deals with the dynamics of database objects, it is never explicitly formulated as a logic for actions.

2.2.2 Transaction Logic

Bonner and Kiefer present a *transaction logic* ($\mathcal{TR}$) that includes a model theory, and a sound and complete SLD-like proof theory ([BK92]). The execution of a transaction ψ is described by an executional entailment which intuitively means that, given transaction axioms gathered in $\mathbf{P}$, the execution of the transaction ψ leads the initial database $\mathbf{D}_0$ to the final database $\mathbf{D}_n$ through a sequence of intermediate states $\mathbf{D}_1, \cdots, \mathbf{D}_{n-1}$.

Like our situation calculus based transaction language, $\mathcal{TR}$ allows the formulation of complex transactions and bulk updates; and its notion of executional entailment corresponds to the logical entailment in the situation calculus. However, $\mathcal{TR}$ differs from our language. First, $\mathcal{TR}$ is both update- and sentence-centered; it allows not only elementary updates, but also additions and removals of rules. By contrast, our approach is solely update-centered.[2] By restricting our concern to updates, we avoid invoking a revision theory. Second, unlike $\mathcal{TR}$, elementary updates in the situation calculus are not predicates, but first order terms instead. Third, unlike $\mathcal{TR}$, which deals with updates at the physical level, the situation calculus deals with updates at the virtual level. In fact, this limitation can be overcome in the situation calculus by progressing the initial database after each elementary update execution ([Rei01]). Finally, unlike the situation calculus , $\mathcal{TR}$ does not consider the frame problem.

To do full justice to $\mathcal{TR}$, one should distinguish between the full logic and its Horn fragment. Many of the limitations mentioned above concern the Horn fragment, not the full logic. For example, [San00] shows how the full logic is used for reasoning about arbitrary elementary and complex actions (not just tuple insertions and deletions) and addresses the frame problem.

2.2.3 Statelog

In [LML96], Ludäscher *et al.* specify nested transactions in a logic of database state change called Statelog that includes a model theory; like ours, this logic allows the formulation of complex transactions, is update-centered, and considers invariance in state changes. However, there are differences between their approach and ours. First, while we allow only primitive updates that are first-order terms, Statelog expresses actions corresponding to $Begin$, End, $Commit$, $Rollback$, etc, as relations. In this respect, Statelog is similar to $\mathcal{TR}$. Second, unlike in Statelog, we appeal solely to the semantics of second order predicate logic; we perceive no need of a special-purpose semantics to account for models of database transactions. Third, unlike Statelog, we do not deal with updates at the physical level. Finally, unlike the situation calculus, Statelog accounts for nested transactions.

2.2.4 Lynch Automata

An early work by Lynch *et al.* reported in [LMWF88] and [LMWF94] describes an automaton-based theoretical framework for reasoning about atomic transactions. This approach is in spirit close to our general philosophy of providing a framework for reasoning about transactions. It shares with our situation calculus based model a common ground. First, both approaches view the execution of a transactional behavior as a sequence of actions. Second, both include the idea of building a complex transactional behavior from existing, simpler ones by using well defined combination constructs. However, our approach

[2]Update-centered approaches specify explicit update operations in the update language; and sentence-centered approaches allow for updates with arbitrary sentences ([Rei95]).

models a transactional behavior in a very different way: while we model these behaviors as ConGolog programs, the approach by Lynch *et al.* models them as automata. We believe that automata used in this approach do not offer the same flexibility that predicate logic does. The situation calculus does in fact have connections with automata in a different way than the one developed by Lynch *et al.*: the decision problem for fragments of the language can be related to automata on infinite trees as done in [Ter99]. Here, automata are used to provide a semantics for (fragments of) the modeling language (i.e. the situation calculus) and not as the modeling language itself.

2.3 Work on Knowledge and Execution Models of Active Databases

The present section aims to review work done in formalizing the knowledge and execution models of active databases. We will be focusing on the general patterns as distinguished from their actual instances or implementation, with a view towards mentioning only work that has not focussed exclusively on some few and special features thereof.

2.3.1 Classification

The formalization proposals examined here are classified using the formalism they use. In general, the formal specification of active behavior can be classified in two groups. The first group uses the denotational semantics ([Ten76]) as formalism to describe the execution models of actual ADBMSs ([Wid92], [CC95], [RPS99]). In the second group, logic is used, especially in form of first-order logic, event calculus, or situation calculus ([Zan93], [Zan95], [LHL95], [BL96], [BLT97],[FWP97]). We also will cover other approaches that exist, but do not fit into the present classification, due to their limited scope.

2.3.2 Denotational Semantics-based Formalization

In [Wid92], Widom formally specifies the execution model of the Starburst ADBMS, using denotational semantics. A denotational semantics is generally defined as a mapping of the syntactic constructs of a formal language into an abstract meaning formalized in a suitable mathematical model ([Ten76]). This mapping is represented as a *meaning function* taking programs of the language as an input value and producing the function computed by those programs. In the context of an ADBMS, a denotational semantics is a meaning function. This takes a set of active rules as an input and produces a function which maps the set of database states and the set of allowed database modification operations into a new set of database states.

The semantics described in [Wid92] assumes both deterministic and non-deterministic selection of which rule to consider first when more than one rule is triggered. It is divided into three parts. The first describes the different domains of various help or supporting functions used to define the meaning function $\mathcal{M}$; the second defines these supporting functions themselves; and the third defines the meaning function $\mathcal{M}$. Here, $\mathcal{M}$ has a set of rules $Rules \in 2^{\mathcal{R}}$ and a rule ordering $o \in \mathcal{O}$ as input, where $\mathcal{R}$ is the set of active rules, $2^{\mathcal{R}}$ is the powerset of $\mathcal{R}$, and $\mathcal{O}$ is the set of rule orderings. The meaning of o and $Rules$, denoted by $\mathcal{M}[Rules, o]$, is defined as a function $\phi : \triangle \times \mathcal{S} \longrightarrow \mathcal{S} \cup \{\perp\}$, where $\triangle$ is the domain of sets of database changes, $\mathcal{S}$ is the domain of database states, and $\perp$ denotes the non-termination of rule execution. Putting it all together, Widom formally defines $\mathcal{M}$ as a function $2^{\mathcal{R}} \times \mathcal{O} \longrightarrow \triangle \times \mathcal{S} \rightarrow \mathcal{S} \cup \{\perp\}$.

To our knowledge, this proposal is the first one to have succeeded in giving a formal foundation to active rules. For this reason, it is widely referred to in the literature. Unfortunately, though Widom argues that her work is providing a denotational semantics for the Starburst rule language, it is in fact providing only a specification of the execution model of the Starburst ADBMS, abstracting from its rule language.

Coupaye and Collet present another attempt to use denotational semantics for formally specifying ADBMSs ([CC95]). They give a formal specification of the execution model of the NAOS system developed at the university of Grenoble. In NAOS, a rule condition can be a query expressed in O_2SQL, which is an object-oriented version of SQL, and a rule action can be a O_2C program. Both O_2SQL and O_2C are languages in the style of Heraclitus (see Section 2.3.4): they may be used to express the formal semantics of ADBMSs.

Coupaye and Collet are essentially applying the set of semantic functions developed by Widom on the NAOS system, taking the object-oriented character of NAOS conditions and actions into account. For example, they extend the definition of the valuation function for the net effect of rule actions by incorporating the effect of object-oriented related operations such as object creation, or method invocation. They also add some new domains for supporting functions that valuate new features found in NAOS such as the dynamic activation or deactivation of rules.

Like the denotational semantics approach, we give a declarative semantics that allows one to reason about the behavior of active rules. Unlike it, ours is a formalization that can be generalized without difficulties, thus allowing a comparison between the various execution models that exist.

2.3.3 Logic-based Formalization

The idea of using first-order logic as a mean to formalize ADBMSs has been advocated first by Widom and Zaniolo ([Wid93], [Zan93]). Some other researchers have tried to give a logical account of the semantics of ADBMSs ([Wid93], [Zan93], [Zan95], [HD93], [FT95], [LHL95], [BL96], [BLT97], [FWP97]). Most of these researchers have been motivated by the existence of a well developed semantics for deductive rules ([VEK76], [Llo88], [GL88], [VGRS88]).

Databases have been formalized in logic now for about 20 years (for a description of the main contributions to the use of logic in databases, the actual state of the art in commercial implementations, and future trends, see [Min96]). We have already pointed out that Reiter (Reiter[1984]) provides the first of these formalizations.

Deductive databases are an extension of relational databases formalized *à la* Reiter. Formally, we have the following

Definition 2.7 *A deductive database is a theory composed of a set of ground assertions (extensional database) and a set of axioms in the form of rules (intensional database) of the form*

$$A \leftarrow L_1, \ldots, L_n,\ n \geq 0,$$

where A is an atom and L_i are literals, for each i.

The name ”Datalog” is also used to refer to databases of this form. Datalog programs are function-free logic programs. Concepts playing an important role in relational databases like integrity constraints are also formalized in logic (for an account, see [Min96]).

Deductive databases were generalized in many ways. One extension was to allow recursion in rules. The main contribution here is a good understanding of how resolution and fixpoint techniques are related to each other and how rewriting techniques make search space of the later identical to the earlier. Another extension permits negated atoms in the body of rules. Here, main results are the development of the theory of stratification for databases without recursion through negated atoms and the definition of semantics for databases having recursion through negated atoms. The best known examples of the later are the well-founded semantics of Van Gelder, Ross, and Schlipf ([VGRS88]), and the stable semantics of Gelfond and Lifschitz ([GL88]).

Considering semantics of deductive databases, some researchers attempt to provide similar semantics to ADBMSs ([Zan93], [Zan95], [HD93], [LHL95],[FWP97]). They feel a need for combining active and deductive DBMSs into a single and reconciled paradigm. This need is being addressed in different ways. Widom argues that deductive rules could be naturally extended to run on active DBMSs ([Wid93]). Zaniolo, Harrison and Dietrich, and Ludäscher *et al.* state that some syntactically and semantically constrained active rules would run on deductive DBMSs ([Zan93, Zan95], [HD93], [LHL95]). Finally, Fernandes *et al.* advocate that active and deductive DBMSs have no intrinsic similarities with respect to their operational semantics, but they can be integrated into one hybrid system ([FWP97]).

Some other researchers are proposing logic-based formalizations that do not directly address the issue of combining the active and deductive DBMSs into a single and reconciled paradigm. Picouet and Vianu address expressiveness and complexity questions such as: the relevance of active features, their impact on the expressive power and the complexity of the systems, and the simplification of execution models and their equivalence ([PV95, PV97]). Baral and Lobo develop a situation calculus-based language called $\mathcal{L}_{active}$ to describe actions and their effects, events, and evaluation modes of active rules ([BL96, BLT97]). Finally, Fraternali and Tanca address the problem of giving a formal semantics based on formal concepts that captures most of the features of known ADBMSs in the similar way fixpoint semantics captures the deductive nature of deductive DBMSs ([FT95]). The remainder of this section is devoted to examining some of these proposals.

Unifying Semantics for Active and Deductive Databases. Widom ([Wid93]) claims that deductive rules and active rules have no obvious separation between them; rather they form two different paradigms at the ends of a common spectrum of rule languages. She defends the view that both paradigms could be unified by constraining DRs to run on an AR engine. However, it remains to be seen how such a constraining process could be achieved in a general and formal way, and not only in some particular case, as the author claims.

Widom's claim is an extreme viewpoint in the debate about whether deductive and ADBMSs could be reconciled. Zaniolo on one side, and Ludäscher *et al.* on the other side are challenging this claim.

Zaniolo ([Zan93, Zan95]) defends the thesis that both active and deductive DBMSs have an underlying conceptual unity. However, he points out that a complete integration of deductive and ADBMSs appears to be a difficult task. The difficulty lies in the lack of a common semantics. Though many semantics have been proposed for both paradigms, there are still inadequacies in their conceptual foundations. On the one side, most of the proposed ADBMSs retain their own operational semantics. On the other side, the semantics proposed for deductive DBMSs can not be easily extended to non-monotonic constructs like negation and database updates. Zaniolo mentions the known advances with respect to the semantics of negation like local stratification, and stable and well-founded models. To him, the proposed logic-based semantics have serious drawbacks: some are not simple and intuitive enough to be grasped by a normal programmer; others are inefficiently implementable.

Zaniolo's proposal is a new kind of stratification called *XY-stratification* that allows non-monotonic constructs in recursive rules ([Zan93]). He uses the notion of XY-stratification to give a formal semantics for updates and triggers in databases. He claims that his approach has the advantage of possessing a formal logical semantics that is simple, intuitive and constructive. Other advantages claimed are the efficiency of its implementation and the simplicity of its concrete semantics, that is, one that a programmer can grasp without understanding the corresponding formal and abstract semantics.

In [Zan95], the behavior of active rules is syntactically captured by rules called *action request rules* written in Datalog$_{1S}$ in which predicates are allowed to have a stage (integer) argument ([BCW93b]). Events are represented by the entries of delta relations. Conditions are represented by queries on the current content of the database. Finally, actions are introduced as action requests. Events and conditions form the body of the rule while actions are situated in the head. In addition to the events and conditions, the body of an action request rule also contains a goal ensuring that the events triggering the rule have durable changes.

Zaniolo first provides a way of defining rules with updates in their heads by re-writing them into equivalent XY-stratified programs that are update-free. Thereafter, he gives a semantics for these programs that is essentially based on the extension of classical fixpoint semantics for deductive DBMSs; it is used to design a specification language for expressing triggers and event detection in an ADBMS.

In general, Datalog$_{1S}$ programs run under the stable models ([GL88]); they either terminate or get ultimately periodic. Fortunately, it can be shown that durable-change semantics guarantees by construction that the computation does not fall into a cyclic behavior.

To summarize, the author provides a new semantics for the problem of modeling updates in deductive DBMSs. The semantics turns out to support event-driven rules as a special case. His approach shows how to keep the classical deductive database engine while simulating active rules by re-writing them as deductive rules with updates. The obvious advantage of the approach is the possibility of reusing old components of a deductive DBMS without major changes. Zaniolo is one of the first researchers who pointed out the need to unify deductive and ADBMSs paradigms. In addition, his approach provides a semantic account for an active rule as an unfragmented unity. However, we stress one drawback in the approach: it is not clear whether all aspects of the active behavior like coupling modes or operational semantics of active rules can be simulated within a pure semantics for deductive database updates.

Ludäscher *et al.* propose a framework that bridges the gap between active and deductive rules by including database states into Datalog. The resulting language, which is called Statelog, has two sorts of rules. In this Framework, *Query rules* are used to model Datalog rules, whereas *transition rules* are used to extend the expressiveness of the language to deal with active data manipulation operations.

Normally, Datalog rules access one state of the database. To include active behavior into Datalog, Ludäscher *et al.* propose a logical framework in which Datalog-like rules have access to more than one state. The main idea is to extend Datalog to be able to refer to different database states. Query rules, which are non-state changing rules, are used to query the actual database state. Transition rules, which are state changing rules, are activated and perform actions which are specified in their heads if they are triggered by the occurrence of one or more external events and if the conditions stated in their body hold. Action execution results in a sequence of virtual states. The computation of the sequence of states terminates if it reaches a fixpoint of those states.

Ludäscher *et al.* provide a model-theoretic semantics for Statelog programs via the concept of Σ-stratification which is a local stratification ([Prz88]) with respect to the set of state identifiers. The semantics is independent of evaluation strategies adopted.

Given this semantics, the authors prove many properties with respect to the expressiveness, complexity, and termination of Statelog programs. Considering a subset of Datalog programs for which termination is proven to be guaranteed, they also give an architecture unifying active and deductive databases.

The main result of the work of Ludäscher *et al.* is a demonstration of a possible unification of active and deductive rules within a common logical framework of a suitably extended version of Datalog.

Statelog is similar to Datalog$_{1S}$, and thus it has the same advantages and drawbacks. The kind of stratification proposed for Statelog is much similar to Zaniolo's XY-stratification. Zaniolo's X-rules are now called *Query rules* and his Y-rules are now *transition rules*. In fact, Ludäscher *et al.* have proven the equivalence of both framework for many properties such as the termination or the ultimate periodicity.

Considering Expressiveness and Complexity Issues. Picouet and Vianu address expressiveness and complexity questions such as: the relevance of active features, their impact on the expressive power and the complexity of the systems, and the simplification of execution models and their equivalence[PV95, PV97]. They describe a generic framework formalizing active databases. In their eyes, this framework highlights the distinctions between the various prototypes much better than the abstract model based on relational machines they introduced in Picouet[1995]. They use it to articulate and factor out common features of the prototypes they are considering. Within their framework, they investigate the impact of the various dimensions of the active behavior on the expressiveness and complexity of active databases.

To develop the aforementioned framework, formalizing the notion of an external program and the execution model of a trigger program in conjunction with an external program is a necessity. The update language $while_N$ ([AHV95]) is used to formalize external programs embedding trigger programs.

The generic framework presented is a common skeleton of the prototypes considered. This skeleton will be used as a specification model of the various existing execution models. A trigger program t is syntactically a 7-tuple $< \mathbf{D}, \mathbf{R}, rules, cpl, ev, pri, \Delta{-}type >$, where $\mathbf{D}$ is a public database schema; $\mathbf{R}$, also denoted $sch(t)$, is the schema of t; $rules$ is a set of rules of the form $condition \rightarrow action$ over $sch(t)$; cpl is a mapping from $rules$ into the set of coupling modes; ev is the event mapping of t; pri is the priority mapping of t, that is, a mapping from $rules$ into the subset $\{1, \ldots, |rules|\}$ of natural numbers; and $\Delta{-}type$ is a mapping from $rules$ into the set $\{global, local{-}fixed, local{-}fluid\}$ which contains consumption modes. The trigger program t uses delta relations; that is, relations containing incremental information on deleted or inserted tuples between two database states.

Two rule queues are maintained to represent sequences of triggered rules: the first queue contains rules with an immediate coupling mode, and the second one contains rules with a deferred coupling mode.

Given a trigger program t and an external program e written in $while_N$, an update $t[e]$ is operationally defined in two phases. In the first phase, the program e starts, taking control of the computation. Each time an update instruction is encountered during the execution of e, t takes control over the computation, detecting events and triggering new rules. The immediate rules are fired until the corresponding queue is empty. A new database state and a queue of deferred rules are generated. Then e retakes control over the computation. This process is repeated until e halts, upon which a new database state and a deferred queue are returned. The second phase starts from the new database state returned in the first phase; deferred rules returned here are executed, yielding a final database state.

Using this framework, Picouet and Vianu obtain expressiveness results relating the prototypes to each other, the main outcome being that HiPAC subsumes all other prototypes considered. They also obtain complexity results relative to the impact of active features. For instance, it turns out on the one hand that immediate triggering with unbounded nesting of queues is in EXPTIME. On the other hand, deferred triggering with bounded nesting of queues is in PSPACE. Generally, deferred triggering is computationally more powerful than immediate triggering.

The main contribution of the work seems in our opinion to be the insight provided into which active features are essential and which are not. A feature is essential if it has an impact on the expressive power and the complexity of the ADBMS.

In our opinion, the impact of omitting the event part of ECA-rules on the expressiveness and the complexity of an active system is an open question in this work. Moreover, the operational semantics abstracted by Picouet and Vianu seems general enough to account for many existing ADBMSs. However, the style of their formalization remains operational. In this thesis, we shall show how to express this kind of execution model that in fact combines immediate and deferred execution models in a purely declarative way.

Using the Event Calculus. Fernandes *et al.* provide a formal semantics for active databases ([FWP97]) by formalizing certain aspects characteristic of active behavior within a first order logical framework, whereas the pure deductive functionality is left unchanged. The formalization relies on Kowalski-Sergot's *event calculus* ([KS86] and tries to integrate active and deductive DBMSs by respecting their differences with respect to the operational semantics they traditionally used to have. The key idea behind the actual realization of their approach is to use Kowalski-Sergot's notion of *history*, which is a kind of transaction log that stores successive database states. Fernandes *et al.* formalize the traditional ECA-rules by specifying events and conditions in a Datalog-like language over the database of event occurrences maintained in the history, while actions are specified as an extension of the history maintained database with new event occurrences.

As presented by Fernandes *et al.*, the most important achievement of the approach is the possibility of keeping the usual ECA structure of active rules unchanged while using the old well-defined operational semantics of deductive rules to define event detection, condition test and action execution. Given an ECA structure of the form given in Definition 2.6, detecting that an event E has occurred amounts to evaluating a deductive query over the set of events in the history which constitutes the extensional part of the intended deductive database. Similarly, testing that a condition holds is equivalent to answering a query over logical consequences of the extensional part of the deductive database. Such logical consequences are the intensional part of the deductive database. They are stored in the form of rules over the history.

It should be noted that the approach still has the drawback of resorting almost exclusively to the operational semantics as a way of describing ECA-rules. What about the declarative and model-theoretical parts of active rules as a whole? By contrast, considering E-, C- and A-features separately hinder a semantical consideration of an active rule as a whole. Although the approach represents a major accomplishment *per se* by only reusing old semantical features of deductive databases, it still remains to see how an active rule as an unfragmented entity could be given a semantical account.

Using the Situation Calculus. Baral and Lobo ([BL96, BLT97]) propose a language called $\mathcal{L}_{active}$ which is based on the $\mathcal{L}_0$ language for action proposed by Baral, Gelfond and Provetti ([BGP97]). The language $\mathcal{L}_0$ makes a distinction between actual and hypothetical actions. This distinction is an important design concept that allows reasoning about the effects of the execution of actions.

In $\mathcal{L}_{active}$, the following axioms are included: causal laws of the form $a(\overline{X})$ $\mathbf{causes}$ $f(\overline{Y})$ $\mathbf{if}$ $p_1(\overline{X_1}), \ldots, p_n(\overline{X_n})$, event definitions of the form $e(\overline{X})$ $\mathbf{after}$ $a(\overline{W})$ $\mathbf{if}$ $e_1(\overline{Y_1}), \ldots, e_m(\overline{Y_m})$, $q_1(\overline{Z_1}), \ldots, q_n(\overline{Z_n})$, and active rules of the form

$$\begin{aligned} r(\overline{X_r}) : e_t(\overline{X_t})\ & \mathbf{consumed}\ (C_s) \\ & \mathbf{initiates}[\alpha]\ \mathbf{at}\ \ e_a(\overline{X_a}) \\ & \mathbf{if}\ p_1(\overline{X_1}), \ldots, p_n(\overline{X_n})\ \mathbf{at}\ \ e_c(\overline{X_c}). \end{aligned}$$

where $a(\overline{X})$ is an action and α a sequence of actions; $f(\overline{Y})$, $p_i(\overline{X_i})$, $1 \leq i \leq n$, $q_j(\overline{X_j})$, $1 \leq j \leq m$, are fluents; $e(\overline{X})$, $e_k(\overline{Y_k})$, $1 \leq k \leq n$, $e_t(\overline{X_t})$, $e_a(\overline{X_a})$, and $e_c(\overline{X_c})$ are events; $r(\overline{X_r})$ is the identity number of the active rule.; $C_s \in \{no,\ local,\ global\}$ expresses the consumption mode.

The intended meaning of some of the symbols appearing in the above constructs is the following. The fluents $p_i(\overline{X_i})$, $1 \leq i \leq n$ in causal laws are the *preconditions* of these, whereas they represent the *condition of a rule* in the rule definition. In an active rule, $e_t(\overline{X_t})$ is the *triggering* event of the rule, $e_c(\overline{X_c})$ and $e_a(\overline{X_a})$ represent the states in which the condition evaluation and the action execution are performed.

Semantically, $\mathcal{L}_{active}$ is characterized by a transition diagram whose states are labeled with sets of fluents called *fluent states* and whose transitions are labeled with actions. Similarly to fluent states, there are also *event states*. An *active database state* is a tuple $< \sigma, \epsilon, \tau, \kappa >$ where σ is an fluent state, ϵ is an event state, τ is a set of triggered rules, and κ is a set of rules to be fired or *considered* rules.

Let $\mathcal{S}$ be the set of DB states. To interpret effects of actions in a given DB state, a (partial) *causal interpretation* function $\Psi : \mathcal{A}^* \times \mathcal{S} \rightarrow \mathcal{S}$ is defined, where $\mathcal{A}$ is the set of actions, and $\mathcal{S}$ is the set of database states. Let $\mathcal{E}$ and $\mathcal{T}$ be the set of events and the set of active rules. Given a set of events ϵ and a set of active rules τ, the causal interpretation function uses a total *action selection* function that selects a rule $r_i \in \tau$ such that the action execution event of r_i belongs to ϵ and returns the action part of r_i.

Let $< \sigma, \phi, \phi, \phi >$ the initial DB state with respect to a selection function S in a given set D of ground instances of an application domain, let α be an initial sequence of actions, and let σ be some initial fluent state. Then a kind of successor DB state $< \sigma', \epsilon, \tau, \kappa > = \Psi(\alpha, < \sigma, \phi, \phi, \phi >)$ is computed using a set of transition functions to compute the resulting fluent state, event set, triggered rule set, and selected rule set.

Queries in $\mathcal{L}_{active}$ are of the form

$$f \textbf{ after } \alpha \textbf{ at } \sigma. \tag{2.1}$$

With (2.1), it is possible to reason about a given ground set D. That is, the very nature of the validity of f after the sequence α of actions has been performed is an hypothetical one. The query (2.1) means that f is true in the hypothetical DB state following the performance of the action sequence α. In order to be actual, changes made to the initial DB states should the explicitly “hardwired” into it.

What Baral and Lobo propose in [BL96, BLT97] can be considered as preliminary work on how the situation calculus may be used to formalize active rules: in these papers, too many aspects such as modeling complex events, complex and concurrent actions, and transactions remain as future work. Beside this limitation, there is another one: having presented their language, Baral and Lobo give a translation method for transforming active database rules in a corresponding logic program expressed in the situation calculus notation. In our opinion, this presents a conceptual problem: their language seems to be a set of metaconstructs that have to be translated to the situation calculus. Why not directly express active rules in the situation calculus? The purpose of introducing a new language seems unclear to us.

In [Pin98], Pinto introduces various notions of action occurrence. He views an occurrence statement as a constraint on the *legal* path in the tree of possible futures. His theory incorporates many ingredients that are partly user-provided facts. Pinto argues that his framework can be used as an effective tool for formalizing active databases, assuming Reiter’s formalization of DB transactions. The idea is to let an action occur in the state following the performance of a triggering action, according to the general (simplified) pattern $\phi \supset occurs(\alpha_2(\bar{x}_2), do(\alpha_1(\bar{x}_1), s))$, where ϕ is the condition of the rule, $do(\alpha, s)$ — as we shall see in the next chapter — means the situation following the execution of action α in situation s, and α_1 triggers α_2. The same idea is found in [MC98]. Pinto and McCarthy’s proposals are yet too fragmentary to allow a comparison with our approach.

Bertossi *et al.* ([BPV99]) propose a situation calculus-based formalization that differs considerably from our approach. They first extend Reiter’s specification of database updates to database transactions, incorporating a formalization of static integrity constraints and the notion of action occurrence introduced in [Pin98]. They then specify active rules as sentences of the situation calculus. This representation of rules forces the action part of their rules to be a primitive database operation. Unlike Bertossi et al., we base our approach on GOLOG, which allows the action component of our rules to be arbitrary GOLOG programs.

Extracting a Common Set of Semantic Dimensions From Existing Systems. Having reviewed many of the best known research prototypes and commercial products including the draft for the proposed standard SQL3, Fraternali and Tanca ([FT95]) extract many dimensions of the active behavior common to most of these systems. They then uniformly encode rules of existing systems in a common syntax called Extended Event-Condition-Action (EECA) syntax. This constitutes a base for a comparison of the different existing systems.

Given their huge variety, the dimensions of active bahaviour are made explicit at the syntactic level to avoid treating them at the execution model level. Thus, Fraternali and Tanca introduce a core model of an active database as an *internal representation* used by the execution engine.

The core model uses two kinds of databases: a database, which stores the application data, and an eventbase, where facts relevant to the database history are recorded. The eventbase stores primitive updates (i.e. $insert$, $delete$, $update$), external events (i.e. $begin$, $commit$, $rollbacks$), time events, etc. Two tables, $ACTIVE$ and $EVENT$, are maintained as eventbase. The former registers all events that occurred during a transaction, whereas the later only registers events still contributing to the triggering of some rules. These relations are considered local to each transactions. An alternative would be to maintain a unique table by adding a further transaction argument to both tables.

The event language must allow event queries (EBQ) and event updates (EBU) as well. An event query language can be any first order language containing the relations corresponding to the tables ACTIVE and EVENT. An event update language includes operations for inserting tuples into and deleting them from tables ACTIVE and EVENT. A table log may be added to the system if the capability of querying past data is needed.

The core active rules have the form

$$\begin{array}{l} EBQ(\vec{y}_1), \\ DB/EBQ(\vec{y}_1, \vec{x}_2, \vec{y}_2), \\ EBU(\vec{x}_2) \\ \longrightarrow \\ TU(\vec{x}_{3_1}), \ldots, TU(\vec{x}_{3_n}), \\ \textbf{before/after}\{core\ rule\ list\} \end{array}$$

where $EBQ(\vec{y}_1)$ is a formula interpreted over the eventbase and expresses the implemented triggering semantics; $DB/EBQ(\vec{y}_1, \vec{x}_2, \vec{y}_2)$ is a query to the database and/or the eventbase; $EBU(\vec{x}_2)$ is sequence of event updates modeling the event consumption; the formulas $TU(\vec{x}_{3_i})$ are noninterruptable update blocks.

The translation from EECA to core is done as follows. The event part of EECA rules is converted into a core event part expressing the granularity and the coupling modes; it must prevent the triggering of rules when other noninterruptable rules are being executed. The condition part of the core expresses the primitives for transaction history inspection and the consumption time consideration. Finally, the action part of the core expresses the execution consumption time and enforce the atomicity of noninterruptable rules.

Semantically, a user transaction submitted to an ADBMS with a set of rules R is considered as a transformation from an original state to a final state. At some points called *rule starting points* the control is passed from the transaction to the rule execution engine. The rule engine returns the control back at some points called *quiescent states* which are points where no more rule are triggered. Fraternali and Tanca consider different aspects of embedding the semantics of rule processing into a framework of update and transaction semantics.

The main result of this work seems to us to be the readable catalogue of dimensions of active behavior that it offers. To our knowledge, this work is the first attempt that formally accounts for the similarities and differences between existing ADBMSs considering a large number of systems. However, such important issues as composite events, and delayed coupling modes remain beyond its scope; moreover, the author concentrate their formalization to the deferred execution model.

2.3.4 Other Approaches

In [HJ91], Hull and Jacobs present a rich language using an operational semantics. Based on the database language Heraclitus, the discussion in this work shows how to theoretically analyze alternative rule processing semantics. As an example of such an analysis, they examine approaches of accessing delta relations. Heraclitus is an imperative language that is statically typed, supports (possibly persistent) relation types and variables, supports a type called delta which stores insertions to, or deletions from, a relation, and has an embedded relational calculus subpart to manipulate relations and delta values.

Constructs of Heraclitus can be used to express rule processing semantics. Rules are expressed in Heraclitus as a function from deltas to deltas. A particular semantics of an existing system can be expressed as a procedure called *rule application template*, which uses rules defined as functions. A special kind of indexing called *indexed families* is used to manipulate and refer to functions; it is a mapping from integers to the appropriate code fragments. The authors exclude considering explicit arrays of rules, since this would introduce rules as first-class citizens.

Using the framework of Heraclitus, Hull and Jacobs specify the semantics of Starburst.

In [BCW93a], an approach based on relational algebra is described, aiming at the analysis and optimization of CA rules. It shows how to propagate changes introduced by the execution of the action part of a rule, for example r_1, to the condition of another rule, say r_2, in order to know whether r_1 may trigger r_2. For example, if the condition of r_2 contains the relational algebra expression $\sigma_{\$1}(R) =$ ”IBM”, and the action of r_1 contains an insertion $insertedR$ to R, then the expression $\sigma_{\$1}(insertedR) =$ ”IBM” is the result of the insertion on the original expression. The propagation process is iterated to yield a final expression which, if satisfiable, indicates that r_1 may trigger r_2. Unfortunately, this approach does not treat crucial issues like the execution model and events.

There are some other approaches that we do not cover in the present present document, due as stated earlier to their limited scope. As an example, graph-based formalism is used, especially in the form of event graphs ([CM91]) or Petri nets ([GD94]) to model and/or detect composite events. In [AHW95], a graph-based approach is also used to model provable properties of sets of active rules, such as confluence, termination, and determinism.

2.4 Summary

The problem of formalizing active databases appears to be a major research challenge. The challenge amounts to providing a unique theory accounting for the various dimensions of active behavior. Though we have seen in the present chapter that some interesting proposals exist, there still appears to be a need for a theory providing a *unique* framework for the event language, the operational semantics, the transactional activity, and the underlying data models of active databases. With the notable exception of Zaniolo's proposal, many of the proposals try merely to integrate or reconcile the concept of rule language developed in active databases with transaction concepts and data models that were developed independently.

We think that providing a unique theory that highlights all these aspects would significantly improve our understanding of active databases. Moreover, we think that the situation calculus ([Rei01]) may be used as such a theory since it allows us to bridge the gap between the various dimensions of active behavior through the uniqueness of the representational framework it offers.

We saw that Baral and Lobo claim to use the situation calculus ([BL96, BLT97]); but it turns out in our opinion that their language is a set of constructs that are not expressed in the situation calculus. We also saw that the situation calculus is used in [Pin98] [MC98]. However, we pointed out that Pinto and McCarthy's proposals are yet too fragmentary to allow a comparison with our approach. The approach described in [BPV99] is promising. However, it differs considerably from ours in a major point: active rules are in our view programs expressed in a situation calculus based language, not sentences of the modeling language. In the next chapter, we will lay down the language of the situation calculus as a logical preliminary for the rest of the thesis.

Chapter 3

Logical Preliminaries: the Situation Calculus

In the introductory chapter, we mentioned many proposals which have highlighted and extended the expressiveness of the situation calculus. In this chapter, we review the situation calculus as enhanced in [Rei01], and we introduce the formal language we shall use for the remainder of this thesis. First we provide an introduction to the language of the situation calculus, together with its syntax. We then present an axiomatization that allows formal reasoning about actions of an application domain and includes a solution to the frame problem and a resulting characterization of an important class of theories called *basic action theories*. The presentation of the solution to the frame problem is followed by that of a model theoretic semantics for this language. Finally, we review the available theoretical results that we will base our work on.

3.1 Situation Calculus: the Core Language

In the situation calculus, we appeal to a language that, for many database formalization tasks, will be largely first order. However, this language will sometimes be — as we shall see — second order when some complex actions are introduced and inductive reasoning about actions is performed. Moreover, the language of the situation calculus will have many sorts of variables, each of which ranging over a distinguished portion of the universe of discourse. Thus, to describe the situation calculus, we will appeal to a many-sorted second order logical language whose syntax we now give.

In this section, we present details of the core language of the situation calculus following [Rei01]. Since much of the arguments presented in this thesis will be of semantical nature, we then add a model theoretic semantical account of this language. Finally, we use this semantical account to justify an existing axiomatization of the calculus.

3.1.1 An Informal Presentation

Informally, the situation calculus forms a suitable logical language within which a general theory of actions in a dynamically changing world can be formulated (see e.g. [LLL$^+$94], [Lif87]). The basic ontological categories of the calculus are actions and situations. Actions are first order terms consisting of an action function symbol and its arguments. In modeling relational databases, these will correspond to the elementary database operations of inserting, deleting and updating relational tuples. Situations are first order terms denoting sequences of actions; they are world histories constituted by a list of actions that have been performed ([Rei96]). In modeling relational databases, these will correspond to the database log.[1] Relations whose truth values vary from situation to situation are called *fluents*, and are denoted by predicate symbols whose last argument is a situation term. The universe of discourse of the language is partitioned into action individuals, situation individuals, and object individuals which are any individuals that are neither action, nor situation. Thus the language will provide variables and constants suitable for the appropriate partition of the universe of discourse.

Less informally, the situation calculus is a many-sorted second order predicate logic with sorts for actions ($\mathcal{A}$), situations ($\mathcal{S}$), and objects ($\mathcal{O}$) of an application domain. There is a distinguished constant S_0 denoting the initial situation. The term $do(a(\vec{x}), s)$ represents the result of executing an action $a(\vec{x})$ in the situation s. Finally, the theory also includes special predicates $Poss$ and $\sqsubset$; $Poss(a(\vec{x}), s)$ means that the action $a(\vec{x})$ is possible in the situation s; and $s \sqsubset s'$ states that the situation s' is reachable from s by performing some sequence of actions — s is said to be a subhistory of s'.

The particular language we use will be constrained in different ways with respect to the space of situations. Constraints will typically be added in the form of axioms. Some axioms, called foundational axioms, will impose a space of situations that has the form of a tree rooted in the initial situation S_0. Figure 3.1 shows a tree of situations rooted in S_0. The world is represented as evolving from S_0 through the execution of actions and can be viewed as described by fluents. In the relational database setting, fluents will be used to represent relations, the truth value of which we are interested in. In this setting, the database may be initially empty, that is, for all possible arguments, all fluents are false in S_0. In subsequent situations, the database may be filled with some content, in which case we can conclude that a database operation must have occurred to change the truth value of some of the fluents from false to true. All of the possible courses of actions form a tree rooted in S_0.

Some other axioms, called precondition axioms, will restricts further the tree of situations to those situations in which resulting situations are obtained if and only if the involved actions are possible in the previous situations.

In relational databases, we will be concerned with the truth value of fluents in a particular situation representing a database state. The intended model theoretic meaning of such fluents that are true in a

[1]In fact, in the database setting, a situation is a sequence involving many histories corresponding to as many transactions.

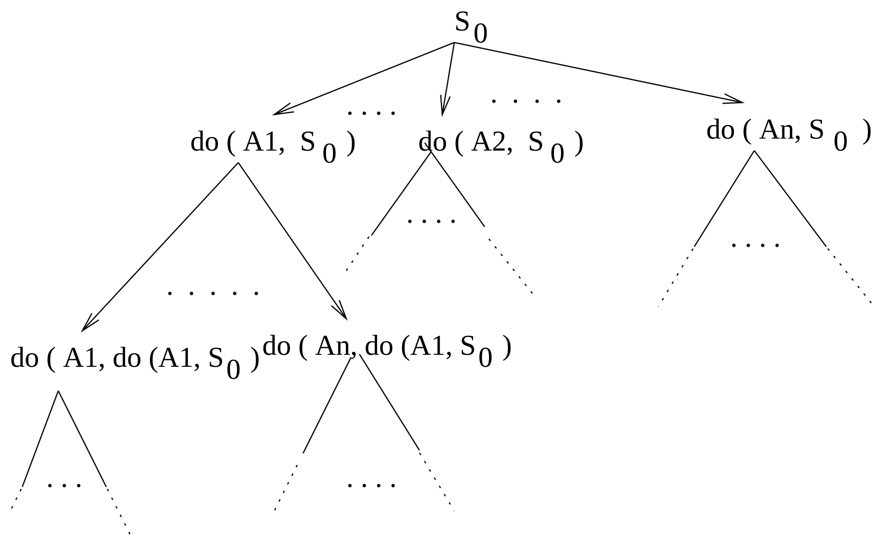


Figure 3.1: The tree of possible situations rooted in the initial situation for a domain with n actions.

certain situation is a database schema instance.

Let us call this language $\mathcal{L}_{sitcalc}$. It is the framework within which we shall formalize active databases in the form of action theories of a particular kind.

3.1.2 The Language

Formally, the language $\mathcal{L}_{sitcalc}$ is a many-sorted second order language with equality represented as a triple $(\mathfrak{A}, \mathfrak{W}, \mathfrak{J})$, where we have:

1. a set $\mathfrak{J} = \{\mathcal{A}, \mathcal{S}, \mathcal{O}\}$ of sorts;
2. an alphabet $\mathfrak{A}$ with
    - individual symbols:
        (a) variables: a, a_1, a_2, …, a', a'', … (for sort $\mathcal{A}$); s, s_1, s_2, …, s', s'', … (for sort $\mathcal{S}$); and $x, y, z, \ldots, x_1, x_2, \ldots$ (for sort $\mathcal{O}$).
        (b) constants: nil (null action), A, A_1, A_2, … (for sort $\mathcal{A}$); and X, Y, Z, …, X_1, X_2, … (for sort $\mathcal{O}$).
    - function symbols:[2]
        (a) variables: We have countably many of these, called *action functions*, of sort $<\mathcal{O}^n, \mathcal{A}>$.

[2]In general, the language of the situation calculus contains two kinds of fluents: relational and functional. Since the later play a marginal role in this thesis, we will introduce them when needed.

   (b) constants: We have two of these: S_0 (start situation) of sort $\mathcal{S}$; do, a binary function of sort $<\mathcal{A}, \mathcal{S}, \mathcal{S}>$, for generating the situation resulting from the execution of an action.

- predicate symbols:
   (a) variables: We have countably many of these, called *relational fluents*, of sort $<\mathcal{O}^n, \mathcal{S}>$.
   (b) constants: We have two of these: $\sqsubset$, an ordering relation of sort $<\mathcal{S}, \mathcal{S}>$ over the space of situations; and $Poss$, of sort $<\mathcal{A}, \mathcal{S}>$, to express what actions are possible in a given situation.
   (c) Equality: $=$.
- punctuation symbols: $($, $)$, $[$, $]$, $\{$, $\}$, $\ldots$
- logical symbols: $\neg$, $\supset$, $\forall_{\mathfrak{j}}$, $TRUE$, and $FALSE$; with $\mathfrak{j} \in \mathfrak{J}$;

3. a set $\mathfrak{W}$ of wellformed formulas (wffs): Terms, atomic formulas, wffs, and sentences are defined in the standard way of second order languages ([End73]). Additional logical constants are introduced as abbreviations in the usual way.

The language $\mathcal{L}_{sitcalc}$ is the basic framework within which one axiomatizes action theories. As a notational convention, we shall often omit any leading universal quantifiers in closed formulas. Moreover, a formula ϕ is said to be *uniform in* s ([Rei01]) iff s is a free situation variable, which is the only situation term mentioned by ϕ, and whenever it mentions a predicate, then that predicate is neither $Poss$, nor $\sqsubset$, nor else an equality on situations. Finally, we also introduce the following convenient abbreviation:

$$do([\,], s) =_{df} s;$$
$$do([a_1, a_2, \cdots, a_n], s) =_{df} do(a_n, do(\cdots, do(a_1, s)\cdots)).$$

Example 3.1 *Consider again the stock trading database of Example 2.1 The database schema is formalized in the following fluents:*

$price(stock\_id, price, time, s)$,
$stock(stock\_id, price, closingprice, s)$, *and*
$customer(cust\_id, balance, stock\_id, s)$,

which are relational fluents.

Primitive actions are of the form $F\_insert(\vec{x})$ and $F\_delete(\vec{x})$, where F is the relational schema being manipulated. For example $price\_insert(stock\_id, price, time)$ ($price\_delete(stock\_id, price, time)$) is an action term that denotes the operation of inserting (deleting) the tuple $(stock\_id, price, time)$ into (from) the relation $price$. We have similar actions for the relations $stock$ and $customer$. The formula

$$price(ST1, \$100, 100100 : 9AM, S_0) \wedge price(ST2, \$60, 100100 : 4PM, S_0)$$

is uniform in S_0.

Finally, we have constants $ST1$, $ST2$, $ST3$, $\$50$, $\$60$, $\$100$, $\$110$, $100100:9AM$, $100100:1PM$, $100100:4PM$, Sue, $Zang$, $\$10000$, *and* $\$5000$. ■

3.2 The Frame Problem and Basic Action Theories

3.2.1 Modeling Dynamic Domains: the Frame Problem

To model a dynamic application domain, it is assumed that axioms are given to describe how and under what conditions the domain is changing or not changing as a result of performing actions. The first sort of axioms, called *action precondition axioms*, describes the general conditions of action execution. They have the form

$$(\forall \vec{x}, s).Poss(\alpha(\vec{x}), s) \equiv \Pi_\alpha(\vec{x}, s) \tag{3.1}$$

for each action function $\alpha(\vec{x}) \in \mathcal{A}$, where $\Pi_\alpha(\vec{x}, s)$ is a first order formula with free variables among $\vec{x}, s$; $\Pi_\alpha(\vec{x}, s)$ describes the execution preconditions of the action $\alpha(\vec{x})$; (3.1) means that the action $\alpha(\vec{x})$ is possible if and only if the preconditions described by the formula $\Pi_\alpha(\vec{x}, s)$ hold. Moreover, $\Pi_\alpha(\vec{x}, s)$ is uniform in s. In the stock trading example, the following axiom states that it is possible to insert a tuple into the $price$ relation relative to the database log s iff, as a result of performing the actions in the log, that tuple would not already be present in the $price$ relation.

$$\begin{aligned} &Poss(price\_insert(stock\_id, price, time), s) \equiv \\ &\qquad\qquad \neg price(stock\_id, price, time, s). \end{aligned} \tag{3.2}$$

The second sort of axioms describes the positive and negative effects of action executions on fluents. For each fluent $F \in \mathcal{F}$, the axiomatization includes a *positive effect axiom* of the form

$$(\forall \vec{x}, a, s).\gamma_F^+(\vec{x}, a, s) \supset F(\bar{x}, do(a, s)) \tag{3.3}$$

and a *negative effect axiom* of the form

$$(\forall \vec{x}, a, s).\gamma_F^-(\vec{x}, a, s) \supset \neg F(\bar{x}, do(a, s)), \tag{3.4}$$

where $\gamma_F^+(\vec{x}, a, s)$ ($\gamma_F^-(\vec{x}, a, s)$) denotes a first order formula with free variables among $\vec{x}, a, s$ describing conditions for the fluent F to become true (false) in the successor situation $do(a, s)$. Moreover, $\gamma_F^+(\vec{x}, a, s)$ and $\gamma_F^-(\vec{x}, a, s)$ are uniform in s. In the stock trading example, the following axiom states that the tuple $(stock\_id, price, time)$ will be in the $price$ relation relative to the log $do(a, s)$ if the last database operation a in the log inserted it there:

$$\begin{aligned} &a = price\_insert(stock\_id, price, time) \supset \\ &\qquad\qquad price(stock\_id, price, time, do(a, s)), \end{aligned} \tag{3.5}$$

and the following states that the tuple $(stock\_id, price, time)$ will be deleted from the $price$ relation relative to the log $do(a, s)$ if the last database operation a in the log deleted it from there:

$$\begin{aligned} &a = price\_delete(stock\_id,price, time) \supset \\ &\qquad\qquad \neg price(stock\_id, price, time, do(a, s)). \end{aligned} \tag{3.6}$$

The third sort of axioms describes under what conditions fluents remain unchanged in the successor situation. To achieve this kind of "inertia", the axiomatization includes a set of *frame axioms* ([MH69]) whose general form is one of the following:

$$(\forall \vec{x}, \vec{y}, a, s).\Pi_\alpha(\vec{x}, s) \wedge \varphi^+_F(\vec{x}, \vec{y}, s) \wedge F(\vec{y}, s) \supset F(\vec{y}, do(\alpha(\vec{x}), s)) \tag{3.7}$$

or

$$(\forall \vec{x}, \vec{y}, a, s).\Pi_\alpha(\vec{x}, s) \wedge \varphi^-_F(\vec{x}, \vec{y}, s) \wedge \neg F(\vec{y}, s) \supset \neg F(\vec{y}, do(\alpha(\vec{x}), s)). \tag{3.8}$$

Here, $\Pi_\alpha(\vec{x}, s)$ is as described for (3.1), and $\varphi^+_F(\vec{x}, \bar{y}, s)$ $(\varphi^-_F(\vec{x}, \vec{y}, s))$ denotes a first order formula with free variables among $\vec{x}, \vec{y}, s$ describing inertia conditions of the fluent F. As an example of (3.7) in our stock trading setting, we have the following axiom which states that the tuple $(stock\_id', price', time')$ will be in the $price$ relation relative to the log $do(a, s)$ if it was already there and the last database operation a in the log did not touch it:

$$\begin{aligned} &a = price\_insert(stock\_id, price, time) \wedge \\ &\quad stock\_id \neq stock\_id' \wedge price \neq price' \wedge time \neq time' \wedge \\ &\qquad price(stock\_id', price', time', s) \supset price(stock\_id', price', time', do(a, s)). \end{aligned} \tag{3.9}$$

With $|\mathcal{A}|$ actions and $|\mathcal{F}|$ fluents, a total of $2|\mathcal{A}||\mathcal{F}|$ of frame axioms is to be provided in the worst case. The frame problem is just the burden of providing **all** these axioms to express what is not changing in the world. With growing $|\mathcal{A}|$ and $|\mathcal{F}|$, the number of frame axioms become more tedious to be given by hand. Thus some way out must be found.

3.2.2 Basic Action Theories

A simple solution to the frame problem, which relies on some proposals of [Sch90] and [Ped89], was provided by Reiter ([Rei91]). He shows how to transform effect and frame axioms in a syntactically equivalent formula of the form

$$(\forall \vec{x}, a, s).F(\vec{x}, do(a, s)) \equiv \gamma^+_F(\vec{x}, a, s) \vee F(\vec{x}, s) \wedge \gamma^-_F(\vec{x}, a, s). \tag{3.10}$$

Such formulas are named *successor state axioms* and are given for each $F \in \mathcal{F}$; these characterize the truth values of the fluent F in the next situation $do(a, s)$ in terms of the current situation s. In the stock trading example, the following successor state axiom states that the tuple $(stock\_id, price, time)$ will be in the $price$ relation relative to the log $do(a,s)$ iff the last database operation a in the log inserted it there, or it was already in the $price$ relation relative to the log s, and a didn't delete it.

$$price(stock\_id, price, time, do(a,s)) \equiv a = price\_insert(stock\_id, price, time) \vee$$
$$price(stock\_id, price, time, s) \wedge a \neq price\_delete(stock\_id, price, time).$$

A domain theory is axiomatized in the situation calculus with five classes of axioms summarized in the following definition:

Definition 3.1 *([LR94]) Let $\mathcal{L}_{sitcalc} = (\mathfrak{A}, \mathfrak{W}, \mathfrak{J})$ be the language of the situation calculus. Then a theory $\mathcal{D} \subseteq \mathfrak{W}$ is a* basic action theory *iff it has a set Act of actions and a set Fl of fluents, and, moreover, it is of the form*

$$\mathcal{D} = \mathcal{D}_f \cup \mathcal{D}_{ap} \cup \mathcal{D}_{ss} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0},$$

where

1. *$\mathcal{D}_f$ is the set of foundational axioms (more details in [PR99] and in Section 3.3).*

2. *$\mathcal{D}_{ap}$ is a set of action precondition axioms of the form (3.1), one for each action in Act.*

3. *$\mathcal{D}_{ss}$ is a set of successor state axioms of the form (3.10), one for each fluent in Fl.*

4. *$\mathcal{D}_{una}$ consists of unique names axioms*[3]

$$C_i \neq C_j, \quad i \neq j$$

for objects, where $C_1, \ldots, C_n$ are all the object constants of $\mathfrak{A}$, together with unique name axioms for actions:

$$A_i(\vec{x}) \neq A_j(\vec{y}), \quad i \neq j,$$

$$(\forall \vec{x}, \vec{y}).A_i(\vec{x}) = A_i(\vec{y}) \supset \vec{x} = \vec{y}, \quad i = 1, \ldots, m$$

where $A_1, \ldots, A_m$ are all the action function constants in Act.

5. *$\mathcal{D}_{S_0}$ is a set of first order sentences uniform in S_0; this specifies the initial state of the domain, in our case the* initial database *state.*

[3]These axioms assume no null values. In fact, if null values are assumed, these axioms can be relaxed to have some constants that are equal (See [Rei86]).

The following is an example of an initial database for the stock trading domain.

$$\begin{aligned}
price(s\_id, pr, t, S_0) \equiv\ & s\_id = ST1 \wedge pr = \$100 \wedge t = 100100:4PM \vee \\
& s\_id = ST2 \wedge pr = \$110 \wedge t = 100100:9AM \vee \\
& s\_id = ST3 \wedge pr = \$50 \wedge t = 100100:1PM, \\
stock(s\_id, pr, clPr, S_0) \equiv\ & s\_id = ST1 \wedge pr = \$100 \wedge clPr = 100 \vee \\
& s\_id = ST2 \wedge pr = \$110 \wedge clPr = 100 \vee \\
& s\_id = ST3 \wedge pr = \$50 \wedge clPr = 60, \\
customer(c\_id, bal, s\_id, S_0) \equiv\ & c\_id = Sue \wedge bal = \$10000 \wedge s\_id = ST1 \vee \\
& c\_id = Zang \wedge bal = \$5000 \wedge s\_id = ST2.
\end{aligned}$$

Notice that while these initial database axioms specify a complete initial database state (as is normal for relational databases), this is not a requirement of the theory we are presenting, except for the basic action theories introduced in Definition 3.1. Therefore our account could, for example, accommodate initial databases with null values.

3.2.3 The projection Problem in the Situation Calculus

Let $\mathcal{D}$ be a background situation calculus axiomatization for some dynamic domain, as described in Definition 3.1, and let $Q(s)$ be a situation calculus formula – the *query* – with one free situation variable s. Moreover, let the situation $do(\alpha_n, do(\alpha_{n-1}, \cdots, do(\alpha_1, S_0)\cdots))$ be a ground situation term, that is one that mentions no free variables. We treat this as a sequence of executed actions, and define the projection problem in the situation calculus as the problem of determining whether

$$\mathcal{D} \models Q(do(\alpha_n, do(\alpha_{n-1}, \cdots, do(\alpha_1, S_0)\cdots))).$$

We define the *answer* to Q *relative to this sequence* to be "yes" iff

$$\mathcal{D} \models Q(do(\alpha_n, do(\alpha_{n-1}, \cdots, do(\alpha_1, S_0)\cdots))).$$

The answer is "no" iff

$$\mathcal{D} \models \neg Q(do(\alpha_n, do(\alpha_{n-1}, \cdots, do(\alpha_1, S_0)\cdots))).$$

So on this definition, evaluating whether Q is true in situation s is performed relative to a ground sequence, and in the most general setting, it is a theorem-proving task.

3.3 A Model Theoretic Semantics for the Situation Calculus

Normally, the standard semantics of second order languages (see, e.g., [End73]) is sufficient for ascribing a meaning to wffs of $\mathcal{L}_{sitcalc}$. However, some aspects specific to the situation calculus need to be explicitly addressed to highlight the dynamics captured by this language. For example, it is necessary to explicitly address the problem of ascribing an appropriate denotation to the distinguished situation terms S_0 and $do(t_{\mathcal{A}}, s)$, where $t_{\mathcal{A}}$ is an action term, and s a situation variable. We map those situation terms to sequences of individuals of sort action. The same idea is also used in [BL96], [PR99], and [Lak96], sometimes under different names.[4]

We define a second order structure $\mathfrak{M}$ for $\mathcal{L}_{sitcalc}$ as a pair $(\mathfrak{U}, \mathfrak{I})$, where $\mathfrak{U}$ is a universe such that $\mathfrak{U} = \mathfrak{U}_{\mathcal{A}} \cup \mathfrak{U}_{\mathcal{O}}$, and $\mathfrak{I}$ is an interpretation function such that:

1. For each individual constant $C^{\mathfrak{j}}$ such that $\mathfrak{j} \in \{\mathcal{A}, \mathcal{O}\}$, $\mathfrak{I}(C^{\mathfrak{j}}) \in \mathfrak{U}_{\mathfrak{j}}$.

2. For the situation constant S_0, $\mathfrak{I}(S_0) = [\,]$.

3. For each action function of the form $a(\vec{x})$, $\mathfrak{I}(a(\vec{x})) = \mathfrak{I}(a) \ : \ \mathfrak{U}_{\mathfrak{j}_1} \times \ldots \times \mathfrak{U}_{\mathfrak{j}_n} \longrightarrow \mathfrak{U}_{\mathcal{A}}$, where $\mathfrak{j}_i = \mathcal{O}$.

4. $\mathfrak{I}(do(A, s)) = \mathfrak{I}(s) \circ [\mathfrak{I}(A)]$, where $\circ$ denotes the list concatenation function, s is a situation variable, and A is an action constant.

5. For each predicate symbol F other than $\sqsubset$ and $Poss$, $\mathfrak{I}(F) \subseteq \mathfrak{U}_{\mathfrak{j}_1} \times \ldots \times \mathfrak{U}_{\mathfrak{j}_n} \times \mathfrak{U}^*_{\mathcal{A}}$, where $\mathfrak{j}_i = \mathcal{O}$.

Let γ, γ_1, and γ_2 be wffs of $\mathcal{L}_{sitcalc}$, $\mathfrak{M}$ a structure for $\mathcal{L}_{sitcalc}$, and $\mathcal{V}$ a mapping from the set of variables into the universe $\mathfrak{U}$; $\mathcal{V}$ is a variable assignment applied to variables of sorts $\mathcal{O}, \mathcal{A}$, and $\mathcal{S}$, and we define it as follows.

1. For each individual variable x, $\mathcal{V}(x) \in \mathfrak{U}_{\mathcal{O}}$

2. For each action variable a, $\mathcal{V}(a) \in \mathfrak{U}_{\mathcal{A}}$

3. For each situation variable s, $\mathcal{V}(s) \in \mathfrak{U}^*_{\mathcal{A}}$

4. For each $n+1$-ary relational fluent variable F, $\mathcal{V}(F) \subseteq \mathfrak{U}^n \times \mathfrak{U}^*_{\mathcal{A}}$, where $\mathfrak{U}^n$ is of the appropriate sort.

5. For each n-ary action function variable $a(\vec{x})$, $\mathcal{V}(a) \subseteq \mathfrak{U}^n \times \mathfrak{U}_{\mathcal{A}}$, where $\mathfrak{U}^n$ is of the appropriate sort.

The interpretation of $\sqsubset$ and $Poss$ is inferred from $\mathfrak{I}$ in a way to be seen later in this section.

The denotation $||t_{\mathfrak{j}}||_{\mathfrak{M},\mathcal{V}}$ of a term $t_{\mathfrak{j}}$ of sort $\mathfrak{j}$ is now defined by:

1. $||t_{\mathcal{A}}||_{\mathfrak{M},\mathcal{V}} = \mathcal{V}(t_{\mathcal{A}})$, if $t_{\mathcal{A}}$ is an action variable.

2. $||t_{\mathcal{A}}||_{\mathfrak{M},\mathcal{V}} = A(||\vec{t}||_{\mathfrak{M},\mathcal{V}})$, if $t_{\mathcal{A}}$ is an action term constant $A(\vec{t})$.

[4]In particular, the semantics used in the proof of Theorem 1 in [PR99] has influenced our semantical account.

3. $||t_{\mathcal{S}}||_{\mathfrak{M},\mathcal{V}} = \mathcal{V}(t_{\mathcal{S}})$, if $t_{\mathcal{S}}$ is a situation variable.

4. $||t_{\mathcal{S}}||_{\mathfrak{M},\mathcal{V}} = [\,]$, if $t_{\mathcal{S}} = S_0$.

5. $||t_{\mathcal{S}}||_{\mathfrak{M},\mathcal{V}} = ||t'_{\mathcal{S}}||_{\mathfrak{M},\mathcal{V}} \circ [||t_{\mathcal{A}}||_{\mathfrak{M},\mathcal{V}}]$, if $t_{\mathcal{S}} = do(t_{\mathcal{A}}, t'_{\mathcal{S}})$.

6. $||t_{\mathcal{A}}||_{\mathfrak{M},\mathcal{V}} = \mathcal{V}(a)(||\vec{t_{\mathcal{A}}}||_{\mathfrak{M},\mathcal{V}})$, if $t_{\mathcal{A}}$ is action term variable $a(\vec{t})$.

Next, we define the semantics of wffs of $\mathcal{L}_{sitcalc}$. Let $\mathfrak{M} = (\mathfrak{U}, \mathfrak{I})$ be a structure for $\mathcal{L}_{sitcalc}$. The truth value of a wff of $\mathcal{L}_{sitcalc}$ in the structure $\mathfrak{M}$ with respect to the variable assignment $\mathcal{V}$ is given by the following semantical rules:

$\models_{\mathfrak{M},\mathcal{V}} F(\vec{t_{\mathcal{O}}}, S_0)$ iff $<||\vec{t_{\mathcal{A}}}||_{\mathfrak{M},\mathcal{V}}, ||S_0||> \in \mathfrak{I}(F)$, where F is a relational fluent constant .

$\models_{\mathfrak{M},\mathcal{V}} Poss(a(\vec{t_{\mathcal{O}}}), S_0)$ iff $\models_{\mathfrak{M},\mathcal{V}} \Pi_A(\vec{t_{\mathcal{O}}}, S_0) \wedge a(\vec{t_{\mathcal{O}}}) = A(\vec{x})$,

where $A(\vec{x})$ has an action precondition axiom of the form $Poss(\vec{x}, s) \equiv \Pi_A(\vec{x}, s)$.

$\models_{\mathfrak{M},\mathcal{V}} f(\vec{t_{\mathcal{O}}}, S_0)$ iff $<||\vec{t_{\mathcal{A}}}||_{\mathfrak{M},\mathcal{V}}, ||S_0||> \in \mathcal{V}(F)$, where f is a relational fluent variable.

$\models_{\mathfrak{M},\mathcal{V}} t_{\mathcal{S}} \sqsubset t'_{\mathcal{S}}$ iff $||t_{\mathcal{S}}||$ is a prefix of $||t'_{\mathcal{S}}||$.

$\models_{\mathfrak{M},\mathcal{V}} t_{\mathcal{A}} = t'_{\mathcal{A}}$ iff $||t_{\mathcal{A}}|| = ||t'_{\mathcal{A}}||$.

$\models_{\mathfrak{M},\mathcal{V}} t_{\mathcal{S}} = t'_{\mathcal{S}}$ iff $||t_{\mathcal{S}}|| = ||t'_{\mathcal{S}}||$.

$\models_{\mathfrak{M},\mathcal{V}} t_{\mathcal{O}} = t'_{\mathcal{O}}$ iff $||t_{\mathcal{O}}|| = ||t'_{\mathcal{O}}||$.

$\models_{\mathfrak{M},\mathcal{V}} \neg\gamma$ iff $\not\models_{\mathfrak{M},\mathcal{V}} \gamma$.

$\models_{\mathfrak{M},\mathcal{V}} \gamma_1 \supset \gamma_2$ iff whenever $\models_{\mathfrak{M},\mathcal{V}} \gamma_1$, then $\models_{\mathfrak{M},\mathcal{V}} \gamma_2$.

$\models_{\mathfrak{M},\mathcal{V}} (\forall a)\gamma$ iff $\models_{\mathfrak{M},\mathcal{V}[a/A]} \gamma$, for all $A \in \mathfrak{U}_{\mathcal{A}}$.

$\models_{\mathfrak{M},\mathcal{V}} (\forall s)\gamma$ iff $\models_{\mathfrak{M},\mathcal{V}[s/S]} \gamma$, for all $S \in \mathfrak{U}^{*}_{\mathcal{A}}$.

$\models_{\mathfrak{M},\mathcal{V}} (\forall x)\gamma$ iff $\models_{\mathfrak{M},\mathcal{V}[x/X]} \gamma$, for all $X \in \mathfrak{U}_{\mathcal{O}}$.

$\models_{\mathfrak{M},\mathcal{V}} F(\vec{t_{\mathcal{O}}}, do(a(\vec{x}), s))$ iff $\models_{\mathfrak{M},\mathcal{V}} \Phi_F(\vec{t_{\mathcal{O}}}, a, \vec{x}, s)$, where F has a successor state axiom of the

form $F(\vec{t_{\mathcal{O}}}, do(a(\vec{x}), s)) \equiv \Phi_F(\vec{t_{\mathcal{O}}}, a, \vec{x}, s)$.

$\models_{\mathfrak{M},\mathcal{V}} f(\vec{t_{\mathcal{O}}}, do(a(\vec{x}), s))$ iff $\models_{\mathfrak{M},\mathcal{V}} \Phi_{\mathcal{V}(f)}(\vec{t_{\mathcal{O}}}, a, \vec{x}, s)$, where $\mathcal{V}(f)$ has a successor state axiom of

the form $\mathcal{V}(f)(\vec{t_{\mathcal{O}}}, do(a(\vec{x}), s)) \equiv \Phi_{\mathcal{V}(f)}(\vec{t_{\mathcal{O}}}, a, \vec{x}, s)$.

$\models_{\mathfrak{M},\mathcal{V}} Poss(a(\vec{t_{\mathcal{O}}}), do(a'(\vec{t'_{\mathcal{O}}}), s))$ iff $\models_{\mathfrak{M},\mathcal{V}} \Pi_A(\vec{t_{\mathcal{O}}}, do(a'(\vec{t'_{\mathcal{O}}}), s)) \wedge a(\vec{t_{\mathcal{O}}}) = A(\vec{x})$,

where $A(\vec{x})$ has an action precondition axiom of the form $Poss(\vec{x}, s) \equiv \Pi_A(\vec{x}, s)$.

In this definition, $\mathcal{V}[\kappa/\mathrm{K}]$ denotes a variant of $\mathcal{V}$ where κ is mapped to K.

A formula γ is true in the structure $\mathfrak{M}$, denoted by $\models_{\mathfrak{M}} \gamma$, iff we have $\models_{\mathfrak{M},\mathcal{V}} \gamma$ for all variable assignments $\mathcal{V}$. A structure $\mathfrak{M}$ of $\mathcal{L}_{sitcalc}$ is a model of the wff γ iff γ is true in $\mathfrak{M}$; one also says that $\mathfrak{M}$ satisfies γ. We say that a wff γ is *valid* (i.e., $\models \gamma$) iff we have $\models_{\mathfrak{M},\mathcal{V}} \gamma$ for all structures $\mathfrak{M}$ and all value assignments $\mathcal{V}$. In the context of relational databases, we will encounter many wffs which are true in a *given* relational database viewed as a model, and a few other wffs which are valid, i.e. true for all possible relational databases. When it is not the case that $\models_{\mathfrak{M}} \gamma$, then γ is false in the structure $\mathfrak{M}$. Finally, a wff γ_2 is a logical consequence of another wff γ_1, written $\gamma_1 \models \gamma_2$, iff every model of γ_1 is also a model of γ_2.

Let Σ be a set of wffs. A structure $\mathfrak{M}$ is a model of Σ iff it is a model of every wff in Σ. A wff γ is a logical consequence of Σ ($\Sigma \models \gamma$) iff every model of Σ is also a model of γ.

Example 3.2 *Consider the Example 3.1. Then the tables in Figure 3.2 define an interpretation for the database formalized there. Both*

$$(\forall st, pr, t, s).price(st, pr, t, s) \supset stockId(st, s) \wedge price(pr) \wedge time(t)$$

and

$$(\forall st, pr, t, s).price(st, pr, t, s). \wedge price(st', pr, t, s) \supset st = st'.$$

are true in this interpretation which, therefore, is a model for these sentences. ■

price			
ST1	$100	100100:4PM	[ ]
ST2	$110	10010:9AM	[ ]
ST3	$50	100100:1PM	[ ]

stock			
ST1	$100	$100	[ ]
ST2	$110	$100	[ ]
ST3	S50	$60	[ ]

customer			
Sue	$10000	ST1	[ ]
Zang	$5000	ST2	[ ]

stockId
ST1
ST2
ST3

price
$100
$110
$60
$50

time
100100:4PM
100100:9AM
100100:1PM

customerId
Sue
Zang

balance
$10000
$5000

Figure 3.2: An interpretation for the stock trading example

It has become usual to validate any proposed semantics for the situation calculus by proving the validity of the foundational axioms in it ([Lak96]). Conversely, it is important to show the validity of any set of foundational axioms used to construct theories of $\mathcal{L}_{sitcalc}$ as – we shall see it – they are an important part these theories. We use the foundational axioms proposed in [Rei01]:

Theorem 3.2 *The following are valid sentences of $\mathcal{L}_{sitcalc}$:*

1. $\models (\forall a_1, a_2, s_1, s_2).do(a_1, s_1) = do(a_2, s_2) \supset a_1 = a_2 \wedge s_1 = s_2$

2. $\models (\forall P).P(S_0) \wedge (\forall a, s)[P(s) \supset P(do(a, s))] \supset \forall s P(s)$

3. $\models (\forall s).\neg(s \sqsubset S_0)$

4. $\models (\forall s).s \sqsubset do(a, s') \equiv s \sqsubseteq s'$

3.4 Basic Results of the Situation Calculus

In the sequel of this thesis, we will use some fundamental results of the situation calculus stated in [Rei01] and proven in [PR99]. This section is devoted to summarizing these results, each of which will be motivated before being formally introduced.

Automated reasoning in the situation calculus has been organized mostly around the mechanism of *regression*. Intuitively, regression is a syntactic manipulation mechanism aimed at reducing the nesting of the complex situation term $do(\alpha, \sigma)$ in any sentence W of the appropriate syntactic form. Suppose W mentions a fluent atom $F(\vec{t}, do(\alpha, \sigma))$ atom with F's successor state axiom being $F(\vec{x}, do(a, s)) \equiv \Phi(\vec{x}, a, s)$. Then we can use regression — in the form of an operator $\mathcal{R}$ applied to W ($\mathcal{R}$[W]) — to determine a logically equivalent variant W' of W in which $F(\vec{t}, do(\alpha, \sigma))$ has been replaced by $\Phi(\vec{t}, \alpha, \sigma)$. The formal definition of the regression operator $\mathcal{R}$ is reviewed in Appendix A. By taking W', which mentions σ, instead of W, which mentions $do(\alpha, \sigma)$, the nesting of do is reduced by one.

In order for the regression operator to be applicable to it, a formula W must be in *regressable form* which, in essence, means that W must have its situations terms grounded in S_0 and it does not mention any predicate $\sqsubset$, nor any equality between situation terms; moreover, for any atom $Poss(\alpha, \sigma)$ mentioned in W, α is of the form $A(t_1, \ldots, t_n)$, where A is an n-ary action function symbol.

Regression uses action precondition axioms and successor state axioms to simplify the situation term of a formula W. In doing so, it (repeatedly) transforms W into a formula $\mathcal{R}[W]$. The importance of this mechanism is shown in the following theorem.

Theorem 3.3 (Regression Theorem) *Suppose W is a regressable sentence of $\mathcal{L}_{sitcalc}$, and $\mathcal{D}$ is a basic action theory.*[5] *Then*

$$\mathcal{D} \models W \textit{ iff } \mathcal{D}_{S_0} \cup \mathcal{D}_{una} \models \mathcal{R}[W].$$

Computationally, Theorem 3.3 is of extraordinary importance. It states that in order to evaluate a sentence W against a basic action theory $\mathcal{D}$, it is necessary and sufficient to evaluate $\mathcal{R}[W]$ in the initial theory $\mathcal{D}_{S_0}$. We shall use it to answer regressable queries in the context of active databases.

[5]See Appendix A for a formal definition of the notion of regression and regressable sentence.

In addition to the Regression Theorem, which furnishes a theoretical justification of posing regressed queries against initial databases, there is a further result (the Implementation Theorem) of the situation calculus that justifies a simple Prolog implementation of dynamical systems formalized via basic action theories. Before stating the Implementation Theorem, we introduce two preliminary concepts that are used in that theorem.

Definition 3.4 (Closed Form Database) *Suppose $\mathcal{D}$ is a basic action theory of $\mathcal{L}_{sitcalc}$. Its initial database $\mathcal{D}_{S_0}$ is in closed form iff*

- *For each fluent F of $\mathcal{D}_{S_0}$, $\mathcal{D}_{S_0}$ contains exactly one sentence of the form $F(\vec{x}, S_0) \equiv \Psi_F(\vec{x}, S_0)$, where $\Psi_F(\vec{x}, S_0)$ is a first order formula uniform in S_0 with free variables among $\vec{x}$.*
- *For each non-fluent predicate symbol P, $\mathcal{D}_{S_0}$ contains exactly one sentence of the form $P(\vec{x}) \equiv \Theta_P(\vec{x})$, where $\Theta_P(\vec{x})$ is a situation independent first order formula with free variables among $\vec{x}$.*
- *The remaining sentences of $\mathcal{D}_{S_0}$ are unique name axioms for the sort $\mathcal{O}$, together with all instances of the schemas $t[s] \neq s$ and $t[a] \neq a$, where $t[s]$ and $t[a]$ are a situation term and an action term other than s and a mentioning s and a, respectively.*

Definition 3.5 (Definitional Theory) *A theory is* definitional *iff each one of its axioms consists of a first order sentence (definition) of the form $(\forall x_1, \ldots, x_n).P(x_1, \ldots, x_n) \equiv \phi$ for each predicate symbol P mentioned, except for equality.*

Theorem 3.6 (Implementation Theorem) *Suppose that $\mathcal{D}$ is a basic action theory of $\mathcal{L}_{sitcalc}$ with the following restrictions:*

- *$\mathcal{L}_{sitcalc}$ has finitely many relational fluents and action function symbols.*
- *$\mathcal{D}_{S_0}$ is in closed form.*

Suppose further that $\mathcal{P}$ is a Prolog program obtained from the following sentences, after transforming them by the revised Lloyd-Topor rules[6]*:*

- *For each definition of a non-fluent predicate of $\mathcal{D}_{S_0}$ of the form $P(\vec{x}) \equiv \Theta_P(\vec{x})$:*

$$\Theta_P(\vec{x}) \supset P(\vec{x}).$$

- *For each equivalence in $\mathcal{D}_{S_0}$ of the form $F(\vec{x}, S_0) \equiv \Psi_F(\vec{x}, S_0)$:*

$$\Psi_F(\vec{x}, S_0) \supset F(\vec{x}, S_0).$$

[6]See Appendix B for further details on these rules.

- *For each action precondition axiom of $\mathcal{D}_{ap}$ of the form $Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$:*

$$\Pi_A(\vec{x}, s) \supset Poss(A(\vec{x}), s).$$

- *For each successor state axiom of $\mathcal{D}_{ss}$ of the form $F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$:*

$$\Phi_F(\vec{x}, a, s) \supset F(\vec{x}, do(a, s)).$$

Then $\mathcal{P}$ is a correct (i.e. sound, but perhaps not complete) Prolog implementation of the basic action theory $\mathcal{D}$ for proving regressable sentences.

To prove a regressable sentence G, first transform it using the revised Lloyd-Topor transformations, then issue the resulting query to $\mathcal{P}$.

Example 3.3 *The stock trading domain includes the following sentences, obtained after transforming the if-halves of the corresponding definitions according to the Implementation Theorem.*

Precondition Axioms. *These are the following sentences.*

$$\neg price(stockId, price, time, s) \supset Poss(price\_insert(stockId, price, time), s),$$
$$\neg stock(stockId, price, closingPrice, s) \supset Poss(stock\_insert(stockId, price, closingPrice), s),$$
$$\neg customer(custId, balance, stockId, s) \supset Poss(customer\_insert(custId, balance, stockId), s),$$
$$price(stockId, price, time, s) \supset Poss(price\_delete(stockId, price, time, s)),$$
$$stock(stockId, price, closingPrice, s) \supset Poss(stock\_delete(stockId, price, closingPrice), s),$$
$$customer(custId, balance, stockId, s) \supset Poss(customer\_delete(custId, balance, stockId), s).$$

Successor State Axioms. *They are the following sentences.*

$$a = price\_insert(stockId, price, time) \supset price(stockId, price, time, do(a, s)),$$
$$price(stockId, price, time, s) \wedge a \neq price\_delete(stockId, price, time) \supset \\ price(stockId, price, time, do(a, s)),$$
$$a = stock\_insert(stockId, price, closingPrice) \supset stock(stockId, price, closingPrice, do(a, s)),$$
$$stock(stockId, price, closingPrice, s) \wedge a \neq stock\_delete(stockId, price, closingPrice) \supset \\ stock(stockId, price, closingPrice, do(a, s)),$$
$$a = customer\_insert(custId, balance, stockId) \supset customer(custId, balance, stockId, do(a, s)),$$
$$customer(custId, balance, stockId, s) \wedge a \neq customer\_delete(custId, balance, stockId) \supset \\ customer(custId, balance, stockId, do(a, s)).$$

Initial Database. *This is*

$price(ST1, \$100, 100100 : 4PM), price(ST2, \$110, 100100 : 9AM), price(ST3, \$100, 100100 : 1PM),$
$stock(ST1, \$100, \$100), stock(ST2, \$110, \$100), stock(ST3, \$50, \$60), customer(Sue, \$10000, ST1),$
$customer(Zang, \$5000, ST2).$

■

3.5 Summary

In the current chapter, we described a knowledge representation scheme based on the situation calculus for modeling dynamic domains. The scheme turns around basic action theories. In particular, we stressed on the importance of the notion of regression in order to answer regressable queries posed against basic action theories and that of correct Prolog implementation of these theories. In the chapters to come, we will use this scheme to represent and reason about active relational databases. To start with this task, the next chapter will integrate this scheme with a theory of database transactions with ACID properties.

Chapter 4

Specifying Database Transactions

Thus far, specifically in the previous chapter, we have treated a general scheme for representing dynamic domains. This scheme has been used in [Rei95] to specify database updates. In this chapter, we extend this specification with a theory of database transactions with ACID properties.

In Chapter 2, we have followed the distinction made in the classical transaction theory between transactions and transaction programs. A transaction can be viewed as a particular execution (or execution trace) of a transaction program which is written in an Algol-like language. It is important to keep this distinction in mind when modeling transactions. The present chapter deals with execution traces as opposed to transaction programs modeled in the next chapter. So the present chapter really models transactions per se, not transaction programs. Moreover, database transaction models covered here are concerned with execution traces, not with programs.

On the other hand, the database community seems confused about the aforementioned distinction when it comes to advanced transactions, since (with the notable exception of some versions of nested transactions) many so-called advanced transactions usually are just programming methodologies for writing transaction programs.

In this chapter, we propose a theory of database transactions that deals only with execution traces, and not with programs, thus a theory of transactions, not of transaction programs.

We start the chapter by extending the general scheme of the previous chapter to non-Markovian theories, which explain change in terms of all past situations and not solely based on the previous situation. Next, Section 4.1.2 introduces the building blocks of our specification framework for representing relational database transactions in the situation calculus. In Section 4.2, we model relational flat databases transactions as second order theories called basic relational theories. Here, we will focus on using basic relational theories to model flat transactions with ACID properties. Section 4.3 deals with modeling of advanced transactions.

4.1 The Specification Language

4.1.1 Non-Markovian Control in the Situation Calculus

Including dynamic aspects into database formalization has emerged as an active area of research. In particular, a variety of proposals have been made concerning the formalization of the evolution of databases with respect to update operations. These approaches can be classified into procedural(essentially due to the work of Abiteboul and Vianu in [SV87], [Abi88], [AV88], [AV90]) and logical ([FUV83], [Win90], [GL90], [Gra91], [KM91], [BK98], [Rei95],[BPV99]). Procedural approaches deal with updates at the level of stored data. Logical approaches generally view updates as a removal and addition of sentences into the logical theory capturing the evolution of the database.

Proposals in [Rei95], and [BPV99] use the language of the situation calculus ([McC63], [Rei01]). These proposals use basic action theories for reasoning about actions that rely on two important assumptions: their axioms describe *deterministic* primitive actions, and their execution preconditions and effects depend solely on the current situation. The later assumption is what the control theoreticians call the *Markov property*. Thus both indeterminate and non-Markovian actions are precluded in the theories introduced in [Rei95] and [BPV99]. However, in formalizing database transactions, one quickly encounters settings where using non-Markovian actions and fluents are unavoidable. For example, a transaction may explicitly execute a $Rollback$ action to go back to the last database state s' in the past in which a $Begin$ action was executed; and an End action is executable only if it closes a bracket opened by a $Begin$ action in the past and no other transaction specific action occurred meanwhile. Thus more than one situation is involved in considering the semantics of actions such as $Begin$ and $Rollback$. Thus one clearly needs to address the issue of non-Markovian actions and fluents explicitly when formalizing database transactions, and researchers using the situation calculus or similar languages to account for updates and transactions are not addressing this need. Moreover, one also needs to accommodate for indeterminate actions that arise in realistic database settings in the form of null values. This issue, however, remains out of the scope of our thesis.

The inability of basic action theories of [Rei95] and [BPV99] to characterize both the truth value of fluents and the actions preconditions in the current situation in terms of more than one past situations is formally caused by the fact that the predicate $\sqsubset$ cannot be used on the right hand side of successor state and action precondition axioms.

Thus, a first step towards formalizing database transactions is to extend action precondition axioms of the form $Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$ by allowing $\Pi_A(\vec{x}, s)$ to be a formula with free variables among $\vec{x}, s$ that may mention the predicate $\sqsubset$. Similarly, we must extend successor state axioms of the form $F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$ by allowing $\Phi_F(\vec{x}, a, s)$ to be a formula with free variables among $\vec{x}, a, s$ that may mention the predicate $\sqsubset$. Moreover, the notion of uniformity in a given situation term seen in the precedent chapter must be extended to this new setting.

Following [Gab00], we call such an extension of action precondition and successor state axioms together with the foundational axioms, unique name axioms, and axioms describing the initial situation of Definition 3.1 a *non-Markovian basic action theory*. This is formally introduced as follows:

Definition 4.1 *([Gab00]) Let $\mathcal{L}_{sitcalc} = (\mathfrak{A}, \mathfrak{W}, \mathfrak{I})$ be the language of the situation calculus. Then a theory $\mathcal{D} \subseteq \mathfrak{W}$ is a* non-Markovian basic action theory *iff it has a set Act of actions and a set Fl of fluents, and, moreover, it is of the form*

$$\mathcal{D} = \mathcal{D}_f \cup \mathcal{D}_{ap} \cup \mathcal{D}_{ss} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0},$$

where $\mathcal{D}_f$, $\mathcal{D}_{ap}$, $\mathcal{D}_{ss}$, $\mathcal{D}_{una}$, and $\mathcal{D}_{S_0}$ are as in Definition 3.1, except that the action precondition axioms and the successor state axioms now mention the predicate $\sqsubset$ on their right hand sides.

Non-Markovian basic action theories are suited for expressing non-Markovian control in the situation calculus.

In order to represent relational database transactions, we need some appropriate restrictions on $\mathcal{L}_{sitcalc}$.

Definition 4.2 *A* basic relational language *is a subset of $\mathcal{L}_{sitcalc}$ that has the following restrictions on the alphabet $\mathfrak{A}$:*

1. *$\mathfrak{A}$ has a finite number of constants, but at least one.*

2. *$\mathfrak{A}$ has a finite number of action functions.*

3. *$\mathfrak{A}$ has a finite number of relational fluents.*

Fluents now contain a further argument specifying which transaction contributed to its truth value in a given log. The domain of this new argument could be arbitrarily set to, e.g., integers. So a *basic relational language* is a finite fragment of the situation calculus that is suitable for modeling relational database transactions. In the sequel of this chapter we shall introduce an extension of non-Markovian basic action theories called *basic relational theories* which is tailored to relational languages and shall devote the subsequent chapters to extending them. Again, at a few occasions, we shall use functional fluents and will introduce them whenever we need them.

For simplicity, we consider basic relational languages whose only primitive update operations correspond to insertion or deletion of tuples into relations. For each such relation of the form $F(\vec{x}, t, s)$, where $\vec{x}$ is a tuple of objects, t is a transaction argument, and s is a situation argument, a *primitive internal action* is a parameterized primitive action of the situation calculus of the form $F\_insert(\vec{x}, t)$ or $F\_delete(\vec{x}, t)$. Intuitively, $F\_insert(\vec{x}, t)$ and $F\_delete(\vec{x}, t)$ denote the actions of inserting the tuple $\vec{x}$ into and deleting it from the relation F by the transaction t, respectively; for convenience, we will abbreviate long symbols when necessary (e.g., $account\_insert(\vec{x}, t)$ will be abbreviated as $a\_insert(\vec{x}, t)$). Below, we will use the following

Abbreviation 4.1

$$writes(a, F, \vec{x}, t) =_{df} a = F\_insert(\vec{x}, t) \vee a = F\_delete(\vec{x}, t),$$

Abbreviation 4.2

$$reads(a, F, \vec{x}, t) =_{df} (\exists s) a = F\_reads(\vec{x}, t, s),$$

one for each fluent. Notice that Abbreviation 4.2 is typical of the database context where we need to include certain test actions in the log. We distinguish the primitive internal actions from *primitive external actions* which are $Begin(t)$, $Commit(t)$, $End(t)$, and $Rollback(t)$, whose meaning will be clear in the sequel of this chapter; these are external as they do not specifically affect the content of the database.[1] The argument t is a unique transaction identifier. Finally, the set of fluents of a relational language is partitioned into two disjoint sets, namely a set of *database fluents* and a set of *system fluents*. Intuitively, the database fluents represent the relations of the database domain, while the system fluents are used to formalize the processing of the domain. Usually, any functional fluent in a relational language will always be a system fluent.

4.1.2 The Specification Framework

In [Chr91], the following building blocks for transaction models are identified: *history*, intertransaction *dependencies*, *conflict* between operations, and *delegation* of responsibility for objects to a transaction. We now show how these building blocks are represented in the situation calculus.

In the situation calculus, the history of [Chr91] corresponds to the log. We extend the basic action theories of [Rei01] to include a specification of relational database transactions, by giving action precondition axioms for external actions such as $Begin(t)$, $End(t)$, $Commit(t)$, $Rollback(t)$, $Spawn(t, t')$, etc. $Commit(t)$ and $Rollback(t)$ are coercive actions that must occur whenever they are possible. We also give successor state axioms that state how change occurs in databases in the presence of both internal and external actions. All these axioms provide the *first dimension* of the situation calculus framework for axiomatizing transactions, namely the axiomatization of the effects of transactions on fluents; they also comprise axioms indicating which transactions are conflicting with each other.

A useful concept that underlies most of the transaction models is that of responsibility over changes operated on data items. For example, in a nested transaction, a parent transaction will take responsibility of changes done by any of its committed children. The only way we can keep track of those responsibilities is to look at the transaction arguments of the actions present in the log. To that end, we introduce a system fluent $responsible(t, a, s)$, which intuitively means that transaction t is responsible for the action a in the log s, which we characterize with an appropriate successor state axiom of the form

[1]The terminology internal versus *external* action is also used in [LMWF94], though with a different meaning.

$responsible(t, a', do(a, s)) \equiv \Phi_{tm}(t, a, a', s)$, where $\Phi_{tm}(t, a, a', s)$ is a transaction model-dependent first order formula whose only free variables are among t, a, a',, and s. For example, in the flat transactions, we will have the following, simple axiom:

$$responsible(t, a, s) \equiv \bigvee_{A \in \mathcal{A}} (\exists \vec{x}) a = A(\vec{x}, t)$$

i.e., each transaction is considered responsible for any action whose last argument bears its name; here, $\mathcal{A}$ is the set of actions of the relational language.

To express conflicts between transactions, we need the predicate $termAct(a, t)$ and the system fluents $updConflict(a, a', s)$ and $transConflict(t, t', s)$, whose intuitive meaning is that the action a is a terminal action of t, the action a is conflicting with the action a' in s, and the transaction t is conflicting with the transaction t' in s; their characterization is as follows:

Abbreviation 4.3

$$termAct(a, t) =_{df} a = Commit(t) \lor a = Rollback(t);$$

Abbreviation 4.4

$$\begin{aligned} updConflict(a, a', s) =_{df} (\exists \vec{x}, t, t').\ & \bigvee_{F \in \mathcal{F}} \neg[F(\vec{x}, t, do(a, do(a', s))) \equiv F(\vec{x}, t', do(a', do(a, s)))] \lor \\ & \bigvee_{F \in \mathcal{F}} [writes(a, F, \vec{x}, t) \land writes(a', F, \vec{x}, t') \lor \\ & \quad writes(a, F, \vec{x}, t) \land reads(a', F, \vec{x}, t') \lor \\ & \quad reads(a, F, \vec{x}, t) \land writes(a', F, \vec{x}, t')]. \end{aligned}$$

In Abbreviation 4.4, $\mathcal{F}$ is the set of database fluents of the relational language; this definition says that two internal actions a and a' conflict in the log s iff the value of the fluents depends on the order in which a and a' appear in s. Moreover, two internal actions conflict if at least one of them is a write operation. Notice that we assumed only reads and writes, to ease presentation. Any further database modifying action that is introduced must be accompanied by a change in the definition which would be adding a new disjunct in the right hand side of the abbreviation above. This is like the treatment of actions beyond reads and writes in [BHG87]. Notice also that if we remove the first disjunct in the right hand side of the abbreviation above, our definition of update conflict will capture the one that the database community uses.

Abbreviation 4.5

$$\begin{aligned} & transConflict(t, t', do(a, s)) \equiv t \neq t' \land responsible(t', a, s) \land \\ & \quad (\exists a', s')[responsible(t, a', s) \land do(a', s') \sqsubseteq s \land updConflict(a', a, s)] \lor \\ & \quad transConflict(t, t', s) \land \neg termAct(a, t); \end{aligned} \tag{4.1}$$

i.e., transaction t conflicts with transaction t' in the log s iff t' executes an internal action a after t has executed an internal action a' that conflicts with a in the log s.

Notice that we define $updConflict(a, a', s)$ in terms of performing action a and action a' one immediately after the other and vice-versa; in the definition of $transConflict(t, t', s)$, however, we allow action a' to be executed long before action a. This does not mean that actions that are performed between a' and a are irrelevant with respect to update conflicts. Rather, the first disjunct in the right hand side of Abbreviation (4.4) just means that actions a and a' conflict whenever executing one immediately after the other *would* results in a discrepancy in the truth value of at least one of the relational fluents; and Abbreviation (4.1) allows for the possibility of other update conflicts arising between a' and other actions before the execution of a.

A further useful system fluent that we provide in the general framework is $readsFrom(t, t', s)$. This is used in most transaction models as a source of dependencies among transactions. In the database tradition, it intuitively means that the transaction t reads a value written by the transaction t' in the log s. In general however, the axiomatizer must provide a successor state axiom for this fluent depending on the application.

The *second dimension* of the situation calculus framework is made of dependencies between transactions. All the dependencies expressed in ACTA ([Chr91]) can also be expressed in the situation calculus. As an example, we have:

Commit Dependency of t on t'

$$do(Commit(t), s) \sqsubset s^* \supset [do(Commit(t'), s') \sqsubseteq s^* \supset do(Commit(t'), s') \sqsubset do(Commit(t), s)];$$

i.e., if t commits in a log s^*, then, whenever t' also commits in s^*, t' commits before t.

Strong Commit Dependency of t on t'

$$(\exists s')do(Commit(t'), s') \sqsubset s^* \supset (\exists s)do(Commit(t), s) \sqsubseteq s^*;$$

i.e., if t' commits in a log s^*, then t must also commit in s^*.

Rollback Dependency of t on t'

$$(\exists s')do(Rollback(t'), s') \sqsubset s^* \supset (\exists s)do(Rollback(t), s) \sqsubseteq s^*;$$

i.e., if t' rolls back in a log s^*, then t must also roll back in that log.

Weak Rollback Dependency of t on t'

$$\begin{aligned}&do(Rollback(t'), s') \sqsubset s^* \supset\\ &\quad \{(\forall s)[s \sqsubset s^* \wedge do(Commit(t), s) \not\sqsubseteq do(Rollback(t'), s')] \supset (\exists s'')do(Rollback(t), s'') \sqsubseteq s^*\};\end{aligned}$$

i.e., if t' rolls back in a log s^*, then, whenever t does not commit before t', t must also roll back in s^*.

Begin on Commit Dependency of t on t'

$$do(Begin(t),s) \sqsubset s^* \supset (\exists s')do(Commit(t'),s') \sqsubset do(Begin(t),s) \sqsubset s^*;$$

i.e., if t begins in a log s^*, then, t' must commit before the beginning of t in s^*.

All these dependencies are properties of legal database logs of various transaction models.

To control dependencies that may develop among running transactions, we use a set of predicates denoting these dependencies. For example, we use $c\_dep(t,t',s)$, $sc\_dep(t,t',s)$, $r\_dep(t,t',s)$, $wr\_dep(t,t',s)$, and and $bc\_dep(t,t',s)$ to denote the commit, strong commit, rollback, weak rollback, and begin on commit dependencies, respectively. These are system fluents whose truth value is changed by the relevant transaction models by taking into account dependencies generated by the execution of its external actions (*external dependencies*) and those generated by the execution of its internal actions (*internal dependencies*). As an example, in the nested transaction model, we have the following successor state axiom for $wr\_dep(t,t',s)$:

$$\begin{aligned} wr\_dep(t,t',do(a,s)) \equiv a = Spawn(t,t') \vee \\ wr\_dep(t,t',s) \wedge \neg termAct(a,t) \wedge \neg termAct(a,t'). \end{aligned}$$

This says that a weak rollback dependency of t on t' arises in $do(a,s)$ when either a is the action of t spawning t', or that dependency existed already in s and neither t nor t' terminated with the action a.

4.2 Specifying Flat Transactions Models

4.2.1 Basic Relational Theories

Flat transactions exhibit ACID properties. This section introduces a characterization of flat transactions in terms of theories of the situation calculus. These theories give axioms of flat transaction models that constrain database logs in such a way that these logs satisfy important correctness properties of database transaction, including the ACID properties.

Definition 4.3 (Flat Transaction) *A sequence of database actions is a* flat transaction *iff it is one of the following:*

1. *Atomic transaction:* $[a_1, \ldots, a_n]$, *where the* a_1 *must be* $Begin(t)$, *and* a_n *must be either* $Commit(t)$, *or* $Rollback(t)$; a_i, $i = 2, \cdots, n$–1, *may be any of the primitive actions, except* $Begin(t)$, $Rollback(t)$, *and* $Commit(t)$; *here, the argument* t *is a unique identifier for the atomic transaction.*

2. *Transaction:* $at_1 \bullet \ldots \bullet at_m$, *where the* at_i, $1 \leq i \leq m$, *are atomic transactions.*[2]

Notice that we do not introduce a term of a new sort for transactions, as is the case in [BPV99]; we treat transactions as run-time activities — execution traces — whose design-time counterparts will be Con-

[2]Given two atomic transactions $A = [A_1, \cdots, A_n]$ and $B = [B_1, \cdots, B_m]$, $A \bullet B$ is an abbreviation for $[A_1, \cdots, A_n, B_1, \cdots, B_m]$.

Golog programs introduced in the next chapter. We refer to transactions by their names that are of sort *object*. Notice also that, on this definition, a transaction is a semantical construct which will be denotations of situations of a special kind called legal logs in the next section.

The axiomatization of a dynamic relational database with flat transaction properties comprises the following classes of axioms:

Foundational Axioms. These are constraints imposed on the structure of database logs ([PR99], See Theorem 3.2):

$$do(a_1, s_1) = do(a_2, s_2) \supset a_1 = a_2 \land s_1 = s_2, \tag{4.2}$$

$$(\forall P).P(S_0) \land (\forall a, s)[P(s) \supset P(do(a, s))] \supset (\forall s)P(s), \tag{4.3}$$

$$\neg(s \sqsubset S_0), \tag{4.4}$$

$$s \sqsubset do(a, s') \equiv s \sqsubseteq s'. \tag{4.5}$$

These characterize database logs as finite sequences of updates and were proven to be valid in the previous chapter. Notice that the second axiom is a second-order induction axiom; the third and fourth axioms characterize the subsequence relation $\sqsubset$.

Integrity Constraints. These are constraints imposed on the data in the database at a given situation s; their set is denoted by $\mathcal{IC}_e$ for constraints that must be enforced at each update execution, and by $\mathcal{IC}_v$ for those that must be verified at the end of the flat transaction.

Update Precondition Axioms. There is one for each internal action $A(\vec{x}, t)$, with syntactic form

$$Poss(A(\vec{x}, t), s) \equiv (\exists t')\Pi_A(\vec{x}, t', s) \land IC_e(do(A(\vec{x}, t), s)) \land running(t, s). \tag{4.6}$$

Here, $\Pi_A(\vec{x}, t, s)$ is a first order formula with free variables among $\vec{x}$, t, and s. Moreover, the formula on the right hand side of (4.6) is uniform in s.[3] These axioms characterize the preconditions of the update A; $IC_e(s)$ and $running(t, s)$ are defined as follows:[4]

Abbreviation 4.6 $IC_e(s) =_{df} \bigwedge_{IC \in \mathcal{IC}_e} IC(s)$.

Abbreviation 4.7

$$\begin{aligned} running(t, s) =_{df}\ & (\exists s').do(Begin(t), s') \sqsubseteq s \land \\ & (\forall a, s'')[do(Begin(t), s') \sqsubset do(a, s'') \sqsubset s \supset a \neq Rollback(t) \land a \neq End(t)]. \end{aligned}$$

[3]See Appendix A for the formal definition of uniformity in a given situation term.

[4]In fact, the definition of $running(t, s)$ could be replaced by the successor state axiom

$$running(t, do(a, s)) \equiv a = Begin(t) \lor running(t, s) \land a \neq Rollback(t) \land a \neq End(t).$$

However, we keep the definition to avoid introducing too much system fluents.

In a banking Credit/Debit example formalized below (in Section 4.4), the following states that it is possible to delete a tuple specifying the teller identity tid and balance $tbal$ into the $tellers$ relation relative to the database log s iff, as a result of performing the actions in the log, that tuple would not already be present in the $tellers$ relation, the integrity constraints are satisfied, and transaction t is running.

$$\begin{aligned}Poss(t\_delete(tid, tbal, t), s) \equiv & (\exists t')tellers(tid, tbal, t', s) \wedge \\ & IC_e(do(t\_delete(tid, tbal, t), s)) \wedge running(t, s).\end{aligned} \tag{4.7}$$

Successor State Axioms. These have the syntactic form

$$\begin{aligned}F(\vec{x}, t, do(a, s)) \equiv & (\exists \vec{t_1})\Phi_F(\vec{x}, a, \vec{t_1}, s) \wedge \neg(\exists t'')a = Rollback(t'') \vee \\ & (\exists t'')a = Rollback(t'') \wedge restoreBeginPoint(F, \vec{x}, t'', s),\end{aligned} \tag{4.8}$$

There is one such axiom for each database relational fluent F. The formula on the right hand side of (4.8) is uniform in s, and $\Phi_F(\vec{x}, a, \vec{t}, s)$ is a formula with free variables among $\vec{x}, a, \vec{t}, s$; $\Phi_F(\vec{x}, a, \vec{t}, s)$ stands for the right hand side of the successor state axioms of Section 3.2.2 and has the following canonical form ([Rei01]):

$$\gamma_F^+(\vec{x}, a, \vec{t}, s) \vee F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, \vec{t}, s), \tag{4.9}$$

where $\gamma_F^+(\vec{x}, a, \vec{t}, s)$ ($\gamma_F^-(\vec{x}, a, \vec{t}, s)$) denotes a first order formula specifying the conditions that make a fluent F true (false) in the situation following the execution of an update a.

The predicate $restoreBeginPoint(F, \vec{x}, t, s)$ is defined as follows:

Abbreviation 4.8

$$\begin{aligned}
& restoreBeginPoint(F, \vec{x}, t, s) =_{df} \\
& \quad \{(\exists a_1, a_2, s', s_1, s_2, t'). \\
& \quad\quad do(Begin(t), s') \sqsubset do(a_2, s_2) \sqsubset do(a_1, s_1) \sqsubseteq s \wedge writes(a_1, F, \vec{x}, t) \wedge writes(a_2, F, \vec{x}, t') \wedge \\
& \quad\quad\ [(\forall a'', s'').do(a_2, s_2) \sqsubset do(a'', s'') \sqsubset do(a_1, s_1) \supset \neg writes(a'', F, \vec{x}, t)] \wedge \\
& \quad\quad\ [(\forall a'', s'').do(a_1, s_1) \sqsubseteq do(a'', s'') \sqsubset s \supset \neg(\exists t'')writes(a'', F, \vec{x}, t'')] \wedge (\exists t'')F(\vec{x}, t'', s_1)]\} \vee \\
& \quad \{(\forall a^*, s^*, s').do(Begin(t), s') \sqsubset do(a^*, s^*) \sqsubseteq s \supset \neg writes(a^*, F, \vec{x}, t)] \wedge (\exists t')F(\vec{x}, t', s)\}.
\end{aligned}$$

Notice that system fluents have successor state axioms that have to be specified on a case by case basis and do not necessarily have the form (4.8). Intuitively, $restoreBeginPoint(F, \vec{x}, t, s)$ means that the system restores the value of the database fluent F with arguments $\vec{x}$ in a particular way:

- The first disjunct in Abbreviation 4.8 captures the scenario where the transactions t and t' running in parallel, and writing into and reading from F are such that t overwrites whatever t' writes before it (t) rolls back. Suppose that t and t' are such that t begins, and eventually writes into F before rolling back; t' begins after t has begun, writes into F before the last write action of t, and commits before t rolls back. Now the second disjunct in 4.8 says that the value of F must be set to the "before image" ([BHG87]) of the first $w(t)$, that is, the value the F had just before the first $w(t)$ was executed.

- The second disjunct in Abbreviation 4.8 captures the case where the value F had at the beginning of the transaction that rolls back is kept.

Given the actual situation s, the successor state axiom characterizes the truth values of the fluent F in the next situation $do(a, s)$ in terms of all the past situations. Notice that Abbreviation 4.8 captures the intuition that $Rollback(t)$ affects all tuples within a table.

In the banking example, the following successor state axiom (4.10) states that the tuple $(tid, tbal)$ will be in the $tellers$ relation relative to the log $do(a, s)$ iff the last database operation a in the log inserted it there, or it was already in the $tellers$ relation relative to the log s, and a didn't delete it; all this, provided that the operation a is not rolling the database back. In the case the operation a is rolling the database back, the $tellers$ relation will get a value according to the logic of (4.8).

$$\begin{aligned} tellers(tid, tbal, t, do(a, s)) \equiv {} & ((\exists t_1)a = t\_insert(tid, tbal, t_1) \vee (\exists t_2)tellers(tid, tbal, t_2, s) \wedge \\ & \neg(\exists t_3)a = t\_delete(tid, tbal, t_3)) \wedge \neg(\exists t')a = Rollback(t') \vee \\ & (\exists t').a = Rollback(t') \wedge restoreBeginPoint(tellers, (tid, tbal), t', s). \end{aligned} \tag{4.10}$$

In this successor state axiom, the formula

$$(\exists t_1)a = t\_insert(tid, tbal, t_1) \vee (\exists t_2)tellers(tid, tbal, t_2, s) \wedge \neg(\exists t_3)a = t\_delete(tid, tbal, t_3)$$

corresponds to the canonical form 4.9.

Precondition Axioms for External Actions. This is a set of action precondition axioms for the transaction specific actions $Begin(t)$, $End(t)$, $Commit(t)$, and $Rollback(t)$. The external actions of flat transactions have the following precondition axioms:[5]

$$Poss(Begin(t), s) \equiv \neg(\exists s')do(Begin(t), s') \sqsubseteq s, \tag{4.11}$$

$$Poss(End(t), s) \equiv running(t, s), \tag{4.12}$$

$$\begin{aligned} & Poss(Commit(t), s) \equiv (\exists s').s = do(End(t), s') \wedge \\ & \bigwedge_{IC \in \mathcal{IC}_v} IC(s) \wedge (\forall t')[sc\_dep(t, t', s) \supset (\exists s'')do(Commit(t'), s'') \sqsubseteq s], \end{aligned} \tag{4.13}$$

[5]It must be noted that, in reality, a part of rolling back and committing lies with the user and another part lies with the system. So, we could in fact have something like $Rollback_{sys}(t)$ and $Commit_{sys}(t)$ on the one hand, and $Rollback_{usr}(t)$ and $Commit_{usr}(t)$ on the other hand. However, the discussion is simplified by considering only the system's role in executing these actions.

$$Poss(Rollback(t), s) \equiv (\exists s')[s = do(End(t), s') \wedge \neg \bigwedge_{IC \in \mathcal{IC}_v} IC(s)] \vee (\exists t', s'')[r\_dep(t, t', s) \wedge do(Rollback(t'), s'') \sqsubseteq s]. \tag{4.14}$$

Notice that our axioms (4.11)–(4.14) assume that the user will only use internal actions $Begin(t)$ and $End(t)$ and the system will execute either $Commit(t)$ or $Rollback(t)$.

Dependency axioms. These are transaction model-dependent axioms of the form

$$dep(t, t', s) \equiv \mathcal{C}(t, t', s), \tag{4.15}$$

where $\mathcal{C}(t, t', s)$ is a condition involving a relationship between transactions t and t', and $dep(t, t', s)$ is one of the dependency predicates $c\_dep(t, t', s)$, $sc\_dep(t, t', s)$, etc. In the classical case, we have the following axioms:

$$r\_dep(t, t', s) \equiv transConflict(t, t', s), \tag{4.16}$$

$$sc\_dep(t, t', s) \equiv readsFrom(t, t', s). \tag{4.17}$$

The first axiom says that a transaction conflicting with another transaction generates a rollback dependency, and the second says that a transaction reading from another transaction generates a strong commit dependency. Axioms (4.16) and (4.17) generate internal dependencies.

Unique Names Axioms. These state that the primitive updates and the objects of the domain are pairwise unequal.

Initial Database. This is a set of first order sentences specifying the initial database state. They are completion axioms of the form

$$(\forall \vec{x}, t).F(\vec{x}, t, S_0) \equiv \vec{x} = \vec{C}^{(1)} \vee \ldots \vee \vec{x} = \vec{C}^{(r)}, \tag{4.18}$$

one for each (database or system) fluent F. Here, the $\vec{C}^i$ are tuples of constants. Also, $\mathcal{D}_{S_0}$ includes unique name axioms for constants of the database. Axioms of the form (4.18) say that our theories accommodate a complete initial database state, which is commonly the case in relational databases as unveiled in [Rei84]. This requirement is made to keep the theory simple and to reflect the standard practice in databases. It has the theoretical advantage of simplifying the establishment of logical entailments in the initial database; moreover, it has the practical advantage of facilitating rapid prototyping of the ATMs using Prolog which embodies negation by failure, a notion close to the completion axioms used here.

One striking feature of our axioms is the use of the predicate $\sqsubset$ on the right hand side of action precondition axioms and successor state axioms. That is, they are capturing the notion of a situation being located in the past relative to the current situation which we express with the predicate $\sqsubset$ in the situation calculus. Thus they are capturing non-Markovian control. We call these axioms a *basic relational theory*, and introduce the following:

Definition 4.4 *(***Basic Relational Theory***) Suppose* $\mathfrak{R} = (\mathfrak{A}, \mathfrak{W})$ *is a basic relational language.*[6] *Then a theory* $\mathcal{D} \subseteq \mathfrak{W}$ *is a* non-Markovian basic relational theory *iff it is of the form*

$$\mathcal{D} = \mathcal{D}_f \cup \mathcal{D}_{IC} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{ss} \cup \mathcal{D}_{FT} \cup \mathcal{D}_{dep} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0}$$

where

1. *$\mathfrak{A}$ comprises, in addition to the internal actions, the external actions $Begin(t)$, $End(t)$, $Commit(t)$, and $Rollback(t)$.*

2. *$\mathcal{D}_f$ is the set of foundational axioms.*

3. *$\mathcal{D}_{IC}$ is a set of integrity constraints $IC(s)$. More specifically, we have built-in ICs ($\mathcal{D}_{IC_e}$) and generic ICs ($\mathcal{D}_{IC^v}$). Built-in ICs are: not null attributes, primary keys, and uniqueness ICs.*

4. *$\mathcal{D}_{ap}$ is a set of non-Markovian action precondition axioms of the form (4.6), one for each primitive internal action of $\mathfrak{R}$.*

5. *$\mathcal{D}_{ss}$ is a set of non-Markovian successor state axioms of the form (4.8), one for each database fluent of $\mathfrak{R}$. Also, $\mathcal{D}_{ss}$ includes successor state axioms for all the system fluents of the flat transaction model.*

6. *$\mathcal{D}_{FT}$ is a set of action precondition axioms for the primitive external actions of $\mathfrak{R}$.*

7. *$\mathcal{D}_{dep}$ is a set of dependency axioms.*

8. *$\mathcal{D}_{una}$ consists of unique names axioms for objects and for actions.*

9. *$\mathcal{D}_{S_0}$ is an initial relational theory, i.e. a set of completion axioms of the form*

   $$(\forall \vec{x}).F(\vec{x}, S_0) \equiv \vec{x} = \vec{C}^{(1)} \vee \ldots \vee \vec{x} = \vec{C}^{(r)},$$

   one for each fluent F whose interpretation contains r n-tuples, together with completion axioms of the form $(\forall \vec{x})\neg F(\vec{x}, S_0)$, one for each fluent F whose interpretation is empty. Also, $\mathcal{D}_{S_0}$ includes unique name axioms for constants of the database.

Definition 4.5 *(***Relational Database***) A relational database is a pair $(\mathfrak{R}, \mathcal{D})$, where $\mathfrak{R}$ is a relational language and $\mathcal{D}$ is a basic relational theory.*

4.2.2 Strict Schedules that Avoid Cascaded Rollback

Cascading rollback is the phenomenon where rolling back a transaction leads to further transactions rolling back. Usually, practical DBMSs try to avoid this phenomenon. Our formalism so far captures

[6]In most of what follows, we omit the $\mathfrak{I}$ component of relational languages whenever the context is clear.

general logs that allow cascading rollbacks. This can be seen by examining Axiom (4.14). This axiom implies that a rollback of a transaction t is possible whenever a transaction on which t is rollback dependent has indeed rolled back. Since – as we will see below – rollback is an action that must occur whenever it is possible, rollback dependency may lead to a cascading rollback. Therefore, we must spend toughts on how to capture the notion of avoiding cascading rollbacks in the situation calculus.

Following [BHG87], a schedule – corresponding to our log – avoids cascading rollback (is strict) if a transaction reads (reads and writes) a database item only if all transactions that previously wrote that item have committed or aborted. Then, a possible way of capturing these restrictions in our framework would be to do the following:

- First, add the condition "if all transactions that previously wrote the item have committed or aborted" to the action precondition axioms (4.6). Doing so, we obtain the following general form for the action precondition axioms:

$$\begin{aligned}
&Poss(A(\vec{x},t),s) \equiv \\
&\quad (\exists t')\Pi_A(\vec{x},t',s) \wedge IC_e(do(A(\vec{x},t),s)) \wedge running(t,s) \wedge \\
&\quad [(\forall F,t',a).[reads(a,F,\vec{x},t') \vee writes(a,F,\vec{x},t')] \supset \\
&\qquad\qquad [(\exists s').do(Commit(t'),s') \sqsubset s \vee do(Rollback(t'),s') \sqsubset s]].
\end{aligned} \tag{4.19}$$

  Here, as before, $\Pi_A(\vec{x},t,s)$ is a first order formula with free variables among $\vec{x},t,$ and s, and the formula on the right hand side of (4.19) is uniform in s. This corresponds to the database practice of suitably delaying read and write operations to achieve strict schedules.

- The successor state axioms will remain as formalized in (4.8) and Abbreviation 4.8.

Notice that the most general case of rollack should expunge the actions of a rolled back transaction t from the history. In the situation calculus, this amounts to striking out t's actions from the log. Now, since many transactions may be running in parallel, the end effect of the this striking out should not disturb the rest of the transactions. We do not pursue this case here further. Also, for ease of presentation, the rest of this thesis will assume the basic logs that allow cascading rollbacks.

4.2.3 Legal Flat Transactions

A fundamental property of $Rollback(t)$ and $Commit(t)$ actions is that, the database system *must* execute them in any database state in which they are possible. In this sense, they are coercive actions, and we call them *system actions*:

Abbreviation 4.9

$$systemAct(a,t) =_{df} a = Commit(t) \vee a = Rollback(t).$$

We constrain legal logs to include these mandatory system actions, as well as the requirement that all actions in the log be possible:

Abbreviation 4.10

$$\begin{aligned} legal(s) =_{df} & (\forall a, s^*)[do(a, s^*) \sqsubseteq s \supset Poss(a, s^*)] \wedge \\ & (\forall a', a'', s', t)[systemAct(a', t) \wedge responsible(t, a', s') \wedge \\ & responsible(t, a'', s') \wedge Poss(a', s') \wedge do(a'', s') \sqsubset s \supset a' = a'']. \end{aligned} \quad (4.20)$$

Let $T = AT_1 \bullet \cdots \bullet AT_m$ be a transaction, where the AT_i are atomic transactions, and recall that $do(T, S_0)$ is an abbreviation for $do(A_n, do(A_{n-1}, \cdots, do(A_1, S_0) \cdots))$. Then, we extend the notion of legality to transactions by requiring that legal transactions be those that lead to legal database situations. Thus whenever we can establish that $\mathcal{D} \models legal(do(T, S_0))$, we will say that T is a legal transaction.

In proving the different properties of transaction models, the following three lemmas exhibiting simple properties of legal logs will be useful.

Lemma 4.6 *Let $\mathcal{D}$ be a basic relational theory. Then*

$$\begin{aligned} \mathcal{D}_f \cup \{(4.20)\} \models (\forall s, a)\{ & legal(S_0) \wedge \\ & [legal(do(a, s)) \equiv legal(s) \wedge Poss(a, s) \wedge \\ & (\forall a', t).systemAct(a', t) \wedge responsible(t, a', s) \wedge Poss(a', s) \supset a = a']\}. \end{aligned}$$

Lemma 4.7 *Suppose $\mathcal{D}$ is a basic relational theory. Then*

$$\mathcal{D} \cup \{(4.20)\} \models legal(s) \supset (\forall s')[s' \sqsubseteq s \supset legal(s')].$$

Lemma 4.8 *Suppose $[a_1, \cdots, a_n]$ is a sequence of ground action terms. Then*

$$\begin{aligned} \mathcal{D}_f \cup \{(4.20)\} \models & legal(do([a_1, \cdots, a_n], s)) \equiv \\ & \bigwedge_{i=1}^{n} \{Poss(a_i, do([a_1, \cdots, a_{i-1}], s)) \wedge \\ & (\forall a, t)[systemAct(a, t) \wedge responsible(t, a, do([a_1, \cdots, a_{i-1}], s)) \wedge \\ & Poss(a, do([a_1, \cdots, a_{i-1}], s)) \supset a_i = a]\}. \end{aligned}$$

4.2.4 Properties

Simple properties such as well-formedness of atomic transactions ([LMWF88]) can be formulated in the situation calculus and proven as logical consequences of basic relational theories. We first introduce the following abbreviation:

Abbreviation 4.11

$$externalAct(a,t) =_{df} a = Begin(t) \vee a = End(t) \vee a = Commit(t) \vee a = Rollback(t).$$

Theorem 4.9 (Well-Formedness of Flat Transactions) *Suppose $\mathcal{D}$ is a basic relational theory. Then*

1. *No external action may occur twice in a legal log; i.e.,*

$$\mathcal{D} \models legal(s) \supset \{do(a,s') \sqsubset s \wedge do(a,s'') \sqsubset s \wedge externalAct(a,t) \supset s' = s''\}.$$

2. *There are no dangling $Commit$ or $Rollback$ actions; i.e.,*

$$\begin{aligned}&\mathcal{D} \models legal(s) \supset \\ &\qquad \{[do(Commit(t),s') \sqsubset s \supset (\exists s'')do(Begin(t),s'') \sqsubset do(Commit(t),s')] \wedge \\ &\qquad [do(Rollback(t),s') \sqsubset s \supset (\exists s'')do(Begin(t),s'') \sqsubset do(Rollback(t),s')]\}.\end{aligned}$$

3. *No transaction may commit and then roll back, and conversely; i.e.,*

$$\begin{aligned}&\mathcal{D} \models legal(s) \supset \\ &\qquad \{[do(Commit(t),s') \sqsubset s \supset \neg(\exists s'')do(Rollback(t),s'') \sqsubset s] \wedge \\ &\qquad [do(Rollback(t),s') \sqsubset s \supset \neg(\exists s'')do(Commit(t),s'') \sqsubset s]\}.\end{aligned}$$

These properties are similar to the fundamental axioms, applicable to all transactions, of [Chr91]. They are well-formedness properties since they rule out all the ill-formed transactions such as

$[Begin(t), a\_insert(A_1,B_1,1000,T_1), Begin(t), a\_delete(A_1,B_1,1000,T_1), Commit(t)]$,

$[Begin(t), a\_insert(A_1,B_1,1000,T_1), Commit(t), a\_delete(A_1,B_1,1000,T_1), Commit(t)]$,

$[Begin(t), a\_insert(A_1,B_1,-1000,T_1), Commit(t), a\_delete(A_1,B_1,-1000,T_1), Rollback(t)]$, etc.

Theorem 4.10 *Suppose $\mathcal{D}$ is a basic relational theory. Then any legal log satisfies the strong commit and rollback dependency properties; i.e.,*

$$\begin{aligned}&\mathcal{D} \models legal(s) \supset \\ &\quad (\forall t,t').\{sc\_dep(t,t',s) \supset [(\exists s')do(Commit(t'),s') \sqsubset s \supset (\exists s^*)do(Commit(t),s^*) \sqsubseteq s]\} \wedge \\ &\quad \{r\_dep(t,t',s) \supset [(\exists s')do(Rollback(t'),s') \sqsubset s \supset (\exists s^*)do(Rollback(t),s^*) \sqsubset s]\}.\end{aligned}$$

Now we turn to the ACID properties, which are the most important properties of flat transactions.

Theorem 4.11 *(**Atomicity**) Suppose $\mathcal{D}$ is a relational theory. Then for every database fluent F*

$$\begin{aligned}&\mathcal{D} \models legal(s) \supset \\ &\qquad (\forall t,a,s_1,s_2)\{[do(Begin(t),s_1) \sqsubset do(a,s_2) \sqsubset s] \wedge \\ &\qquad\quad (\exists a^*,s^*,\vec{x})[do(Begin(t),s_1) \sqsubset do(a^*,s^*) \sqsubset do(a,s_2) \wedge writes(a^*,F,\vec{x},t)] \supset \\ &\qquad\qquad [(a = Rollback(t) \supset ((\exists t_1)F(\vec{x},t_1,do(a,s_2)) \equiv (\exists t_2)F(\vec{x},t_2,s_1))) \wedge \\ &\qquad\qquad (a = Commit(t) \supset ((\exists t_1)F(\vec{x},t_1,do(a,s_2)) \equiv (\exists t_2)F(\vec{x},t_2,s_2)))]\}.\end{aligned}$$

This says that rolling back restores any database fluent to the value it had just before the last $Begin(t)$ action, and committing endorses the value it had in the situation just before the $Commit(t)$ action.

Theorem 4.12 (Consistency) *Suppose $\mathcal{D}$ is a relational theory. Then all integrity constraints are satisfied at committed logs; i.e.,*

$$\mathcal{D} \models legal(s) \supset \{do(Commit(t), s') \sqsubseteq s \supset \bigwedge_{IC \in \mathcal{IC}_v \cup \mathcal{IC}_e} IC(do(Commit(t), s'))\}.$$

Theorem 4.13 *$\mathcal{D}$ is satisfiable iff $\mathcal{D}_{S_0} \cup \mathcal{D}_{una} \cup \mathcal{D}_{IC}[S_0]$ is.[7] In other words, provided the constraints are consistent with the initial database state and unique names for actions, then the entire relational theory is satisfiable, and conversely.*

Some properties of transactions need the notions of committed and rolled back updates. With the predicates $committed(a, s)$ and $rolledBack(a, s)$, we express these notions in the situation calculus using the following definitions:

$$committed(a, s) =_{df} (\exists t, s').responsible(t, a, s) \wedge do(Commit(t), s') \sqsubseteq s, \tag{4.21}$$

$$rolledBack(a, s) =_{df} (\exists t, s').responsible(t, a, s) \wedge do(Rollback(t), s') \sqsubseteq s. \tag{4.22}$$

Theorem 4.14 (Durability) *Suppose $\mathcal{D}$ is a relational theory. Then whenever an update is committed or rolled back by a transaction, another transaction can not change this fact:*

$$\begin{aligned} \mathcal{D} \models & legal(s) \supset \{do(Rollback(t), s') \sqsubseteq s \supset \\ & \qquad [committed(a, s') \equiv committed(a, do(Rollback(t), s'))] \wedge \\ & \qquad [rolledBack(a, s') \equiv rolledBack(a, do(Rollback(t), s'))]. \end{aligned}$$

4.2.5 Flat Transactions with Savepoints

Flat transactions with savepoints are a variation of flat transactions which provides the user with a new external action $Save(t)$ to establish savepoints in the database log ([GA95]). The user program can roll back to those savepoints from later database logs. A flat transaction with savepoints is a sequence $[a_1, \ldots, a_n]$ of primitive actions, where a_1 must be $Begin(t)$, and a_n must be either $Commit(t)$, or $Rollback(t)$; $a_i, i = 2, \cdots, n-1$, may be any of the primitive actions including $Save(t)$ and $Rollback(t, n)$, except $Begin(t)$, $Commit(t)$, and $Rollback(t)$; a_{n-2} may be $End(t)$.

The external action $Rollback(t, n)$, where t is a transaction, and n is a monotonically increasing number – the savepoint –, brings the database back to the database state corresponding to that savepoint. With this action, we now can roll back with respect to savepoints; thus the precondition axiom for $Rollback(t, n)$, which now has a savepoint as argument, must be specified accordingly. If a $Rollback(t, n)$

[7]Here, $\mathcal{D}_{IC}[S_0]$ is the set $\mathcal{D}_{IC}$ relativized to the situation S_0.

action is executed in situation s, its effect is that we ignore any situation between some s' and s, where s' is the database log corresponding to the savepoint n. One way of doing this is to maintain a predicate $Ignore(t, s', s)$ in order to know which parts of the log to skip over. The following action precondition axioms and definition reflect these changes to the corresponding axioms for flat transactions of Section 4.2.1:

$$Poss(Save(t), s) \equiv running(t, s), \tag{4.23}$$

$$\begin{aligned} Poss(Rollback(t), s) \equiv\ & (\exists s')[s = do(End(t), s') \wedge \neg \bigwedge_{IC \in \mathcal{IC}_v} IC(s)] \vee \\ & (\exists t', s'')[r\_dep(t, t', s) \wedge do(Rollback(t'), s'') \sqsubseteq s] \vee \\ & (\exists t', n, s', s^*, s^{**})[s' \sqsubseteq s^* \sqsubseteq s^{**} \wedge r\_dep(t, t', s^*) \wedge \\ & \quad s' = sitAtSavePoint(t', n) \wedge do(Rollback(t', n), s^{**}) \sqsubseteq s], \end{aligned} \tag{4.24}$$

$$\begin{aligned} Poss(Rollback(t, n), s) \equiv\ & running(t, s) \wedge (\exists s').s' = sitAtSavePoint(t, n) \wedge \\ & s' \sqsubset s \wedge numOfSavePoints(t, s) \geq n \wedge \neg(\exists s^*, s^{**}).s^* \sqsubseteq s' \sqsubseteq s^{**} \wedge \\ & Ignore(t, s^*, s^{**}), \end{aligned} \tag{4.25}$$

$$\begin{aligned} Ignore(t, s', do(a, s)) \equiv\ & s' \sqsubseteq do(a, s) \wedge \\ & (\exists n).sitAtSavePoint(t, n) = s' \wedge a = Rollback(t, n), \end{aligned} \tag{4.26}$$

$$\begin{aligned} numOfSavePoints(t, do(a, s)) = n \equiv\ & a = Begin(t) \wedge n = 1 \vee \\ & a = Save(t) \wedge n = numOfSavePoints(t, s) + 1 \vee \\ & numOfSavePoints(t, s) = n \wedge a \neq Begin(t) \wedge a \neq Save(t), \end{aligned} \tag{4.27}$$

$$\begin{aligned} sitAtSavePoint(t, n) = s =_{df}\ & (\exists a, s').numOfSavePoints(t, s) = n \wedge \\ & s = do(a, s') \wedge (a = Begin(t) \vee a = Save(t)). \end{aligned} \tag{4.28}$$

In flat transactions with save points, successor state axioms for relations have the following form that reflects changes introduced by the external $Rollback(t, n)$ action.

$$\begin{aligned} & F(\vec{x}, t, do(a, s)) \equiv \\ & (\exists \vec{t_1}).\Phi_F(\vec{x}, a, \vec{t_1}, s) \wedge \neg(\exists t'')a = Rollback(t'') \wedge \neg(\exists t'', n)a = Rollback(t'', n) \vee \\ & (\exists t'')a = Rollback(t'') \wedge restoreBeginPoint(F, \vec{x}, t'', s) \vee \\ & (\exists n, t'').a = Rollback(t'', n) \wedge restoreSavePoint(F, \vec{x}, n, t'', s), \end{aligned} \tag{4.29}$$

one for each relation F, where $\Phi_F(\vec{x}, a, \vec{t_1}, s)$ is a formula with free variables among $a, s, \vec{x}, \vec{t_1}$; Abbreviation (4.8) defines $restoreBeginPoint(F, \vec{x}, t'', s)$, and $restoreSavePoint(F, \vec{x}, n, t'', s)$ is defined as follows:

Abbreviation 4.12

$$\begin{aligned}&restoreSavePoint(F, \vec{x}, n, t, s) =_{df}\\ &\qquad (\exists s').s' \sqsubset s \wedge sitAtSavePoint(t, n) = s' \wedge (\exists t')F(\vec{x}, t', s'),\end{aligned} \tag{4.30}$$

where $sitAtSavePoint(t, n)$ is a function returning the log relative to the transaction t at the savepoint n, defined by (4.28); $restoreSavePoint(F, \vec{x}, n, t, s)$ means that the value of the fluent F with arguments $\vec{x}$ is set back to the value it had at the sublog of s corresponding to the savepoint n established by the transaction t.

The dependency axioms have to be adapted to this new setting where dependencies that held previously may no longer hold as a consequence of the partial rollback mechanism; these axioms are now of the form

$$\begin{aligned}&dep(t, t', do(a, s)) \equiv \mathcal{C}(t, t', s) \wedge a \neq Rollback(t) \wedge a \neq Rollback(t') \wedge\\ &\quad \neg(\exists n, s')[(a = Rollback(t, n) \vee a = Rollback(t', n)) \wedge\\ &\quad sitAtSavePoint(t', n) = s' \wedge (\forall s'').s' \sqsubseteq s'' \sqsubseteq s \supset \neg dep(t, t', s'')],\end{aligned} \tag{4.31}$$

where $\mathcal{C}(t, t')$ and $dep(t, t', s)$ are defined as in (4.15); we have one such axiom for each dependency predicate.

A basic relational theory for flat transactions with savepoints is as in Definition 4.4, but where the relational language includes $Save(t)$ and $Rollback(t, n)$ as further actions, the axioms (4.23) – (4.28) are added to $\mathcal{D}_{FT}$, the set $\mathcal{D}_{ss}$ is a set of successor state axioms of the form (4.29), and the set $\mathcal{D}_{dep}$ is a set of dependency axioms of the form (4.31). All the other axioms of Definition 4.4 remain unchanged.

Properties

Now we turn to the ACID properties of flat transactions with savepoints. The introduction of the $Rollback(t, n)$ action modifies some of the previous theorems.

Lemma 4.15 *Suppose $\mathcal{D}$ is a relational theory. Then for every relational fluent F*

$$\begin{aligned}&\mathcal{D} \models legal(s) \supset\\ &\{[do(Begin(t), s_1) \sqsubset do(a, s_2) \sqsubset s] \wedge\\ &\quad (\exists a^*, s^*, \vec{x})[do(Begin(t), s_1) \sqsubset do(a^*, s^*) \sqsubset do(a, s_2) \wedge writes(a^*, F, \vec{x}, t)] \supset\\ &\qquad [(\exists n)(a = Rollback(t, n) \wedge sitAtSavePoint(t, n) = s') \supset\\ &\qquad\qquad (((\exists t_1)F(\vec{x}, t_1, do(a, s_2)) \equiv (\exists t_2)F(\vec{x}, t_2, s')))]\}.\end{aligned}$$

This tells us that $Rollback(t, n)$ does not fall under the all-or-nothing logic that characterizes flat transactions since the situation at a given savepoint of a transaction is not necessarily the same as the situation at the beginning of that transaction.

Note that Theorem 4.11 continues to hold for flat transactions with savepoints. Hence, from Theorem 4.11 and Lemma 4.15, we have the following

Corollary 4.16 *(***Atomicity of Transactions with Savepoints***) Suppose $\mathcal{D}$ is a relational theory. Then for every database fluent F*

$$
\begin{aligned}
&\mathcal{D} \models legal(s) \supset \\
&\qquad \{[do(Begin(t), s_1) \sqsubset do(a, s_2) \sqsubset s] \land \\
&\qquad\qquad (\exists a^*, s^*, \vec{x})[do(Begin(t), s_1) \sqsubset do(a^*, s^*) \sqsubset do(a, s_2) \land writes(a^*, F, \vec{x}, t)] \supset \\
&\qquad\qquad [[a = Rollback(t) \supset ((\exists t_1)F(\vec{x}, t_1, do(a, s_2)) \equiv (\exists t_2)F(\vec{x}, t_2, s_1))] \land \\
&\qquad\qquad [(\exists n)(a = Rollback(t, n) \land sitAtSavePoint(t, n) = s') \supset \\
&\qquad\qquad\qquad (((\exists t_1)F(\vec{x}, t_1, do(a, s_2)) \equiv (\exists t_2)F(\vec{x}, t_2, s')))] \\
&\qquad\qquad [a = Commit(t) \supset ((\exists t_1)F(\vec{x}, t_1, do(a, s_2)) \equiv (\exists t_2)F(\vec{x}, t_2, s_2))]]\}.
\end{aligned}
$$

Theorem 4.14, which also holds for flat transactions with savepoints, characterizes the durability of flat transactions with Savepoints. The consistency Theorem 4.12 also holds for flat transactions with savepoints, as does Theorem 4.13.

The following theorem establishes a fundamental property of transactions with savepoints: if a transaction rolls back to a given savepoint, say, n, all the updates on the way back to the situation corresponding to n are aborted, and no future rollback to the situations generated by these updates are possible.

Theorem 4.17 *Suppose $\mathcal{D}$ is a relational theory. Then*

$$
\begin{aligned}
&\mathcal{D} \models legal(s) \supset \\
&\qquad \{do(Rollback(t, n), s') \sqsubset s \quad \supset \\
&\qquad\quad [\neg(\exists n^*, s^*).do(Rollback(t, n), s') \sqsubset do(Rollback(t, n^*), s^*) \sqsubset s \land \\
&\qquad\qquad sitAtSavePoint(t, n) \sqsubseteq sitAtSavePoint(t, n^*) \sqsubset do(Rollback(t, n), s')]\}.
\end{aligned}
$$

4.2.6 Chained Flat Transactions

A chained flat transaction is a sequence $[a_1, \ldots, a_n]$ of primitive actions, where a_1 must be $Begin(t)$, and a_n must be either $Commit(t)$, or $Rollback(t)$; $a_i, i = 2, \cdots, n-1$, may be any of the primitive actions including $Chain(t)$, except $Begin(t)$ and $Commit(t)$.

Chained flat transactions are equivalent to the special case of flat transactions with save points, where only the most recent savepoint is restored. The intuition behind chained transactions is to allow committing work done so far, thus waiving any further right to execute a rollback over the committed logs ([GA95]). The new external action $Chain(t)$ is used by the programmer to commit work done so far and continue with work yet to be done. For any s, we call the situation $do(Chain(t), s)$ a chaining situation of transaction t.

The following action precondition axioms capture the essence of chained flat transactions:

$$\begin{aligned}
Poss(Rollback(t), s) \equiv\ & (\exists s')[s = do(End(t), s') \wedge \neg \bigwedge_{IC \in \mathcal{IC}_v} IC(s)] \vee \\
& (\exists t', s^*, s^{**}).do(Chain(t'), s^*) \sqsubset do(Rollback(t'), s^{**}) \sqsubseteq s \wedge \\
& \quad [(\forall a, s'').do(Chain(t'), s^*) \sqsubset do(a, s'') \sqsubset s^{**} \supset a \neq Chain(t)] \wedge \\
& \quad [(\exists s'').do(Chain(t'), s^*) \sqsubset s'' \sqsubset s^{**} \wedge r\_dep(t, t', s'')],
\end{aligned} \tag{4.32}$$

$$\begin{aligned}
Poss(Chain(t), s) \equiv\ & \bigwedge_{IC \in \mathcal{IC}_v} IC(s) \wedge running(t, s) \wedge \\
& (\forall t').c\_dep(t, t', s) \supset \\
& \quad (\exists s'')\{do(Commit(t'), s'') \sqsubseteq s \vee [do(Chain(t'), s'') \sqsubseteq s \wedge \\
& \quad\quad (\forall a^*, s^*)(do(Chain(t'), s'') \sqsubset do(a^*, s^*) \sqsubset s \supset \\
& \quad\quad\quad a^* \neq Chain(t) \wedge \neg c\_dep(t, t', s^*))]\}.
\end{aligned} \tag{4.33}$$

The later axiom is particularly critical: it prevents the user from chaining a transaction t that is commit-dependent on another transaction t' that has not committed before the last chaining situation of t. Axioms for $Begin(t)$, $End(t)$ and $Commit(t)$ remain unchanged.

Successor state axioms for fluents of chained flat transactions have the form (4.8), but with a different definition for $restoreBeginPoint$:

Abbreviation 4.13

$restoreBeginPoint(F, \vec{x}, t, s) =_{df}$

(1) $(\exists a, s').\{(a = Begin(t) \vee a = Chain(t)) \wedge$

(2) $(\forall a^*, s^*)[s' \sqsubset do(a^*, s^*) \sqsubset s \supset a^* \neq Chain(t) \wedge a^* \neq Begin(t)] \wedge$

(3) $\{\{(\exists a_1, a_2, s', s_1, s_2, t').$

(4) $do(a, s') \sqsubset do(a_2, s_2) \sqsubset do(a_1, s_1) \sqsubseteq s \wedge writes(a_1, F, \vec{x}, t) \wedge writes(a_2, F, \vec{x}, t') \wedge$

(5) $[(\forall a'', s'').do(a_2, s_2) \sqsubset do(a'', s'') \sqsubset do(a_1, s_1) \supset \neg writes(a'', F, \vec{x}, t)] \wedge$

(6) $[(\forall a'', s'').do(a_1, s_1) \sqsubseteq do(a'', s'') \sqsubset s \supset \neg(\exists t'')writes(a'', F, \vec{x}, t'')] \wedge (\exists t'')F(\vec{x}, t'', s)]\} \vee$

(7) $\{(\forall a^*, s^*, s').do(a, s') \sqsubset do(a^*, s^*) \sqsubseteq s \supset \neg writes(a^*, F, \vec{x}, t)] \wedge (\exists t')F(\vec{x}, t', s)\}\}.$

In the abbreviation above, lines (1) and (2) ensure that restoring the values of fluents is not done by rolling back over chaining situations. Lines (3)–(7) are as in Abbreviation 4.8.

The dependency axioms must now be adapted to this setting where dependencies that held previously may no longer hold as a consequence of the system rolling back to the last chaining situation; these axioms are now of the form

$$\begin{aligned} &dep(t,t',do(a,s)) \equiv \\ &\quad \mathcal{C}(t,t',s) \wedge a \neq Rollback(t) \wedge a \neq Rollback(t') \wedge a \neq Chain(t'), \end{aligned} \tag{4.34}$$

one for each dependency predicate; $\mathcal{C}(t,t')$ and $dep(t,t',s)$ are defined as in (4.15).

A basic relational theory for chained flat transactions is as in Definition 4.4, where $\mathfrak{A}$ comprises $Chain(t)$ as a further action, the set $\mathcal{D}_{FT}$ is modified accordingly to accommodate the new axioms (4.32)–(4.33), the set $\mathcal{D}_{ss}$ is now a set of successor state axioms that reflects the changes brought by Abbreviation 4.13, and the set $\mathcal{D}_{dep}$ is a set of dependency axioms of the form (4.34). All the other axioms of Definition 4.4 remain unchanged.

The following is a property specific to chained transactions. It captures the intuition behind chained transactions which is that, whenever chained, a database transaction can never roll back over the last chaining situation.

Theorem 4.18 *(***Durability of Chaining Situations***) Suppose $\mathcal{D}$ is a relational theory for chained flat transactions. Then, for all database fluents F*

$$\begin{aligned} \mathcal{D} \models [&do(Chain(t),s') \sqsubset do(Rollback(t),s'') \sqsubseteq s \wedge \\ &(\exists a^*,s^*,\vec{x})[do(Chain(t),s') \sqsubset do(a^*,s^*) \sqsubset do(Rollback(t),s'') \wedge writes(a^*,F,\vec{x},t)] \\ &\neg(\exists s^*)do(Chain(t),s') \sqsubset do(Chain(t),s^*) \sqsubset do(Rollback(t),s'')] \supset \\ &\qquad ((\exists t')F(\vec{x},t',do(Rollback(t),s'')) \equiv (\exists t'')F(\vec{x},t'',do(Chain(t),s'))). \end{aligned}$$

4.3 Specifying Advanced Transaction Models

4.3.1 Closed Nested Transactions

The main idea conveyed by the notion of nested transactions is that of levels of abstractions: each nesting in the hierarchy of nested transactions corresponds to a level of abstraction from the details of the action execution.

Nested transactions ([Mos85]) are the best known example of ATMs. A nested transaction is a set of transactions (called subtransactions) forming a tree structure, meaning that any given transaction, the parent, may spawn a subtransaction, the child, nested in it. A child commits only if its parent has committed. If a parent transaction rolls back, all its children are rolled back. However, if a child rolls back, the parent may execute a recovery procedure of its own. Each subtransaction, except the root, fulfills the A, C, and I among the ACID properties. The root (level 1) of the tree structure is the only transaction to satisfy all of the ACID properties. This version of nested transactions is called closed because of this inability of subtransactions to durably commit independently of the outcome of the root transaction

([WS92]). This section deals with closed nested transactions (CNTs), open nested transactions will be the topic of the next section.

A root transaction t is a sequence $[a_1, \ldots, a_n]$ of primitive actions, where a_1 must be $Begin(t)$, and a_n must be either $Commit(t)$, or $Rollback(t)$; $a_i, i = 2, \cdots, n-1$, may be any of the primitive actions, except $Begin(t)$, $Commit(t)$, and $Rollback(t)$, but including $Spawn(t, t')$, $Rollback(t')$, and $Commit(t')$, with $t \neq t'$. A child transaction t is a sequence $[a_1, \ldots, a_n]$ of primitive actions, where a_1 must be $Spawn(t', t)$, and a_n must be either $Commit(t)$, or $Rollback(t)$; $a_i, i = 2, \cdots, n-1$, may be any of the primitive actions, except $Spawn(t, t')$, $Commit(t)$, and $Rollback(t)$, but including $Spawn(t^*, t^{**})$, $Rollback(t^{**})$, and $Commit(t^{**})$, with $t \neq t^{**}$. We capture the typical relationships that hold between transactions in the hierarchy of a nested transaction with the system fluents $transOf(t, a, s)$, $parent(t, t', s)$ and $ancestor(t, t', s)$, which are introduced in the following successor state axiom and abbreviation, respectively:

$$transOf(t, a, s) \equiv \bigvee_{A \in \mathcal{A}} (\exists \vec{x}) a = A(\vec{x}, t), \tag{4.35}$$

$$\begin{aligned} parent(t, t', do(a, s)) \equiv\ & a = Spawn(t, t') \vee \\ & parent(t, t', s) \wedge \neg termAct(a, t) \wedge \neg termAct(a, t'), \end{aligned} \tag{4.36}$$

$$\begin{aligned} ancestor(t, t', s) =_{df}\ & (\forall B)[(\forall t) B(t, t, s) \wedge \\ & (\forall s, t, t', t'')[B(t, t'', s) \wedge parent(t'', t', s) \supset B(t, t', s)] \supset B(t, t', s)]. \end{aligned} \tag{4.37}$$

In (4.35), $\mathcal{A}$ denotes the set of actions of the domain.

Responsibility over actions that are executed and conflicts between transactions are specified with the following axioms:

$$\begin{aligned} responsible(t, a', do(a, s)) \equiv\ & transOf(t, a', s) \wedge \neg(\exists t^*) parent(t, t^*, s) \vee \\ & (\exists t^*)[parent(t, t^*, s) \wedge a = Commit(t^*) \wedge responsible(t^*, a')] \vee \\ & responsible(t, a', s) \wedge \neg termAct(a, t), \end{aligned} \tag{4.38}$$

$$\begin{aligned} transConflictNT(t, t', do(a, s)) \equiv\ & t \neq t' \wedge responsible(t', a, s) \wedge \\ & (\exists a', s')[responsible(t, a', s) \wedge updConflict(a', a, s) \wedge do(a', s') \sqsubset s] \wedge \\ & \neg responsible(t, a, s) \wedge running(t', s) \wedge ((\exists t'') parent(t', t'', s) \supset \\ & \qquad \neg ancestor(t, t', s)) \vee \\ & transConflictNT(t, t', s) \wedge \neg termAct(a, t). \end{aligned} \tag{4.39}$$

Intuitively, (4.39) means that transaction t conflicts with transaction t' in the log s iff t and t' are not equal, internal actions they are responsible for are conflicting in s, t is not responsible for the action of t' it is conflicting with, t' is running; moreover, a transaction cannot conflict with actions his ancestors are responsible for. Due to the presence of the new external action $Spawn$, we need to redefine $running(t,s)$ as follows:

Abbreviation 4.14

$$\begin{aligned} running(t,s) =_{df} &(\exists s').\{do(Begin(t),s') \sqsubseteq s \wedge \\ &(\forall a,s'')[do(Begin(t),s') \sqsubset do(a,s'') \sqsubset s \supset a \neq Rollback(t) \wedge a \neq End(t)] \vee \\ &(\exists t').do(Spawn(t',t),s') \sqsubseteq s \wedge \\ &(\forall a,s'')[do(Spawn(t',t),s') \sqsubset do(a,s'') \sqsubset s \supset a \neq Rollback(t) \wedge a \neq End(t)]\}. \end{aligned}$$

Now the external actions of closed nested transactions have the following precondition axioms:

$$\begin{aligned} Poss(Begin(t),s) \equiv &\neg(\exists t')parent(t',t,s) \wedge \\ &[s = S_0 \vee (\exists s',t').t \neq t' \wedge do(Begin(t'),s') \sqsubset s], \end{aligned} \tag{4.40}$$

$$\begin{aligned} Poss(Spawn(t,t'),s) \equiv &\, t \neq t' \wedge \\ &(\exists s',t'')[do(Begin(t),s') \sqsubset s \vee do(Spawn(t'',t),s') \sqsubset s], \end{aligned} \tag{4.41}$$

$$Poss(End(t),s) \equiv running(t,s), \tag{4.42}$$

$$\begin{aligned} Poss(Commit(t),s) \equiv &(\exists s').s = do(End(t),s') \wedge \bigwedge_{IC \in \mathcal{IC}_v} IC(s) \wedge \\ &(\forall t')[sc\_dep(t,t',s) \supset (\exists s'')do(Commit(t'),s'') \sqsubseteq s] \wedge \\ &(\forall t')[c\_dep(t,t',s) \wedge \neg(\exists s^*)do(Rollback(t'),s^*) \sqsubseteq s \supset (\exists s')do(Commit(t'),s') \sqsubset s)], \end{aligned} \tag{4.43}$$

$$\begin{aligned} Poss(Rollback(t),s) \equiv &(\exists s')[s = do(End(t),s') \wedge \bigwedge_{IC \in \mathcal{IC}_v} IC(s)] \vee \\ &(\exists t',s')[r\_dep(t,t',s) \wedge do(Rollback(t'),s') \sqsubset s] \vee \\ &(\exists t',s')[wr\_dep(t,t',s) \wedge do(Rollback(t'),s') \sqsubset s \wedge \\ &\quad \neg(\exists s^*)do(Commit(t),s^*) \sqsubset do(Rollback(t'),s')]. \end{aligned} \tag{4.44}$$

Dependency axioms characterizing the system fluents $r\_dep(t,t',s)$, $c\_dep(t,t',s)$, $sc\_dep(t,t',s)$, and $wr\_dep(t,t',s)$ are:

$$r\_dep(t,t',s) \equiv transConflictNT(t,t',s), \tag{4.45}$$

$$sc\_dep(t,t',s) \equiv readsFrom(t,t',s), \tag{4.46}$$

$$\begin{aligned} c\_dep(t,t',do(a,s)) \equiv &\, a = Spawn(t,t') \vee \\ &c\_dep(t,t',s) \wedge \neg termAct(a,t) \wedge \neg termAct(a,t'), \end{aligned} \tag{4.47}$$

$$\begin{aligned} wr\_dep(t,t',do(a,s)) \equiv &\, a = Spawn(t',t) \vee \\ &wr\_dep(t,t',s) \wedge \neg termAct(a,t) \wedge \neg termAct(a,t'). \end{aligned} \tag{4.48}$$

As an example of what they mean, the last axiom says that a transaction spawning another transaction generates a weak rollback dependency of the later one on the first one, and this dependency ends when either transactions execute a terminating action.

The successor state axioms for nested transactions are of the form:

$$\begin{aligned}&F(\vec{x},t,do(a,s)) \equiv (\exists \vec{t_1})\Phi_F(\vec{x},a,\vec{t_1},s) \wedge \neg(\exists t'')a = Rollback(t'') \vee \\ &\quad [(\exists t'').a = Rollback(t'') \wedge \neg(\exists t^*)parent(t^*,t'',s) \wedge restoreBeginPoint(F,\vec{x},t'',s)] \vee \\ &\quad [(\exists t'').a = Rollback(t'') \wedge (\exists t^*)parent(t^*,t'',s) \wedge restoreSpawnPoint(F,\vec{x},t'',s)],\end{aligned} \tag{4.49}$$

one for each database fluent of the relational language. Here $\Phi_F(\vec{x},a,\vec{t},s)$ is a formula with free variables among $\vec{x}, a, \vec{t}$, and s; $restoreBeginPoint(F,\vec{x},t,s)$ is as in Abbreviation 4.8, and we replace $Begin(t)$ by $Spawn(t',t)$ in Abbreviation 4.8 to define $restoreSpawnPoint(F,\vec{x},t,s)$:

Abbreviation 4.15

$$\begin{aligned}&restoreSpawnPoint(F,\vec{x},t,s) =_{df} \\ &\quad \{(\exists a_1,a_2,s',s_1,s_2,t',t^*). \\ &\qquad do(Spawn(t^*,t),s') \sqsubset do(a_2,s_2) \sqsubset do(a_1,s_1) \sqsubseteq s \wedge writes(a_1,F,\vec{x},t) \wedge writes(a_2,F,\vec{x},t') \wedge \\ &\qquad [(\forall a'',s'').do(a_2,s_2) \sqsubset do(a'',s'') \sqsubset do(a_1,s_1) \supset \neg writes(a'',F,\vec{x},t)] \wedge \\ &\qquad [(\forall a'',s'').do(a_1,s_1) \sqsubseteq do(a'',s'') \sqsubset s \supset \neg(\exists t'')writes(a'',F,\vec{x},t'')] \wedge (\exists t'')F(\vec{x},t'',s_1)]\} \vee \\ &\quad \{(\forall a^*,s^*,s',t^*).do(Spawn(t^*,t),s') \sqsubset do(a^*,s^*) \sqsubseteq s \supset \neg writes(a^*,F,\vec{x},t)] \wedge (\exists t')F(\vec{x},t',s)\}.\end{aligned}$$

A basic relational theory for nested transactions is defined as in Section 4.2, but where the relational language includes $Spawn(t,t')$ as a further action, and the axioms (4.40) – (4.41) replace axioms (4.11) – (4.14), the axioms (4.45) – (4.48) replace the axioms (4.16) – (4.17), and the set $\mathcal{D}_{ss}$ is a set of successor state axioms of the form (4.49). All the other axioms of Section 4.2 remain unchanged.

Now we state some of the properties of nested transactions as an illustration of how such properties are formulated in the situation calculus. Similarly to Theorem 4.10, we can show that a basic relational theory for nested transactions logically implies the commit and weak rollback dependency properties.

Theorem 4.19 *(***Atomicity of Nested Transactions***) Suppose $\mathcal{D}$ is a relational theory for nested transactions. Then for every database fluent F*

$$\begin{aligned}&\mathcal{D} \models legal(s) \supset \\ &\quad (\forall t,t',s_1,s_2)\{[[s' = do(Begin(t),s_1) \vee s' = do(Spawn(t,t'),s_1)] \wedge \\ &\qquad s' \sqsubset do(a,s_2) \sqsubset s \wedge \\ &\qquad (\exists a^*,s^*)[s' \sqsubset do(a^*,s^*) \sqsubset do(a,s_2) \wedge writes(a^*,F,t)]] \supset \\ &\qquad\qquad [(a = Rollback(t) \supset ((\exists t_1)F(\vec{x},t_1,do(a,s_2)) \equiv (\exists t_2)F(\vec{x},t_2,s_1))) \wedge \\ &\qquad\qquad (a = Commit(t) \supset ((\exists t_1)F(\vec{x},t_1,do(a,s_2)) \equiv (\exists t_2)F(\vec{x},t_2,s_2)))]\}.\end{aligned}$$

Theorem 4.20 (No-Orphan-Commit: [Chr91]) *Suppose $\mathcal{D}$ is a relational theory. Then, whenever a child's parent terminates before the parent does, the child is rolled back;i.e.,*

$$\begin{aligned}\mathcal{D} \models\ & legal(s) \supset \\ & \quad \{parent(t,t',s) \wedge termAct(a,t) \wedge \\ & \qquad do(Commit(t'),s') \not\sqsubseteq do(a,s'') \sqsubset s \supset (\exists s^*)do(Rollback(t'),s^*) \sqsubset s\}.\end{aligned}$$

This property, combined with the atomicity of all subtransactions of the nested transaction tree (i.e. Theorem 4.19), leads to the fact that, should a root transaction roll back, then so must all its subtransactions, also the committed ones. This is where the D in the ACID acronym is relaxed for subtransactions.

4.3.2 Cooperative Transaction Hierarchy

The *cooperative transaction hierarchy* (CTH: [NRZ92]) model has been proposed for supporting cooperative applications in the context of CAD.[8] A cooperative nesting of transactions is a nesting, where sibling subtransactions are allowed to interact. A cooperative transaction hierarchy is structured as a rooted tree whose leaf nodes, the *cooperative transactions* (CTs), represent the transactions at the level of individual designers,and whose internal nodes, the *transaction groups* (TGs), are each a set of children (CTs or TGs) that cooperate to perform a single task. There is no central correctness criterion in a CTH; instead, each TG has its own, user-defined correctness criteria. A TG is not atomic; it performs specific tasks via its members, enforces its own correctness criterion, and organizes cooperation among its members; moreover, it keeps private copies of the objects that its members acquire at creation time, and is a unit of recovery by managing its own recoverability. A TG correctness is expressed in terms of patterns and conflicts. Patterns specify interleavings of actions that must occur, and conflicts specify those interleavings that must not occur. A TG's log is correct iff it satisfies all its pattern specifications and satisfies none of its conflict specifications. A child passes its copy to the parent upon committing, at which time that copy subsumes the parent's copy. Ultimately, the copy of the root TG will be subsumed when the entire design commits. We now give a situation calculus characterization of CTHs.

We have two new external actions: $Join(t,t',n)$, and $Leave(t,t')$, where t and t' are transactions, and n indicates whether the joining node is CT or TG. A root transaction t is a sequence $[a_1,\ldots,a_n]$ of primitive actions, where a_1 must be $Begin(t)$, and a_n must be either $Commit(t)$, or $Rollback(t)$; $a_i, i=2,\cdots,n-1$, may be any of the primitive actions, except $Begin(t)$, $Commit(t)$, and $Rollback(t)$, but including $Join(t',t,n)$, $Rollback(t')$, and $Commit(t')$, with $t \neq t'$. A CT or a TG t is a sequence $[a_1,\ldots,a_n]$ of primitive actions, where a_1 must be $Join(t,t')$, and a_n must be either $Commit(t)$, or $Rollback(t)$; $a_i, i=2,\cdots,n-1$, may be any of the primitive actions, except $Join(t,t')$, $Commit(t)$, and $Rollback(t)$, but including $Join(t^*,t^{**})$, $Rollback(t^*)$, and $Commit(t^*)$, with $t \neq t'$, $t^* \neq t^{**}$,

[8]Note that we shall give no theorems stating properties of this model. Neither won't we prove properties of the open nested transactions. These properties are formulated and proven in a similar way as those of flat and closed nested transactions.

and $t^* \neq t$.

The external actions of roots, and CTs and TGs, are enumerated as follows, respectively.

Abbreviation 4.16

$$externalActR(a,t) =_{df} a = Begin(t) \vee a = End(t) \vee a = Commit(t) \vee a = Rollback(t).$$

Abbreviation 4.17

$$\begin{gathered} externalActC(a,t) =_{df} (\exists t', n)a = Join(t,t',n) \vee (\exists t')a = Leave(t,t') \vee a = End(t) \vee \\ a = Commit(t) \vee a = Rollback(t). \end{gathered}$$

We continue to capture the typical relationships that hold between transactions in the CTH model with the same fluents $parent(t,t',s)$ and $ancestor(t,t',s)$ as in nested transactions, but now with a slightly different successor state axiom for $parent(t,t',s)$.

$$parent(t,t',do(a,s)) \equiv (\exists n)a = Join(t',t,n) \vee parent(t,t',s) \wedge \neg a \neq Leave(t',t). \tag{4.50}$$

Furthermore, we need the fluents $transGroup(t,s)$ and $coopTrans(t,s)$ which intuitively tell whether a transaction is a TG or a CT; these have the following successor state axioms:

$$\begin{aligned} transGroup(t,do(a,s)) \equiv\ & (\exists t')a = Join(t,t',TG) \vee \\ & transGroup(t,s) \wedge \neg(\exists t')a = Leave(t,t'), \end{aligned} \tag{4.51}$$

$$\begin{aligned} coopTrans(t,do(a,s)) \equiv\ & (\exists t')a = Join(t,t',CT) \vee \\ & coopTrans(t,s) \wedge \neg(\exists t')a = Leave(t,t'). \end{aligned} \tag{4.52}$$

A user-defined predicate $transConflictCTH(t,t',s)$ must be provided, where t and t' are transactions; intuitively, $transConflictCTH(t,t',s)$ means that transactions t and t' conflict in the log s. As an example of such a definition of this predicate, the following one captures the cooperative serializability property of [MP90]:

Abbreviation 4.18

$$\begin{aligned} & transConflictCTH(t,t',s) \equiv (\exists t'').t \neq t' \wedge transGroup(t'',s) \wedge \\ & \quad \{\neg parent(t'',t) \wedge \neg parent(t'',t') \wedge transConflictNT(t,t',s) \vee \\ & \quad (\exists t^*)[\neg parent(t'',t) \wedge parent(t'',t') \wedge parent(t'',t^*) \wedge transConflictNT(t,t^*,s)] \vee \\ & \quad (\exists t^*)[parent(t'',t) \wedge \neg parent(t'',t') \wedge parent(t'',t^*) \wedge transConflictNT(t^*,t',s)]\}. \end{aligned} \tag{4.53}$$

Intuitively, (4.18) means that transaction t conflicts with transaction t' in the log s iff t and t' are not equal and there is a transaction group t^* such that: (1) either t and t' do not have t'' as parent, in which case they conflict in the usual way of closed nested transactions; or (2) t' has t'' as parent and t does not, but t' has a sibling t^* with which t is conflicting in the usual way of nested transactions; or else (3) t has t'' as parent and t' does not, but t has a sibling t^* which is conflicting with t' in the usual way of nested transactions.

The pattern specifications that must be verified and the conflict specifications that must be avoided are captured in action precondition axioms. Suppose $\mathcal{P}(t,s)$ and $\mathcal{C}(t,s)$ denote the pattern and conflict specifications for a TG t, respectively. Then precondition axioms for internal actions are of the form

$$\begin{aligned}&Poss(A(\vec{x},t),s) \equiv \Pi_A(\vec{x},t,s) \wedge IC_e(do(A(\vec{x},t),s)) \wedge \\ &\quad \{[(\exists t^*).parent(t^*,t,s) \wedge \mathcal{P}(t^*,do(A(\vec{x},t),s)) \wedge \neg\mathcal{C}(t^*,do(A(\vec{x},t),s)) \wedge running(t,s)] \vee \\ &\quad [\neg(\exists t^*).parent(t^*,t,s) \wedge \mathcal{P}(t,do(A(\vec{x},t),s)) \wedge \neg\mathcal{C}(t,do(A(\vec{x},t),s)) \wedge running(t,s)]\}. \end{aligned} \tag{4.54}$$

In CTHs, the following dependencies must be maintained among transactions: a rollback dependency of a child on its parent, and a weak commit dependency of a parent on all its children. A **Weak Commit Dependency** of t on t' is characterized as follows:

$$\begin{aligned}&do(Commit(t),s) \sqsubset s^* \supset \\ &\qquad\qquad [do(Rollback(t'),s') \not\sqsubseteq s^* \supset do(Commit(t'),s') \sqsubset do(Commit(t),s)];\end{aligned}$$

i.e., If t commits in a log s^*, then, whenever t' does not roll back in s^*, t' commits before t.

Now the external actions of a CTH have the following precondition axioms:

$$\begin{aligned}&Poss(Begin(t),s) \equiv \neg(\exists t')parent(t',t,s) \wedge \\ &\quad [s = S_0 \vee (\exists s',t').t \neq t' \wedge do(Begin(t'),s') \sqsubset s],\end{aligned} \tag{4.55}$$

$$\begin{aligned}&Poss(End(t),s) \equiv (\exists s')\{do(Begin(t),s') \sqsubseteq s \wedge \neg(\exists s'')do(End(t),s'') \sqsubseteq s \wedge \\ &\quad (\forall a,s^*)[do(Begin(t),s') \sqsubset do(a,s^*) \sqsubset s \supset \neg externalActR(a,t)]\} \vee \\ &\quad (\exists s',t',n)\{do(Join(t',t,n),s') \sqsubseteq s \wedge \neg(\exists s'')do(End(t),s'') \sqsubseteq s' \wedge \\ &\quad (\forall a,s^*)[(do(Join(t',t,n),s') \sqsubset do(a,s^*) \sqsubset s) \supset \neg externalActC(a,t)]\},\end{aligned} \tag{4.56}$$

$$\begin{aligned}&Poss(Commit(t),s) \equiv (\exists s').s = do(End(t),s') \wedge \bigwedge_{IC \in \mathcal{IC}_v} IC(s) \wedge \\ &\quad (\forall t').wc\_dep(t,t',s) \wedge \neg(\exists s^*)do(Rollback(t'),s^*) \sqsubseteq s' \supset \\ &\qquad\qquad (\exists s'')do(Commit(t'),s'') \sqsubseteq s',\end{aligned} \tag{4.57}$$

$$\begin{aligned}&Poss(Rollback(t),s) \equiv (\exists s').s = do(End(t),s') \wedge \neg \bigwedge_{IC \in \mathcal{IC}_v} IC(s) \vee \\ &\qquad (\exists t',s'').r\_dep(t,t',s) \wedge do(Rollback(t'),s'') \sqsubset s',\end{aligned} \tag{4.58}$$

$$\begin{aligned}&Poss(Join(t,t',n),s) \equiv t \neq t' \wedge \\ &\quad (\exists s',t'',n')[do(Begin(t'),s') \sqsubset s \vee do(Join(t',t'',n'),s') \sqsubset s \wedge n' \neq CT].\end{aligned} \tag{4.59}$$

Dependency axioms characterizing the fluents r_dep and wc_dep are:

$$\begin{aligned} r\_dep(t,t',do(a,s)) \equiv {} & (\exists n).a = Join(t,t',n) \vee \\ & transConflictCTH(t,t',do(a,s)) \vee r\_dep(t,t',s) \wedge a \neq Leave(t,t'), \end{aligned} \tag{4.60}$$

$$wc\_dep(t,t',do(a,s)) \equiv a = Join(t',t) \vee wc\_dep(t,t',s) \wedge a \neq Leave(t,t'). \tag{4.61}$$

The successor state axioms for CTHs are of the form:

$$\begin{aligned} F(\vec{x},t,do(a,s)) \equiv {} & \Phi_F(\vec{x},a,t,s) \wedge \neg(\exists t'')a = Rollback(t'') \vee \\ & (\exists t')[a = Commit(t') \wedge parent(t,t',s) \wedge F(\vec{x},t',s)] \vee \\ & (\exists t',n)[a = Join(t,t',n) \wedge F(\vec{x},t',s)] \vee \\ & (\exists t'')a = Rollback(t'') \wedge \neg(\exists t')parent(t',t'',s) \wedge restoreBeginPoint(F,\vec{x},t'',s) \vee \\ & (\exists t'')a = Rollback(t'') \wedge (\exists t')parent(t',t'',s) \wedge restoreJoinPoint(F,\vec{x},t'',s), \end{aligned} \tag{4.62}$$

one for each relation of the relational language, where $\Phi_F(\vec{x},a,s)$ is a formula with free variables among $a,s,\vec{x}$; $restoreBeginPoint(F,\vec{x},t,s)$ is defined in Abbreviation 4.8, and $restoreJoinPoint(F,\vec{x},t,s)$ is the following:

Abbreviation 4.19

$$\begin{aligned} & restoreJoinPoint(F,\vec{x},t,s) =_{df} \\ & \quad \{(\exists a_1,a_2,s',s_1,s_2,t',t^*,n). \\ & \qquad do(Join(t,t^*,n),s') \sqsubset do(a_2,s_2) \sqsubset do(a_1,s_1) \sqsubseteq s \wedge writes(a_1,F,\vec{x},t) \wedge writes(a_2,F,\vec{x},t') \wedge \\ & \qquad [(\forall a'',s'').do(a_2,s_2) \sqsubset do(a'',s'') \sqsubset do(a_1,s_1) \supset \neg writes(a'',F,\vec{x},t)] \wedge \\ & \qquad [(\forall a'',s'').do(a_1,s_1) \sqsubseteq do(a'',s'') \sqsubset s \supset \neg(\exists t'')writes(a'',F,\vec{x},t'')] \wedge (\exists t'')F(\vec{x},t'',s_1)]\} \vee \\ & \quad \{(\forall a^*,s^*,s',t^*).do(Join(t,t^*,n),s') \sqsubset do(a^*,s^*) \sqsubseteq s \supset \neg writes(a^*,F,\vec{x},t)] \wedge (\exists t')F(\vec{x},t',s)\}. \end{aligned}$$

A basic relational theory for CTHs is as in Definition 4.4, but where the relational language includes $Join(t,t',n)$ as a further external action, and the axioms (4.55) – (4.59) replace axioms (4.11) – (4.14), the axioms (4.60) – (4.61) replace the axiom (4.15), and the successor state axioms in $\mathcal{D}_{ss}$ are of the form (4.62). All the other axioms of Definition 4.4 remain unchanged.

4.3.3 Open Nested Transactions

Open nested transactions ([WS92]) are a generalized version of nested transactions. An open nested transaction is a system of component transactions forming an unbalanced tree whose nodes represent specific tasks that are performed by executing their children. All the other components with exception of the root of the system are called subtransactions. The leaves of such a tree are constituted by primitive actions. Given a node t of an open nested transaction with n children $t_1, \cdots, t_n$, these are classified into the following types:

- **Open Subtransactions**. These allow for unilateral commit and rollback independently of the parent transaction t. In other words, they are a collection of top-level transactions that may act independently from each other; but, if t rolls back, they also must rollback. So they are allowed to make their updates visible to other subtransactions before committing.

- **Closed Subtransactions**. These are structured like in Moss nested transactions ([Mos85]) (See Section 4.2). They do not allow for unilateral commit independently of the parent transaction t. For this reason, they are not allowed to make their updates visible to any other subtransactions before committing, at which point they only make their results available to their parent t. Any time, if t rolls back, they also must rollback.

- **Compensatable Subtransactions**. These are associated with compensating subtransactions; that is, rather than simply vanishing when its parent t rolls back, a compensatable t_i that has already committed triggers a compensating transaction $comp_i$ whose semantics is described below. Since their actions can always be undone, they can in general be allowed to be open.

- **Compensating Subtransactions**. These undo any changes done by the committed compensatable subtransactions. So a compensating transaction $comp_i$ undoes any changes done by the corresponding compensatable subtransaction t_i. A compensating transaction can again be another open nested transaction. The order of compensating subtransactions must be compatible with the order of their corresponding compensatable transactions; that is, if t_i commits before t_j then whenever $comp_i$ begins, it does so only after $comp_j$ has committed.

- **Non-Compensatable Subtransactions**. These are subtransaction that cannot be compensated for whenever they have already committed. For this reason, they can in general not be allowed to be open.

We now give a situation calculus characterization of open nested transactions.

The external actions are: $Begin(t, t', m, c)$, $End(t)$, $Commit(t)$, and $Rollback(t)$. Among these, $Begin(t, t', m, c)$ is new; intuitively, it means that the (sub)transaction t begins as a component of the (sub)transaction t' in mode m and class c, where the mode argument can be one of $OPEN$, $CLOSED$ and INV, and the class argument can be one of $COMP$, $NONCOMP$ and INV. We introduce a predicate $compensates(t, comp_t)$, meaning that $comp_t$ compensates the actions of t.

A root transaction t is a sequence $[a_1, \ldots, a_n]$ of primitive actions, where the action a_1 must be $Begin(t, NIL, INV, INV)$, and a_n must be either $Commit(t)$, or $Rollback(t)$; $a_i, i = 2, \cdots, n-1$, may be any of the primitive actions, except $Begin(t, t', m, c)$, $Commit(t)$, and $Rollback(t)$. A subtransaction t is a sequence $[a_1, \ldots, a_n]$ of primitive actions, where a_1 must be $Begin(t, t', m, c)$, and

a_n must be either $Commit(t)$, or $Rollback(t)$; $a_i, i = 2, \cdots, n-1$, may be any of the primitive actions, except $Begin(t^*, t^{**}, m^*, c^*)$, $Commit(t^*)$, and $Rollback(t^*)$.

The external actions of (sub)transactions are enumerated as follows.

Abbreviation 4.20

$$externalAct(a,t) =_{df}$$
$$(\exists t', m, c)a = Begin(t, t', m, c) \vee a = End(t) \vee a = Commit(t) \vee a = Rollback(t).$$

We have to slightly reconsider the axiom for the fluent $parent(t, t', s)$, and the abbreviation $running(t, s)$:

$$\begin{aligned} &parent(t, t', do(a, s)) \equiv (\exists m, c)a = Begin(t, t', m, c) \vee \\ &\quad parent(t, t', s) \wedge a \neq Rollback(t) \wedge a \neq Rollback(t') \wedge a \neq Commit(t) \wedge a \neq Commit(t'); \end{aligned} \tag{4.63}$$

$$\begin{aligned} &running(t, s) =_{df} (\exists s', t', m, c)\{[s^* = do(Begin(t, t', m, c), s') \vee s^* = do(Begin(t', t, m, c), s')] \wedge \\ &\quad s^* \sqsubseteq s \wedge (\forall a, s'')[s^* \sqsubset do(a, s'') \sqsubset s \supset a \neq Rollback(t) \wedge a \neq End(t)]\}. \end{aligned} \tag{4.64}$$

We also add a predicate $compensatable(t)$ with the characterization

$$compensatable(t) =_{df} (\exists t')compensates(t', t).$$

Furthermore, we need the fluents $closed(t, s)$, and $comp(t, s)$, which intuitively tell whether a transaction is closed or compensatable, respectively. These fluents have the following successor state axioms:

$$\begin{aligned} &closed(t, do(a, s)) \equiv \\ &\quad (\exists t', m, c)a = Begin(t, t', m, c) \wedge t \neq NIL \wedge m = CLOSED \wedge c \neq INV \vee \\ &\quad closed(t, s) \wedge a \neq Commit(t) \wedge a \neq Rollback(t), \end{aligned} \tag{4.65}$$

$$\begin{aligned} &comp(t, do(a, s)) \equiv \\ &\quad (\exists t', m, c)a = Begin(t, t', m, c) \wedge t \neq NIL \wedge m \neq INV \wedge c = COMP \vee \\ &\quad closed(t, s) \wedge a \neq Commit(t) \wedge a \neq Rollback(t). \end{aligned} \tag{4.66}$$

The conflict predicate $transConflictNT(t, t', s)$ defined in (4.39) still captures the nature of conflict that may arise in the open nested transaction context.

In open nested transactions, the following dependencies must be maintained among transactions: a weak rollback dependency of a child on its parent, a commit dependency of a parent on all its children, a strong commit dependency of compensating transactions on their corresponding compensatable transactions, and dependency of the order of the beginnings of compensating transactions on the commitments of their corresponding compensatable transactions.

Now the external actions of an open nested transaction have the following precondition axioms:

$$\begin{aligned}
&Poss(Begin(t,t',m,c),s) \equiv \neg(m = OPEN \wedge c = NONCOMP) \wedge t \neq t' \wedge \\
&\quad \{s = S_0 \vee \\
&\quad (\exists s', t^*, t'', m^*, c^*)[t^* \neq t \wedge do(Begin(t^*, t'', m^*, c^*), s') \sqsubset s] \wedge \\
&\quad [(\forall t'').bc\_dep(t,t'',s) \supset (\exists s'')do(Commit(t''),s'') \sqsubseteq s]\},
\end{aligned} \tag{4.67}$$

$$Poss(End(t),s) \equiv running(t,s), \tag{4.68}$$

$$\begin{aligned}
&Poss(Commit(t),s) \equiv (\exists s').s = do(End(t),s') \wedge \bigwedge_{IC \in \mathcal{IC}_v} IC(s) \wedge \\
&\quad (\forall t', s')[c\_dep(t,t',s) \supset (do(Commit(t'),s') \sqsubseteq s^* \supset do(Commit(t'),s') \sqsubset s)] \wedge \\
&\quad (\forall t', s').sc\_dep(t,t',s) \supset (\exists s'')do(Commit(t'),s'') \sqsubseteq s,
\end{aligned} \tag{4.69}$$

$$\begin{aligned}
&Poss(Rollback(t),s) \equiv (\exists s')[s = do(End(t),s') \wedge \neg \bigwedge_{IC \in \mathcal{IC}_v} IC(s)] \vee \\
&\quad (\exists t', s'')[r\_dep(t,t',s) \wedge do(Rollback(t'),s'') \sqsubseteq s] \vee \\
&\quad (\exists t', s')[wr\_dep(t,t',s) \wedge do(Rollback(t'),s') \sqsubseteq s \wedge \\
&\qquad (\forall s'')(do(Commit(t),s'') \not\sqsubseteq do(Rollback(t'),s'))],
\end{aligned} \tag{4.70}$$

Now we give dependency axioms characterizing the fluents $r\_dep(t,t',s)$, $wr\_dep(t,t',s)$, $sc\_dep(t,t',s)$, $c\_dep(t,t',s)$, and $bc\_dep(t,t',s)$:

$$r\_dep(t,t',s) \equiv transConflictNT(t,t',s), \tag{4.71}$$

$$\begin{aligned}
&wr\_dep(t,t',do(a,s)) \equiv (\exists m,c)a = Begin(t,t',m,c) \wedge t' \neq NIL \wedge c = CLOSED \vee \\
&\quad wr\_dep(t,t',s) \wedge \neg termAct(a,t) \wedge \neg termAct(a,t'),
\end{aligned} \tag{4.72}$$

$$\begin{aligned}
&sc\_dep(t,t',do(a,s)) \equiv readsFrom(t,t') \vee \\
&\quad (\exists t'').parent(t'',t',s) \wedge compensates(t,t') \wedge a = Rollback(t'') \vee \\
&\quad sc\_dep(t,t',s) \wedge \neg termAct(a,t) \wedge \neg termAct(a,t'),
\end{aligned} \tag{4.73}$$

$$\begin{aligned}
&c\_dep(t,t',do(a,s)) \equiv (\exists m,c)a = Begin(t',t,m,c) \wedge m = CLOSED \vee \\
&\quad transConflictNT(t,t',do(a,s)) \vee \\
&\quad c\_dep(t,t',s) \wedge \neg termAct(a,t) \wedge \neg termAct(a,t'),
\end{aligned} \tag{4.74}$$

$$\begin{aligned}
&bc\_dep(t,t',do(a,s)) \equiv (\exists s', s'', t^*, t^{**}).compensates(t,t^*) \wedge \\
&\quad compensates(t',t^{**}) \wedge do(Commit(t^*),s') \sqsubseteq s \wedge a = Commit(t^{**}) \vee \\
&\quad bc\_dep(t,t',s) \wedge a \neq Commit(t) \wedge a \neq Commit(t').
\end{aligned} \tag{4.75}$$

The successor state axioms for open nested transactions are of the form:

$$\begin{aligned}F(\vec{x}, t, do(a, s)) \equiv{}& \Phi_F(\vec{x}, a, t, s) \wedge \neg(\exists t'')a = Rollback(t'') \vee \\ & (\exists t')[a = Commit(t') \wedge parent(t, t', s) \wedge closed(t, s) \wedge F(\vec{x}, t', s)] \vee \\ & (\exists t'')a = Rollback(t'') \wedge restoreBeginPoint(F, \vec{x}, t'', s).\end{aligned} \tag{4.76}$$

one for each fluent F of the relational language, where $\Phi_F(\vec{x}, a, t, s)$ is a formula with free variables among $a, t, s, \vec{x}$; we introduce $restoreBeginPoint(F, \vec{x}, t, s)$ as in Abbreviation 4.8 with $Begin(t)$ replaced by $Begin(t, t', m, c)$ and some straightforward quantifier changes.

A basic relational theory for open nested transactions is as in Definition 4.4, but where the relational language includes $Begin(t, t', m, c)$ as a further action, and the axioms (4.67) – (4.70) replace axioms (4.11) – (4.14), the axioms (4.71) – (4.75) replace the axiom (4.15), and the successor state axioms in $\mathcal{D}_{ss}$ are of the form (4.76). All the other axioms of Definition 4.4 remain unchanged.

It is important to note that open nested transactions have one more system action which is $Begin(t, t', m, c)$ in the special case where t' compensates some other transaction. Thus our definition for $systemAct(a, t)$ must capture this novelty:

Abbreviation 4.21

$$\begin{aligned}systemAct(a, t) =_{df}{}& a = Commit(t) \vee a = Rollback(t) \vee \\ & (\exists s, t', t'')bc\_dep(t, t', s) \wedge parent(t'', t', s) \supset a = Begin(t'', t, OPEN, INV).\end{aligned}$$

4.4 Example

We consider a Debit/Credit example which illustrates how to formulate a relational theory for closed nested transactions.

The database involves a relational language with:

Fluents: $served(aid, s), branches(bid, bbal, bname, t, s), tellers(tid, tbal, t, s),$ $accounts(aid, bid, abal, t, s)$.

Situation Independent Predicate: $requested(aid, req)$.

Action Functions: $b\_insert(bid, bbal, bname, t), b\_delete(bid, bbal, bname, t), t\_insert(tid, tbal, t),$ $t\_delete(tid, tbal, t), a\_insert(aid, bid, abal, tid, t), a\_delete(aid, bid, abal, tid, t), report(aid)$.

Constants: Ray, $Iluju$, $Misha$, Ho, etc.

The meaning of the arguments of fluents are self explanatory; and the relational language also includes the external actions of nested transactions. Among the fluents above, $served(aid, s)$ is a system fluent, and the remaining ones are database fluents.

To be brief, we skip unique name axioms and concentrate ourself on the remaining axioms of the basic relational theory. We enforce the following ICs ($\mathcal{IC}_e$):

$$accounts(aid, bid, abal, tid, t, s) \land accounts(aid, bid', abal', tid', t', s) \supset$$
$$bid = bid' \land abal = abal' \land tid = tid',$$
$$branches(bid, bbal, bname, t, s) \land branches(bid, bbal', bname', t', s) \supset$$
$$bbal = bbal' \land bname = bname',$$
$$tellers(tid, tbal, t, s) \land tellers(tid, tbal', t', s) \supset tbal = tbal';$$

and we have to verify the IC ($\mathcal{IC}_v$)

$$accounts(aid, bid, abal, tid, t, s) \supset abal \geq 0$$

at transaction's end. The following are the update precondition axioms:

$$Poss(a\_insert(aid, bid, abal, tid, t), s) \equiv \neg(\exists t')accounts(aid, bid, abal, tid, t', s) \land$$
$$IC_e(do(a\_insert(aid, bid, abal, tid, t), s)) \land running(t, s),$$
$$Poss(a\_delete(aid, bid, abal, tid, t), s) \equiv (\exists t')accounts(aid, bid, abal, tid, t', s) \land$$
$$IC_e(do(a\_delete(aid, bid, abal, tid, t), s)) \land running(t, s),$$
$$Poss(b\_insert(bid, bbal, bname, t), s) \equiv \neg(\exists t')branches(bid, bbal, bname, t', s) \land$$
$$IC_e(do(b\_insert(bid, bbal, bname, t), s)) \land running(t, s),$$
$$Poss(b\_delete(bid, bbal, bname, t), s) \equiv (\exists t')branches(bid, bbal, bname, t', s) \land$$
$$IC_e(do(b\_delete(bid, bbal, bname, t), s)) \land running(t, s),$$
$$Poss(t\_insert(tid, tbal, t), s) \equiv \neg(\exists t')tellers(tid, tbal, t', s) \land$$
$$IC_e(do(t\_insert(tid, tbal, t), s)) \land running(t, s),$$
$$Poss(t\_delete(tid, tbal, t), s) \equiv (\exists t')tellers(tid, tbal, t', s) \land$$
$$IC_e(do(t\_delete(tid, tbal, t), s)) \land running(t, s).$$

The successor state axioms ($\mathcal{D}_{ss}$) are:

$$accounts(aid, bid, abal, tid, t, do(a, s)) \equiv$$
$$((\exists t_1)a = a\_insert(aid, bid, abal, tid, t_1) \lor (\exists t_2)accounts(aid, bid, abal, tid, t_2, s) \land$$
$$\neg(\exists t_3)a = a\_delete(aid, bid, abal, tid, t_3)) \land \neg(\exists t')a = Rollback(t') \lor$$
$$(\exists t').a = Rollback(t') \land \neg(\exists t'')parent(t'', t', s) \land$$
$$restoreBeginPoint(accounts, (aid, bid, abal, tid), t', s) \lor$$
$$a = Rollback(t') \land (\exists t'')parent(t'', t', s) \land$$
$$restoreSpawnPoint(accounts(aid, bid, abal, tid), t', s),$$

$$
\begin{aligned}
&branches(bid, bbal, bname, t, do(a,s)) \equiv \\
&\quad ((\exists t_1)a = b\_insert(bid, bbal, bname, t_1) \lor (\exists t_2)branches(bid, bbal, bname, t_2, s) \land \\
&\qquad\qquad \neg(\exists t_3)a = b\_delete(bid, bbal, bname, t_3)) \land \neg(\exists t')a = Rollback(t') \lor \\
&\quad (\exists t').a = Rollback(t') \land \neg(\exists t'')parent(t'', t', s) \land \\
&\qquad\qquad restoreBeginPoint(branches, (bid, bbal, bname), t', s) \lor \\
&\quad a = Rollback(t') \land (\exists t'')parent(t'', t', s) \land restoreSpawnPoint(branches, (bid, bbal, bname), t', s),
\end{aligned}
$$

$$
\begin{aligned}
&tellers(tid, tbal, do(a,s)) \equiv \\
&\quad ((\exists t_1)a = t\_insert(tid, tbal, t_1) \lor (\exists t_2)tellers(tid, tbal, t_2, s) \land \\
&\qquad\qquad \neg(\exists t_3)a = t\_delete(tid, tbal, t_3)) \land \neg(\exists t')a = Rollback(t') \lor \\
&\quad (\exists t').a = Rollback(t') \land \neg(\exists t'')parent(t'', t', s) \land restoreBeginPoint(tellers, (tid, tbal), t', s) \lor \\
&\quad a = Rollback(t') \land (\exists t'')parent(t'', t', s) \land restoreSpawnPoint(tellers, (tid, tbal), t', s).
\end{aligned}
$$

4.5 Summary

We have given a framework for specifying relational database transactions as non-Markovian theories formulated in a finite fragment of the situation calculus. Main results in this chapter are:

- semantics of various versions of the classical flat transactions using relational theories;
- semantics of closed and open nested transactions using basic relational theories;
- proof of properties of ATMs by establishing logical consequences of sentences of the situation calculus capturing these properties;

One must distinguish between our approach which is a purely logical, abstract specification in which all system properties are formulated relative to the database log, and an implementation which normally materializes the database using progression ([Rei01]). This is the distinguishing feature of our approach. The log is a first class citizen of the logic, and the semantics of external actions are defined with respect to it.

A further point that we must acknowledge is that our axiomatization requires a lot of formulas, many of which seem to be low level. However, we believe that this complexity is inherent to the ATMs. We provide one with a framework whose expressiveness allows to exactly state the properties of ATMs in a way that accurately describes their inherent complexity.

Thus far, we have given axioms that accommodate a complete initial database state. This, however, is not a requirement of the theory we are presenting. Therefore our account could, for example, accommodate initial databases with null values, open initial database states, initial databases accounting for object orientation, or initial semistructured databases. These are just a examples of some of the generalizations that our initial databases could admit.

Finally, it is important to notice that the only place where the second order nature of our framework is needed is in the proof of the properties of the transaction models that rely on the second order induction principle (4.2).

The framework described in this chapter will be extended in the next chapter where we will consider modeling active rules and different active rule processing mechanisms. A substantial part of the building blocks for doing so has been constructed in this chapter.

Chapter 5

Specifying Knowledge Models

In the previous chapter, we have specified relational database transactions models as basic relational theories, which are non-Markovian theories formulated in a finite fragment of the situation calculus. The present chapter is devoted to extending the basic relational theories to model the representational component of active behaviors.[1] The new theories we will introduce in this chapter are called active relational theories. As active databases are intimately related to transactions, a substantial building block of these new theories is made of basic relational theories. As we shall see, an active relational theory precisely encompasses a basic relational theory capturing a specific ATM and axioms for typical active database fluents that are induced by the original database fluents of the domain.

Events are traditionally described using an event algebra. Virtually every proposed ADBMS brings about a different event algebra. This makes it very difficult to analyze these proposals in a uniform way by spelling out what they may have in common, or how they may differ. Typically, logic might act as a framework for dealing with these issues. This chapter treats events as (somewhat constrained) formulas of the situation calculus. We provide a framework for devising the semantics of complex events in the situation calculus. Such semantics, formulated as a class of axioms of active relational theories, are used for reasoning about the occurrence and consumption modes of events.

The chapter is organized as follows. Section 5.1 introduces the syntax of ECA rules, and Section 5.2 specifies the notions of transition tables and net effect policy. In Section 5.3, a metamodel for abstracting from event algebras is presented. Section 5.4 formally defines the active relational theories. Finally, Section 5.5 summarizes the chapter.

[1]Those readers from Knowledge Representation might get confused by the qualification “Knowledge” in the title of trhis chapter. It seems that a title like “Specifying Events” or ”Specifying Active Relational Theories” would be a more appropriate title of the chapter. However, due to [Pat99], “Knowledge model” is now prevalent in active databases to denote all aspects of active behavior related to events.

5.1 ECA-Rules

An *ECA rule* is a construct of the following form:

$$<t : R : \tau : \zeta(\vec{x}) \rightarrow \alpha(\vec{y})> . \tag{5.1}$$

In this construct, t specifies the transaction that fires the rule, τ specifies events that trigger the rule, and R is a constant giving the rule's identification number (or name). A rule is triggered if the event specified in its event part occurrs. The relationship between the event occurrence and the triggering of a rule is dictated by *consumption modes*. In its simplest form, the semantics of event consumption is that a rule is triggered if the event specified in its event part occurred since the beginning of the open transaction in which that event part is evaluated. Events are one of the predicates $F\_inserted(r, t, s)$ and $F\_deleted(r, t, s)$, called *event fluents,* or a combination thereof using logical connectives. The ζ part specifies the rule's condition; it mentions predicates $F\_inserted(r, \vec{x}, t, s)$ and $F\_deleted(r, \vec{x}, t, s)$ called *transition fluents,* which denote the transition tables ([WC96]) corresponding to insertions into and deletions from the relation F. In (5.1), arguments t, R, and s are suppressed from all the fluents; the two first ones are restored when (5.1) is translated to a ConGolog program, and s is restored at run time. Finally, α gives a ConGolog program which will be executed upon the triggering of the rule once the specified condition holds. Actions also may mention transition fluents. Notice that $\vec{x}$ are free variables mentioned by ζ and contain all the free variables $\vec{y}$ mentioned by α. Details of ConGolog programs will be introduced in the next chapter.

Example 5.1 *Consider a stock trading database (adapted from [WC96]), whose schema contains the relations:* $price(stock\_id, price, time, trans, s)$, $stock(stock\_id, price, closingprice, trans, s)$, *and* $customer(cust\_id, balance, stock\_id, trans, s)$, *which are relational fluents. The explanation of the attributes is as follows:* $stock\_id$ *is the identification number of a stock,* $price$ *the current price of a stock,* $time$ *the pricing time,* $closingprice$ *the closing price of the previous day,* $cust\_id$ *the identification number of a customer,* $balance$ *the balance of a customer, and* $trans$ *is a transaction identifier.*

Now consider the following active behavior. For each customer, his stock is updated whenever new prices are notified. When current prices are being updated, the closing price is also updated if the current notification is the last of the day; moreover, suitable trade actions are initiated if some conditions become true of the stock prices, under the constraint that balances cannot drop below a certain amount of money. Two rules for our example are shown in Figure 5.1. ■

5.2 Transition Fluents and Net Effect Policy

To characterize the notion of transition tables and events, we introduce the fluent $considered(r, t, s)$ which intuitively means that the rule r can be considered for execution in situation s with respect to the

$$
\begin{aligned}
&<trans : Update\_stocks : price\_inserted : \\
&\quad (\exists c, time, bal, price')[price\_inserted(s\_id, price, time) \land \\
&\qquad customer(c, bal, s\_id) \land stock(s\_id, price', clos\_pr)] \\
&\quad \rightarrow \\
&\quad stock\_insert(s\_id, price, clos\_pr) >
\end{aligned}
$$

$$
\begin{aligned}
&<trans : Buy\_100shares : price\_inserted : \\
&\quad (\exists\ new\_price, time, bal, pr, clos\_pr)[price\_inserted(s\_id, new\_price, time) \land \\
&\qquad customer(c, bal, s\_id) \land stock(s\_id, pr, clos\_pr) \land new\_price < 50 \land clos\_pr > 70] \\
&\quad \rightarrow \\
&\quad buy(c, s\_id, 100) >
\end{aligned}
$$

Figure 5.1: Rules for updating stocks and buying shares

transaction t. The following gives an abbreviation for $considered(r, t, s)$:

$$considered(r, t, s) =_{df} (\exists t').running(t', s) \land ancestor(t', t, s). \tag{5.2}$$

Intuitively, this means that, as long as an ancestor of t is running, any rule r may be considered for execution. In actual systems this concept is more sophisticated than this scheme.[2]

For each database fluent $F(\vec{x}, t, s)$, we introduce the *transition fluents* $F\_inserted(r, \vec{x}, t, s)$ and $F\_deleted(r, \vec{x}, t, s)$. The following successor state axioms characterizes them: $F\_inserted(r, \vec{x}, t, s)$:

$$
\begin{aligned}
F\_inserted(r, \vec{x}, t, do(a, s)) \equiv considered(r, t, s) \land (\exists t')a = F\_insert(\vec{x}, t') \lor \\
F\_inserted(r, \vec{x}, t, s) \land \neg(\exists t'')a = F\_delete(\vec{x}, t'').
\end{aligned} \tag{5.3}
$$

$$
\begin{aligned}
F\_deleted(r, \vec{x}, t, do(a, s)) \equiv considered(r, t, s) \land (\exists t')a = F\_delete(\vec{x}, t') \lor \\
F\_deleted(r, \vec{x}, t, s) \land \neg(\exists t'')a = F\_insert(\vec{x}, t'').
\end{aligned} \tag{5.4}
$$

Axiom (5.3) means that a tuple $\vec{x}$ is considered inserted in situation $do(a, s)$ iff the internal action $F\_insert(\vec{x}, t')$ was executed in the situation s while the rule r was considered, or it was already inserted and a is not the internal action $F\_delete(\vec{x}, t')$; here, t' is transaction that can be different than t. This captures the notion of *net effects* ([WC96]) of a sequence of actions. Such net effects are accumulating only changes

[2]For example, in Starburst ([WC96]), r will be considered in the future course of actions only from the time point where it last stopped being considered.

that really affect the database; in this case, if a record is deleted after being inserted, this amounts to nothing having happened. Further net effect policies can be captured in this axiom.

5.3 Event Logics

5.3.1 Primitive and Complex Event Fluents

Events that trigger ECA rules are generally associated with the DML of the underlying database. In the situation calculus, for each database fluent $F(\vec{x}, t, s)$, we introduce the *primitive event fluents* $F\_inserted(r, t, s)$ and $F\_deleted(r, t, s)$.

The primitive event fluent $F\_inserted(r, t, s)$ corresponding to an insertion into the relation F has the following successor state axiom:

$$\begin{aligned} &F\_inserted(r,t, do(a, s)) \equiv \\ &\qquad (\exists \vec{x}, t')a = F\_insert(\vec{x}, t') \wedge considered(r, t, s) \vee F\_inserted(r, t, s). \end{aligned} \tag{5.5}$$

The primitive event fluent $F\_deleted(r, t, s)$ corresponding to a deletion from the relation F has a similar successor state axiom:

$$\begin{aligned} &F\_deleted(r,t, do(a, s)) \equiv \\ &\qquad (\exists \vec{x}, t')a = F\_delete(\vec{x}, t') \wedge considered(r, t, s) \vee F\_deleted(r, t, s). \end{aligned} \tag{5.6}$$

Definition 5.1 (Primitive Event Occurrence) *A primitive event e occurs in situation s with respect to a rule r and a transaction t iff $\mathcal{D} \models e[r, t, s]$. Here $\mathcal{D}$ is a relational theory incorporating the successsor state axioms for the primitive event fluents.*

So, on this definition, an event occurrence (or, equivalently, event detection) in a situation calculus query in the sense of Section 3.2.3. Following [BM01], we call this an *event query*.

In many ADBMSs, complex events are built from simpler, and ultimately, the primitive ones using some *event algebra* ([BM01], [ZU01]). Using logical means, we now specify the semantics of complex events that accounts for the active dimension of consumption mode. This development will ultimately lead to a logic for events, instead of an algebra.

That complex events are built from simpler ones is just one of intuitive assumptions one can make about events. In [ZU01], Zimmer and Unland make five basic assumptions about events, which we adopt in the context of the situation calculus as follows:

- Events are interpreted over a set of situations (logs).
- Primitive events are detected at situations, in the order at which they occured.

Fluent	Informal semantics
$seq\_ev(r, t, e_1, e_2, s)$	event e_1 occurs before event e_2 in s
$simult\_ev(r, t, e_1, e_2, s)$	events e_1 and e_2 occur simultaneoulsy in s
$conj\_ev(r, t, e_1, e_2, s)$	events e_1 and e_2 occur together in any order in s
$disj\_ev(r, t, e_1, e_2, s)$	either event e_1 or event e_2 occurs in s
$neg\_ev(r, t, e, s)$	event e does not occur in s

Table 5.1: Informal semantics of basic complex events

- Complex events are built from primitive ones (components) using logical connectives, and many complex events can independently be built from the same set of simpler ones.
- The situation at which a complex action is considered to have occured is the situation at which the very last of its components occurs; here, "last" means the ordering of situations mentionned above.
- Many events may occur at the same situation, that is, simultaneously.

In order to build complex events, the usual logical connectives and symbols $\wedge$, $\vee$, $\neg$, $\forall$, as well as the ordering predicate $\sqsubset$. These logical symbols and predicates will be used to introduce complex events in the form of abbreviations. The following fluents are used to express some basic constructs for building complex events: $seq\_ev(r, t, e_1, e_2, s)$, $simult\_ev(r, t, e_1, e_2, s)$, $conj\_ev(r, t, e_1, e_2, s)$, $disj\_ev(r, t, e_1, e_2, s)$, and $neg\_ev(r, t, e, s)$. Table 5.1 gives the informal semantics of these fluents.

In what follows concerning execution semantics, it is appropriate to define what counts as a term or a formula whose rule and transaction arguments have been either suppressed or restored. In the same spirit as for Definitions 6.1 and 6.7, we can introduce the concepts of *rule id and transaction id suppressed* terms and formulas, and *rule id and transaction id restored* terms and formulas, respectively.

Definition 5.2 (Rid and Tid-Suppressed Terms and Formulas) *Suppose $\mathfrak{R}$ is a relational language. Then the rid and tid suppressed-terms (rts-terms) and formulas (rts-formulas) of $\mathfrak{R}$ are inductively given by a procedure similar to Definition 6.1 whose details should by now be obvious. So we omit these.*

Definition 5.3 (Rid and Tid-Restored Terms and Formulas) *Suppose $\mathfrak{R}$ is a relational language. Then the rid and tid restored-terms (rtr-terms) and formulas (rtr-formulas) of $\mathfrak{R}$ are inductively given by a procedure similar to Definition 6.7. Again, details of such a procedure should be clear by now and we omit* *them. Whenever t and ϕ are rts-term and rts-formula, respectively, and r and t are rule and transaction names, respectively, we use the notation $t[r,t]$ and $\phi[r,t]$ to denote the corresponding rtr-term and rtr-formula, respectively.*

With reference to the syntax of an ECA rule (see (5.1)), the notation $\tau(\vec{x})[r,t]$ means the result of restoring the arguments r and t to all event fluents mentioned by τ, $\zeta(\vec{x})[r,t]$ means the result of restoring the arguments r and t to all transition fluents mentioned by ζ, and $\alpha(\vec{x})[r,t]$ means the result of restoring the arguments r and t to all transition fluents and passing t to actions mentioned by α. For example, if τ is the complex event

$$price\_inserted \wedge customer\_inserted,$$

then $\tau[r,t]$ is

$$price\_inserted(r,t) \wedge customer\_inserted(r,t).$$

In the absence of consumption modes, the formal situation calculus based-semantics of complex events in terms of simpler ones is as follows:

$$neg\_ev(r,t,e,s) =_{df} (\exists r')\neg e[r',t,s], \tag{5.7}$$

$$seq\_ev(r,t,e_1,e_2,s) =_{df} (\exists r')e_2[r',t,s] \wedge (\exists r'',s').s' \sqsubset s \wedge e_1[r'',t,s'], \tag{5.8}$$

$$simult\_ev(r,t,e_1,e_2,s) =_{df} (\exists r')e_1[r',t,s] \wedge (\exists r'')e_2[r'',t,s], \tag{5.9}$$

$$conj\_ev(r,t,e_1,e_2,s) =_{df} (\exists r_1)seq\_ev(r_1,t,e_1,e_2,s) \vee \tag{5.10}$$

$$(\exists r_2)seq\_ev(r_2,t,e_2,e_1,s) \vee (\exists r_3)simult\_ev(r_3,t,e_1,e_2,s), \tag{5.11}$$

$$disj\_ev(r,t,e_1,e_2,s) =_{df} (\exists r')e_1[r',t,s] \vee (\exists r'')e_2[r'',t,s], . \tag{5.12}$$

Definition 5.4 (Complex Event Occurrence) *A complex event e occurs in situation s with respect to a rule r and a transaction t iff $\mathcal{D} \models e[r,t,s]$. Here $\mathcal{D}$ is a relational theory incorporating the abbreviations above for the complex event fluents.*

In spirit of [ZU01], the following good language design principles are emphasized with respect to complex events of any logic for events:

- **Minimality**: the logic must provide a very small minimal core of constructs that are such that different constructs express different semantics.
- **Symmetry**: The semantics of the constructs is context free.
- **Orthogonality**: The core language must allow every meaningful complex event to be expressible.

From the basic constructs (5.7–5.12) above, the set $\{seq\_ev(r,t,e_1,e_2,s), e_1, \cdots, e_n\}$ is the minimal core from which all the others complex events are built, where the e_i, $i = 1, \cdots, n$, are primitive event fluents. More precisely, all the other complex events can be defined using this set. Any other construct not belonging to that core must satisfy the good language design principles of symmetry and orthogonality listed above.

5.3.2 Event Fluents and Consumption Modes

Once we have specified a way of building a complex event e from simpler ones, we still have to specify which occurrences of the component of e must be selected in order for e to occur (*event occurrence selection*), and what to do with those occurrences once they have been used in the occurrence of e (*occurrence consumption*). *Consumption modes* are used to determine the event occurrence selection and consumption of the events.

Presumably, it suffices to assign consumption modes to the minimal core $\{seq\_ev(r, t, e_1, e_2, s), e_1, \cdots, e_n\}$ of the logic for events.

As for primitive event fluents, occurrence selection is trivial: from axioms (5.5) and (5.6) we see clearly that the first occurrence of a primitive event fluent may trigger any considered ECA rule. From axioms (5.5) and (5.6), we also see that a primitive event fluent remains unconsumed for any later considered rule. So this way we achieve a no-consumption scope (See Section 2.1.4). To achieve a local consumption scope, we change (5.5) and (5.6) respectively to

$$\begin{aligned} F\_inserted(r, t, do(a, s)) \equiv & (\exists \vec{x}, t').a = F\_insert(\vec{x}, t') \wedge considered(r, t, s) \vee \\ & F\_inserted(r, t, s) \wedge \neg(\exists \vec{y}, t'')(a = F\_insert(\vec{y}, t') \wedge t' = t''), \end{aligned} \tag{5.13}$$

and

$$\begin{aligned} F\_deleted(r, t, do(a, s)) \equiv & (\exists \vec{x}, t')a = F\_delete(\vec{x}, t') \wedge considered(r, t, s) \vee \\ & F\_deleted(r, t, s) \wedge \neg(\exists \vec{y}, t'')(a = F\_delete(\vec{y}, t') \wedge t' = t''). \end{aligned} \tag{5.14}$$

Finally, we achieve a global consumption scope by changing (5.5) and (5.6) respectively to

$$\begin{aligned} F\_inserted(r, t, do(a, s)) \equiv & (\exists \vec{x}, t')a = F\_insert(\vec{x}, t') \wedge considered(r, t, s) \vee \\ & F\_inserted(r, t, s) \wedge \neg(\exists r')(F\_inserted(r', t, s) \wedge r \neq r'), \end{aligned} \tag{5.15}$$

and

$$\begin{aligned} F\_deleted(r, t, do(a, s)) \equiv & (\exists \vec{x}, t')a = F\_delete(\vec{x}, t') \wedge considered(r, t, s) \vee \\ & F\_deleted(r, t, s) \wedge \neg(\exists r')(F\_deleted(r', t, s) \wedge r \neq r'). \end{aligned} \tag{5.16}$$

A particular consumption mode is imposed upon the sequence fluent $seq\_ev(r, t, e_1, e_2, s)$ by defining a conjunt $\Psi_{CM}(t, \vec{s})$ such that

$$seq\_ev(r, t, e_1, e_2, s') =_{df} (\exists \vec{s})\Psi_{seq}(t, \vec{s}, s') \wedge \Psi_{CM}(t, \vec{s}), \tag{5.17}$$

where $\Psi_{seq}(t, \vec{s}, s')$ is a situation calculus formula specifying the semantics of $seq\_ev(r, t, e_1, e_2, s)$ (i.e, the right-hand side of (5.8)); $\Psi_{CM}(t, \vec{s})$ is a situation calculus formula that specifies the consumption mode used.

If $\mathcal{L}$ is a distinguished fragment of the situation calculus such that $\Psi_{CM}(t, \vec{s}) \in \mathcal{L}$, then this induces the *consumption mode class* $CM_{\mathcal{L}}$. In general, $\mathcal{L}$ can be any fragment of the situation calculus. However, as we shall see in the sequel of this section, formulas $\Psi_{CM}(t, \vec{s})$ used in practice belong to logics $\mathcal{L}$ that enjoy particularly desirable properties (e.g., decidability) with respect to specific problems such as the equivalence of two given complex events ([BM01]).

To deal with consumption modes for sequences, we introduce further terminology adapted from [ZU01]. Suppose $e = seq\_ev(r, t, e_1, e_2, s)$; then e_1 is called the *initiator* and e_2 the *terminator* of e. A component e' of a sequence e is said to be *consummed* iff it no longer can contribute to the detection of e.

By virtue of the Zimmer-Unland assumptions about events, a sequence $seq\_ev(r, t, e_1, e_2, s)$ occurs when its terminator e_2 occurs, provided that its initiator occurred according to a given consumption mode.

Some possible consumption modes for event sequences are (many of these can be found in [ZU01] and [BM01]):

- **First**: Selects the oldest occurrence of the initiator, after which this occurrence is consumed.
- **Consumed Last**: Selects the most recent occurrence of the initiator, after which this occurrence is consumed.
- **Non-Consumed Last**: Selects the most recent occurrence of the initiator, which remains unconsumed as long as there is no occurrence of the initiator.
- **Cumulative**: Selects all occurrences of the initiator up to the situation where the terminator occurs, after which all these occurrences of the initiator are consummed.
- **FIFO**: Selects the earliest occurrence of the initiator that has not yet been consumed, after which this occurrence is consumed.
- **LIFO**: Selects the latest occurrence of the initiator that has not yet been consumed, after which this occurrence is consumed.

Example 5.2 *Suppose we have have 17 situations and the following occurrences of events E_1 and E_2:*

$$\begin{aligned} E_1 &: S_0, S_1, S_2, S_3, S_4, S_8, S_9, S_{11}, S_{15}, S_{16} \\ E_2 &: S_5, S_6, S_7, S_{10}, S_{12}, S_{13}, S_{14}, S_{17} \end{aligned}$$

The six consumption modes considered in this section can now be illustrated as in Figure 5.2. An arrow $E_1 \rightarrow E_2$ means that E_1 is selected as initiator when E_2 occurs in order for $seq\_ev(r, t, E_1, E_2, s)$ to occur.

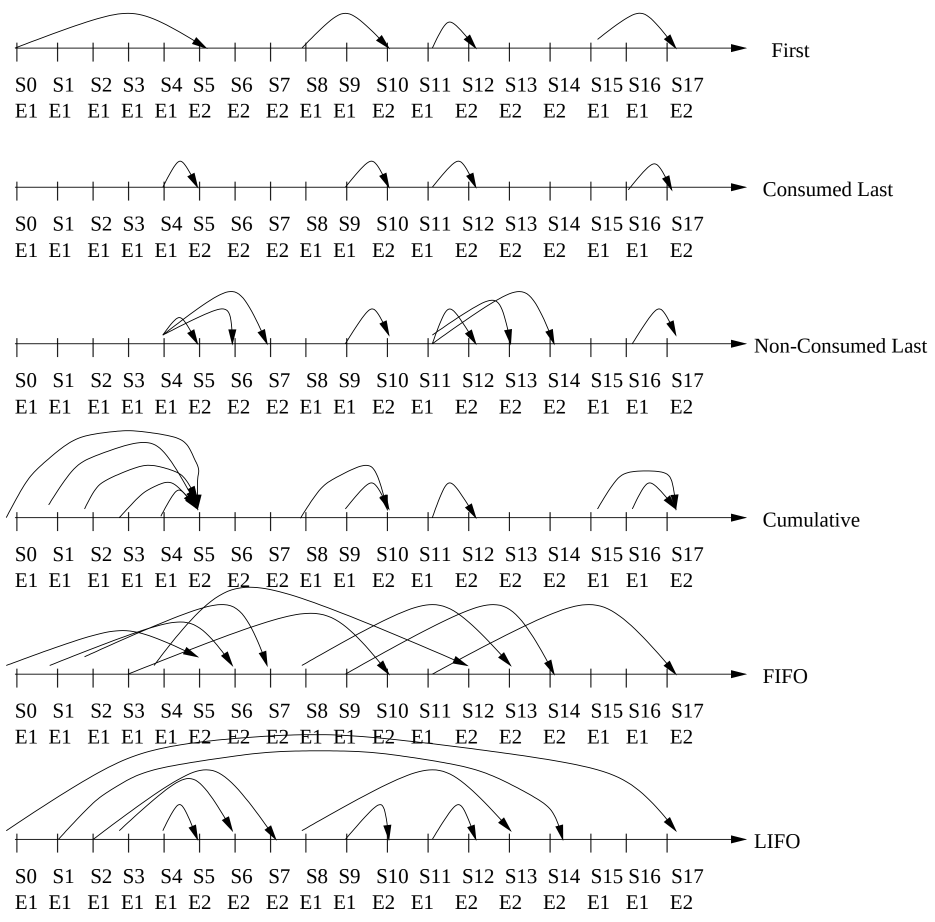

First
Consumed Last
Non-Consumed Last
Cumulative
FIFO
LIFO
S0 S1 S2 S3 S4 S5 S6 S7 S8 S9 S10 S11 S12 S13 S14 S15 S16 S17
E1 E1 E1 E1 E1 E2 E2 E2 E1 E1 E2 E1 E2 E2 E2 E1 E1 E2

Figure 5.2: Informal semantics of basic complex events

Now we spell out details of these consumption modes.

First. We express this by taking $\Psi_{CM}(t, \vec{s})$ in (5.17) as

$$(\forall s^*).s' \sqsubset s^* \sqsubset s \supset ((\exists r_1)e_1[r_1, t, s^*] \vee \neg(\exists r_2)e_2[r_2, t, s^*]). \tag{5.18}$$

So to detect the sequence under this mode, we have to establish the entailment

$$\begin{aligned}\mathcal{D} \models & (\exists r')e_2[r', t, s] \wedge (\exists s', r'')[s' \sqsubset s \wedge e_1[r'', t, s'] \wedge \\ & (\forall s^*).s' \sqsubset s^* \sqsubset s \supset ((\exists r_1)e_1[r_1, t, s^*] \vee \neg(\exists r_2)e_2[r_2, t, s^*])].\end{aligned} \tag{5.19}$$

Consumed Last. We express this by taking $\Psi_{CM}(t, \vec{s})$ in (5.17) as

$$(\forall s^*).s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)e_1[r_1, t, s^*] \wedge \neg(\exists r_2)e_2[r_2, t, s^*]. \tag{5.20}$$

So to detect the sequence under this mode, we have to establish the entailment

$$\begin{aligned}\mathcal{D} \models &(\exists r')e_2[r', t, s] \land (\exists s', r'')[s' \sqsubset s \land e_1[r'', t, s'] \land \\ &(\forall s^*).s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)e_1[r_1, t, s^*] \land \neg(\exists r_2)e_2[r_2, t, s^*]].\end{aligned} \tag{5.21}$$

Non-Consumed Last. We express this by taking $\Psi_{CM}(t, \vec{s})$ in (5.17) as

$$(\forall s^*).s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)e_1[r_1, t, s^*]. \tag{5.22}$$

So to detect the sequence under this mode, we have to establish the entailment

$$\begin{aligned}\mathcal{D} \models &(\exists r')e_2[r', t, s] \land (\exists s', r'')[s' \sqsubset s \land e_1[r'', t, s'] \land \\ &(\forall s^*).s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)e_1[r_1, t, s^*]].\end{aligned} \tag{5.23}$$

Cumulative: Here, we take $\Psi_{CM}(t, \vec{s})$ in (5.17) as

$$(\forall s^*).s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_2)e_2[r_2, t, s^*]. \tag{5.24}$$

So to detect the sequence under this mode, we have to establish the entailment

$$\begin{aligned}\mathcal{D} \models &(\exists r')e_2[r', t, s] \land (\exists s', r'')[s' \sqsubset s \land e_1[r'', t, s'] \land \\ &(\forall s^*).s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_2)e_2[r_2, t, s^*]].\end{aligned} \tag{5.25}$$

FIFO: Here, $\Psi_{CM}(t, \vec{s})$ in (5.17) is

$$\begin{aligned}&(\forall s^*)[s^* \sqsubset s \supset \neg((\exists r_1)e_1[r_1, t, s^*] \land seq\_ev(r, t, e_1, e_2, s^*))] \land \\ &(\forall s^*)[s^* \sqsubset s' \supset [(\exists r_1)e_1[r_1, t, s^*] \supset (\exists s^{**})s^{**} \sqsubset s \land seq\_ev(r, t, e_1, e_2, s^{**})]].\end{aligned} \tag{5.26}$$

So to detect the sequence under this mode, we must establish the entailment

$$\begin{aligned}\mathcal{D} \models &(\exists r')e_2[r', t, s] \land (\exists s', r'')[s' \sqsubset s \land e_1[r'', t, s'] \land \\ &(\forall s^*)[s^* \sqsubset s \supset \neg(e_1[r, t, s^*] \land seq\_ev(r, t, e_1, e_2, s^*))] \land \\ &(\forall s^*)[s^* \sqsubset s' \supset [(\exists r_1)e_1[r_1, t, s^*] \supset (\exists s^{**})s^{**} \sqsubset s \land seq\_ev(r, t, e_1, e_2, s^{**})]]].\end{aligned} \tag{5.27}$$

LIFO: Here, $\Psi_{CM}(t, \vec{s})$ in (5.17) is

$$\begin{aligned}&(\forall s^*)[s^* \sqsubset s \supset \neg((\exists r_1)e_1[r_1, t, s^*] \land seq\_ev(r, t, e_1, e_2, s^*))] \land \\ &(\forall s^*)[s' \sqsubset s^* \sqsubset s \supset [(\exists r_1)e_1[r_1, t, s^*] \supset (\exists s^{**})s^{**} \sqsubset s \land seq\_ev(r, t, e_1, e_2, s^{**})]].\end{aligned} \tag{5.28}$$

So to detect the sequence under this mode, we must establish the entailment

$$\begin{aligned}\mathcal{D} \models &(\exists r')e_2[r', t, s] \land (\exists s', r'')[s' \sqsubset s \land e_1[r'', t, s'] \land \\ &(\forall s^*)[s^* \sqsubset s \supset \neg(e_1[r, t, s^*] \land seq\_ev(r, t, e_1, e_2, s^*))] \land \\ &(\forall s^*)[s' \sqsubset s^* \sqsubset s \supset [(\exists r_1)e_1[r_1, t, s^*] \supset (\exists s^{**})s^{**} \sqsubset s \land seq\_ev(r, t, e_1, e_2, s^{**})]]].\end{aligned} \tag{5.29}$$

For the purpose of characterizing (some of) the consumption modes, and, later, specifying properties of ECA rule sets, the set of operators of first order past temporal logic can be introduced using a set of appropriate abbreviations as follows: ([AB98]):

Definition 5.5 (First Order Past Temporal Logic)

$$previously(\phi, s) =_{df} (\exists s')(\exists a).s = do(a, s') \wedge \phi(s'),$$
$$past(\phi, s) =_{df} (\exists s').S_0 \sqsubseteq s' \sqsubset s \wedge \phi(s'),$$
$$always(\phi, s) =_{df} (\forall s').S_0 \sqsubseteq s' \sqsubset s \supset \phi(s'),$$
$$since(\phi, \psi, s) =_{df} (\exists s')[S_0 \sqsubseteq s' \sqsubset s \wedge \psi(s') \wedge (\forall s'').s' \sqsubset s'' \sqsubseteq s \supset \phi(s')].$$

First order past temporal formulas expressed in the situation calculus are formulas that may include the logical connectives $\neg$, $\wedge$, $\vee$ and $\supset$, quantification over individuals of sort $objects$, and the predicates abbreviated above. In the abbreviations above, ϕ and ψ are first order past temporal formulas.

Now we may characterize (some of) the consumption modes above by stating the following:

Proposition 5.6 *Each of the First, Consumed-Last, Non-Consumed-Last, and Cumulative consumption modes are expressible in the the past temporal fragment of the situation calculus.*

In the context of the situation calculus, the whole development above leads to the concept of an *event logic* which we now formally express as a definition.

Definition 5.7 (Event Logic) *An event logic is a triple $(E, C, \mathcal{L})$, where E is a set of event fluents, C is a set of event connectives, together with the predicate $\sqsubset$, and $\mathcal{L}$ is a fragment of the situation calculus specifying the consumption mode associated with event sequences.*

Definition 5.8 (Implication and Equivalence Problems for an Event Logic) *Suppose $e[r, t, s]$ and $e'[r, t, s]$ are two events of a given event logic $\mathcal{E}$. Then the implication and equivalence problems for $\mathcal{E}$ are the problems of establishing whether, for given R and T, $\mathcal{D} \models (\forall s).e[R, T, s] \supset e'[R, T, s]$, and $\mathcal{D} \models (\forall s).e[R, T, s] \equiv e'[R, T, s]$, respectively. Here $\mathcal{D}$ specifies the semantics of events according to the event logic $\mathcal{E}$.*

Assume the fluents $seq\_ev^F(r, t, e_1, e_2, s)$, $seq\_ev^{CL}(r, t, e_1, e_2, s)$, $seq\_ev^{NL}(r, t, e_1, e_2, s)$, and $seq\_ev^{CUMUL}(r, t, e_1, e_2, s)$ denote event sequence fluents with the consumption modes First, Consumed-Last, Non-Consumed-Last, and Cumulative, whose semantics have been given above. Then we have the following result:

Theorem 5.9 *Suppose $\mathcal{E} = (E, C, \mathcal{L})$ is the event logic given by:*

- $E = \{F\_inserted(r, t, s), F\_inserted(r, t, s), seq\_ev^{CM}(r, t, e_1, e_2, s), simult\_ev(r, t, e_1, e_2, s),$
  $conj\_ev(r, t, e_1, e_2, s), disj\_ev(r, t, e_1, e_2, s), neg\_ev(r, t, e, s)\},$
  with $CM \in \{F, CL, NL, CUMUL\}$*;*
- $C = \{\neg, \wedge, \sqsubset\}$*;*

- *$\mathcal{L}$ is the past temporal fragment of the situation calculus.*

Then both the implication and the equivalence problems for $\mathcal{E}$ are PSPACE-hard.

5.4 Active Relational Theories

An *active relational language* is a relational language extended in the following way: for each $n+2$-ary fluent $F(\vec{x}, t, s)$, we introduce two n+3-ary transition fluents $F\_inserted(r, \vec{x}, t, s)$ and $F\_deleted(r, \vec{x}, t, s)$, and two 3-ary event fluents $F\_inserted(r, t, s)$ and $F\_deleted(r, t, s)$.

Definition 5.10 (Active Relational Theory for a transaction model M) *Suppose $\mathcal{L} = (\mathfrak{R}, \mathfrak{W})$ is an active relational language. Then a theory $\mathcal{D} \subseteq \mathfrak{W}$ is an* active relational theory *for a transaction model M iff $\mathcal{D}$ is of the form $\mathcal{D} = \mathcal{D}_{brt} \cup \mathcal{D}_{tf} \cup \mathcal{D}_{ef}$, where*

1. *$\mathcal{D}_{brt}$ is a basic relational theory for the transaction model M;*
2. *$\mathcal{D}_{tf}$ is the set of axioms for transition fluents;*
3. *$\mathcal{D}_{ef}$ is the set of axioms and definitions for simple and complex event fluents which are expressed in a given event logic.*

Definition 5.11 *(***Active Relational Database***) An active relational database is a pair $(\mathfrak{R}, \mathcal{D})$, where $\mathfrak{R}$ is an active relational language and $\mathcal{D}$ is an active relational theory.*

Assume the same notations $seq\_ev^{F}(r, t, e_1, e_2, s)$, $seq\_ev^{CL}(r, t, e_1, e_2, s)$, $seq\_ev^{NL}(r, t, e_1, e_2, s)$, and $seq\_ev^{CUMUL}(r, t, e_1, e_2, s)$ with meanings as in Theorem 5.9, and suppose $seq\_ev^{FIFO}(r, t, e_1, e_2, s)$ and $seq\_ev^{LIFO}(r, t, e_1, e_2, s)$ denote event sequence fluents with the consumption modes FIFO and LIFO, respectively. Then we have the following result:

Theorem 5.12 *Suppose $\mathcal{D}$ is an active relational theory with global consumption scope for the primitive event fluents. Then the following equivalences can be established:*

1. *First, Consumed-Last and cumulative consumption modes are equivalent; i.e.,*

   $\mathcal{D} \models seq\_ev^{F}(r, t, e_1, e_2, s)$ *iff* $\mathcal{D} \models seq\_ev^{CL}(r, t, e_1, e_2, s)$ *iff* $\mathcal{D} \models seq\_ev^{CUMUL}(r, t, e_1, e_2, s)$.

2. *Non-Consumed-Last, LIFO, and FIFO consumption modes are equivalent; i.e.,*

   $\mathcal{D} \models seq\_ev^{NL}(r, t, e_1, e_2, s)$ *iff* $\mathcal{D} \models seq\_ev^{FIFO}(r, t, e_1, e_2, s)$ *iff* $\mathcal{D} \models seq\_ev^{LIFO}(r, t, e_1, e_2, s)$.

It is important to make clear what the equivalences above means. Intuitively, the logical equivalence of two consumption modes M_1 and M_2 amounts to the fact that any given sequence will occur at exactly the same situations under both M_1 and M_2. This ultimately leads to the same active behavior under both M_1 and M_2. Notice that the theorem asssume global consumption scope. It still is open whether these equivalences still hold in the case of local consumption scope.

5.5 Summary

In this chapter, we have extended the basic relational theories of the previous chapter to active relational theories that capture the reactive and execution models of active behaviors. As we saw, an active relational theory precisely extends a basic relational theory capturing a specific ATM with axioms for typical active database fluents such as event fluents that are induced by the original database fluents of the domain.

More specifically, we have introduced further building blocks that are specific to active databases into the specification framework laid down so far in this chapter. These building blocks include event logics, fragments of the situation calculus used to capture and specify event algebras in logic. Then, we have formally defined the active relational theories and show some of the properties of event logics. The main result here is a classification theorem for the various consumption modes identified. This theorem says roughly which consumption modes are equivalent and which are not.

In a nutshell, we have achieved the following results:

- specification of event algebras as logics in the situation calculus;
- semantics of the following dimensions of active behavior: event consumption modes, and net effects;
- classification theorem for the various consumption modes identified.

Chapter 6

Specifying Execution Models

The previous chapter was devoted to extending basic relational theories to model the reactive models of active behaviors. The new theories introduced there were called active relational theories.

Up to this point, we have uniquely dealt with transactions, informally viewed as execution traces. It is now time to turn our attention to what kind of programs we are supposed to execute in order to get the execution traces we have characterized in Chapter 4, and to how we execute these programs. So we continue to keep the distinction between transactions which are sequences of database actions and transaction programs which must be executed in order to get those sequences of database actions. In the present chapter, we specify transaction programs as well formed programs written in ConGolog, a situation calculus based programming language. Such *well formed ConGolog programs* are executed using a special ternary predicate $Do(P, s, s')$ which will serve as an abstract interpreter; $Do(P, s, s')$, introduced in [LRL$^+$97], intuitively means: s is a situation reached by executing program P in the situation s'. The predicate Do is defined such that the situations reached by executing well formed ConGolog programs are all legal in the sense of Chapter 4. Therefore, these situations will have the various (relaxed) ACID properties of the ATMs.

We specify a given execution model of active behavior by compiling a set of given ECA rules into a ConGolog program called *rule program* whose structure is constrained according to that given execution model. Now, the semantics of the predicate $Do(P, s, s')$ is given in a way such that the rule program is implicitly executed whenever a transaction program is executed. It is important to notice that an execution model of the active behavior of transactions is still concerned with execution traces, not with programs, as we still are concerned with the situations — thus with sequences of database actions — reached by executing transaction programs.

Section 6.1 shows how well formed ConGolog programs are used to capture transaction programs and how to abstractly execute the well formed ConGolog programs using the basic relational theories corresponding to the various ATMs as background axioms. Next, Sections 6.2 and 6.3 specify various execution models of active behaviors as rule programs. Section 6.4 contains the main results of this chapter, namely classification theorems for the various execution models of active behavior that we consider. We specify the notion of rule priorities in Section 6.5. Finally, Section 6.6 tackles the issue of properties of a set of rules.

6.1 Non-Markovian ConGolog

GOLOG, introduced in [LRL+97] and enhanced with parallelism in [DGLL97] and [DGLL00] to yield ConGolog, is a situation calculus-based programming language for defining complex actions in terms of a set of primitive actions axiomatized in the situation calculus. It has the following Algol-like control structures:

- nil, the empty program;
- *sequence* ($[\alpha\ ;\ \beta]$; do action α, followed by action β);
- *test actions* ($\phi?$; test the truth value of expression ϕ in the current situation);
- *nondeterministic action choice* ($\alpha \mid \beta$; do α or β);
- *nondeterministic choice of arguments* ($(\pi\ \vec{x})\alpha$; nondeterministically pick a value for $\vec{x}$, and for that value of $\vec{x}$, do action α);
- *conditionals* and *while* loops; and
- *procedures*, including recursion.

The following are ConGolog constructs for expressing parallelism:

- *Concurrency* ($[\alpha\ \ ||\ \ \beta]$; do α and β in parallel);
- *Concurrent iteration* ($\alpha^{||}$; do α zero or more times in parallel).

The purpose of this section is to show how ConGolog programs are used to capture transaction programs and how the semantics of this programs is used to simulate the ATMs.

6.1.1 Well-formed ConGolog Programs

ConGolog syntax is built using constructs that suppress any reference to situations in which test are evaluated. These will be restored at run time by the ConGolog interpreter. The following is a restriction to relational languages of a similar definition given in [PR99].

Definition 6.1 (Situation-Suppressed Terms and Formulas) *Suppose $\mathfrak{R}$ is a relational language. Then the* situation-suppressed *terms (ss-terms) of $\mathfrak{R}$ are given by:*

1. *Any variable or constant of sort $\mathcal{A}$, $\mathcal{O}$, or $\mathcal{S}$ of $\mathfrak{R}$ is an ss-term.*

2. *Whenever F is a functional ss-fluent of $\mathfrak{R}$ and $t_1, \cdots, t_n$ are ss-terms of the appropriate sort, then $F(t_1, \cdots, t_n)$ is an ss-term.*

3. *If a is an action function symbol of $\mathfrak{R}$, and $t_1, \cdots, t_n$ are variables or constants of $\mathfrak{R}$, then $a(t_1, \cdots, t_n)$ is a ss-term.*

4. *For any situation term σ and any action term a, $do(a, \sigma)$ is an ss-term.*

The situation-suppressed formulas (ss-formulas) of $\mathfrak{R}$ are inductively given as follows:

1. *Whenever t, t' are ss-terms of the same sort, then $t = t'$ is an ss-formula. Notice that an ss-formula here, contrary to [PR99], may mention an equality between terms of sort* situations.

2. *Whenever t is an ss-term of sort $\mathcal{A}$, then $Poss(t)$ is an ss-formula.*

3. *Whenever F is an $n+1$-ary relational fluent of $\mathfrak{R}$ and $t_1, \cdots, t_n$ are ss-terms of sort $\mathcal{O}$, then $F(t_1, \cdots, t_n)$ is a ss-formula.*

4. *Whenever P is an m-ary situation independent predicate of $\mathfrak{R}$ and $t_1, \cdots, t_n$ are ss-terms of sort $\mathcal{O}$, then $P(t_1, \cdots, t_n)$ is a ss-formula.*

5. *Whenever t and t' are situation terms of $\mathfrak{R}$, then $t \sqsubset t'$ is an ss-formula.*

6. *Are ϕ and ψ ss-formulas of $\mathfrak{R}$, so are also $\neg\phi$, $\phi \wedge \psi$, and $(\exists x)\phi$ for any variable x.*

Calling situation terms like S_0, $do(A, S_0)$, etc "situation"-suppressed might sound counterintuitive. However, this definition just means that ss-formulas are first order and may still mention situation terms, but never as last argument of fluents; therefore ss-formulas quantify only over those situations that are mentioned in equalities between terms of sort $situations$ and in $\sqsubset$-atoms. For example, the following is an ss-formula:

$$S_0 \sqsubset do(A, (do(B, S_0))) \wedge (\forall x, y, z, w, t)[accounts(x, y, z, w, t) \supset z \geq 0],$$

since the fluent $accounts(x, y, z, w, t, s)$ has its situation argument removed, whereas the following is not:

$$S_0 \sqsubset do(A, (do(B, S_0))) \wedge (\forall x, y, z, w, t, s)[accounts(x, y, z, w, t, s) \supset z \geq 0].$$

Definition 6.2 (Well formed ConGolog Program for Flat Transactions) *A ConGolog program for flat transactions has the following syntax:*[1]

$$\begin{aligned} \langle prog \rangle \ ::= \ & \langle internal\ action \rangle \mid \langle test\ action \rangle ? \mid (\langle prog \rangle ; \langle prog \rangle) \mid (\langle prog \rangle | \langle prog \rangle) \mid \\ & (\langle prog \rangle \parallel \langle prog \rangle) \mid \langle prog \rangle^{||} \mid (\pi x)\langle prog \rangle \mid \langle prog \rangle^{*} \mid \langle procedure\ call \rangle \mid \\ & (\textbf{proc}\ P_1(\vec{x}_1)\langle prog \rangle\ \textbf{endProc}\ ; \cdots ;\ \textbf{proc}\ P_n(\vec{x}_n)\langle prog \rangle\ \textbf{endProc}\ ;\ \langle prog \rangle) \end{aligned}$$

Notice that

1. $\langle internal\ action \rangle$ *is a situation-suppressed internal action term.*
2. $\langle test\ action \rangle$ *is an ss-formula.*
3. *The variable x in $(\pi x)\langle prog \rangle$ must be of sort* actions *or* objects*, never of sort $situations$.*
4. $\langle procedure\ call \rangle$ *is a predicate — a procedure name — of the form $P(t_1, \cdots, t_n)$ where the t_i are ss-terms whose sorts match those of the n arguments in the declaration of P.*

A well formed ConGolog program for flat transactions is syntactically defined as follows:

$$\begin{aligned} \langle wfprog \rangle ::= \ & (\textbf{proc}\ P_1(\vec{x}_1)\langle prog \rangle\ \textbf{endProc}\ ; \cdots ;\ \textbf{proc}\ P_n(\vec{x}_n)\langle prog \rangle\ \textbf{endProc}\ ; \\ & \quad Begin(t); \langle prog \rangle; End(t)) \mid \\ & \langle wfprog \rangle \parallel \langle wfprog \rangle \end{aligned}$$

Definition 6.3 (Well formed ConGolog Program for CNTs)[2] *A ConGolog program for CNTs has the following syntax:*

$$\begin{aligned} \langle prog \rangle \ ::= \ & \langle internal\ action \rangle \mid \langle test\ action \rangle ? \mid (\langle prog \rangle ; \langle prog \rangle) \mid (\langle prog \rangle | \langle prog \rangle) \mid \\ & (\langle prog \rangle \parallel \langle prog \rangle) \mid \langle prog \rangle^{||} \mid (\pi x)\langle prog \rangle \mid \langle prog \rangle^{*} \mid \\ & (Spawn(t,t')\ ;\ \langle prog \rangle\ ;\ End(t')) \mid \langle procedure\ call \rangle \mid \\ & (\textbf{proc}\ P_1(\vec{x}_1)\langle prog \rangle\ \textbf{endProc}\ ; \cdots ;\ \textbf{proc}\ P_n(\vec{x}_n)\langle prog \rangle\ \textbf{endProc}\ ;\ \langle prog \rangle) \end{aligned}$$

Here, $\langle internal\ action \rangle$, $\langle test\ action \rangle$, the variable x in $(\pi x)\langle prog \rangle$, and $\langle procedure\ call \rangle$ are defined as in Definition 6.2.

A well formed ConGolog program for CNTs is syntactically defined as follows:

$$\begin{aligned} \langle wfprog \rangle ::= \ & (\textbf{proc}\ P_1(\vec{x}_1)\langle prog \rangle\ \textbf{endProc}\ ; \cdots ;\ \textbf{proc}\ P_n(\vec{x}_n)\langle prog \rangle\ \textbf{endProc}\ ; \\ & \quad Begin(t); \langle prog \rangle; End(t)) \end{aligned}$$

[1]As in [LRL$^+$97], loops and conditionals can be defined in terms of the constructs given here.

[2]Recall that CNT in this definition stands for closed nested transactions. In the next definition, ONT will stand for open nested transactions.

Notice that we express a transaction behavior as a main program that is enclosed between a $Begin(t)$ and an $End(t)$ action. Such a main program may call use $Spawn(t, t')$ in its body to spawn subtransactions.

Definition 6.4 (Well formed ConGolog Program for ONTs) *A ConGolog program for ONTs has the same syntax as in Definition 6.3, except that the EBNF alternative*

$$(Spawn(t, t') \ ; \ \langle prog \rangle \ ; \ End(t'))$$

is replaced by

$$(Begin(t, t', m, c) \ ; \ \langle prog \rangle \ ; \ End(t')),$$

where the argument m of $Begin(t, t', m, c)$ can be $OPEN$, $CLOSED$, or INV, and the argument c can be $COMP$, $NONCOMP$, or INV.

A well formed ConGolog program for ONTs is syntactically defined as follows:

$$\langle wfprog \rangle ::= (\textbf{proc } P_1(\vec{x}_1)\langle prog \rangle \textbf{ endProc} \ ; \cdots ; \ \textbf{proc } P_n(\vec{x}_n)\langle prog \rangle \textbf{ endProc} \ ;$$
$$Begin(t, NIL, INV, INV); \langle prog \rangle; End(t))$$

From now on, when writing about a well formed ConGolog program for a given transaction model, we shall omit any mention of the transaction model and ConGolog. The context will clearly determine the appropriate transaction model.

6.1.2 Semantics of Well Formed ConGolog Programs

With the ultimate goal of handling database transactions, it is appropriate to adopt an operational semantics of well formed ConGolog programs based on a single-step execution of these programs; such a semantics is introduced in ([DGLL97]). First, two special predicates $Trans$ and $Final$ are introduced. $Trans(\delta, s, \delta', s')$ means that program δ may perform one step in situation s, ending up in situation s', where program δ' remains to be executed. $Final(\delta, s)$ means that program δ may terminate in situation s. A single step here is either a primitive or a testing action. Then the two predicates are characterized by appropriate axioms. These axioms contain, for example, the following cases (See [DGLL97] and Appendix C for full details):

$$Trans(\delta_1; \delta_2, s, \delta, s') \equiv Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta, s') \vee$$
$$(\exists \gamma).\delta = (\gamma; \delta_2) \wedge Trans(\delta_1, s, \gamma, s'),$$
$$Trans(\delta_1 | \delta_2, s, \delta, s') \equiv Trans(\delta_1, s, \delta, s') \vee Trans(\delta_2, s, \delta, s')$$

to express the semantics of sequences and nondeterministic choice of actions, respectively.

Our axioms for $Trans$ differs from that of [DGLL97] with respect to the handling of primitive and test actions:

Definition 6.5 (Semantics of $Trans$)

$$
\begin{aligned}
Trans(a, s, a', s') \equiv\ & Poss(a, s) \wedge a' = nil \wedge \\
& \{(\exists a'', s'', t)[s'' = do(a, s) \wedge systemAct(a'', t) \wedge Poss(a'', s'') \wedge s' = do(a'', s'')] \vee \\
& \quad s' = do(a, s) \wedge [(\forall a'', t) systemAct(a'', t) \supset \neg Poss(a'', s')]\}, \qquad (6.1)\\
Trans(\phi?, s, a', s') \equiv\ & Holds(\phi, s, s') \wedge a' = nil. \qquad (6.2)
\end{aligned}
$$

In the characterization above, we take particularities of system actions into account when processing primitive actions. These actions must occur whenever possible, so the interpreter must test for their possibility upon each performance of a primitive action. The formula (6.1) captures this requirement; it intuitively means that the primitive action a may legally execute one step in the log s, ending in log s' where a' remains to be executed iff a is possible, the remaining action a' is the empty transaction, and either any possible system action a'' is executed immediately after the primitive action a has been executed and the log s' contains the action a followed by the system action a'', or no system action is possible and the log s' contains only the action a. The formula (6.2) says that the test action $\phi?$ may legally be performed in one or more steps in the log s, ending in log s' where a' remains to be executed iff ϕ holds in s, yielding a log s' in a way to be explained below, and a' is an empty program.

Given situation calculus axioms of a domain theory, an execution of a program δ in situation s is the task of finding a situation s' such that there is a final configuration (δ', s'), for some remaining program δ', after performing a couple of transitions from δ, s to δ', s'. Program execution is captured by using the abbreviation $Do(\delta, s, s')$ ([Rei01]). In the single-step semantics, $Do(\delta, s, s')$ intuitively means that program δ is single-stepped until the remainder of program δ may terminate in situation s'; and s' is one of the logs reached by single-stepping the program δ, beginning in a given situation s. Formally, we have ([DGLL97]):

$$
Do(\delta, s, s') =_{df} (\exists \delta').Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s'), \qquad (6.3)
$$

where $Trans^*$ denotes the transitive closure of $Trans$. Finally, a program execution starting in situation S_0 is formally the task of finding a situation s' such that $\mathcal{D} \models Do(\delta, S_0, s')$, where $\mathcal{D}$ is the domain theory.

Definition 6.6 *The notation $\phi[s]$ denotes the situation calculus formula obtained from a given formula ϕ by restoring the situation argument s in all the fluents (as their last argument) occurring in ϕ.*

The predicate $Holds(\phi, s, s')$ captures the revised Lloyd-Topor transformations of [Rei01]; these are transformations in the style of Lloyd-Topor([Llo88]), but without its auxiliary predicates. The predicate $Holds(\phi, s, s')$ takes a formula ϕ and establish whether it holds in the log s or not. If ϕ is a fluent literal, then the next log s' will be $do(\phi, s)$; if it is a nonfluent literal, then $s' = s$; otherwise revised Lloyd-Topor transformations are performed on ϕ until we reach literals.

To formally define the $Holds(\phi, s, s')$ predicate, we need to define concepts of situation-restored term and situation-restored formula whose semantics are the opposite of those of ss-terms and ss-formulas, respectively.

Definition 6.7 (Situation-Restored Terms and Formulas) *Suppose $\mathfrak{R}$ is a relational language. Then the* situation-restored *terms (sr-terms) of $\mathfrak{R}$ are inductively given by:*[3]

1. *Any variable or constant of sort $\mathcal{A}$, $\mathcal{O}$, or $\mathcal{S}$ of $\mathfrak{R}$ is an sr-term.*
2. *Whenever F is a functional ss-fluent of $\mathfrak{R}$, σ is a situation term, and $t_1, \cdots, t_n$ are sr-terms of the appropriate sort, then $F(t_1, \cdots, t_n, \sigma)$ is an sr-term.*
3. *If a is an action function symbol of $\mathfrak{R}$, and $t_1, \cdots, t_n$ are variables or constants of $\mathfrak{R}$, then $a(t_1, \cdots, t_n)$ is a sr-term.*
4. *For any situation term σ and any action term a, $do(a, \sigma)$ is an sr-term.*

The situation-restored *formulas (sr-formulas) of $\mathfrak{R}$ are inductively given as follows:*

1. *Whenever t, t' are sr-terms of the same sort, then $t = t'$ is an sr-formula.*
2. *Whenever t is an sr-term of sort $\mathcal{A}$, then $Poss(t)$ is an sr-formula.*
3. *Whenever F is an $n+1$-ary relational ss-fluent of $\mathfrak{R}$ and $t_1, \cdots, t_n$ are sr-terms of sort $\mathcal{O}$, and σ is a situation term, then $F(t_1, \cdots, t_n, \sigma)$ is a sr-formula.*
4. *Whenever P is an m-ary situation independent predicate of $\mathfrak{R}$ and $t_1, \cdots, t_n$ are sr-terms of sort $\mathcal{O}$, then $P(t_1, \cdots, t_n)$ is a sr-formula.*
5. *Whenever σ and σ' are situation terms of $\mathfrak{R}$, then $\sigma \sqsubset \sigma'$ is an sr-formula.*
6. *Are ϕ and ψ sr-formulas of $\mathfrak{R}$, so are also $\neg\phi$, $\phi \wedge \psi$, and $(\exists x)\phi$ for any variable x.*

Whenever t and ϕ are a ss-term and a ss-formula, respectively, and σ is a situation term, we use the notation $t[\sigma]$ and $\phi[\sigma]$ to denote the corresponding sr-term and sr-formula, respectively.

Definition 6.8 (Semantics of $Holds$) [4]

$$Holds(\phi, s, s') =_{df} \phi[s] \wedge s' = do(\phi\_reads[s], s), \text{ when } \phi \text{ is a database fluent literal},$$

$$Holds(\phi, s, s') =_{df} \phi[s] \wedge s' = s, \text{ when } \phi \text{ is a nonfluent literal},$$

$$Holds((\phi_1 \wedge \phi_2), s, s') =_{df} (\exists s'').Holds(\phi_1, s, s'') \wedge Holds(\phi_2, s'', s'),$$

[3]Any fluents whose situation is suppressed will be called ss-fluents.

[4]In the sequel, we will sligthly abuse the notation of read actions: whenever we will have a read action $F\_reads(\vec{x}, t, s)$, we will just write $F(\vec{x}, t, s)?$.

$$Holds((\phi_1 \vee \phi_2), s, s') =_{df} Holds(\phi_1, s, s') \vee Holds(\phi_2, s, s'),$$

$$Holds((\phi_1 \supset \phi_2), s, s') =_{df} Holds(\neg\phi_1 \vee \phi_2, s, s'),$$

$$Holds((\phi_1 \equiv \phi_2), s, s') =_{df} Holds((\phi_1 \supset \phi_2) \wedge (\phi_2 \supset \phi_1), s, s'),$$

$$Holds((\forall x)\phi, s, s') =_{df} Holds(\neg(\exists x)\neg\phi, s, s'),$$

$$Holds((\exists x)\phi, s, s') =_{df} (\exists x)Holds(\phi, s, s'),$$

$$Holds(\neg\neg\phi, s, s') =_{df} Holds(\phi, s, s'),$$

$$Holds(\neg(\phi_1 \wedge \phi_2), s, s') =_{df} Holds(\neg\phi_1, s, s') \vee Holds(\neg\phi_2, s, s'),$$

$$Holds(\neg(\phi_1 \vee \phi_2), s, s') =_{df} (\exists s'').Holds(\neg\phi_1, s, s'') \wedge Holds(\neg\phi_2, s'', s'),$$

$$Holds(\neg(\phi_1 \supset \phi_2), s, s') =_{df} Holds(\neg\neg(\phi_1 \vee \phi_2), s, s'),$$

$$Holds(\neg(\phi_1 \equiv \phi_2), s, s') =_{df} Holds(\neg[(\phi_1 \supset \phi_2) \wedge (\phi_2 \supset \phi_1)], s, s'),$$

$$Holds(\neg(\forall x)\phi, s, s') =_{df} Holds((\exists x)\neg\phi, s, s'),$$

$$Holds(\neg(\exists x)\phi, s, s') =_{df} \neg(\exists x)Holds(\phi, s, s').$$

Definition 6.8 expresses a particular semantics for test actions that is appropriate for handling database transactions. It is important to notice how our test actions are different than those of [DGLL97] and why they are needed. Our test actions differ from those of ConGolog ([DGLL97]) in two ways. First of all, unlike in ConGolog, ours are genuine actions and not merely tests that may be forgotten as soon as they are executed. We record test actions in the log; i.e. performing a test changes the situation. Second, depending on the syntactic form of the formula in the test, we may end up executing more than just a "single step". More precisely, more than one single actions are added to the log whenever more than one tests of fluent literals are involved in the formula being tested. This semantics is dictated by the very nature of ATMs. Here, many test actions correspond to database reading actions. A transaction has no means of remembering which transaction it had read from other than to record reading actions in the log. This cannot be done with the semantics for test action found in [DGLL97]. In other words, in the absence of test actions in the log, the semantics of [DGLL97] has no straightforward way to express such things as transaction T_1 reads data from transaction T_2.

6.1.3 Simulation of Well Formed ConGolog Programs

We use the ConGolog language as a transaction language for specifying and simulating ATMs at the logical level. To simulate a specific ATM, we first pick the appropriate basic relational theory $\mathcal{D}$ corresponding to that ATM. Then, we write a well formed ConGolog program T expressing the desired transactional behavior. Simulating the program T amounts to the task of establishing the entailment

$$\mathcal{D} \models (\exists s')\ Do(T, S_0, s'). \tag{6.4}$$

To establish the entailment (6.4), we need to accommodate non-Markovian tests. These are tests involving the predicate $\sqsubset$; they allow to test whether a log is a sublog of another log. Henceforth, the regression operator (see Appendix A) must incorporate a case handling the predicate $\sqsubset$. Such a regression operator is defined in [Gab00] and adapted to active relational theories in Appendix A.

Example 6.1 *Consider the Debit/Credit example of Section 4.4. In addition to the axioms given in Secction 4.4, we have the following successor state axiom characterizing the system fluent $served(aid, s)$ which is used for synchronization purposes:*

$$served(aid, do(a, s)) \equiv report(aid) \vee served(aid, s).$$

The action $report(aid)$, whose precondition axiom is

$$Poss(report(aid), s) \equiv true,$$

is used to make the fluent $served(aid, s)$ true by indicating that a request emitted by the owner of the account aid has been granted. The situation independent predicate $requested(aid, req)$ registers such requests, where req is a positive or negative real number corresponding to a deposit or a withdrawal of that amount of money.

Now we give the following ConGolog procedures which are well-formed and capture the essence of the debit/credit example:

```
proc $a\_update(t, aid, amt)$
  $(\pi\ bid, abal, abal', tid)[accounts(aid, bid, abal, tid, t)?$ ;
    $[abal' = abal + amt]?$ ;
    $a\_del(aid, bid, abal, tid, t)$ ;
    $a\_ins(aid, bid, abal', tid, t)]$
endProc
proc $execDebitCredit(t, bid, tid, aid, amt)$
    $a\_update(aid, amt)$ ;
    $(\pi\ abal)\ [accounts(aid, bid, abal, tid, t)?$ ;
            $t\_update(t, tid, amt)$ ; $b\_update(t, bid, amt)]$
endProc
proc $processReq(t, tid, aid, amt)$
  $(\pi\ bid, abal)[accounts(aid, bid, abal, tid, t)?$ ; $execDebitCredit(t, bid, tid, aid, amt)]$
endProc
```

```
proc $processTrans(t)$
  $Begin(t);$
    $[(\pi\ bid, aid, abal, tid, req).$
        $\{accounts(aid, bid, abal, tid, t) \wedge requested(aid, req) \wedge \neg served(aid)\}?$ ;
        $report(aid)$ ; $Spawn(t, aid)$ ; $processReq(t, tid, aid, req)$ ; $End(aid)]^{||}$ ;
    $\neg((\exists\ aid, req).requested(aid, req) \wedge served(aid))?$ ;
  $End(t)$
endProc
```

Similarly to the first procedure, we can give procedures $t\_update(tid, amt)$ and $b\_update(bid, amt)$ for updating teller and branch balances, respectively. ■

The ACI(D) properties are enforced by the interpreter that either commits work done so far or rolls it back whenever the database general ICs are violated. Thus, well formed programs are a specification of transactions with the full scale of a programming language at the logical level. Notice that a formula ϕ in a test $\phi?$ is in fact an ss-formula whose situation argument is restored at run-time by the interpreter. Notice also the use of the concurrent iteration in the last procedure; this spawns a new child transaction for each account that emitted a request but has not yet been served. For simplicity in this example, we have assumed that each account has at most one request; this allows us to use the account identifiers aid to denote spawn subtransactions.

Now we can simulate the program, say $processTrans(T)$ of Example 6.1, by performing the theorem proving task of establishing the entailment

$$\mathcal{D} \models (\exists s')\ Do(processTrans(T), S_0, s'),$$

where S_0 is the initial, empty log, and $\mathcal{D}$ is the basic relational theory for nested transactions that comprises the axioms above; this exactly means that we look for some log that is generated by the program T. We are interested in any instance of s resulting from the proof obtained by establishing this entailment. Such an instance is obtained as a side-effect of this proof.

In Definition 6.5, we take particularities of system actions into account. These actions must occur whenever they are possible, so the interpreter must test for their possibility upon each performance of a primitive action. Definition 6.5 captures this requirement.

Definition 6.9 (Universal Possibility Assumption (UPA) for Test Actions) *This is the sentence*

$$(\forall F, \vec{x}, t, s)Poss(F\_reads(\vec{x}, t)[s], s). \tag{6.5}$$

The UPA allows unrestricted carrying out of test actions which in the database setting are database queries. Using the UPA and Definition 6.5, we can show that Do generates only legal situations:

Theorem 6.10 *Suppose $\mathcal{D}$ is a relational theory (either for CNTs, or for ONTs), and let T be a well formed ConGolog program for CNTs or for ONTs. Then,*

$$\mathcal{D} \models (\forall s).Do(T, S_0, s) \supset legal(s). \tag{6.6}$$

6.2 Specifying the Execution Models with Flat Transactions

In this section, we specify the execution models of active databases by assuming that the underlying transaction model is that of flat transactions.

6.2.1 Classification

The three components of the ECA rule — i.e. **E**vent, **C**ondition, and **A**ction — are the main dimensions of the representational component of active behavior. Normally, either an indication is given in the rule language as to how the ECA rules are to be processed, or a given processing model is assumed by default for all the rules of the ADBMS. To ease our presentation, we assume the execution models by default.

An execution model is tightly related to the coupling modes specifying the timing constraints of the evaluation of the rule's condition and the execution of the rule's action relative to the occurrence of the event that triggers the rule. We consider the following coupling modes (See Section 2.1.4):[5]

1. EC coupling modes:

   Immediate: Evaluate C immediately after the ECA rule is triggered.
   Delayed: Evaluate C at some delayed time point, usually after having performed many other database operations since the time point at which the rule has been triggered.

2. CA coupling modes:

   Immediate: Execute A immediately after C has been evaluated.
   Delayed: Execute A at some delayed time point, usually after having performed many other database operations since the time point at which C has been evaluated.

As we saw in Section 2.1.4, the execution model is also tightly related to the concept of transaction. In fact, the question of determining when to process the different components of an ECA rule is also answered by determining the transactions within which – if any – the C and A components of the ECA rule are evaluated and executed, respectively. In other words, the transaction semantics offer the means for controlling the coupling modes by allowing one the flexibility of processing the rule components in

[5]They were first introduced in the HiPAC system ([HLM88]) and have since been widely used in most ADBMS proposals ([Pat99]). Our presentation is slightly more general than the original one, in which the relationships between coupling modes and execution models, and thoses between transactions and execution models were not conceptually separated.

different, well-chosen transactions. In the sequel, the transaction triggering a rule will be called *triggering transaction* and any other transaction launched by the triggering transaction will be called *triggered transaction*. We assume that all database operations are executed within the boundaries of transactions. From this point of view, we obtain the following refinement for the delayed coupling mode:

1. **Delayed** EC coupling mode: Evaluate C at the end of the triggering transaction T, after having performed all the other database operations of T, but before T's terminal action.

2. **Delayed** CA coupling mode: Execute A at the end of the triggering transaction T, after having performed all the other database operations of T and after having evaluated C, but before T's terminal action.

In presence of flat transactions, we also obtain the following refinement of the immediate coupling mode:

1. **Immediate** EC coupling mode: Evaluate C within the triggering transaction immediately after the ECA rule is triggered.

2. **Immediate** CA coupling mode: Execute A within the triggering transaction immediately after evaluating C.

Notice that the semantics of flat transactions rules out the possibility of nested transactions. For example, we can not process C in a flat transaction and then process A in a further flat transaction, since we quickly encounter the necessity of nesting transactions whenever the execution of a rule triggers further rules. Also, we can not have a delayed CA coupling mode such as: Execute A at the end of the triggering transaction T in a triggered transaction T', after having performed all the other database operations of T, after T's terminal action, and after the evaluation of C. The reason is that, in the absence of nesting of transactions, we will end up with a large set of flat transactions which are independent from each other. This would make it difficult to relate these independent flat transactions as belonging to the processing of a few single rules.

The refinements above yield for each of the EC and CA coupling modes two possibilities: (1) immediate, and (2) delayed. There are exactly 4 combinations of these modes. We will denote these combinations by pairs (i, j) where i and j denote an EC and a CA coupling modes, respectively. For example, $(1, 2)$ is a coupling mode meaning a combination of the immediate EC and delayed CA coupling modes. Moreover, we will call the pairs (i, j) interchangeably coupling modes or execution models. The context will be clear enough to determine what we are writing about. However, we have to consider these combinations with respect to the constraint that we always execute A strictly after C is evaluated.[6] The

[6]This constraint is in fact stricter than a similar constraint found in [HLM88], where it is stated that "A cannot be executed before C is evaluated". The formulation of [HLM88], however, does not rule out simultaneous action executions and condition evaluations, a situation that obviously can lead to disastrous behaviors.

following combinations satisfy this constraint: $(1,1)$, $(1,2)$, and $(2,2)$; the combination $(2,1)$, on the contrary, does not satisfy the constraint.

6.2.2 Immediate Execution Model

Here, we specify the execution model $(1,1)$. This can be formulated as: **Evaluate C immediately after the ECA rule is triggered and execute A immediately after evaluating C within the triggering transaction.**

Suppose we have a set $\mathcal{R}$ of n ECA rules of the form (5.1). Then the following GOLOG procedure captures the immediate execution model $(1,1)$:

$$\begin{array}{l}
\textbf{proc } Rules(t) \\
\quad (\pi \vec{x}_1, \vec{y}_1)[\tau_1[R_1,t]? \ ; \ \zeta_1(\vec{x}_1)[R_1,t]? \ ; \ \alpha_1(\vec{y}_1)[R_1,t]] | \\
\qquad\qquad\qquad \vdots \\
\quad (\pi \vec{x}_n, \vec{y}_n)[\tau_n[R_n,t]? \ ; \ \zeta_n(\vec{x}_n)[R_n,t]? \ ; \ \alpha_n(\vec{y}_n)[R_n,t]] | \\
\quad \neg[(\exists \vec{x}_1)(\tau_1[R_1,t] \wedge \zeta_1(\vec{x})[R_1,t]) \vee \ldots \vee (\exists \vec{x}_n)(\tau_n[R_n,t] \wedge \zeta_n(\vec{x}_n)[R_n,t])] \ ? \\
\textbf{endProc} \ .
\end{array} \tag{6.7}$$

So a rule is a GOLOG program which we can execute using Do. Notice that the procedure (6.7) above formalizes *how* rules are processed using the immediate model examined here: the procedure $Rules(t)$ nondeterministically selects a rule R_i (hence the use of |), tests if an event $\tau_i[R_i,t]$ occurred (hence the use of ?), in which case it immediately tests whether the condition $\zeta_i(\vec{x}_i)[R_i,t]$ holds (hence the use of ;), at which point the action part $\alpha_i(\vec{y}_i)$ is executed. The last test condition of (6.7) permits to exit from the rule procedure when none of the rules is triggered.

6.2.3 Delayed Execution Model

Now, we specify the execution model $(2,2)$ that has both EC and CA coupling being delayed modes. This asks to **evaluate C and execute A at the end of a transaction between the transaction's last action and either its commitment or its failure**. However, notice that the constraint of executing A after C has been evaluated must be enforced.

Let the interval between the end of a transaction (i.e., the situation $do(End(t),s)$, for some s) and its termination (i.e., the situation $do(Commit(t),s)$ or $do(Rollback(t),s)$, for some s) be called *assertion interval*. We use the fluent $assertionInterval(t,s)$ to capture the notion of assertion interval. The following successor state axiom characterizes this fluent:

$$\begin{aligned} assertionInterval(t, do(a,s)) \equiv & \, a = End(t) \vee \\ & assertionInterval(t,s) \wedge \neg termAct(a,t). \end{aligned} \tag{6.8}$$

Now, the following GOLOG procedure captures the delayed execution model $(2, 2)$:

$$
\begin{array}{l}
\textbf{proc } Rules(t) \\
\quad (\pi \vec{x}_1, \vec{y}_1)[\tau_1[R_1, t]? \ ; \ (\zeta_1(\vec{x}_1)[R_1, t] \wedge assertionInterval(t))? \ ; \ \alpha_1(\vec{y}_1)]| \\
\quad \quad \vdots \\
\quad (\pi \vec{x}_n, \vec{y}_n)[\tau_n[R_n, t]? \ ; \ (\zeta_n(\vec{x}_n)[r_n, t] \wedge assertionInterval(t))? \ ; \ \alpha_n(\vec{y}_n)]| \\
\quad \neg\{[(\exists \vec{x}_1)(\tau_1[R_1, t] \wedge \zeta_1(\vec{x})[R_1, t]) \vee \ldots \\
\quad \quad \vee (\exists \vec{x}_n)(\tau_n[R_n, t] \wedge \zeta_n(\vec{x}_n)[R_n, t])] \wedge assertionInterval(t)\} \ ? \\
\textbf{endProc}.
\end{array}
\tag{6.9}
$$

Here, both the C and A components of triggered rules are executed at assertion intervals.

6.2.4 Mixed Execution Model

Here, we specify the execution model $(1, 2)$ that mix both immediate EC and delayed CA coupling modes. This execution model asks to **evaluate C immediately after the ECA rule is triggered and to execute A after evaluating C in the assertion interval.** This model has the semantics

$$
\begin{array}{l}
\textbf{proc } Rules(t) \\
\quad (\pi \vec{x}_1, \vec{y}_1)[\tau_1[R_1, t]? \ ; \ \zeta_1(\vec{x}_1)[R_1, t]? \ ; \ assertionInterval(t)? \ ; \ \alpha_1(\vec{y}_1)]| \\
\quad \quad \vdots \\
\quad (\pi \vec{x}_n, \vec{y}_n)[\tau_n[R_n, t]? \ ; \ \zeta_n(\vec{x}_n)[r_n, t]? \ ; \ assertionInterval(t))? \ ; \ \alpha_n(\vec{y}_n)]| \\
\quad \neg\{[(\exists \vec{x}_1)(\tau_1[R_1, t] \wedge \zeta_1(\vec{x})[R_1, t]) \vee \ldots \\
\quad \quad \vee (\exists \vec{x}_n)(\tau_n[R_n, t] \wedge \zeta_n(\vec{x}_n)[R_n, t])] \wedge assertionInterval(t)\} \ ? \\
\textbf{endProc}.
\end{array}
\tag{6.10}
$$

Here, only the A components of triggered rules are executed at assertion intervals.

Example 6.2 *Consider the stock trading database of Example 5.1, and also the active behavior described there. That is, customer stocks are updated whenever new prices are notified. When a current price of a stock is being updated, its closing price is also updated if the current price notification is the last of the day. Suitable trade actions are initiated whenever some conditions become true of a stock price, under the specific constraint that the customer balance cannot drop below a certain amount. Under the delayed execution model, the two rules shown in Figure 5.1 can be compiled into the rule program shown in Appendix E.*

6.3 Specifying the Execution Models with ONTs

In this section, we specify the execution models of active databases by assuming that the underlying transaction model is that of ONTs.

6.3.1 Classification

We consider the same coupling modes (i.e., immediate and delayed) as in Section 6.2. However, as for the relationship to the concept of transaction, we obtain a different refinement introduced first in HiPAC ([HLM88]) for the delayed coupling mode:

1. Delayed EC coupling modes:

   Deferred: Evaluate C at the end of the triggering transaction T, after having performed all the other database operations of T, but before T's terminal action.
   Partially detached: Evaluate C at the end of the triggering transaction T in a triggered transaction T', after having performed all the other database operations of T, but before T's terminal action.
   Fully detached: Evaluate C at the end of the triggering transaction T in a triggered transaction T', after having performed all the other database operations of T, and after T's terminal action.

2. Delayed CA coupling modes:

   Deferred: Execute A at the end of the triggering transaction T, after having performed all the other database operations of T, but before T's terminal action.
   Partially detached: Execute A at the end of the triggering transaction T in a triggered transaction T', after having performed all the other database operations of T, but before T's terminal action.
   Fully detached: Execute A at the end of the triggering transaction T in a triggered transaction T', after having performed all the other database operations of T, and after T's terminal action.

In presence of transactions, we also obtain the following refinement of the immediate coupling mode:

1. Immediate EC coupling modes:

   Fully immediate: Evaluate C within the triggering transaction immediately after the ECA rule is triggered.
   Partially immediate: Evaluate C within a triggered transaction immediately after the ECA rule is triggered.

2. Immediate CA coupling modes:

   Fully immediate: Execute A within the triggering transaction immediately after evaluating C.
   Partially immediate: Execute A within a triggered transaction immediately after evaluating C.

So the refinements above yield for each of the EC and CA coupling modes five possibilities: (1) fully immediate, (2) partially immediate, (3) deferred, (4) partially detached, and (5) fully detached. There are exactly 5^2 combinations of these modes. As in Section 6.2, we will denote these combinations by pairs (i, j) where i and j denote an EC and a CA coupling modes, respectively. We continue to consider these combinations with respect to the constraint that we always execute A strictly after C has been evaluated. The following combinations satisfy this constraint:

$$(1,1), (1,2), (1,3), (1,4), (1,5), (2,1), (2,2), (2,3), (2,4), (2,5),$$
$$(3,1), (3,2), (3,3), (3,4), (3,5), (4,1), (4,2), (4,3), (4,4), (4,5), (5,5).$$

Some of these combinations are not yet fine-grained enough. Take, for example, $(5,5)$. Depending on whether the triggered transactions for the C and A components are the same or not, we get two different coupling modes $(5,5)_1$ and $(5,5)_2$, respectively.

6.3.2 Immediate Execution Models

Here, we specify those execution models (i, j) that have both i and j being immediate coupling modes. These are $(1,1)$, $(1,2)$, $(2,1)$, and $(2,2)$.

$(1,1)$**: Evaluate C and execute A immediately after the ECA rule is triggered within the triggering transaction.** Suppose we have a set $\mathcal{R}$ of n ECA rules of the form (5.1). Then the GOLOG procedure that captures the immediate execution model $(1,1)$ is the one given in (6.7).

$(1,2)$**: Evaluate C within the triggering transaction and execute A in a triggered transaction immediately after evaluating C.** The following GOLOG procedure captures the immediate execution model $(1,2)$:

$$
\begin{array}{l}
\textbf{proc}\ Rules(t) \\
\quad (\pi \vec{x}_1, \vec{y}_1, t')[\tau_1[R_1,t]?\ ; \zeta_1(\vec{x}_1)[R_1,t]?\ ;\ t' = setId?\ ; \\
\quad\quad Begin(t,t',CLOSED,NONCOMP)\ ;\ \alpha_1(\vec{y}_1)[R_1,t']\ ;\ End(t')]\ | \\
\quad\quad\quad\quad \vdots \\
\quad (\pi \vec{x}_n, \vec{y}_n, t')[\tau_n[R_n,t]?\ ; \zeta_n(\vec{x}_n)[R_n,t]?\ ;\ t' = setId?\ ; \\
\quad\quad Begin(t,t',CLOSED,NONCOMP)\ ;\ \alpha_n(\vec{y}_n)[R_n,t']\ ;\ End(t')]\ | \\
\quad \neg[(\exists \vec{x}_1)(\tau_1[R_1,t] \wedge \zeta_1(\vec{x})[R_1,t]) \vee \ldots \vee (\exists \vec{x}_n)(\tau_n[R_n,t] \wedge \zeta_n(\vec{x}_n)[R_n,t])]\ ? \\
\textbf{endProc}.
\end{array}
\tag{6.11}
$$

Here, $setId(s)$ is a system fluent whose intuitive meaning is to set a fresh identity number for a subtransaction to spawn. To characterize $setId(s)$, we introduce a relational system fluent $numOfTrans$ that is used to record the number of transactions so far in the log. The successor state axiom for $numOfTrans$ is given as follows:

$$\begin{aligned} & numOfTrans(n, do(a,s)) \equiv \\ & \quad (\exists n', t, t', m, c)[numOfTrans(n', s) \wedge a = Begin(t, t', m, c) \wedge n = n' + 1 \vee \\ & \qquad numOfTrans(n', s) \wedge a = Begin(t', t, m, c) \wedge n = n' + 1] \vee \\ & \quad numOfTrans(n, s). \end{aligned} \tag{6.12}$$

This axiom says that the number of transactions is obtained by counting the $Begin(t, t', m, c)$ actions in the log. Now $setId(s)$ is introduced as an abbreviation:

$$setId(s) = id =_{df} (\exists n).numOfTrans(n, s) \wedge id = n + 1. \tag{6.13}$$

So, we assign a fresh identity number to a transaction by monotonically increasing by one the identity numbers already assigned so far.

$(2, 1)$**: Evaluate C within a triggered transaction and execute A in the triggering transaction immediately after evaluating C.** This semantics is captured by the following GOLOG procedure:

$$\begin{aligned} & \textbf{proc } Rules(t) \\ & \quad (\pi \vec{x}_1, \vec{y}_1, t')[\tau_1[R_1, t]? \ ; \ t' = setId? \ ; \\ & \qquad Begin(t, t', CLOSED, NONCOMP) \ ; \zeta_1(\vec{x}_1)[R_1, t']? \ ; \ End(t') \ ; \ \alpha_1(\vec{y}_1)[R_1, t]] \mid \\ & \qquad\qquad \vdots \\ & \quad (\pi \vec{x}_n, \vec{y}_n, t')[\tau_n[R_n, t]? \ ; \ t' = setId? \ ; \\ & \qquad Begin(t, t', CLOSED, NONCOMP) \ ; \zeta_n(\vec{x}_n)[R_n, t']? \ ; \ End(t') \ \alpha_n(\vec{y}_n)[R_n, t]] \mid \\ & \quad \neg[(\exists \vec{x}_1)(\tau_1[R_1, t] \wedge \zeta_1(\vec{x})[R_1, t]) \vee \ldots \vee (\exists \vec{x}_n)(\tau_n[R_n, t] \wedge \zeta_n(\vec{x}_n)[R_n, t])] \ ? \\ & \textbf{endProc}. \end{aligned} \tag{6.14}$$

$(2, 2)$**: Evaluate C within a triggered transaction and execute A in a triggered transaction immediately after evaluating C.** Depending on whether the transactions for the C and A components are the same or not, we get the coupling models $(2,2)_1$ and $(2,2)_2$ whose semantics is captured as follows

```
proc $Rules(t)$
$(\pi \vec{x}_1, \vec{y}_1, t', t'')[\tau_1[R_1, t]? ;$
  $t' = setId? ; \ Begin(t, t', CLOSED, NONCOMP) ; \zeta_1(\vec{x}_1)[R_1, t']? ; \ End(t') ;$
  $t'' = setId? ; \ Begin(t, t'', CLOSED, NONCOMP) ; \zeta_1(\vec{x}_1)[R_1, t'']? ; \ \alpha_1(\vec{y}_1)[R_1, t''] ; \ End(t'')] |$
                    $\vdots$
$(\pi \vec{x}_n, \vec{y}_n, t', t'')[\tau_n[R_n, t]? ;$                                          (6.15)
  $t' = setId? ; \ Begin(t, t', CLOSED, NONCOMP) ; \zeta_n(\vec{x}_1)[R_n, t']? ; \ End(t') ;$
  $t'' = setId? ; \ Begin(t, t'', CLOSED, NONCOMP) ; \zeta_n(\vec{x}_1)[R_n, t'']? ; \ \alpha_n(\vec{y}_n)[R_n, t''] ; \ End(t'')] |$
$\neg[(\exists \vec{x}_1)(\tau_1[R_1, t] \wedge \zeta_1(\vec{x})[R_1, t]) \vee \ldots \vee (\exists \vec{x}_n)(\tau_n[R_n, t] \wedge \zeta_n(\vec{x}_n)[R_n, t])] ?$
endProc
```

for the mode $(2,2)_1$, and as

```
proc $Rules(t)$
    $(\pi \vec{x}_1, \vec{y}_1, t')[\tau_1[R_1, t]? ; \ t' = setId? ;$
      $Begin(t, t', CLOSED, NONCOMP) ; \zeta_1(\vec{x}_1)[R_1, t']? ; \ \alpha_1(\vec{y}_1) ; \ End(t')] |$
                    $\vdots$
    $(\pi \vec{x}_n, \vec{y}_n, t')[\tau_n[R_n, t]? ; \ t' = setId? ;$                           (6.16)
      $Begin(t, t', CLOSED, NONCOMP) ; \zeta_n(\vec{x}_n)[R_n, t']? ; \ \alpha_n(\vec{y}_n)[R_n, t'] ; \ End(t')] |$
    $\neg[(\exists \vec{x}_1)(\tau_1[R_1, t] \wedge \zeta_1(\vec{x})[R_1, t]) \vee \ldots \vee (\exists \vec{x}_n)(\tau_n[R_n, t] \wedge \zeta_n(\vec{x}_n)[R_n, t])] ?$
endProc
```

for the mode $(2,2)_2$.

6.3.3 Deferred Execution Model

Now, we specify the execution models (i,j) that have both i and j being deferred coupling modes. There is just one such mode, namely $(3,3)$. This asks to **evaluate C and execute A at the end of a transaction between the transaction's last action and its commitment**.

The deferred execution model is captured by the following procedure:

$$\begin{array}{l}
\textbf{proc } Rules(t) \\
\quad (\pi \vec{x}_1, \vec{y}_1)[\tau_1[R_1,t]? \ ; \ (\zeta_1(\vec{x}_1)[R_1,t] \wedge assertionInterval(t))? \ ; \ \alpha_1(\vec{y}_1)]| \\
\qquad \vdots \\
\quad (\pi \vec{x}_n, \vec{y}_n)[\tau_n[R_n,t]? \ ; \ (\zeta_n(\vec{x}_n)[r_n,t] \wedge assertionInterval(t))? \ ; \ \alpha_n(\vec{y}_n)]| \\
\quad \neg\{[(\exists \vec{x}_1)(\tau_1[R_1,t] \wedge \zeta_1(\vec{x})[R_1,t]) \vee \ldots \\
\qquad \vee (\exists \vec{x}_n)(\tau_n[R_n,t] \wedge \zeta_n(\vec{x}_n)[R_n,t])] \wedge assertionInterval(t)\} \ ? \\
\textbf{endProc}.
\end{array} \tag{6.17}$$

Here, the fluent $assertionInterval(t, s)$ is as introduced in (6.8).

6.3.4 Detached Execution Models

Here, we specify those execution models (i, j) that have both i and j being detached coupling modes. These are $(4, 4)$, $(4, 5)$, and $(5, 5)$.

$(4, 4)$**: Evaluate C in a triggered transaction nested in the assertion interval of the triggering transaction and execute A in a triggered transaction nested in the same assertion interval.** Again, depending on whether the transactions for the C and A components are the same or not, we get the coupling models $(4, 4)_1$ and $(4, 4)_2$ whose semantics is captured as follows. The model $(4, 4)_1$ has the semantics

$$\begin{array}{l}
\textbf{proc } Rules(t) \\
\quad (\pi \vec{x}_1, \vec{y}_1, t')[\tau_1[R_1,t]? \ ; \ assertionInterval(t)? \ ; \ t' = setId? \ ; \\
\qquad Begin(t, t', OPEN, NONCOMP) \ ; \zeta_1(\vec{x}_1)[R_1,t']? \ ; \ \alpha_1(\vec{y}_1)[R_1,t'] \ ; \ End(t')] \ | \\
\qquad \vdots \\
\quad (\pi \vec{x}_n, \vec{y}_n, t')[\tau_n[R_n,t]? \ ; \ assertionInterval(t)? \ ; \ t' = setId? \ ; \\
\qquad Begin(t, t', OPEN, NONCOMP) \ ; \zeta_n(\vec{x}_n)[R_n,t']? \ ; \ \alpha_n(\vec{y}_n)[R_n,t'] \ ; \ End(t')] \ | \\
\quad \neg\{[(\exists \vec{x}_1)(\tau_1[R_1,t] \wedge \zeta_1(\vec{x})[R_1,t]) \vee \ldots \\
\qquad \vee (\exists \vec{x}_n)(\tau_n[R_n,t] \wedge \zeta_n(\vec{x}_n)[R_n,t])] \wedge assertionInterval(t)\} \ ? \\
\textbf{endProc}.
\end{array} \tag{6.18}$$

Here, triggered rules are executed at assertion intervals as open and noncompensatable transactions whose semantics has been developed in Section 4.3.3.

The model $(4,4)_2$ has the semantics

$$
\begin{array}{l}
\textbf{proc } Rules(t) \\
(\pi \vec{x}_1, \vec{y}_1, t', t'')[\tau_1[R_1,t]? \ ; \ assertionInterval(t)? \ ; \\
\quad t' = setId? \ ; \ Begin(t,t',CLOSED,NONCOMP) \ ; \zeta_1(\vec{x}_1)[R_1,t']? \ ; \ End(t') \ ; \\
\quad t'' = setId? \ ; \ Begin(t,t'',CLOSED,NONCOMP) \ ; \zeta_1(\vec{x}_1)[R_1,t'']? \ ; \ \alpha_1(\vec{y}_1)[R_1,t''] \ ; \ End(t'')]| \\
\qquad\qquad\qquad\qquad \vdots \\
(\pi \vec{x}_n, \vec{y}_n, t', t'')[\tau_n[R_n,t]? \ ; \ assertionInterval(t)? \ ; \\
\quad t' = setId? \ ; \ Begin(t,t',CLOSED,NONCOMP) \ ; \zeta_n(\vec{x}_1)[R_n,t']? \ ; \ End(t') \ ; \\
\quad t'' = setId? \ ; \ Begin(t,t'',CLOSED,NONCOMP) \ ; \zeta_n(\vec{x}_1)[R_n,t'']? \ ; \ \alpha_n(\vec{y}_n)[R_n,t''] \ ; \ End(t'')]| \\
\neg\{[(\exists \vec{x}_1)(\tau_1[R_1,t] \wedge \zeta_1(\vec{x})[R_1,t]) \vee \ldots \\
\qquad\quad \vee \ (\exists \vec{x}_n)(\tau_n[R_n,t] \wedge \zeta_n(\vec{x}_n)[R_n,t])] \wedge assertionInterval(t)\} \ ? \\
\textbf{endProc}.
\end{array}
\tag{6.19}
$$

$(4,5)$**: Evaluate C in a triggered transaction nested in the assertion interval of the triggering transaction and execute A in a triggered transaction spawned after the same assertion interval.**

$$
\begin{array}{l}
\textbf{proc } Rules(t) \\
(\pi \vec{x}_1, \vec{y}_1, t', t'')[\tau_1[R_1,t]? \ ; \ assertionInterval(t)? \ ; \\
\quad t' = setId? \ ; \ Begin(t,t',CLOSED,NONCOMP) \ ; \zeta_1(\vec{x}_1)[R_1,t']? \ ; \ End(t') \ ; \\
\quad \neg assertionInterval(t)? \ ; \\
\quad t'' = setId? \ ; \ Begin(t,t'',CLOSED,NONCOMP) \ ; \zeta_1(\vec{x}_1)[R_1,t'']? \ ; \ \alpha_1(\vec{y}_1)[R_1,t''] \ ; \ End(t'')]| \\
\qquad\qquad\qquad\qquad \vdots \\
(\pi \vec{x}_n, \vec{y}_n, t', t'')[\tau_n[R_n,t]? \ ; \ assertionInterval(t)? \ ; \\
\quad t' = setId? \ ; \ Begin(t,t',CLOSED,NONCOMP) \ ; \zeta_n(\vec{x}_1)[R_n,t']? \ ; \ End(t') \ ; \\
\quad \neg assertionInterval(t)? \ ; \\
\quad t'' = setId? \ ; \ Begin(t,t'',CLOSED,NONCOMP) \ ; \zeta_n(\vec{x}_1)[R_n,t'']? \ ; \ \alpha_n(\vec{y}_n)[R_n,t''] \ ; \ End(t'')]| \\
\neg\{[(\exists \vec{x}_1)(\tau_1[R_1,t] \wedge \zeta_1(\vec{x})[R_1,t]) \vee \ldots \vee \ (\exists \vec{x}_n)(\tau_n[R_n,t] \wedge \zeta_n(\vec{x}_n)[R_n,t])] \wedge assertionInterval(t)\} \ ? \\
\textbf{endProc}.
\end{array}
\tag{6.20}
$$

$(5,5)$**: Evaluate C in a triggered transaction spawned after the assertion interval of the triggering transaction and execute A in a triggered transaction spawned after the same assertion interval.**

The model $(5,5)_1$ has the semantics

$$\begin{array}{l}
\textbf{proc } Rules(t) \\
\quad (\pi \vec{x}_1, \vec{y}_1, t')[\tau_1[R_1,t]? \; ; \; \neg assertionInterval(t)? \; ; \; t' = setId? \; ; \\
\qquad Begin(t,t',OPEN,NONCOMP) \; ; \zeta_1(\vec{x}_1)[R_1,t']? \; ; \; \alpha_1(\vec{y}_1)[R_1,t'] \; ; \; End(t')] \mid \\
\qquad\qquad\qquad \vdots \\
\quad (\pi \vec{x}_n, \vec{y}_n, t')[\tau_n[R_n,t]? \; ; \; \neg assertionInterval(t)? \; ; \; t' = setId? \; ; \\
\qquad Begin(t,t',OPEN,NONCOMP) \; ; \zeta_n(\vec{x}_n)[R_n,t']? \; ; \; \alpha_n(\vec{y}_n)[R_n,t'] \; ; \; End(t')] \mid \\
\quad \neg\{[(\exists \vec{x}_1)(\tau_1[R_1,t] \wedge \zeta_1(\vec{x})[R_1,t]) \vee \ldots \vee (\exists \vec{x}_n)(\tau_n[R_n,t] \wedge \zeta_n(\vec{x}_n)[R_n,t])] \wedge assertionInterval(t)\} \; ? \\
\textbf{endProc}.
\end{array} \tag{6.21}$$

The model $(5,5)_2$ has the semantics

$$\begin{array}{l}
\textbf{proc } Rules(t) \\
(\pi \vec{x}_1, \vec{y}_1, t', t'')[\tau_1[R_1,t]? \; ; \; \neg assertionInterval(t)? \; ; \\
\; t' = setId? \; ; \; Begin(t,t',CLOSED,NONCOMP) \; ; \zeta_1(\vec{x}_1)[R_1,t']? \; ; \; End(t') \; ; \\
\neg assertionInterval(t)? \; ; \\
\; t'' = setId? \; ; \; Begin(t,t'',CLOSED,NONCOMP) \; ; \zeta_1(\vec{x}_1)[R_1,t'']? \; ; \; \alpha_1(\vec{y}_1)[R_1,t''] \; ; \; End(t'')] \mid \\
\qquad\qquad\qquad \vdots \\
(\pi \vec{x}_n, \vec{y}_n, t', t'')[\tau_n[R_n,t]? \; ; \; \neg assertionInterval(t)? \; ; \\
\; t' = setId? \; ; \; Begin(t,t',CLOSED,NONCOMP) \; ; \zeta_n(\vec{x}_1)[R_n,t']? \; ; \; End(t') \; ; \\
\neg assertionInterval(t)? \; ; \\
\; t'' = setId? \; ; \; Begin(t,t'',CLOSED,NONCOMP) \; ; \zeta_n(\vec{x}_1)[R_n,t'']? \; ; \; \alpha_n(\vec{y}_n)[R_n,t''] \; ; \; End(t'')] \mid \\
\neg\{[(\exists \vec{x}_1)(\tau_1[R_1,t] \wedge \zeta_1(\vec{x})[R_1,t]) \vee \ldots \vee (\exists \vec{x}_n)(\tau_n[R_n,t] \wedge \zeta_n(\vec{x}_n)[R_n,t])] \wedge assertionInterval(t)\} \; ? \\
\textbf{endProc}.
\end{array} \tag{6.22}$$

6.4 Semantics of Rule Programs

6.4.1 Abstract Execution of Rule Programs

Given the program $Rules(t)$ specified as in (6.7)–(6.22), we can now complete the logical characterization of the execution models by showing how the predicate $Trans(\delta, s, \delta', s')$ of [DGLL97] must be modified to handle primitive actions:

$$\begin{array}{l}
Trans(a,s,a',s') \equiv (\exists a^*, s'', s^*, t).transOf(a,t,s) \wedge Poss(a,s) \wedge a' = nil \wedge \\
\quad \{[s'' = do(a,s) \wedge systemAct(a^*,t) \wedge Poss(a^*,s'') \wedge s' = do(a^*,s'')] \vee \\
\quad\; [s^* = do(a,s) \wedge [(\forall a'', t')systemAct(a'',t') \supset \neg Poss(a',s^*) \wedge Do(Rules(t), s^*, s')]]\}.
\end{array} \tag{6.23}$$

With the last conjunct, we interleave the execution of each action with the execution of $Rules(t)$. The *while*-loop picks one of the triggered rules, according to (6.7)–(6.22), executes it, and comes back at the beginning of $Rules(t)$; it does so until the last test condition of (6.7)–(6.22) becomes true; the semantics (6.7)–(6.22) will make sure that rule execution follows the appropriate execution model.[7]

We execute a GOLOG program T embodying an active behavior by performing the theorem proving task of establishing entailments of the form (6.4), where $\mathcal{D}$ is now the active relational theory for an appropriate transaction model.

Using the notion of well formed ConGolog programs introduced in Definitions 6.3 and 6.4, together with the notion of legal database log defined in (4.20), we can show the following:

Theorem 6.11 *Suppose $\mathcal{D}$ is an active relational theory for ONTs, and let T be a well formed ConGolog program for ONTs. Then,*

$$\mathcal{D} \models (\forall s).Do(T, S_0, s) \supset legal(s), \tag{6.24}$$

and, more generally,

$$\mathcal{D} \models (\forall s, s').legal(s) \supset [Do(T, s, s') \supset legal(s')]. \tag{6.25}$$

6.4.2 Classification Theorems for Execution Models: the Flat Transactions Case

There is a natural question which arises with respect to the different execution models whose semantics have been given above: is it possible to reduce the set of execution models described above to a handful of classes by virtue of some equivalence mechanism? To answer this question, we must develop a (logical) notion of equivalence between two given execution models. Suppose that we are given two programs $Rules^{(i,j)}(t)$ and $Rules^{(k,l)}(t)$ corresponding to the execution models (i, j) and (k, l), respectively.

Definition 6.12 (Database versus system queries) *Suppose Q is a situation calculus query. Then Q is a* database query *iff the only fluents it mentions are database fluents. A* system query *is one that mentions at least one system fluent.*

Intuitively, establishing an equivalence between the programs $Rules^{(i,j)}(t)$ and $Rules^{(k,l)}(t)$ with respect to a background active relational theory $\mathcal{D}$ amounts to establishing that, for all database queries $Q(s)$ and transactions t, whenever the answer to $Q(s)$ is "yes" in a situation resulting from the execution of $Rules^{(i,j)}(t)$ in S_0, executing $Rules^{(k,l)}(t)$ in S_0 results in a situation yielding "yes" to $Q(s)$.

Definition 6.13 (Implication of Execution Models) *Suppose $\mathcal{D}$ is an active relational theory, and let $Rules^{(i,j)}(t)$ and $Rules^{(k,l)}(t)$ be ConGolog programs corresponding to the execution models (i, j) and*

[7]Notice that this semantics means that transitions may in fact be big leaps involving many actions. This may prevent some desirable concurrency. We leave this problem out of the scope of this document.

(k,l)*, respectively. Moreover, suppose that for all database queries* Q*, we have*[8]

$$(\forall s, s', s'', t).Do(Rules^{(m,n)}(t), s, s') \wedge Do(Rules^{(m,n)}(t), s, s'') \supset Q[s'] \equiv Q[s''],$$

where (m,n) *is* (i,j) *or* (k,l)*. Then a rule program* $Rules^{(i,j)}(t)$ *implies another rule program* $Rules^{(k,l)}(t)$ *(*$Rules^{(i,j)}(t) \Longrightarrow Rules^{(k,l)}(t)$*) iff, for every database query* Q*,*

$$\begin{aligned}(\forall t, s)\{[(\exists s').Do(Rules^{(i,j)}(t), s, s') \wedge Q[s']] \supset \\ [(\exists s'').Do(Rules^{(k,l)}(t), s, s'') \wedge Q[s'']]\}.\end{aligned} \tag{6.26}$$

Definition 6.14 (Equivalence of execution models) *Assume the conditions and notations of Definition 6.13. Then* $Rules^{(i,j)}(t)$ *and* $Rules^{(k,l)}(t)$ *are equivalent (*$Rules^{(i,j)}(t) \cong Rules^{(k,l)}(t)$*) iff, for every database query* Q*,*

$$(\forall t, s)\{[(\exists s').Do(Rules^{(i,j)}(t), s, s') \wedge Q[s']] \equiv [(\exists s'').Do(Rules^{(k,l)}(t), s, s'') \wedge Q[s''])]\}.$$

It is important to see why we restrict our attention to database queries. We do so since we are interested in the final state of the content of the database, regardless of the final values of the system fluents. Consider the execution models $(1,1)$ and $(1,2)$ and suppose that we use the Do predicate to execute a well formed program T assuming the rules in Figure 5.1. Assume that the classic flat transaction model is used for the transaction program T. Therefore the execution model $(1,1)$ will involve no system fluent since the flat transaction model does not involve any. On the contrary, the execution model $(1,2)$ will involve a system fluent if the triggered transaction associated with this execution model involves any system fluent. In general, different execution models are most likely to involve different system fluents so that, from the point of view of these, virtually no two execution models would be equivalent. Fortunately, the content of the database is usually what matters to the user, not the internal state of the system which may be considered as a black box. This justifies restricting our attention to database queries in establishing the relationships among execution models.

Theorem 6.15 *Assume the conditions of Definition 6.13. Then* $Rules^{(2,2)}(t) \Longrightarrow Rules^{(1,1)}(t)$.

Theorem 6.16 *Assume the conditions of Definition 6.13. Then* $Rules^{(1,2)}(t) \cong Rules^{(2,2)}(t)$.

Corollary 6.17 *Assume the conditions of Definition 6.13. Then* $Rules^{(1,2)}(t) \Longrightarrow Rules^{(1,1)}(t)$.

6.4.3 Classification Theorems for Execution Models: the ONTs Case

In the case of ONTs, the execution model $(1,1)$ seems to be the one that is the most natural and simplest one; it is the same as for the case of flat transactions and is supported in virtually all existing ADBMSs.

[8]We will come back to this sentence later in Section 6.6. The sentence expresses the so-called confluence property of active rules.

Thus it also appears natural to consider $(1,1)$ as the basic execution model with respect to which one may wonder whether, whenever any given execution model (i,j) leads to some final situation s' then, if $(1,1)$ also leads to some final situation s'', the database quesries true in s' are also true in s''; or whether $(1,1)$ and (i,j), whenever they terminate, do so in situations in which exactly the same sentences are true. This motivates the following two definitions.

Definition 6.18 (Correctness of execution models) *Assume the conditions of Definition 6.13. Then a rule program $Rules^{(i,j)}(t)$ is correct iff $Rules^{(i,j)}(t) \Longrightarrow Rules^{(1,1)}(t)$.*

Definition 6.19 (Completeness of execution models) *Assume the conditions of Definition 6.13. Then a rule program $Rules^{(i,j)}(t)$ is complete iff $Rules^{(i,j)}(t) \cong Rules^{(1,1)}(t)$.*

There are also cases where the roles of $(1,1)$ and (i,j) in the implication (6.26) are reversed:

Definition 6.20 (Hypercompleteness of execution models) *Assume the conditions of Definition 6.13. Then a rule program $Rules^{(i,j)}(t)$ is hypercomplete iff $Rules^{(1,1)}(t) \Longrightarrow Rules^{(i,j)}(t)$.*

Theorem 6.21 *All the immediate, deferred, and detached execution models considered above are correct.*

Conjecture 6.22 *No execution model is either complete or hypercomplete.*

The theorems above can be considered as positive classification results. They are just a few examples of many other positive classification results that can be obtained when comparing execution models. However, in practice, it is equally important to know which execution models cannot achieve what we can achieve with the basic immediate model. In this sense, it is important to be able to validate or invalidate the conjecture above. We believe that this conjecture is true, as none of the execution models that we have considered is either complete, or hypercomplete. The reason is that any nested transaction requires a situation in which the integrity constraints are satisfied in order to commit. However, whenever one obtains a log following the execution of the rule program $Rule^{(1,1)}(t)$, that situation does not give any guarantee whatsoever that its sublogs satisfy the integrity constraints; therefore, there is no guarantee that the log generated by the execution of $Rule^{(1,1)}(t)$ would allow any insertion of internal actions.

6.5 Priorities

Rules are assigned priorities by the programmer who provides an explicit (partial) order among rules. Given a set $\mathcal{R} \neq \emptyset$ of rules, this amounts to partitioning the set $\mathcal{R}$ into subsets $\mathcal{R}_i$, $1 \leq i \leq k$, such that rules in $\mathcal{R}_i$ all have equal priority, and rules in $\mathcal{R}_i$ have priority higher than the rules in $\mathcal{R}_j$, for all j such that $i < j$. As an example, we partition the set of rules of Example 5.1 into two subsets; the first subset contains the rule $Update\_stocks$ and the second one contains $Buy\_100shares$.

Suppose that the set $\mathcal{R}$ of rules has been partitioned into subsets $\mathcal{R}_i = \{r_{i_1}, \ldots, r_{i_{l_i}}\}$, $1 \leq i \leq k$, in the fashion explained above. Then the procedure below represents the set $\mathcal{R}$ of rules with priorities:

```
**proc** $Rules(t)$
    $r_{1_1} | r_{1_2} | \ldots | r_{1_{l_1}} |$
    $\{\neg[(\tau_{r_{1_1}} \wedge (\exists\vec{x})\zeta_{r_{1_1}}(\vec{x})) \vee \ldots \vee (\tau_{r_{1_{l_1}}} \wedge (\exists\vec{x})\zeta_{r_{1_{l_1}}}(\vec{x}))]?\ ;$
        $[r_{2_1} | r_{2_2} | \ldots | r_{2_{l_2}} |$
        $(\neg[(\tau_{r_{2_1}} \wedge (\exists\vec{x})\zeta_{r_{2_1}}(\vec{x})) \vee \ldots \vee (\tau_{r_{2_{l_2}}} \wedge (\exists\vec{x})\zeta_{r_{2_{l_2}}}(\vec{x}))]?\ ;\ Rules_{rest})]\}$
**endProc**,
```

where $\tau_{r_{1_j}}$ and $\zeta_{r_{1_j}}(\vec{x})$ denote the event and the condition parts of rule r_{1_j}, which is the j-th rule of the subset $\mathcal{R}_1$ of $\mathcal{R}$, respectively; $Rules_{rest}$ is a GOLOG program representing the remaining rules and their priorities in $\mathcal{R}_3 \cup \ldots \cup \mathcal{R}_k$; and $Rules_{rest}$ iterates the construction in the body of the procedure $Rules(t)$.

Rules within a subset $\mathcal{R}_i$ are selected nondeterministically until one is found, at which point their action parts are executed according to the semantics expressed in (6.7)–(6.22). The test action at the end of $\mathcal{R}_i$ means that if no triggered rule of $\mathcal{R}_i$ has a true condition, the rule processing stops for that subset and continues with rules of lower priorities. Notice that if the processing of rules in $\mathcal{R}_i$ leads to the triggering of one of the rules r_{j_k}, such that $j > i$, control goes back to the rules of higher priority.

Example 6.3 *Using the immediate execution model, the procedure for the prioritized rules in Example 5.1 is given in Figure 6.1: Upon the signaling of a $price\_inserted$ event, rule $Update\_stocks$ updates the stock price in the database for some customer if this stock is being monitored. Rule Buy_100_shares is also triggered by the same $price\_inserted$ event. If the price of some monitored stock has been updated and the new price lies between some threshold, then a suitable trading action is performed. However, $Update\_stocks$ has priority over Buy_100_shares and will be processed first. In this example, instead of using the notation $\phi[\vec{y}]$, we have restored the arguments $\vec{y}$ involved, in this case the arguments $Update\_stocks$, Buy_100_shares, and t (for "transaction").* ■

6.6 Properties of Rule Programs

Usually, even a relatively small number of ECA rules can display a complex and unpredictable run-time behavior, such as non-termination, and discrepancies in the final states of the database depending on how rules are selected for execution. Therefore, in designing ECA rules, it is important to be able to predict run-time behavior of rule at design-time. This is done by analyzing the set of ECA rules. Rule analysis

```
**proc** $Rules(t)$
$(\pi\ s\_id, pr', clos\_pr)[price\_inserted(Update\_stocks, t)?\ ;$
$(\exists\ c, time, bal, pr')[price\_inserted(Update\_stocks, s\_id, pr, time, t) \wedge$
  $customer(c, bal, s\_id, t) \wedge stock(s\_id, pr', clos\_pr, t)]?\ ;$
  $stock\_insert(s\_id, pr, clos\_pr, t)] \quad |$
  $\{\neg[price\_inserted(Update\_stocks, t) \wedge$
   $(\exists\ s\_id, pr, c, time, bal, pr', clos\_pr)[price\_inserted(s\_id, pr, time, t) \wedge$
   $customer(c, bal, s\_id, t) \wedge stock(s\_id, pr', clos\_pr, t)]]?\ ;$
   $(\pi\ c, s\_id, 100)[price\_inserted(Buy\_100\_shares, t)?\ ;$
   $(\exists\ new\_pr, time, bal, pr, clos\_pr)[price\_inserted(Buy\_100\_shares, s\_id, new\_pr, time, t) \wedge$
    $customer(c, bal, s\_id, t) \wedge stock(s\_id, pr, clos\_pr, t) \wedge new\_pr < 50 \wedge clos\_pr > 70]?\ ;$
    $buy(c, s\_id, 100, t)] \quad |$
    $\{\neg[price\_inserted(Buy\_100\_shares, t) \wedge (\exists\ c, s\_id, new\_pr, time, bal, pr, clos\_pr)$
     $[price\_inserted(Buy\_100\_shares, s\_id, new\_pr, time, t) \wedge customer(c, bal, s\_id, t) \wedge$
     $stock(s\_id, pr, clos\_pr, t) \wedge new\_pr < 50 \wedge clos\_pr > 70]]? \quad \}\}$
**endProc** .
```

Figure 6.1: Prioritized rules for updating stocks and buying shares

is a design-time inspection of rules for compliance with a set of desired properties. The most important properties rule designers must care about are ([WC96]):

- *Termination*: This ensures that a database state is reached in which no further rules are triggered.
- *Confluence*: This ensures that whenever a rule program reaches two final database states, then the two states are the same, independently of the order of the execution of non-prioritized rules.
- *Observable determinism*: This ensures that a rule program always performs the same visible actions, independently of the execution order of non-prioritized rules.

This section shows how the first two of these properties can be expressed in the situation calculus. The last one is left out. The general treatment of these properties deserves a full thesis of its own. Here, we just show how to formulate the properties in the situation calculus without reasoning about them.

6.6.1 General Properties

Here, we briefly illustrate the use of our framework for specifying properties of well formed ConGolog programs. We appeal to a well known hierarchy of properties expressible in temporal logic of Manna and Pnueli ([MP91]) who distinguish two classes of properties: *Safety*, and *Progress*. In the situation calculus, a safety property is syntactically characterized by a formula of the form $(\forall \vec{s})\phi$, where ϕ is any first order past temporal formula expressed in the situation calculus. A progress property is syntactically characterized by a formula of the form $(Q_1 s_1) \cdots (Q_n s_n)\phi$, where ϕ is any first order past temporal formula expressed in the situation calculus, and the Q_i must contain at least one occurrence of $\exists$.

A classical example of safety property is the *partial correctness* of a given program T: if T terminates, then it does so in a situation satisfying a desirable property, say ϕ; i.e.

$$(\forall s).Do(T, S_0, s) \supset \phi(s).$$

Notice that ATM systems are designed to terminate. So they constitute a domain where this kind of property can be considered.

Checking partial correctness amounts to establishing the entailment

$$\mathcal{D} \models (\forall s).Do(T, S_0, s) \supset \phi(s). \tag{6.27}$$

As an illustration of partial correctness checking, we have the entailment (G.16) from Theorem 6.10. Another illustration is

$$\mathcal{D} \models (\forall s).Do(T, S_0, s) \supset always(\phi, s),$$

where T is a given well formed ConGolog program, and ϕ is any first order past temporal formula expressed in the situation calculus . Suppose we have the following abbreviation for $well\text{–}formed\text{–}sit(s)$:

$$well\text{–}formed\text{–}sit(s) =_{df} (\forall s')[do(Commit(t), s') \sqsubset s \supset \neg(\exists s'')do(Rollback(t), s'') \sqsubset s].$$

Then the following is a further example of checking correctness:

$$\mathcal{D} \models (\forall s).Do(T, S_0, s) \supset well\text{–}formed\text{–}sit(s),$$

with T being a given well formed ConGolog program.

Given a relational theory $\mathcal{D}$, a property $\phi(s)$, and a well formed ConGolog program T, it is important to look for computationaly feasible ways of establishing the entailment (6.27). The answer to this question would take us too far aside; nevertheless, we can mention one possible mechanism for doing this: in order to check the property $\phi(s)$, we must answer a historical query, i.e. check the validity of $\phi(s)$ over a finite log starting in S_0 and involving only ground operations. We do this by first generating a ground log S such that $Do(T, S_0, S)$ using the relational theory $\mathcal{D}$. Then we use the regression mechanism for non-Markovian situation calculus theories defined in [Gab00] to reduce our task to establishing an entailment involving only the initial database. Regressing the given formula $\phi(s)$ means using $\mathcal{D}$ to transform it into a logically equivalent formula $\phi'(s')$ which mentions a shorter sublog s' of s. Repeatedly performing this mechanism leads ultimately to a formula whose only log is S_0. So, thanks to the regression, checking a property in the log resulting from the execution of a transaction amounts to checking that property – as a theorem proving task – in the initial database.

6.6.2 Termination and Correctness

Recall that termination ensures that a rule program reaches a database state in which no further rules are triggered. In the situation calculus, this can be expressed by writing a formula that captures the fact that a rule program $Rules^(i,j)(t)$ reaches a final situation with the Do predicate. Formally, suppose $\mathcal{D}$ is an active relational theory, and let $Rules^{(i,j)}(t)$ be the ConGolog program corresponding to the execution model (i,j). Then a rule program $Rules^{(i,j)}(t)$ terminates iff

$$(\forall t, s)(\exists s')Do(Rules^{(i,j)}(t), s, s'). \tag{6.28}$$

So, termination of active rules is a progress properties since one of the quantifiers of the formula (6.28) is an existential quantifier.

Strictly speaking, termination just means that rule processing ceases at some point. That is what the formula (6.28) expresses. If we strictly follow the way termination is defined above, that is, as a property that "ensures that a rule program reaches a database state in which no further rules are triggered", we can view termination as a correctness property. In fact, it is easy to see that the following holds:

Proposition 6.23 *The following is a valid situation calculus formula:*

$$(\forall t, s, s').Do(Rules^{(i,j)}(t), s, s') \supset \neg\Phi^{(i,j)},$$

where $\Phi^{(i,j)}$ is the formula in the last test condition of $Rules^{(i,j)}$; i.e., for $Rules^{(1,1)}$, $\Phi^{(i,j)}$ is the formula

$$(\exists\vec{x}_1)(\tau_1[R_1,t]\wedge\zeta_1(\vec{x})[R_1,t]) \vee \ldots \vee (\exists\vec{x}_n)(\tau_n[R_n,t]\wedge\zeta_n(\vec{x}_n)[R_n,t]).$$

6.6.3 Confluence

Confluence ensures that a rule program never reaches two divergent database states, independently of the order of the execution of non-prioritized rules. Notice that, informally, a database state is the set of all database fluents, together with their truth values. Confluence must ensure that all possible executions of a rule program reach the same database state. We may express this property in the situation calculus as follows:

$$\begin{aligned}(\forall Q)(\forall s, s', s'', t).Do(Rules^{(i,j)}(t), s, s') \wedge Do(Rules^{(i,j)}(t), s, s'') \supset \\ Q[s'] \equiv Q[s''],\end{aligned} \tag{6.29}$$

where Q is a database query, and i and j are fixed. The formula (6.29) above shows that confluence is a safety property since all the quantifiers involved in it are universal quantifiers.

6.7 Summary

In this chapter, we have compiled sets of ECA rules into ConGolog programs that capture the execution models of active behavior and whose (abstract) execution semantics is given by extending the semantics of the ternary predicate Do originally introduced in [LLL$^+$94]; Do serves as an abstract interpreter for well formed ConGolog programs that capture transaction programs. In executing the well formed programs using Do, the basic relational theories of Chapter 4 corresponding to the various ATMs serve as background axioms. The main results here is are classification theorems for the various semantics of execution models for active behavior that we have identified. This theorem says roughly which semantics are equivalent and which are not. Finally, we have shown how to formulate some standard (e.g. confluence, and termination) and non-standard properties of a set of rules.

In summary, we have achieved the following results:

- method for simulating ATMs using ConGolog;
- semantics of rule priorities
- specification of various execution models of active behavior in the situation calculus, together with their coupling modes: immediate, deferred, and detached execution models;
- classification theorems for the semantics of some of the various execution models.

Chapter 7

Method for Implementing Active Relational Theories

In the preceding Chapters 2 through 6, we have been involved in a theoretical development that ultimately yields active relational theories of Chapter 5. Recall that these are theories of the situation calculus that capture the dynamic of (relational) active databases in the context of database transactions. Moreover, Chapter 6 expresses ECA rules as ConGolog programs, whose abstract execution is also given a semantics in the situation calculus.

This chapter extends a method for implementing basic action theories in Prolog presented by Reiter in [Rei01] to active relational theories. The justification for Reiter's method lies in a consequence of a fundamental theorem of logic programming due to K. Clark (See [Llo88] for Clark's result). Suppose that we have a first order theory in definitional form, in the sense that all its axioms are formulas of the form $(\forall\vec{x})P(\vec{x}) \equiv \phi$ (Definition 3.5). Clark's theorem says that, whenever a logical program `P` obtained from a definitional theory $\mathcal{T}$ by taking the if-halves of the sentences $(\forall\vec{x})P(\vec{x}) \equiv \phi$ yields the answer "yes" on a sentence ψ, then ψ is logically entailed by $\mathcal{T}$; also, whenever `P` yields the answer "no" on ψ, then $\neg\psi$ is logically entailed by $\mathcal{T}$. The Implementation Theorem 3.6 is a corollary of Clark's theorem.

Our first objective will be to extend the implementation theorem to basic and active relational theories. The second one is to give simple and straightforward implementation methods for active relational theories. A third objective is to model some aspects of the knowledge and execution models of SQL3, the ECA rule standard, following the guidelines set in Chapters 4, 5, and 6.

7.1 Implementing Basic Relational Theories

In this section, we extend the implementation theorem of [Rei01] to basic relational theories. We restrict ourself to the case of flat transactions and closed nested transactions.

7.1.1 Some Preliminaries on the Syntax of Logic Programs

Logic programs are usually formulated in a first order language over a vocabulary of variables, constants, and predicates and functions symbols of various arities. In this thesis, we are concerned with logic programs that contain functions. Our definitions follow the exposition in [Llo88].

Definition 7.1 (Terms, Atoms, and Literals) *A* term *is inductively defined as follows: variables and constants are terms, and, for each n-ary function symbol f and terms $t_1, \cdots, t_n$, $f(t_1, \cdots, t_n)$ is a term. A* ground *term is one that mentions no variables.*

An atom *is formula $P(t_1, \cdots, t_n)$, where P is an n-ary predicate symbol and the t_i are terms. A* ground atom *is one whose terms are all ground.*

A literal *is an atom or the negation of an atom.*

Definition 7.2 (Normal and General Programs) *A clause is a formula of the form*

$$(\forall \vec{x}).A_1 \vee \cdots \vee A_k \leftarrow L_1 \wedge \ldots \wedge L_n \;\; k, n \geq 0,$$

where each A_i and L_i is a literal, and $\vec{x}$ are all the variables occurring in the A_i and L_i.[1]

A normal program clause *is a clause of the form*

$$(\forall \vec{x}).A \leftarrow L_1 \wedge \cdots \wedge L_n \;\; n \geq 0,$$

where A is an atom, and each L_i is a literal; A is called the head, and $L_1 \wedge \ldots \wedge L_n$ is called the body of the clause. If the L_i are all atoms, then we have a definite program clause. *If $m = 0$, then the Horn clause is called a* fact. *A clause mentioning only ground terms is a* ground clause. *A* normal logic program *is a finite set of normal program clauses.*

A general logic program statement *(or just* logic program statement*) is a first order formula of the form $A \leftarrow W$, where A is an atom and W is an* **arbitrary** *first order formula. Finally, a* general logic program *(or just a* logic program*) is a finite set of logic program statements.*

Definition 7.3 (Normal and General Goals) *A* normal goal *is a clause of the form $L_1, \cdots, L_n$, where each L_i is a literal and is called a* subgoal. *If the L_i are all atoms, we have a* definite goal. *A* general goal *(or just* goal*) is an arbitrary first order formula.*

7.1.2 Revised Lloyd-Topor Transformations for non-Markovian Sentences

The Lloyd-Topor transformations are syntactical transformations that take an arbitrary first order sentence and transform it into a set of clauses suitable for a straightforward translation into Prolog syntax.

[1]As noticed earlier, leading universal will always be droped and the resulting free variable will implicitly be assumed to be universally quantifies.

Let P be the input program of the transformation. The set of sentences corresponding to the output P′ of the transformation forms a program called the normal form of P. In [Llo88], Lloyd-Topor rules are used to transform general logical programs of the form $A \leftarrow W$ and general goals into normal forms for which the conditions of the generalized Clark's theorem (to be seen below) holds.

In this document, we use the revised Lloyd-Topor transformations given in [Rei01] and collected in Appendix B. However, we have to accommodate the $\sqsubset$-atom that is of high interest for expressing non-Markovian control. To start with this task, we introduce a (simplified) notion of *bounded formula* proposed in [Gab02a].

Definition 7.4 (Bounded Formula) *A formula W of $\mathcal{L}_{sitcalc}$ is bounded by a situation term σ iff any further situation term σ' that it mentions is such that $W \models S_0 \sqsubseteq \sigma' \sqsubseteq \sigma$.*

Intuitively, a formula bounded by a situation term σ is one whose situation terms are all confined to the past of σ. All the non-Markovian sentences of our relational theories are of this sort.

The following abbreviations, based on [Gab02a], are introduced for handling bounded formulas:

Abbreviation 7.1

$$\begin{array}{l}
(\exists s : \sigma' \sqsubset \sigma)W =_{df} (\exists s).\sigma' \sqsubset \sigma \wedge W,\\
(\exists s : \sigma' = \sigma)W =_{df} (\exists s).\sigma' = \sigma \wedge W,\\
(\forall s : \sigma' \sqsubset \sigma)W =_{df} \neg(\exists s).\sigma' \sqsubset \sigma \wedge \neg W,\\
(\forall s : \sigma' = \sigma)W =_{df} \neg(\exists s).\sigma' = \sigma \wedge \neg W,\\
(\exists s : \sigma' \sqsubseteq \sigma)W =_{df} (\exists s : \sigma' \sqsubset \sigma)W \vee (\exists s : \sigma' = \sigma)W,\\
(\forall s : \sigma' \sqsubseteq \sigma)W =_{df} (\forall s : \sigma' \sqsubset \sigma)W \wedge (\forall s : \sigma' = \sigma)W.
\end{array}$$

Definition 7.5 (Lloyd-Topor Transforms for non-Markovian Formulas)

1. *When W is a literal that does not mention $\sqsubset$, $lt(W) = W$.*

2. $lt(\sigma = \sigma) = TRUE, lt(\sigma \sqsubset do(a, \sigma)) = TRUE, lt(\sigma_1 \sqsubset do(a, \sigma_2)) = lt(\sigma_1 \sqsubset \sigma_2)$.

3. $lt((\exists s : do(a, \sigma) = do(a', \sigma'))W) = a = a' \wedge lt((\exists s : \sigma = \sigma')W)$.

4. $lt((\exists s : \sigma' \sqsubset do(a, \sigma))W) = lt((\exists s : \sigma' = \sigma)W) \vee lt((\exists s : \sigma' \sqsubset \sigma)W)$.

5. $lt((\exists s : \sigma' \sqsubseteq \sigma)W) = lt((\exists s : \sigma' = \sigma)W) \vee lt((\exists s : \sigma' \sqsubset \sigma)W)$.

6. $lt((\exists s : s = s')W) = s = s' \wedge lt(W)$.

7. $lt((\forall s : Bound)W) = lt(\neg(\exists s : Bound)\neg W)$,
   where $Bound$ is a $\sqsubset$-literal or an equality between situations.

8. $lt(\neg(\forall s : Bound)W) = lt((\exists s : Bound)\neg W)$,
where $Bound$ is a $\sqsubset$-literal or an equality between situations.

9. $lt(\neg(\exists s : Bound)W) = \neg lt((\exists s : Bound)W)$,
where $Bound$ is a $\sqsubset$-literal or an equality between situations.

Suppose we have a logic program P with program statements of the form $W \supset A$, where A is an atom and W is a first order formula possibly mentioning the $\sqsubset$-atom. Then we next transform P into a normal program, called the *normal form* of P as follows:

1. When W is a formula that does not mention $\sqsubset$, we replace $W \supset A$ by a formula of the form $lt(W) \supset A$, where $lt(W)$ is a formula inductively obtained by the transformations in Appendix B.

2. When W is a formula that mentions $\sqsubset$ or any equality between situations, any bounded subformula W' of W is replaced by a formula of the form $lt(W) \supset A$, where $lt(W)$ is a formula inductively obtained by the transformations in Definition 7.5. In the implementations of our theories, however, whenever W mentions $\sqsubset$ or any equality between situations, we will use the predicate $Holds(\phi, s, s')$ which is given in Definition 7.6 and extends Definition 6.8.

Definition 7.6 (Extension of the Semantics of $Holds$)

$Holds(s' = s', s, s)$,

$Holds(s' \sqsubset do(a, s'), s, s)$,

$Holds(s' \sqsubset do(a, s''), s, s) =_{df} Holds(s' \sqsubset s'', s, s)$,

$Holds((\exists s' : s' = s'')\phi, s, s_1) =_{df} (\exists s^*).Holds(s' = s'', s, s^*) \wedge Holds(\phi, s^*, s_1)$,

$Holds((\exists s' : do(a, s_1) = do(a', s_2))\phi, s, s_3) =_{df} a = a' \wedge Holds((\exists s' : s_1 = s_2)\phi, s, s_3)$,

$Holds((\exists s' : s_1 \sqsubset do(a, s_2))\phi, s, s_3) =_{df} Holds((\exists s' : s_1 = s_2)\phi, s, s_3) \vee Holds((\exists s' : s_1 \sqsubset s_2)\phi, s, s_3)$,

$Holds((\exists s' : s_1 \sqsubseteq s_2)\phi, s, s_3) =_{df} Holds((\exists s' : s_1 = s_2)\phi, s, s_3) \vee Holds((\exists s' : s_1 \sqsubset s_2)\phi, s, s_3)$,

$Holds((\forall s' : s_1 \sqsubseteq s_2)\phi, s, s_3) =_{df} Holds(\neg(\exists s' : s_1 \sqsubseteq s_2)\neg\phi, s, s_3)$,

$Holds(\neg(\forall s' : s_1 \sqsubseteq s_2)\phi, s, s_3) =_{df} Holds((\exists s' : s_1 \sqsubseteq s_2)\neg\phi, s, s_3)$,

$Holds(\neg(\exists s' : s_1 \sqsubseteq s_2)\phi, s, s_3) =_{df} \neg Holds((\exists s' : s_1 \sqsubseteq s_2)\phi, s, s_3)$.

7.1.3 Correct Answers, and Non-Floundering SLDNF-Resolution

In order to justify a Prolog implementation of our basic and active relational theories, we use definitional theories (Definition 3.5), together with an equality theory comprising standard axioms for equality and unique name axioms for individuals of sorts $\mathcal{A}$, $\mathcal{S}$, and $\mathcal{O}$. Such an equality theory will be made more precise in the conditions of the generalized Clark's theorem. For any given logic program P, there is a definitional theory T_{P} that can be built from it ([Llo88]).

Definition 7.7 (Completion of a Program) *Suppose* P *is a program. Then the* completion $T_{\mathtt{P}}$ *of* P *is the definitional theory obtained from* P*, together with an appropriate equality theory.*

Definition 7.8 (Answers and Correct Answers) *Suppose* P *is a normal program and* G *is a normal goal. Then an* answer θ *is an arbitrary substitution for variables in* $\mathtt{P} \cup G$*; and* θ *is a* correct answer *for* P *iff* $T_{\mathtt{P}} \models (\forall)G\theta$*, where* $T_{\mathtt{P}}$ *is the completion of* P*, and* $(\forall)G\theta$ *denotes the result of universally quantifying all the free variables (if any) of the goal* G *on which the answer* θ *has been applied.*

The concept of correct answer is a declarative one. This declarative concept has been complemented with a procedural concept called SLDNF-resolution which is based on Kowalski's SLDNF-resolution procedure ([Kow74]), augmented with negation as failure ([Cla78]). Since Clark's theorem will be assumed without proof in this thesis, we restrict ourself to an informal presentation of SLDNF-resolution.

Selected linear, definite clause (SLD) resolution is a goal-driven top-down proof procedure that is focussed on deriving a given definite goal G from a definite program P. To prove G, we choose a program clause $A \leftarrow L_1 \wedge \cdots \wedge L_n$ with head A such that G and A can be unified with θ as the most general unifier (MGU) such that $G\theta = A\theta$. Then we attempt to prove the body of $(L_1 \wedge \cdots \wedge L_n)\theta$ which is the body $L_1 \wedge \cdots \wedge L_n$ on which the MGU θ has been applied. We iterate this task for each of the $L_i\theta$ until we either reach facts or fail to choose an appropriate program clause. The composition of all the MGUs accumulated during such iterations is the SLD-computed answer for the goal G.

Clark's negation as failure principle states that if, for a given goal G, the SLD-proof procedure finitely generates no answers, then infer $\neg G$. By adding such a rule to SLD-resolution, one obtains *SLDNF-computed answers* for normal goals proven from normal programs.

SLDNF-resolution is the base of many state-of-the-art Prolog implementations, including Eclipse Prolog, the logic programming language that we use to implement some of the sample relational theories given throughout this thesis. These systems make an important assumption about the SLDNF-resolution they use: at every time during the computation, no goal is reached that contains only non-ground negative literals. Any SLDNF-resolution procedure that can't guarrantee this is said to *flounder* at some point, meaning that it can reach a point where only non-ground literals are encountered.

7.1.4 Generalized Clark's Theorem

The generalized Clark's theorem states the soundness of the SLDNF-resolution. For our exposition, we follow both [Rei01] and [Llo88]. Though not explicitely said in the formulation below, Clark's theorem assumes a non-floundering SLDNF-resolution procedure.

Theorem 7.9 (Generalized Clark's Theorem) *Suppose that* $T_{\mathtt{P}}$ *is a definitional theory corresponding to a logic program* P *that contains a definition for every predicate symbol given by some finite first order language, together with the following equality theory:*

- *For every n-ary function symbols f and g of the language,*

$$f(\vec{x}) \neq g(\vec{y}).$$

- *For every n-ary function symbol f of the language,*

$$f(x_1, \cdots, x_n) = f(y_1, \cdots, y_n) \supset x_1 = y_1 \wedge \cdots \wedge x_n = y_n$$

- *For each term $t[x]$ that mentions x and that is different from x,*

$$t[x] \neq x.$$

Suppose further that $lt(\mathtt{P})$ is a program obtained by applying the revised Lloyd-Topor transformations to $\mathtt{P}$ and that G is a normal goal. Then

- *Every SLDNF-computed answer θ for $\mathtt{P} \cup G$ is a correct answer for $\mathtt{P}$ and vice-versa.*
- *Whenever the SLDNF-resolution fails for G, then $T_{\mathtt{P}} \models (\forall)\neg G$.*

This theorem will serve as basis for our implementation of relational theories in the following way. First, we will transform a given relational theory to obtain a definitional theory in the sense of Definition 3.5, together with an appropriate equality theory; doing so we meet the conditions of Clark's theorem. Then, we must show that, using such a definitional theory to prove a regressable sentence, we do not loose anything with respect to proving that the same sentence logically follows from the original relational theory. This was already shown in [Rei01] for basic action theories and we have to extend the result of [Rei01] to relational theories. Finally, we will conclude that the normal program clauses obtained from the if-halves of the sentences of the definitional theory using the revised Lloyd-Topor transformations will provide a correct Prolog (more precisely: SLDNF) implementation of our original relational theory in the sense of Clark's theorem.

7.1.5 Implementing BRTs for Flat Transactions

Our first task is to extend the definition of closed initial databases for basic action theories (Definition 3.4) to basic relational theories.

Definition 7.10 (Closed Initial Database for BRTs) *Suppose $\mathcal{D}$ is a basic relational theory with relational language $\mathfrak{R}$. Its initial database $\mathcal{D}_{S_0}$ is in closed form iff*

- *For each database fluent F of $\mathfrak{R}$, $\mathcal{D}_{S_0}$ contains exactly one sentence of the form*

$$F(\vec{x}, t, S_0) \equiv \Psi_F(\vec{x}, t, S_0), \tag{7.1}$$

where $\Psi_F(\vec{x}, t, S_0)$ is a first order formula uniform in S_0 with free variables among $\vec{x}$,t.

- *For each system fluent F of $\mathfrak{R}$, $\mathcal{D}_{S_0}$ contains exactly one sentence of the form $F(\vec{x}, S_0) \equiv \Psi_F(\vec{x}, S_0)$, where $\Psi_F(\vec{x}, S_0)$ is a first order formula uniform in S_0 with free variables among $\vec{x}$.*
- *For each non-fluent predicate symbol P of $\mathfrak{R}$, except $Poss$ and $\sqsubset$, $\mathcal{D}_{S_0}$ contains exactly one sentence of the form $P(\vec{x}) \equiv \Theta_P(\vec{x})$, where $\Theta_P(\vec{x})$ is a situation independent first order formula with free variables among $\vec{x}$.*
- *For each dependency predicate dep of $\mathfrak{R}$, $\mathcal{D}_{S_0}$ contains exactly one sentence of the form $\neg dep(t, t', S_0)$, with free variables t, t'.*
- *The remaining sentences of $\mathcal{D}_{S_0}$ are unique name axioms for the sorts $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{O}$.*

Notice that a closed initial database for BRTs is slightly more than a definitional theory in the sense of Definition 3.5. It captures dynamical domains in terms not only of definitional equivalences, but also of the unique name axioms for names of all the sorts involved in the relational language. However, this restriction suits perfectly to relational database domains where, generally, a closed world assumption ([Rei78]) is made. As is the case for basic action theories in [Rei01], the equivalences in Definition 7.10 will serve to derive definitional forms for fluents of BRTs, and the unique name axioms will be necessary to derive an implementation theorem for BRTs. To that end, we progressively recast all our BRT axioms into definitions and unique name axioms for sorts $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{O}$.

Definition for Action Precondition Axioms

Suppose we have a relational language $\mathfrak{R}$ that has n internal actions $A(\vec{x}_1), \ldots, A(\vec{x}_n)$, together with the external actions $Begin(t)$, $End(t)$, $Commit(t)$, and $Rollback(t)$. Moreover, let the action precondition axioms for the n internal actions have the form (4.6) as follows:

$$Poss(A_1(\vec{x}_1, t), s) \equiv (\exists t')\Pi_{A_1}(\vec{x}_1, t', s) \wedge IC_e(do(A_1(\vec{x}_1, t), s)) \wedge running(t, s),$$
$$\vdots$$
$$Poss(A_n(\vec{x}_n, t), s) \equiv (\exists t')\Pi_{A_n}(\vec{x}_n, t', s) \wedge IC_e(do(A_n(\vec{x}_n, t), s)) \wedge running(t, s),$$

and those for external actions

$$Poss(Begin(t), s) \equiv \Pi_{Begin}(t, s),$$
$$Poss(End(t), s) \equiv \Pi_{End}(t, s),$$
$$Poss(Commit(t), s) \equiv \Pi_{Commit}(t, s),$$
$$Poss(Rollback(t), s) \equiv \Pi_{Rollback}(t, s).$$

Here, $\Pi_{Begin}(t, s)$, $\Pi_{End}(t, s)$, $\Pi_{Commit}(t, s)$, and $\Pi_{Rollback}(t, s)$ are the first order formulas uniform in s representing the right hand side of the action precondition axioms for $Begin(t)$, $End(t)$, $Commit(t)$, and $Rollback(t)$, given in (4.11), (4.12), (4.13), and (4.14), repectively.

Theorem 7.11 (Definition for $Poss$) *The action precondition axioms for internal and external axioms of a basic relational theory logically entail the following sentence:*

$$\begin{aligned}
&Poss(a, s) \equiv \\
&\quad (\exists \vec{x}, t, t')\{[\bigvee_{i=1}^{n}(a = A_i(\vec{x}, t) \land \Pi_{A_i}(\vec{x}_i, t', s) \land IC_e(do(A_i(\vec{x}_i, t), s)) \land running(t, s))] \lor \\
&\qquad [a = Begin(t) \land \Pi_{Begin}(t, s)] \lor [a = End(t) \land \Pi_{End}(t, s)] \lor \\
&\qquad [a = Commit(t) \land \Pi_{Commit}(t, s)] \lor [a = Rollback(t) \land \Pi_{Rollback}(t, s)]\}.
\end{aligned} \tag{7.2}$$

Corollary 7.12 *The if-half of the definition (7.2) is logically equivalent to the conjunction of the following sentences:*

$$\begin{gathered}
(\exists t')\Pi_{A_1}(\vec{x}_1, t', s) \land IC_e(do(A_1(\vec{x}_1, t), s)) \land running(t, s) \supset Poss(A_1(\vec{x}_1, t), s), \\
\vdots \\
(\exists t')\Pi_{A_n}(\vec{x}_n, t', s) \land IC_e(do(A_n(\vec{x}_n, t), s)) \land running(t, s) \supset Poss(A_n(\vec{x}_n, t), s), \\
\Pi_{Begin}(t, s) \supset Poss(Begin(t), s), \\
\Pi_{End}(t, s) \supset Poss(End(t), s), \\
\Pi_{Commit}(t, s) \supset Poss(Commit(t), s), \\
\Pi_{Rollback}(t, s) \supset Poss(Rollback(t), s).
\end{gathered}$$

Definition for Fluents

Suppose $\mathcal{D}$ is a basic relational theory with closed initial database. Then, for each fluent, $\mathcal{D}$ contains the axioms (4.8) and (7.1) which we recall:

$$\begin{aligned}
F(\vec{x}, t, do(a, s)) \equiv\ & (\exists \vec{t_1})\Phi_F(\vec{x}, a, \vec{t_1}, s) \land \neg(\exists t'')a = Rollback(t'') \lor \\
& (\exists t'')a = Rollback(t'') \land restoreBeginPoint(F, \vec{x}, t'', s),
\end{aligned} \tag{7.3}$$

$$F(\vec{x}, t, S_0) \equiv \Psi_F(\vec{x}, t, S_0). \tag{7.4}$$

Theorem 7.13 (Definition for Fluents) *For each fluent F, the axioms (4.8) and (7.1) are logically equivalent to the following sentence:*

$$\begin{aligned}
&F(\vec{x}, t, s) \equiv s = S_0 \land \Psi_F(\vec{x}, t, S_0) \lor \\
&\quad (\exists a, s', \vec{t_1})\{[s = do(a, s') \land \Phi_F(\vec{x}, a, \vec{t_1}, s) \land \neg(\exists t'')a = Rollback(t'')] \lor \\
&\qquad [(\exists t'').a = Rollback(t'') \land restoreBeginPoint(F, \vec{x}, t'', s')]\}.
\end{aligned} \tag{7.5}$$

Corollary 7.14 *The if-half of the definition (7.13) is logically equivalent to the conjunction of the following three sentences:*

$$\Psi_F(\vec{x}, t, S_0) \supset F(\vec{x}, t, S_0),$$
$$(\exists \vec{t_1})\Phi_F(\vec{x}, a, \vec{t_1}, s) \wedge \neg(\exists t'')a = Rollback(t'') \supset F(\vec{x}, t, do(a, s)),$$
$$(\exists t'')a = Rollback(t'') \wedge restoreBeginPoint(F, \vec{x}, t'', s) \supset F(\vec{x}, t, do(a, s)).$$

Definition for Dependency Predicates

Suppose $\mathcal{D}$ is a basic relational theory with closed initial database. Then, by definition, $\mathcal{D}$ contains the axioms $\neg r\_dep(t, t', S_0)$ and $\neg sc\_dep(t, t', S_0)$, as well as the dependency axioms (4.16) and (4.46).

Theorem 7.15 (Definition for Dependency Predicates) *Axioms $\neg r\_dep(t, t', S_0)$ and (4.16), together with $\neg sc\_dep(t, t', S_0)$ and (4.17) are logically equivalent, in that order, to the following sentences:*

$$r\_dep(t, t', s) \equiv s \neq S_0 \wedge transConflict(t, t', s), \tag{7.6}$$
$$sc\_dep(t, t', s) \equiv s \neq S_0 \wedge readsFrom(t, t', s). \tag{7.7}$$

The if-halves of axioms (7.6) and (7.7) are trivially

$$s \neq S_0 \wedge transConflict(t, t', s) \supset r\_dep(t, t', s),$$
$$s \neq S_0 \wedge readsFrom(t, t', s) \supset sc\_dep(t, t', s).$$

7.1.6 Theoretical Basis of the Implementation

In Section 3.4, we have seen that automated reasoning in the situation calculus is organized around a computational mechanism called *regression*. A regression operator due to Reiter ([Rei01] is reviewed in Appendix A, where we also give an extension of this mechanism to basic relational theories by adapting a general mechanism for non-Markovian basic action theories given in [Gab00].

Recall that, intuitively, regression is a syntactic manipulation mechanism aimed at reducing the nesting of the complex situation term $do(\alpha, \sigma)$ in any given sentence W in an appropriate syntactic form. In the present setting, W is a non-Markovian formula of the situation calculus. Suppose W mentions a fluent $F(\vec{X}, T, do(\alpha, \sigma))$ with F's successor state axiom being $F(\vec{x}, t, do(a, s)) \equiv (\exists \vec{t_1})\Phi(\vec{x}, \vec{t_1}, a, s)$, where $\Phi(\vec{x}, \vec{t_1}, a, s)$ is a non-Markovian first order sentence. Then we can apply the regression operator $\mathcal{R}$ to W — $\mathcal{R}$[W] — to determine a logically equivalent variant W' of W in which $F(\vec{X}, T, do(\alpha, \sigma))$ has been replaced by $(\exists \vec{t_1})\Phi(\vec{X}, \vec{t_1}, a, s)$. If we take W', which mentions $do(\alpha, \sigma)$, instead of W, which mentions σ, we will have reduced the nesting of do by one.

As in the case of Markovian basic action theories, the regression operator is applicable to formulas that are in *regressable* form which, for non-Markovian formulas, means the following:

Definition 7.16 *(non-Markovian Regressable Formulas) A formula W of $\mathcal{L}_{sitcalc}$ is non-Markovian regressable iff*

1. *W is a first order formula whose terms of sort $\mathcal{S}$ are all of the syntactic form $do([\alpha_1, \cdots, \alpha_n], S_0)$, where $n \geq 0$ and $\alpha_1, \cdots, \alpha_n$ are of sort $\mathcal{A}$.*

2. *In every atom $Poss(\alpha, \sigma)$ mentionned by W, α is of the form $A(X_1, \cdots, X_m, t)$, where $m \geq 0$ and A is some $m + 1$-ary action function symbol of $\mathcal{L}_{sitcalc}$.*

3. *W may quantify over situations and mention the predicate symbol $\sqsubset$, or equality atoms over situation terms.*

There is a theorem justifiying the regression as a computational mechanism for general non-Markovian basic action theories:

Theorem 7.17 (One-Step-Regression Theorem for non-Markovian Basic Action Theories)([Gab00]) *Suppose W is a regressable sentence of $\mathcal{L}_{sitcalc}$, and $\mathcal{D}$ is a non-Markovian basic action theory. Then $\mathcal{R}[W]$ is uniform in S_0, and $\mathcal{D} \models (\forall)W \equiv \mathcal{R}[W]$.*

Now, From Theorems 4.13 and 7.17, we use non-Markovian regression to repeatedly transform a given non-Markovian query W into a formula $\mathcal{R}[W]$ that mentions only S_0 as official situation arguments of the fluents. Therefore we have the following:

Corollary 7.18 (Regression Theorem for Basic Relational Theories) *Suppose W is a regressable sentence of $\mathcal{L}_{sitcalc}$, and $\mathcal{D}$ is a basic relational theory. Then*

$$\mathcal{D} \models W \textit{ iff } \mathcal{D}_{S_0} \cup \mathcal{D}_{una} \cup \mathcal{D}_{IC}[S_0] \models \mathcal{R}^*[W],$$

where $\mathcal{R}^*[W]$ is the result of repeatedly applying $\mathcal{R}$ on W.

As for basic action theories of [Rei01], Corollary 7.18 is computationally important. It states that in order to evaluate a sentence W against a basic relational theory $\mathcal{D}$, it is necessary and sufficient to evaluate $\mathcal{R}[W]$ against $\mathcal{D}_{S_0} \cup \mathcal{D}_{una} \cup \mathcal{D}_{IC}[S_0]$.

In addition to this Regression Theorem, which furnishes a theoretical justification of posing regressed non-Markovian queries against initial databases, we need a result that justifies a simple Prolog implementation of active databases formalized via basic relational theories. The following two results are a generalization of similar results of [Rei01] for basic action theories to basic relational theories.

Theorem 7.19 (Definition Theorem for Basic Relational Theories) *Suppose that $\mathcal{D}$ is a basic relational theory of $\mathcal{L}_{sitcalc}$ with relational language $\mathfrak{R}$ and whose initial database $\mathcal{D}_{S_0}$ is in closed form. Suppose further that $\mathcal{D}^\Delta = \mathcal{D}^\Delta_{S_0} \cup \mathcal{D}^\Delta_{ap} \cup \mathcal{D}^\Delta_{ss} \cup \mathcal{D}^\Delta_{dep} \cup \mathcal{D}_{IC} \cup \mathcal{D}_{una} \cup \mathcal{D}_{uns} \cup \mathcal{D}_{uno}$ such that*

- $\mathcal{D}^{\Delta}_{S_0}$ *consists of* $\mathcal{D}S_0$ *where all equivalences for fluents have been removed;*
- $\mathcal{D}^{\Delta}_{ap}$ *consists of the definition for* $Poss$ *given in Theorem 7.2;*
- $\mathcal{D}^{\Delta}_{ss}$ *consists of the definitions for each of the fluents of* $\mathfrak{R}$ *of the form given in Theorem 7.13;*
- $\mathcal{D}^{\Delta}_{dep}$ *consists of the definitions (7.6) and (7.7) for the dependency predicates;*
- $\mathcal{D}_{IC}$ *consists of the integrity contraints of the domain. Without loss of generality, we assume that they are all checked by an oracle*[2]*;*
- $\mathcal{D}_{una}$, $\mathcal{D}_{uns}$, *and* $\mathcal{D}_{uno}$ *consists of the unique name axioms for actions, for situations, and for objects, respectively.*

Then, whenever G *is a non-Markovian regressable sentence of* $\mathfrak{R}$*,*

$$\mathcal{D} \models G \quad \textit{iff} \quad \mathcal{D}^{\Delta} \models G.$$

Corollary 7.20 (Implementation Theorem for Basic Relational Theories) *Suppose that* $\mathcal{D}$ *is a basic relational theory of* $\mathcal{L}_{sitcalc}$ *with the restrictions of Theorem 7.19. Suppose further that* `P` *is a Prolog program obtained from the following sentences, after transforming them by the revised Lloyd-Topor transformations*[3]*:*

- *For each definition of a non-fluent predicate of* $\mathcal{D}_{S_0}$ *of the form* $P(\vec{x}) \equiv \Theta_P(\vec{x})$*:*

$$\Theta_P(\vec{x}) \supset P(\vec{x}).$$

- *For each equivalence in* $\mathcal{D}_{S_0}$ *of the form* $F(\vec{x}, t, S_0) \equiv \Psi_F(\vec{x}, t, S_0)$*:*

$$\Psi_F(\vec{x}, t, S_0) \supset F(\vec{x}, t, S_0).$$

- *For each action precondition axiom of* $\mathcal{D}_{ap}$ *of the form*

$$Poss(A(\vec{x}, t), s) \equiv (\exists t')\Pi_A(\vec{x}, t', s) \wedge IC_e(do(A(\vec{x}, t), s)) \wedge running(t, s) :$$

$$(\exists t')\Pi_A(\vec{x}, t', s) \wedge IC_e(do(A(\vec{x}, t), s)) \wedge running(t, s) \supset Poss(A(\vec{x}), s).$$

- *For the action precondition axioms of* $\mathcal{D}_{FT}$*:*

$$\Pi_{Begin}(t, s) \supset Poss(Begin(t), s),$$

$$\Pi_{End}(t, s) \supset Poss(End(t), s),$$

$$\Pi_{Commit}(t, s) \supset Poss(Commit(t), s),$$

$$\Pi_{Rollback}(t, s) \supset Poss(Rollback(t), s).$$

[2]The predicate $Holds$ seen in Section 6.1 is such an oracle.
[3]See Appendix B.

- *For each successor state axiom of $\mathcal{D}_{ss}$ of the form*

$$\begin{aligned} F(\vec{x}, t, do(a, s)) \equiv & (\exists \vec{t_1}) \Phi_F(\vec{x}, a, \vec{t_1}, s) \wedge \neg(\exists t'') a = Rollback(t'') \vee \\ & (\exists t'') a = Rollback(t'') \wedge restoreBeginPoint(F, \vec{x}, t'', s) : \\ & [(\exists \vec{t_1}) \Phi_F(\vec{x}, a, \vec{t_1}, s) \wedge \neg(\exists t'') a = Rollback(t'') \vee \\ & (\exists t'') a = Rollback(t'') \wedge restoreBeginPoint(F, \vec{x}, t'', s)] \supset F(\vec{x}, t, do(a, s)). \end{aligned}$$

- *For the dependency axioms (4.16) and (4.17): (7.6) and (7.7).*

Then `P` *is a correct Prolog implementation of the basic relational theory $\mathcal{D}$ for proving regressable sentences. Moreover, to prove a regressable sentence G, first transform it using the revised Lloyd-Topor transformations, then issue the resulting query to* `P`.

7.1.7 Implementing BRTs for Closed Nested Transactions

To implement basic relational theories for closed nested transactions, we extend the implementation of basic relational theories for flat transactions in a straightforward way. Therefore, we will simply state the various definitions needed without proofs.

Definition for Action Precondition Axioms

Recall that closed nested transactions have one more external action than the corresponding flat models, namely $Spawn(t, t')$. So the only change we have is that (7.1.5) now has the axiom $Poss(Spawn(t, t'), s) \equiv \Pi_{Spawn}(t, t', s)$ for the new external action. Therefore, Corollary 7.12 will have to accommodate this fact by adding the sentence

$$\Pi_{Spawn}(t, t', s) \supset Poss(Spawn(t, t'), s)$$

to those that it mentions. Also, $\Pi_{Begin}(t, s)$, $\Pi_{End}(t, s)$, $\Pi_{Commit}(t, s)$, and $\Pi_{Rollback}(t, s)$ are the first order formulas uniform in s representing the right hand side of the action precondition axioms for $Begin(t)$, $End(t)$, $Commit(t)$, and $Rollback(t)$, given in (4.40), (4.42), (4.43), and (4.44), repectively.

Definition for Fluents

In the case of closed nested transactions a basic relational theory $\mathcal{D}$ with closed initial database has, for each fluent, the axioms (4.49) and (7.1). So the only change we have is that Theorem 7.13 and Corollary 7.14 must now accommodate the external action $Spawn(t, t')$. With this change, the definition of a fluent F is

$$\begin{aligned}F(\vec{x},t,s) \equiv\, & s = S_0 \wedge \Psi_F(\vec{x},t,S_0) \vee \\ & (\exists a, s', \vec{t_1})\{[s = do(a,s') \wedge \Phi_F(\vec{x},a,\vec{t_1},s) \wedge \neg(\exists t'')a = Rollback(t'')] \vee \\ & \quad [(\exists t'').a = Rollback(t'') \wedge \neg(\exists t^*)parent(t'',t^*,s) \wedge restoreBeginPoint(F,\vec{x},t'',s') \vee \\ & \quad [(\exists t'').a = Rollback(t'') \wedge (\exists t^*)parent(t'',t^*,s) \wedge restoreSpawnPoint(F,\vec{x},t'',s').\end{aligned} \tag{7.8}$$

The if-half of the definition (7.8) is logically equivalent to the conjunction of the following four sentences:

$$\Psi_F(\vec{x},t,S_0) \supset F(\vec{x},t,S_0),$$
$$(\exists \vec{t_1})\Phi_F(\vec{x},a,\vec{t_1},s) \wedge \neg(\exists t'')a = Rollback(t'') \supset F(\vec{x},t,do(a,s)),$$
$$(\exists t'').a = Rollback(t'') \wedge \neg(\exists t^*)parent(t'',t^*,s) \wedge restoreBeginPoint(F,\vec{x},t'',s) \supset F(\vec{x},t,do(a,s))$$
$$(\exists t'').a = Rollback(t'') \wedge (\exists t^*)parent(t'',t^*,s) \wedge restoreSpawnPoint(F,\vec{x},t'',s) \supset F(\vec{x},t,do(a,s)).$$

Definition for Dependency Predicates

Suppose $\mathcal{D}$ is a basic relational theory with closed initial database. Then $\mathcal{D}$ contains the axioms $\neg r\_dep(t,t',S_0)$, $\neg sc\_dep(t,t',S_0)$, $\neg c\_dep(t,t',S_0)$, and $\neg wr\_dep(t,t',S_0)$, as well as the dependency axioms (4.45)–(4.48).

Theorem 7.21 (Definition for Dependency Predicates) *Axioms $\neg r\_dep(t,t',S_0)$ and (4.45), $\neg sc\_dep(t,t',S_0)$ and (4.46), $\neg c\_dep(t,t',S_0)$ and (4.47), and $\neg wr\_dep(t,t',S_0)$ and (4.48) are logically equivalent, in that order, to the following sentences:*

$$r\_dep(t,t',s) \equiv s \neq S_0 \wedge transConflictNT(t,t',s), \tag{7.9}$$

$$sc\_dep(t,t',s) \equiv s \neq S_0 \wedge readsFrom(t,t',s), \tag{7.10}$$

$$\begin{aligned}c\_dep(t,t',s) \equiv (\exists a,s')[s \neq S_0 \wedge s = do(a,s') \wedge a = Spawn(t,t') \vee \\ c\_dep(t,t',s') \wedge \neg termAct(a,t) \wedge \neg termAct(a,t')],\end{aligned} \tag{7.11}$$

$$\begin{aligned}wr\_dep(t,t',s) \equiv (\exists a,s')[s \neq S_0 \wedge s = do(a,s') \wedge a = Spawn(t',t) \vee \\ wr\_dep(t,t',s') \wedge \neg termAct(a,t) \wedge \neg termAct(a,t')].\end{aligned} \tag{7.12}$$

Corollary 7.22 *The if-halves of axioms (7.9)–(7.12) are equivalent to*

$$s \neq S_0 \wedge transConflictNT(t,t',s) \supset r\_dep(t,t',s),$$
$$s \neq S_0 \wedge readsFrom(t,t',s) \supset sc\_dep(t,t',s),$$
$$[a = Spawn(t,t') \vee c\_dep(t,t',s') \wedge \neg termAct(a,t) \wedge \neg termAct(a,t')] \supset c\_dep(t,t',do(a,s)),$$
$$[a = Spawn(t,t') \vee c\_dep(t',t,s') \wedge \neg termAct(a,t) \wedge \neg termAct(a,t')] \supset wr\_dep(t,t',do(a,s)).$$

7.2 Implementing Active Relational Theories

Recall that each active relational theory has a subset that is a basic relational theory. So our task here is reduced to giving definitional forms for transition and event fluents.

First, we extend the definition of closed initial databases for basic relational theories (Definition 7.10) to active relational theories.

Definition 7.23 (Closed Initial Database for ARTs) *Suppose $\mathcal{D}$ is an active relational theory with relational language $\mathfrak{R}$, and let $\mathcal{D}_{brt}$ be its basic relational subset. The initial database $\mathcal{D}_{S_0}$ of $\mathcal{D}$ is in closed form iff it is of the form $\mathcal{D}_{S_0} = \mathcal{D}_{brt_{S_0}} \cup \mathcal{D}_{tf_{S_0}} \cup \mathcal{D}_{ef_{S_0}}$, where*

1. *$\mathcal{D}_{brt_{S_0}}$ is the closed form of $\mathcal{D}_{brt}$;*
2. *$\mathcal{D}_{tf_{S_0}}$ is a set of axioms such that, for each database fluent of $\mathfrak{R}$, includes the two sentences $(\forall r, \vec{x}, t)\neg F\_inserted(r, \vec{x}, t, S_0)$ and $(\forall r, \vec{x}, t)\neg F\_deleted(r, \vec{x}, t, S_0)$;*
3. *$\mathcal{D}_{ef_{S_0}}$ is a set of axioms such that, for each database fluent of $\mathfrak{R}$, includes the two sentences $(\forall r, t)\neg F\_inserted(r, t, S_0)$ and $(\forall r, t)\neg F\_deleted(r, t, S_0)$. Moreover, for each complex event fluent F, $\mathcal{D}_{ef_{S_0}}$ contains exactly one sentence of the form $(\forall r, t)\neg F(r, t, S_0)$.*

7.2.1 Definitions for Transition and Event Fluents

Theorem 7.24 (Definition for Transition Fluents) *Axioms $(\forall r, \vec{x}, t)\neg F\_inserted(r, \vec{x}, t, S_0)$ and (5.3), together with $(\forall r, \vec{x}, t)\neg F\_deleted(r, \vec{x}, t, S_0)$ and (5.4) are logically equivalent, in that order, to the following sentences:*

$$\begin{aligned}
&F\_inserted(r, \vec{x}, t, s) \equiv \\
&\quad (\exists a, s').s \neq S_0 \wedge s = do(a, s') \wedge considered(r, t, s') \wedge (\exists t')a = F\_insert(\vec{x}, t') \vee \\
&\quad F\_inserted(r, \vec{x}, t, s) \wedge a \neq F\_delete(\vec{x}, t),
\end{aligned} \tag{7.13}$$

$$\begin{aligned}
&F\_deleted(r, \vec{x}, t, do(a, s)) \equiv \\
&\quad (\exists a, s').s \neq S_0 \wedge considered(r, t, s) \wedge (\exists t')a = F\_delete(\vec{x}, t')] \vee \\
&\quad F\_deleted(r, \vec{x}, t, s') \wedge a \neq F\_insert(\vec{x}, t).
\end{aligned} \tag{7.14}$$

The if-halves of (7.13) and (7.14) are trivially

$$\begin{aligned}
&(\exists a, s').s \neq S_0 \wedge s = do(a, s') \wedge considered(r, t, s') \wedge (\exists t')a = F\_insert(\vec{x}, t') \vee \\
&\qquad F\_inserted(r, \vec{x}, t, s) \wedge a \neq F\_delete(\vec{x}, t) \supset F\_inserted(r, \vec{x}, t, s),
\end{aligned} \tag{7.15}$$

$$\begin{aligned}
&(\exists a, s').s \neq S_0 \wedge considered(r, t, s) \wedge (\exists t')a = F\_delete(\vec{x}, t') \vee \\
&\qquad F\_deleted(r, \vec{x}, t, s') \wedge a \neq F\_insert(\vec{x}, t) \supset F\_deleted(r, \vec{x}, t, do(a, s)).
\end{aligned} \tag{7.16}$$

By a transformation similar to that applied to transition fluents, we get if-halves of definitions for simple event fluents that are of the form

$$(\exists \vec{x}, a, s', t').s \neq S_0 \wedge s = do(a, s') \wedge a = F\_insert(\vec{x}, t') \wedge considered(r, t, s') \vee \\ F\_inserted(r, t, s) \supset F\_inserted(r, t, s), \tag{7.17}$$

$$(\exists \vec{x}, a, s', t').s \neq S_0 \wedge s = do(a, s') \wedge a = F\_delete(\vec{x}, t') \wedge considered(r, t, s) \vee \\ F\_deleted(r, t, s') \supset F\_deleted(r, t, s). \tag{7.18}$$

Since complex event fluents in fact are abbreviations standing for complex non-Markovian formulas, it is easy to introduce their definitional forms and to derive if-halves of the definitions that characterize these complex event fluents. We leave the details out.

7.2.2 Example

In the Debit/Credit example of Section 6.1, for the fluent $tellers(tid, tbal, t, s)$, we will have the following if-halves corresponding to sentences 7.15 and 7.16, respectively:

$$(\exists a, s').s \neq S_0 \wedge s = do(a, s') \wedge considered(r, t, s') \wedge (\exists t')a = tellers\_insert(tid, tbal, t') \vee \\ tellers\_inserted(r, tid, tbal, t, s) \wedge a \neq tellers\_delete(\vec{x}, t) \supset tellers\_inserted(r, tid, tbal, t, s),$$

$$(\exists a, s').s \neq S_0 \wedge considered(r, t, s) \wedge (\exists t')a = tellers\_delete(tid, tbal, t') \vee \\ tellers\_deleted(r, tid, tbal, t, s') \wedge a \neq F\_insert(\vec{x}, t) \supset tellers\_deleted(r, tid, tbal, t, do(a, s)).$$

Similarly, we have the following sentences corresponding to sentences 7.17 and 7.18, respectively:

$$(\exists tid, tbal, a, s', t').s \neq S_0 \wedge s = do(a, s') \wedge a = tellers\_insert(tid, tbal, t') \wedge considered(r, t, s') \vee \\ tellers\_inserted(r, t, s) \supset tellers\_inserted(r, t, s),$$

$$(\exists tid, tbal, a, s', t').s \neq S_0 \wedge s = do(a, s') \wedge a = tellers\_delete(tid, tbal, t') \wedge considered(r, t, s) \vee \\ tellers\_deleted(r, t, s') \supset tellers\_deleted(r, t, s).$$

7.3 A non-Markovian ConGolog Interpreter for Well Formed Programs with BRTs as Background Axioms

Below, we give a non-Markovian ConGolog interpreter written in Prolog for well formed programs that have basic relational theories as background axioms. This interpreter is based on the following assumptions[4]:

[4]Many of these assumptions are inherited from the basic ConGolog and GOLOG interpreters of [DGLL00] and [LRL$^+$97], respectively.

1. ConGolog programs are represented by the following Prolog terms:

   - `nil`: empty programs.
   - `?(p)`: test whether the condition `p` is true.
   - a_1`:`a_2: sequence.
   - a_1`#`a_2: nondeterministic action choice.
   - `pi(x,a)`: nondeterministic choice of argument `x` which is a Prolog constant standing for a ConGolog variable; `a` will use `x`.
   - `if(p,a)`: conditional without else part.
   - `if(p,`a_1`,`a_2`)`: conditional with else part.
   - `star(a)`: nondeterministic iteration.
   - `while(p,a)`: while loop.
   - a_1`#=`a_2: concurrency.
   - `procedure(args)`: procedure name `procedure` and arguments `args`.

2. Test actions are performed on conditions of the form:

   - an atomic formula.
   - `-p` : negation.
   - p_1`&`p_2: conjunction.
   - p_1`v`p_2: disjunction.
   - p_1`=>`p_2: implication.
   - p_1`<=>`p_2: equivalence.
   - σ_1`<<`σ_2: subsequence.
   - σ_1`<<=`σ_2: subsequence or equality.
   - `some(x,p)`: universal quantification over database domain objects, where `x` is a Prolog constant used in `p`.
   - `somes(x,`p_1`,`p_2`)`: universal quantification over situations, where `x` is a Prolog constant used in p_1 and p_2, p_1 is a formula of the form σ_1`<<`σ_2, σ_1`<<=`σ_2, or σ_1`=`σ_2, and p_2 is an arbitrary regressable formula of the situation calculus.
   - `all(x,p)`: universal quantification over database domain objects, with `x` and `p` as above.
   - `alls(x,`p_1`,`p_2`)` (universal quantification over situations), with `x`, p_1, and p_2 as above.

3. The predicate `holds/3` evaluate conditions in test actions, while loops, and conditionals. Notice this predicate evaluate non-Markovian formulas.

4. `final/2`, `trans/4`, `trans*/4`, and `do/3` are a straightforward Prolog rendering of the predicates of the situation calculus with same names and arities seen in Appendix C. The predicate `sub/4` performs term substitutions in formulas ([LRL+97]).

5. All fluents are relational. Therefore any functional fluent must be translated into a relational one.

6. The background axioms are constituted by the following:

   - a collection of Prolog clauses representing the initial database, that is, all the fluent atoms that are true in S_0.
   - a collection of Prolog clauses representing the if-halves of the situation calculus axioms for $Poss$, one for each external and internal action.
   - a collection of Prolog clauses representing the if-halves of the situation calculus axioms for database fluents, event fluents, transition fluents, and dependency predicates. Typically, these involves the use of the predicate `holds/3` in the body of Prolog clauses in each subformulas of the form described in Abbreviation 7.1.
   - a collection of Prolog facts of the form `proc(pName(t,arg`$_1$`,`$\cdots$`,arg`$_n$`), body)` describing a ConGolog procedure. Here, `t` is a transaction argument, `pName` is the procedure name, `arg`$_1$`,`$\cdots$`,arg`$_n$ are its arguments, and `body` is its body. The transaction argument is very important for the implementation to work, since mentioning that transaction argument will always ensure its propagation downwards to the primitive updates and test actions mentioned in the procedure. Also, the only free Prolog variables allowed in the body `body` are those among `t,arg`$_1$`,`$\cdots$`,arg`$_n$.

```
/* A non-Markovian ConGolog interpreter for running programs with BRTs. */

:- set_flag(print_depth,100).
:- nodbgcomp.  % turns the debugger down.
:- dynamic(proc/2).            /* Compiler directives. They MUST   */
:- set_flag(all_dynamic, on).  /* be loaded first. */

:- op(750, xfy, [<<]).  /* Subsequence */
:- op(750, xfy, [<<=]). /* Subsequence and equality */
:- op(800, xfy, [&]).   /* Conjunction */
:- op(850, xfy, [v]).   /* Disjunction */
:- op(870, xfy, [=>]).  /* Implication */
:- op(880, xfy, [<=>]). /* Equivalence */
:- op(890, xfy, [#=]).  /* concurrency */
:- op(950, xfy, [:]).   /* Action sequence */
:- op(960, xfy, [#]).   /* Nondeterministic action choice */

/* final(Prog,S) */

final(nil,_).
final(A1 : A2,S) :- final(A1,S), final(A2,S).
final(A1 # A2,S) :- final(A1,S); final(A2,S).
final(pi(_,A),S) :- final(A,S).
final(star(_),_).
final(if(P,A1,A2),S) :- holds(P,S,_), !, final(A1,S); final(A2,S).
final(while(P,A),S) :- holds(P,S,_), !, final(A,S).
final(while(_,_),_).
final(A1 #= A2,S) :- final(A1,S), final(A2,S).
final(!(A),S).

/* trans(Prog,S,Prog1,S1) */

trans(A,S,R,S1) :- primitive_action(A), poss(A,S), !, R=nil, S2 = do(A,S),
              (systemAct(A1,T), poss(A1,S2), S1 = do(A1,S2) ;
               not (systemAct(A2,T2), poss(A2,S2)),S1 = S2).
trans(?(P),S,R,S1) :- holds(P,S,S1), !, R=nil.
trans(A1 : A2,S,R,S1) :- final(A1,S), trans(A2,S,R,S1).
trans(A1 : A2,S,R1:A2,S1) :- trans(A1,S,R1,S1).
trans(A1 # A2,S,R,S1) :- trans(A1,S,R,S1); trans(A2,S,R,S1).
trans(pi(V,A),S,R,S1) :- sub(V,_,A,A1), trans(A1,S,R,S1).
trans(star(A),S,R : star(A),S1) :- trans(A,S,R,S1).
trans(if(P,A),S,R,S1) :- trans((?(P) : A1),S,R,S1).
trans(if(P,A1,A2),S,R,S1) :- trans((?(P) : A1) # (?(-P) : A2),S,R,S1).
trans(while(P,A),S,R,S1) :- trans(star(?(P) : A) : ?(-P),S,R,S1).
trans(A1 #= A2,S,R,S1) :- final(A1,S), !,
  (trans(A2,S,R,S1); trans(A1,S,R1,S1), R=(R1 #= A2)).
trans(A1 #= A2,S,R,S1) :- trans(A1,S,R1,S1), R=(R1 #= A2);
  trans(A2,S,R1,S1), R=(R1 #= A1).
trans(!(A),S, R #= !(A), S1) :- trans(A,S,R,S1).
trans(Proc,S,R,S1) :- proc(Proc,Body), !, trans(Body,S,R,S1).

/* Do(Prog,S,S1) */
```

```
trans*(A,S,A,S).
trans*(A,S,R,S1) :- trans(A,S,R2,S2), trans*(R2,S2,R,S1).

do(A,S,S1) :- trans*(A,S,R,S1), final(R,S1).


/* sub(Name,New,Term1,Term2): Term2 is Term1 with Name replaced by New. */

sub(X1,X2,T1,T2) :- var(T1), T2 = T1.
sub(X1,X2,T1,T2) :- not var(T1), T1 = X1, T2 = X2.
sub(X1,X2,T1,T2) :- not T1 = X1, T1 =..[F|L1], sub_list(X1,X2,L1,L2),
                    T2 =..[F|L2].
sub_list(X1,X2,[],[]).
sub_list(X1,X2,[T1|L1],[T2|L2]) :- sub(X1,X2,T1,T2), sub_list(X1,X2,L1,L2).

/* The holds predicate implements the revised Lloyd-Topor
   transformations on test conditions.  */

holds(S1 = S1,S,S).
holds( S1 << do(_,S1),S,S).
holds( S2 << do(_,S1),S,S) :- holds(S2 << S1,S,S).

holds(P & Q,S,S2) :- holds(P,S,S1), holds(Q,S1,S2).
holds(P v Q,S,S1) :- holds(P,S,S1); holds(Q,S,S1).
holds(P => Q,S,S1) :- holds(-P v Q,S,S1).
holds(P <=> Q,S,S1) :- holds((P => Q) & (Q => P),S,S1).
holds(-(-P),S,S1) :- holds(P,S,S1).
holds(-(P & Q),S,S1) :- holds(-P v -Q,S,S1).
holds(-(P v Q),S,S1) :- holds(-P & -Q,S,S1).
holds(-(P => Q),S,S1) :- holds(-(-P v Q),S,S1).
holds(-(P <=> Q),S,S1) :- holds(-((P => Q) & (Q => P)),S,S1).
holds(-all(V,P),S,S1) :- holds(some(V,-P),S,S1).
holds(-some(V,P),S,S1) :- not holds(some(V,P),S,S1).  /* Negation */
holds(-P,S,S1) :- isAtom(P), not holds(P,S,S1).     /* by failure */
holds(all(V,P),S,S1) :- holds(-some(V,-P),S,S1).
holds(some(V,P),S,S1) :- sub(V,_,P,P1), holds(P1,S,S1).

holds(somes(S1,SigmaS1 = S,P),S3,S4) :-
                   sub(S1,SNew,SigmaS1,SigmaS2), sub(S1,SNew,P,P1),
                   holds(SigmaS2=S,S3,S2), holds(P1,S2,S4).
holds(somes(S1,do(A,S1)=do(B,S),P),S3,S4) :- A = B,
                            holds(somes(S1,S1=S,P),S3,S4).

holds(somes(S1, SigmaS1 << do(A,S),P),S2,S3) :-
                          holds(somes(S1, SigmaS1 = S,P),S2,S3) ;
                          holds(somes(S1, SigmaS1 << S,P),S2,S3).

holds(somes(S1, SigmaS1 <<= S,P),S2,S3) :-
                         holds(somes(S1, SigmaS1 = S,P),S2,S3) ;
                         holds(somes(S1, SigmaS1 << S,P),S2,S3).

holds(alls(S1, SigmaS1 <<= S,P),S2,S3) :-
                        holds(somes(S1, SigmaS1 = S,P),S2,S3),
                        holds(somes(S1, SigmaS1 << S,P),S2,S3).

holds(-somes(S1,Bound,P),S2,S3) :- not holds(somes(S1,Bound,P),S2,S3).

holds(alls(S1,Bound,P),S2,S3) :- holds(-somes(S1,Bound,-P),S2,S3).


/* The following clause treats the holds predicate for non fluents, including
   Prolog system predicates. For this to work properly, the ConGolog programmer
   must provide, for all fluents, a clause giving the result of restoring
   situation arguments to situation-suppressed terms, for example:
         restoreSitArg(ontable(X),S,ontable(X,S)).             */

holds(A,S,S1) :- restoreSitArg(A,S,F), F, S1=do(A,S) ;
              not restoreSitArg(A,S,F), isAtom(A), A, S1 = S.

isAtom(A) :- not (A = -W ; A = (W1 & W2) ; A = (W1 => W2) ;
    A = (W1 <=> W2) ; A = (W1 v W2) ; A = some(X,W) ; A = all(X,W) ;
    A = (S1 = S2) ; A = (S1 << S2) ; A = (S1 <<= S2) ; A = somes(_,_,_) ;
    A = alls(_,_,_) ).

restoreSitArg(poss(A),S,poss(A,S)).

/* Transaction external actions */

primitive_action(begin(_)). primitive_action(commit(_)).
primitive_action(end(_)). primitive_action(rollback(_)).
```

```
externalAct(begin(T),T). externalAct(commit(T),T).
externalAct(end(T),T). externalAct(rollback(T),T).

termAct(commit(T),T). termAct(rollback(T),T).

systemAct(commit(T),T). systemAct(rollback(T),T).


/* Action precondition axioms for external actions. For this to work
   properly, the ConGolog programmer must provide, for all programs, a clause
   "ic(do(A,S))" giving the built-in integrity constraints and a clause
   "gic(S)" giving the generic integrity constraints. */

poss(begin(T),S) :-
    holds(-somes(s1, do(begin(T),s1)<<=S,true),S,_).
poss(end(T),S) :- running(T,S).
poss(commit(T),S) :-
   holds(somes(s1, do(end(T),s1)=S,true) & gic(S) &
         all(t1, sc_dep(T,t1,S) =>
           somes(s2, do(commit(t1),s2)<<=S,true)),S,_).
poss(rollback(T),S) :-
   holds(somes(s1, do(end(T),s1)=S,true) & -gic(S) v
     some(t1, somes(s2, r_dep(T,t1,S) & do(rollback(t1),s2)<<=S,true)),S,_).

running(T,do(A,S)) :- A = begin(T);
  running(T,S), not A = rollback(T), not A = commit(T).

/* Dependency axioms*/

r_dep(T,T1,S) :- transConflict(T,T1,S).

sc_dep(T,T1,S) :- readsFrom(T,T1,S).

transConflict(T,T1,do(A,S)) :- not T=T1, responsible(T1,A,S),
  holds(some(a1,somes(s1, do(a1,s1)<<do(A,S) & responsible(T,a1,S) &
                                          updConflict(a1,A,S),true)), S, _);
  transConflict(T,T1,S), not termAct(A,T).

readsFrom(T,T1,do(A,S)) :- readAct(A,F,T),
        holds(some(a1,somes(s1, do(a1,s1)<<=S, writes(a1,F,T1))), S, _).
```

```
               /* Notice that writes has 3 arguments here. We droped
                  the tuple argument to ease the simulations. */

/*  Utilities (from Reiter’s book).  */

prettyPrintSituation(S) :- makeActionList(S,Alist), nl, write(Alist), nl.

makeActionList(s0,[]).
makeActionList(do(A,S), L) :- makeActionList(S,L1), append(L1, [A], L).

exec(T) :- do(T,s0,S), prettyPrintSituation(S), askForMore.

askForMore :- write(’More? ’), read(n).
```

To illustrate our methodology, a Prolog implementation of the basic relational theory for the Debit/Credit example is given in Appendix D. Using that example, we get the following sample final situations:

```
[eclipse 3]: exec(begin(2):  execDebitCredit(2,b1,t1_3,a1,300) : end(2)#=
                 begin(1):  execDebitCredit(1,b1,t1_3,a1,300) : end(1)).

[begin(1), accounts(a1, b1, 1000, t1_1, 1), a_delete(a1, b1, 1000, t1_1, 1),
 a_insert(a1, b1, 1300, t1_1, 1), accounts(a1, b1, 1300, t1_1, 1),
 tellers(t1_3, 5000, 1), t_delete(t1_3, 5000, 1), t_insert(t1_3, 5300, 1),
 branches(b1, 10000, collegeStr, 1), b_delete(b1, 10000, collegeStr, 1),
 b_insert(b1, 10300, collegeStr, 1), end(1), begin(2), commit(1),
 accounts(a1, b1, 1300, t1_1, 2), a_delete(a1, b1, 1300, t1_1, 2),
 a_insert(a1, b1, 1600, t1_1, 2), accounts(a1, b1, 1600, t1_1, 2),
 tellers(t1_3, 5300, 2), t_delete(t1_3, 5300, 2), t_insert(t1_3, 5600, 2),
 branches(b1, 10300, collegeStr, 2), b_delete(b1, 10300, collegeStr, 2),
 b_insert(b1, 10600, collegeStr, 2), end(2), commit(2)]

[eclipse 4]: exec(begin(1):  execDebitCredit(1,b1,t1_3,a1,500) : end(1) #=
                 begin(2):  execDebitCredit(2,b1,t1_3,a1,300)   : end(2)).

[begin(1), accounts(a1, b1, 1000, t1_1, 1), a_delete(a1, b1, 1000, t1_1, 1),
 a_insert(a1, b1, 1500, t1_1, 1), accounts(a1, b1, 1500, t1_1, 1),
 tellers(t1_3, 5000, 1), t_delete(t1_3, 5000, 1), t_insert(t1_3, 5500, 1),
 branches(b1, 10000, collegeStr, 1), b_delete(b1, 10000, collegeStr, 1),
 b_insert(b1, 10500, collegeStr, 1), end(1), begin(2), commit(1),
 accounts(a1, b1, 1500, t1_1, 2), a_delete(a1, b1, 1500, t1_1, 2),
 a_insert(a1, b1, 1800, t1_1, 2), accounts(a1, b1, 1800, t1_1, 2),
 tellers(t1_3, 5500, 2), t_delete(t1_3, 5500, 2), t_insert(t1_3, 5800, 2),
 branches(b1, 10500, collegeStr, 2), b_delete(b1, 10500, collegeStr, 2),
 b_insert(b1, 10800, collegeStr, 2), end(2), commit(2)]

[eclipse 5]: exec(begin(2):  execDebitCredit(2,b1,t1_3,a1,300)   : end(2) #=
                 begin(1):  execDebitCredit(1,b1,t1_3,a1,500) : end(1)).

[begin(2), accounts(a1, b1, 1000, t1_1, 2), a_delete(a1, b1, 1000, t1_1, 2),
 a_insert(a1, b1, 1300, t1_1, 2), accounts(a1, b1, 1300, t1_1, 2),
 tellers(t1_3, 5000, 2), t_delete(t1_3, 5000, 2), t_insert(t1_3, 5300, 2),
 branches(b1, 10000, collegeStr, 2), b_delete(b1, 10000, collegeStr, 2),
 b_insert(b1, 10300, collegeStr, 2), end(2), begin(1), commit(2),
 accounts(a1, b1, 1300, t1_1, 1), a_delete(a1, b1, 1300, t1_1, 1),
 a_insert(a1, b1, 1800, t1_1, 1), accounts(a1, b1, 1800, t1_1, 1),
 tellers(t1_3, 5300, 1), t_delete(t1_3, 5300, 1), t_insert(t1_3, 5800, 1),
 branches(b1, 10300, collegeStr, 1), b_delete(b1, 10300, collegeStr, 1),
 b_insert(b1, 10800, collegeStr, 1), end(1), commit(1)]

[eclipse 6]: exec(begin(2):  execDebitCredit(2,b1,t1_3,a1,-9000)   : end(2)).

[begin(2), accounts(a1, b1, 1000, t1_1, 2), a_delete(a1, b1, 1000, t1_1, 2),
a_insert(a1, b1, -8000, t1_1, 2), accounts(a1, b1, -8000, t1_1, 2),
tellers(t1_3, 5000, 2), t_delete(t1_3, 5000, 2), t_insert(t1_3, -4000, 2),
branches(b1, 10000, collegeStr, 2), b_delete(b1, 10000, collegeStr, 2),
b_insert(b1, 1000, collegeStr, 2), end(2), rollback(2)]

[eclipse 7]: exec(begin(1):  execDebitCredit(1,b1,t1_3,a1,500) : end(1)).

[begin(1), accounts(a1, b1, 1000, t1_1, 1), a_delete(a1, b1, 1000, t1_1, 1),
a_insert(a1, b1, 1500, t1_1, 1), accounts(a1, b1, 1500, t1_1, 1),
tellers(t1_3, 5000, 1), t_delete(t1_3, 5000, 1), t_insert(t1_3, 5500, 1),
branches(b1, 10000, collegeStr, 1), b_delete(b1, 10000, collegeStr, 1),
b_insert(b1, 10500, collegeStr, 1), end(1), commit(1)]

[eclipse 8]: exec((begin(1): a_update(1,a1,-12000) : end(1)) #=
     (begin(2): a_update(2,a2,50) : end(2))).
```

```
[begin(1), accounts(a1, b1, 1000, t1_1, 1), a_delete(a1, b1, 1000, t1_1, 1),
 a_insert(a1, b1, -11000, t1_1, 1), begin(2), accounts(a2, b1, 100, t1_1, 2),
 a_delete(a2, b1, 100, t1_1, 2), a_insert(a2, b1, 150, t1_1, 2), end(2),
 rollback(2), end(1), commit(1)]

[eclipse 9]: exec((begin(2): a_update(2,a2,50) : end(2)) #=
                  (begin(1): a_update(1,a1,-12000) : end(1))).

[begin(2), accounts(a2, b1, 100, t1_1, 2), a_delete(a2, b1, 100, t1_1, 2),
 a_insert(a2, b1, 150, t1_1, 2), end(2), begin(1), commit(2),
 accounts(a1, b1, 1000, t1_1, 1), a_delete(a1, b1, 1000, t1_1, 1),
 a_insert(a1, b1, -11000, t1_1, 1), end(1), rollback(1)]
```

Notice that this interpreter is serializing transactions run concurrently. One way to force finer-grained interleavings is to randomly chose the actions to execute (e.g. [Gab02b]) as follows:

```
trans(A1 # A2,S,R,S1) :- frandom(N),
                         (N<0.5 ->trans(A1,S,R,S1) ; trans(A2,S,R,S1)) ;
                         (trans(A2,S,R,S1) ; trans(A1,S,R,S1)).

trans(A1 #= A2,S,A1r #= A2r,S1) :- frandom(N),
  (N=<0.5 -> ((trans(P1,S,P1r,Sr), P2r=P2) ; (trans(P2,S,P2r,Sr), P1r=P1)) ;
  ((trans(P2,S,P2r,Sr), P1r=P1) ; (trans(P1,S,P1r,Sr), P2r=P2))).
```

This however turns out to be inefficient. We leave this aspect out.

7.4 Illustrating the Methodology: A Further Example

This section introduces a portfolio management example ([CS94],[PD99]) to illustrate how to construct a normal program suitable for a SLDNF-based Prolog system according to the conditions of the Implementation Theorem for active relational theories. We assume the closed nested transations model in this example. The example is intended to illustrate the whole methodology for implementing active relational theories used as background theories for the ConGolog abstract interpreter given in Appendix C for which primitive actions are interpreted as shown in Section 6.4.1, and test actions executed using the predicate $Holds(\phi, s, s')$ given in Definitions 6.8 and 7.6.

Database Fluents:

$holder(holder\#, hname, country, value, t, s)$,

$stock(stock\#, sname, price, qty, t, s)$,
$owns(holder\#, stock\#, qty, t, s)$.

System Fluents:

$vseQuotation(stock\#, price, s), tseQuotation(stock\#, price, s)$.

Internal Actions:

$h\_insert(holder\#, hname, country, value, t), h\_delete(holder\#, hname, country, value, t)$,
$s\_insert(stock\#, sname, price, qty, t), s\_delete(stock\#, sname, price, qty, t)$,
$o\_insert(holder\#, stock\#, qty, t), o\_delete(holder\#, stock\#, qty, t)$,
$notify(stock\#)$.

Constants: Ray, $Iluju$, $Misha$, Ho, etc.

The fluent $holder(holder\#, hname, country, value, t, s)$ represents data about holders, individuals or companies, that owns stocks. Each stock holder has an identification number $holder\#$, a name $hname$, a country $country$, and a total value $value$ of owned stocks. All the stocks that are currently on the market are recorded in $stock(stock\#, sname, price, qty, t, s)$. Each stock has an identification number $stock\#$, a name $sname$, a current price $price$, and a quantity qty currently available. Finally, any holder $holder\#$ which owns a certain quantity qty of a given stock $stock\#$ is recorded in the fluent $owns(holder\#, stock\#, qty, t, s)$.

Unique Name Axioms are given in the usual way; thus we concentrate ourself on the remaining axioms.

Integrity Constraints: We enforce the following IC ($\in \mathcal{IC}_e$):

$$\begin{aligned}&holder(holder\#, hname, country, value, t, s) \wedge \\ &\qquad holder(holder\#, hname', country', value', t', s) \supset bid = bid', abal = abal', tid = tid',\end{aligned}$$

and similar ICs for $stock$ and $owns$; and we will verify the IC ($\in \mathcal{IC}_v$)

$$holder(holder\#, hname, country, value, t, s) \supset value \geq 0$$

at transaction's end.

Update Precondition Axioms:

$$\begin{aligned}&Poss(h\_insert(holder\#, hname, country, value, t), s) \equiv \\ &\qquad \neg(\exists t')holder(holder\#, hname, country, value, t', s) \wedge \\ &\qquad IC_e(do(h\_insert(holder\#, hname, country, value, t), s)) \wedge running(t, s),\end{aligned} \tag{7.19}$$

$$\begin{aligned}&Poss(h\_delete(holder\#, hname, country, value, t), s) \equiv \\ &\qquad (\exists t')holder(holder\#, hname, country, value, t', s) \wedge \\ &\qquad IC_e(do(h\_delete(holder\#, hname, country, value, t), s)) \wedge running(t, s).\end{aligned} \tag{7.20}$$

Similar precondition axioms are given for the remaining internal actions. Here, we do not avoid cascading rollbacks.

Successor State Axioms:

$$\begin{array}{l} holder(holder\#, hname, country, value, t, do(a,s)) \equiv \\ \quad (\exists t_1)a = h\_insert(holder\#, hname, country, value, t_1) \lor \\ \quad (\exists t_2)holder(holder\#, hname, country, value, t_2, s) \land \\ \quad \neg(\exists t_3)a = h\_delete(holder\#, hname, country, value, T_3)) \land \neg(\exists t')a = Rollback(t') \lor \\ \quad (\exists t').a = Rollback(t') \land \neg(\exists t'')parent(t'', t', s) \land \\ \qquad\qquad restoreBeginPoint(holder, (holder\#, hname, country, value), t', s) \lor \\ \quad a = Rollback(t') \land (\exists t'')parent(t'', t', s) \land \\ \qquad\qquad restoreSpawnPoint(holder(holder\#, hname, country, value), t', s). \end{array}$$

Similar successor state axioms are given for the remaining database fluents. Precondition axioms for external actions are those given in Theorem 7.11, taking into account the action $Spawn(t, t')$ (See Section 7.1.7). Axioms for dependency predicates are the ones given in Theorem 7.21.

The following is an example of an initial database for the portfolio domain:

$$\begin{array}{l} holder(h\_id, hn, c, val, t, S_0) \equiv h\_id = C_1 \land hn = Smith \land c = Canada \land val = \$30000 \lor \\ \qquad h\_id = C_2 \land hn = diouf \land c = Senegal \land val = \$20000 \lor \\ \qquad h\_id = C_3 \land hn = Brown \land c = USA \land val = \$9000, \end{array}$$

$$\begin{array}{l} stock(s\_id, sn, pr, qty, S_0) \equiv s\_id = ST_1 \land sn = IBM \land pr = \$100 \land qty = 5000 \lor \\ \qquad s\_id = ST_2 \land sn = ORACLE \land pr = \$30 \land qty = 1000, \end{array}$$

$$\begin{array}{l} owns(h\_id, s\_id, qty, t, S_0) \equiv h\_id = C_1 \land s\_id = IBM \land qty = 300 \lor \\ \qquad h\_id = C_2 \land s\_id = IBM \land qty = 200 \lor \\ \qquad h\_id = C_3 \land s\_id = ORACLE \land qty = 300, \end{array}$$

$$\begin{array}{l} vseQuotation(IBM, 110, S_0), vseQuotation(GM, 10, S_0), \\ tseQuotation(ORACLE, 50, S_0), tseQuotation(FORD, 40, S_0). \end{array}$$

The fluent $vseQuotation(Sid, Pr, s)$ and $tseQuotation(Sid, Pr, s)$ have the successor state axioms

$$\begin{array}{l} vseQuotation(Sid, Pr, do(a,s)) \equiv vseQuotation(Sid, Pr, s), \\ tseQuotation(Sid, Pr, do(a,s)) \equiv tseQuotation(Sid, Pr, s). \end{array}$$

The fluent $served(id, s)$ introduced in Section refSimulationWellFormedProg will be used for synchonization. The action $notify(stock\#)$, whose precondition axiom is $Poss(notify(stock\#), s) \equiv true$, is used to indicate that the price of the stock $stock\#$ has been updated. It's effect is to make the fluent $served(Sid, s)$ true.

For the database fluent $owns(h\_id, s\_id, qty, t, s)$, the transition and event fluents are:

$$\begin{aligned} &owns\_inserted(r, h\_id, s\_id, qty, t, s) \equiv \\ &\quad (\exists a, s').s \neq S_0 \wedge s = do(a, s') \wedge considered(r, t, s') \wedge \\ &\quad (\exists t')a = owns\_insert(h\_id, s\_id, qty, t') \vee \\ &\quad F\_inserted(r, h\_id, s\_id, qty, t, s) \wedge a \neq owns\_delete(h\_id, s\_id, qty, t), \end{aligned} \tag{7.21}$$

$$\begin{aligned} &owns\_inserted(r, t, s) \equiv \\ &\quad (\exists h\_id, s\_id, qty, a, s', t').s \neq S_0 \wedge s = do(a, s') \wedge \\ &\quad a = owns\_insert(, h\_id, s\_id, qty, t') \wedge considered(r, t, s') \vee F\_inserted(r, t, s). \end{aligned} \tag{7.22}$$

The following sentences are all of the form $W \supset A$ and obtained using the revised Lloyd-Topor transformations applied to the if-halves of the sample axioms given above.

$$\begin{aligned} &\neg(\exists t')holder(holder\#, hname, country, value, t', s) \wedge \\ &\quad IC_e(do(h\_insert(holder\#, hname, country, value, t), s)) \wedge running(t, s) \supset \\ &\qquad Poss(h\_insert(holder\#, hname, country, value, t), s), \end{aligned} \tag{7.23}$$

$$\begin{aligned} &\{(\exists t_1)a = h\_insert(holder\#, hname, country, value, t_1) \vee \\ &\quad (\exists t_2)holder(holder\#, hname, country, value, t_2, s) \wedge \\ &\quad \neg(\exists t_3)a = h\_delete(holder\#, hname, country, value, T_3)) \wedge \neg(\exists t')a = Rollback(t') \vee \\ &(\exists t').a = Rollback(t') \wedge \neg(\exists t'')parent(t'', t', s) \wedge \\ &\quad (\exists s^*)Holds(restoreBeginPoint(holder, (holder\#, hname, country, value), t', s), s, s^*) \vee \\ &a = Rollback(t') \wedge (\exists t'')parent(t'', t', s) \wedge \\ &\quad (\exists s^*)Holds(restoreSpawnPoint(holder(holder\#, hname, country, value), t', s), s, s^*)\} \supset \\ &\qquad holder(holder\#, hname, country, value, t, do(a, s)) \end{aligned} \tag{7.24}$$

$$\begin{aligned} &vseQuotation(stock\#, price, s) \wedge a \neq notify(stock\#) \supset \\ &\qquad vseQuotation(stock\#, price, do(a, s)), \end{aligned} \tag{7.25}$$

$$\begin{aligned} &vseQuotation(stock\#, price, s) \wedge a \neq notify(stock\#) \supset \\ &\qquad tseQuotation(stock\#, price, do(a, s)), \end{aligned} \tag{7.26}$$

$$\begin{aligned}&(\exists a, s').s \neq S_0 \wedge s = do(a, s') \wedge considered(r, t, s') \wedge \\ &\quad (\exists t')a = owns\_insert(h\_id, s\_id, qty, t') \vee \\ &\quad F\_inserted(r, h\_id, s\_id, qty, t, s) \wedge a \neq owns\_delete(h\_id, s\_id, qty, t) \supset \\ &\qquad\qquad owns\_inserted(r, h\_id, s\_id, qty, t, s),\end{aligned} \tag{7.27}$$

$$\begin{aligned}&(\exists h\_id, s\_id, qty, a, s', t').s \neq S_0 \wedge s = do(a, s') \wedge \\ &\quad a = owns\_insert(, h\_id, s\_id, qty, t') \wedge considered(r, t, s') \vee F\_inserted(r, t, s) \supset \\ &\qquad\qquad owns\_inserted(r, t, s).\end{aligned} \tag{7.28}$$

Notice the use of the predicate $Holds(\phi, s, s')$ in the if-half of the successor state axiom for the fluent $holder(holder\#, hname, country, value, t, s)$: it is used to express the Lloyd-Topor transformation of a subformula that in fact is an abbreviation standing for a non-Markovian formula. This is a methodological choice that we make in transforming any given formula ϕ into a Lloyd-Topor form: if ϕ has the form of any of the formulas abbreviated in Definition 7.1, we will always treat it by using the predicate $Holds(\phi, s, s')$ given in Definition 7.1.

Now we give the following sample ConGolog procedures which are well-formed to update the different fluents of our example:

```
proc $hValueUpdate(t, holder\#, amount)$
 $(\pi\ hname, country, value, value')[holder(holder\#, hname, country, value, t)?$ ;
  $[value' = value + amount]?$ ; $h\_delete(holder\#, hname, country, value, t)$ ;
    $h\_insert(holder\#, hname, country, value', t)]$
endProc
proc $sQtyUpdate(t, stock\#, amount)$
 $(\pi\ sname, price, qty, qty')[stock(stock\#, sname, price, qty, t)?$ ;
  $[qty' = qty + amount]?$ ; $s\_delete(stock\#, sname, price, qty, t)$ ;
    $s\_insert(stock\#, sname, price, qty', t)]$
endProc
proc $tQtyUpdate(t, holder\#, stock\#, amount)$
 $(\pi\ qty, qty')[owns(holder\#, stock\#, qty, t)?$ ;
  $[qty' = qty + amount]?$ ; $o\_delete(holder\#, stock\#, qty, t)$ ;
    $o\_insert(holder\#, stock\#, qty', t)]$
endProc
```

```
**proc** $processQuotations(t, se)$
    $Begin(t);$
      $[(\pi\ stock\#, price).$
       **If** $se = TSE$ **then**
         $Spawn(t, stock\#)$ ;
         $(\pi newPrice)[tseQuotation(stock\#, newPrice)?$ ;
                   $sPriceUpdate(t, stock\#, newPrice)]$ ; $notify(stock\#)$ ;
         $End(stock\#)$
       **else**
         $Spawn(t, stock\#)$ ;
         $(\pi newPrice)[vseQuotation(stock\#, price)?$ ;
                   $sPriceUpdate(t, stock\#, newPrice)]$ ; $notify(stock\#)$ ;
         $End(stock\#)]^{||}$ ;
      $\neg[(\exists\ stock\#, price).\ (vseQuotation(stock\#, price) \vee$
                   $tseQuotation(stock\#, price)) \wedge \neg served(sid)]?$ ;
    $End(t)$
**endProc**
```

The given procedures $hValueUpdate(t, holder\#, amount)$, $sQtyUpdate(t, stock\#, amount)$, and $tQtyUpdate(t, holder\#, stock\#, amount)$ update the value, and quantity by the specified amount for the fluents $holder$, and $stock$ and $owns$, respectively. Further procedures for updating the values of the other arguments may be given in a similar way. Notice what the procedure $processQuotations(t, se)$ does: for any given stock exchange se, the procedure checks whether se is the Toronto or the Vancouver stock exchange, after which it spawns a child transaction for each of the stocks quoted on the stock exchange. Children are spawned in parallel. The child transaction updates the stock price according to the available quotation for that stock. The update process continues until no more quotations are left.

Below are a few sample ECA rules for the portfolio example of the last section.

$$\begin{array}{l} <trans\ :\ Rule_1\ :\ stock\_inserted\ : \\ \quad (\exists pr, sn, qty_1)[stock\_inserted(st\#, sn, pr, qty_1) \wedge \\ \qquad owns(h\#, st\#, qty_2) \wedge pr = 0] \\ \quad \rightarrow \\ \quad owns\_delete(h\#, st\#, qty_2) > \end{array}$$

$<trans\ :\ Rule_2\ :\ stock\_deleted\ :$

$(\exists pr, sn, qty_1) stock\_deleted(st\#, sn, pr, qty_1)$

$\rightarrow$

$(\pi\ h\#, qty_2)[owns(h\#, st\#, qty_2)?\ ;\ owns\_delete(h\#, st\#, qty_2)\ >$

$<trans\ :\ Rule_3\ :\ owns\_inserted\ :$

$(\exists\ h\#, st\#, qty_1) owns\_inserted(h\#, st\#, qty_1)$

$\rightarrow$

$(\pi\ amt, sn, pr, h\#, st\#, qty_1, qty_2)[stock(st\#, sn, pr, h\#, st\#, qty_1, qty_2)?\ ;\ (amt = pr * qty_1)?\ ;$

$hValueUpdate(trans, h\#, amt)\ ;\ sQtyUpdate(trans, st\#, -qty_1)\ \ >$

The first rule captures the behavior that, whenever there is an insertion into the relation $stock$, if there is any stock whose price is zero and that has been inserted into the relation $stock$, anyone who owns that stock must be deleted from the relation $owns$. The rationale behind this rule is that any stock with a null price is worthless, and therefore should not be kept by anyone. The second rule captures the behavior that, whenever there is a deletion from the relation $stock$, if there is a stock that has been deleted from the relation $stock$, then check whether there is anyone who owns that stock and delete the tuple corresponding to him from the relation $owns$. The third rule captures the behavior that, whenever there is an insertion into the relation $owns$, if there is a new onwer that has been inserted into $owns$, then check the price of the stocks that he owns in the relation $stock$ and update both the value of his portfolio in the relation $holder$ and the remaining quantity of stocks in the relation $stock$ accordingly.

The if-sentences given above are in a form readily expressible in Prolog. The entire program can be given in the same way as the Debit/Credit example given in Appendix D.

The interpreter for running rule programs is different than the one given above for basic action theories in the way it handles primitive actions. To modify the interpreter, replace the clause

```
trans(A,S,R,S1) :- primitive_action(A), poss(A,S), !, R=nil, S2 = do(A,S),
              (systemAct(A1,T), poss(A1,S2), S1 = do(A1,S2) ;
               not (systemAct(A2,T2), poss(A2,S)),S1 = S2).
```

by the following:

```
trans(A,S,nil,S1) :- primitive_action(A), transOf(A,T), poss(A,S), S2=do(A,S),
               (systemAct(A1,T), poss(A1,S2), S1 = do(A1,S2) ;
                not (systemAct(A2,T), poss(A2,S2)),do(rules(T),S2,S1)).
```

Because of the new predicate `transOf`, taking an action and a transaction as arguments, augment the interpreter by the clauses:

```
transOf(begin(T),T). transOf(end(T),T).
transOf(commit(T),T). transOf(rollback(T),T).
```

Next, we need to give axioms characterizing the system fluent $considered$ introduced earlier in Chapter5. To do so, we introduce two primitive actions $beginCon(rid, t)$ and $stopCons(rid, t)$ where rid and t are a rule identification name and a transaction, respectively. These actions are used for bookkeeping and their action precondition axioms are:

$$Poss(beginCons(rid, t), s) \equiv \neg considered(rid, t, s),$$
$$Poss(stopCons(rid, t), s) \equiv considered(rid, t, s).$$

The successor state axiom for $considered$ used in the interpreter is

$$considered(rid, t, do(a, s)) \equiv a = beginCons(rid, t).$$

Initialy, we have $(\forall r, t) considered(r, t, S_0)$. Intuitively, $considered(r, t, s)$ means the most recent situations in which rule r is considered for execution is s with respect to transaction t. This is a semantics usually found in ADBMS, e.g. in Starburst.

Below, we give the non-Markovian ConGolog interpreter written in Prolog for well formed programs that have active relational theories as background axioms. This interpreter is based on the same assumptions as the one in Section 7.3.

```
/* A non-Markovian ConGolog interpreter for programs with ARTs. */

:- set_flag(print_depth,100).
:- nodbgcomp.
:- dynamic(proc/2).            /* Compiler      */
:- set_flag(all_dynamic, on).  /* directives    */

:- op(750, xfy, [<<]).  /* Subsequence */
:- op(750, xfy, [<<=]). /* Subsequence and equality */
:- op(800, xfy, [&]).   /* Conjunction */
:- op(850, xfy, [v]).   /* Disjunction */
:- op(870, xfy, [=>]).  /* Implication */
:- op(880, xfy, [<=>]). /* Equivalence */
:- op(890, xfy, [#=]).  /* concurrency */
:- op(950, xfy, [:]).   /* Action sequence */
:- op(960, xfy, [#]).   /* Nondeterministic action choice */

/* final(Prog,S) */

final(nil,S).
final(A1 : A2,S) :- final(A1,S), final(A2,S).
final(A1 # A2,S) :- final(A1,S); final(A2,S).
final(pi(V,A),S) :- sub(V,_,A,A1),final(A1,S).
final(star(_),_).
final(if(P,A1,A2),S) :- holds(P,S,_), !, final(A1,S); final(A2,S).
final(while(P,A),S) :- holds(P,S,_), !, final(A,S).
final(while(_,_),_).
final(A1 #= A2,S) :- final(A1,S), final(A2,S).
final(!(A),S).

/* trans(Prog,S,ProgRest, S) */

trans(A,S,nil,S1) :- primitive_action(A), transOf(A,T), poss(A,S), S2=do(A,S),
               (systemAct(A1,T), poss(A1,S2), S1 = do(A1,S2) ;
                not (systemAct(A2,T), poss(A2,S2)), do(rules(T),S2,S1)).
trans(?(P),S,nil,S1) :- holds(P,S,S1).
trans(A1 : A2,S,R,S1) :- final(A1,S), trans(A2,S,R,S1).
trans(A1 : A2,S,R1:A2,S1) :- trans(A1,S,R1,S1).
trans(A1 # A2,S,R,S1) :- trans(A1,S,R,S1); trans(A2,S,R,S1).
trans(pi(V,A),S,R,S1) :- sub(V,_,A,A1), trans(A1,S,R,S1).
trans(star(A),S,R : star(A),S1) :- trans(A,S,R,S1).
trans(if(P,A),S,R,S1) :- trans((?(P) : A1),S,R,S1).
trans(if(P,A1,A2),S,R,S1) :- trans((?(P) : A1) # (?(-P) : A2),S,R,S1).
trans(while(P,A),S,R,S1) :- trans(star(?(P) : A) : ?(-P),S,R,S1).
trans(A1 #= A2,S,R,S1) :- final(A1,S), !,
  (trans(A2,S,R,S1); trans(A1,S,R1,S1), R=(R1 #= A2)).
trans(A1 #= A2,S,R,S1) :- trans(A1,S,R1,S1), R=(R1 #= A2);
  trans(A2,S,R1,S1), R=(R1 #= A1).
trans(!(A),S, R #= !(A), S1) :- trans(A,S,R,S1).
trans(Proc,S,R,S1) :- proc(Proc,Body), !, trans(Body,S,R,S1).

trans*(A,S,A,S).
trans*(A,S,R,S1) :- trans(A,S,R2,S2), trans*(R2,S2,R,S1).

do(A,S,S1) :- trans*(A,S,R,S1), final(R,S1).


/* sub(Name,New,Term1,Term2): Term2 is Term1 with Name replaced by New. */

sub(X1,X2,T1,T2) :- var(T1), T2 = T1.
sub(X1,X2,T1,T2) :- not var(T1), T1 = X1, T2 = X2.
sub(X1,X2,T1,T2) :- not T1 = X1, T1 =..[F|L1], sub_list(X1,X2,L1,L2),
                    T2 =..[F|L2].
sub_list(X1,X2,[],[]).
sub_list(X1,X2,[T1|L1],[T2|L2]) :- sub(X1,X2,T1,T2), sub_list(X1,X2,L1,L2).

/* The holds predicate implements the revised Lloyd-Topor
   transformations on test conditions.  */

holds(S1 = S1,S,S).
holds( S1 << do(_,S1),S,S).
holds( S2 << do(_,S1),S,S) :- holds(S2 << S1,S,S).

holds(P & Q,S,S2) :- holds(P,S,S1), holds(Q,S1,S2).
holds(P v Q,S,S1) :- holds(P,S,S1); holds(Q,S,S1).
holds(P => Q,S,S1) :- holds(-P v Q,S,S1).
holds(P <=> Q,S,S1) :- holds((P => Q) & (Q => P),S,S1).
holds(-(-P),S,S1) :- holds(P,S,S1).
holds(-(P & Q),S,S1) :- holds(-P v -Q,S,S1).
holds(-(P v Q),S,S1) :- holds(-P & -Q,S,S1).
holds(-(P => Q),S,S1) :- holds(-(-P v Q),S,S1).
holds(-(P <=> Q),S,S1) :- holds(-((P => Q) & (Q => P)),S,S1).
holds(-all(V,P),S,S1) :- holds(some(V,-P),S,S1).
holds(-some(V,P),S,S1) :- holds(some(V,P),S,S1), fail, !. /* Negation */
holds(-some(V,P),S,S1) :- S = S1.                          /*          */
holds(-P,S,S1) :- isAtom(P), holds(P,S,S1), fail, !.      /* by       */
holds(-P,S,S1) :- isAtom(P), S = S1.                       /* failure  */
holds(all(V,P),S,S1) :- holds(-some(V,-P),S,S1).
holds(some(V,P),S,S1) :- sub(V,_,P,P1), holds(P1,S,S1).

holds(somes(S1,SigmaS1 = S,P),S3,S4) :-
                     sub(S1,SNew,SigmaS1,SigmaS2), sub(S1,SNew,P,P1),
                     holds(SigmaS2=S,S3,S2), holds(P1,S2,S4).

holds(somes(S1,do(A,S1)=do(B,S),P),S3,S4) :- A = B,
                                       holds(somes(S1,S1=S,P),S3,S4).

holds(somes(S1, SigmaS1 << do(A,S),P),S2,S3) :-
holds(somes(S1, SigmaS1 = S,P),S2,S3) ;
holds(somes(S1, SigmaS1 << S,P),S2,S3).

holds(somes(S1, SigmaS1 <<= S,P),S2,S3) :-
holds(somes(S1, SigmaS1 = S,P),S2,S3) ;
holds(somes(S1, SigmaS1 << S,P),S2,S3).

holds(alls(S1, SigmaS1 <<= S,P),S2,S3) :-
holds(somes(S1, SigmaS1 = S,P),S2,S3) ,
holds(somes(S1, SigmaS1 << S,P),S2,S3).

holds(-somes(S1,Bound,P),S2,S3) :- holds(somes(S1,Bound,P),S2,S3), fail, !.
holds(-somes(S1,Bound,P),S2,S3) :- S2 = S3.

holds(alls(S1,Bound,P),S2,S3) :- holds(-somes(S1,Bound,-P),S2,S3).


/* The following clause treats the holds predicate for non fluents, including
   Prolog system predicates. For this to work properly, the ConGolog programmer
   must provide, for all fluents, a clause giving the result of restoring
   situation arguments to situation-suppressed terms, for example:
         restoreSitArg(ontable(X),S,ontable(X,S)).             */

holds(A,S,S1) :- db_fluent(A), restoreSitArg(A,S,F), S1=do(A,S), F.
holds(A,S,S) :- not db_fluent(A), restoreSitArg(A,S,F), F.
holds(A,S,S) :- not restoreSitArg(A,S,F), isAtom(A), A.

isAtom(A) :- not (A = -W ; A = (W1 & W2) ; A = (W1 => W2) ;
    A = (W1 <=> W2) ; A = (W1 v W2) ; A = some(X,W) ; A = all(X,W) ;
    A = (S1 = S2) ; A = (S1 << S2) ; A = (S1 <<= S2) ;
    A = somes(_,_,_) ; A = alls(_,_,_) ).

restoreSitArg(poss(A),S,poss(A,S)).

/* Transaction external actions */

primitive_action(begin(_)). primitive_action(commit(_)).
primitive_action(end(_)). primitive_action(rollback(_)).

externalAct(begin(T),T). externalAct(commit(T),T).
externalAct(end(T),T). externalAct(rollback(T),T).

termAct(commit(T),T). termAct(rollback(T),T).

systemAct(commit(T),T). systemAct(rollback(T),T).
```

```
transOf(begin(T),T). transOf(end(T),T).
transOf(commit(T),T). transOf(rollback(T),T).

/* Action precondition axioms for external actions. For this to work
   properly, the ConGolog programmer must provide, for all programs, a clause
   "ic(do(A,S))" giving the built-in integrity constraints and a clause
   "gic(S)" giving the generic integrity constraints. */

poss(begin(T),S) :-
    holds(-somes(s1, do(begin(T),s1)<<=S,true),S,_).
poss(end(T),S) :- running(T,S).
poss(commit(T),S) :-
   holds(somes(s1, do(end(T),s1)=S,true) & gic(S) &
         all(t1, sc_dep(T,t1,S) =>
           somes(s2, do(commit(t1),s2)<<=S,true)),S,_).
poss(rollback(T),S) :-
   holds(somes(s1, do(end(T),s1)=S,true) & -gic(S) v
     some(t1, somes(s2, r_dep(T,t1,S) & do(rollback(t1),s2)<<=S,true)),S,_).

running(T,do(A,S)) :- A = begin(T);
  running(T,S), not A = rollback(T), not A = commit(T).

/* Dependency axioms*/

r_dep(T,T1,S) :- transConflict(T,T1,S).

sc_dep(T,T1,S) :- readsFrom(T,T1,S).

transConflict(T,T1,do(A,S)) :- not T=T1, responsible(T1,A,S),
  holds(some(a1,somes(s1, do(a1,s1)<<do(A,S) & responsible(T,a1,S) &
                                          updConflict(a1,A,S),true)), S, _);
  transConflict(T,T1,S), not termAct(A,T).

readsFrom(T,T1,do(A,S)) :- readAct(A,F,T),
        holds(some(a1,somes(s1, do(a1,s1)<<=S, writes(a1,F,T1))), S, _).

/* Bookkeeping actions and fluents */

primitive_action(beginCons(_,_)).
primitive_action(stopCons(_,_)).
```

```
poss(beginCons(Rid,T),S) :- not considered(Rid,T,S).
poss(stopCons(Rid,T),S) :- considered(Rid,T,S).

considered(Rid,T,do(A,S)) :-  A=beginCons(Rid,T);
                            considered(Rid,T,S), not A=stopCons(Rid,T).

transOf(stopCons(_,T), T).
transOf(beginCons(_,T), T).

/*  Utilities. */

prettyPrintSituation(S) :- makeActionList(S,Alist), nl, write(Alist), nl.

makeActionList(s0,[]).
makeActionList(do(A,S), L) :- makeActionList(S,L1), append(L1, [A], L).

exec(T) :- do(T,s0,S), prettyPrintSituation(S), askForMore.

askForMore :- write('More? '), read(n).


/*  Here ends the non-Markovian ConGolog interpreter. */
```

Using the Prolog axioms obtained from the present section as background theory, the ConGolog interpreter for active relational theories runs the procedures above, and, doing so, simulates the portfolio domain. Recall that, after each execution of a primitive action, the interpreter calls a rule procedure that is an appropriate compilation of the ECA rules given above. For simplicity and shortness, a sample program that uses flat transactions is given in Appendix D. Some sample runs are[5]:

```
[eclipse 2]: exec(begin(1): s_insert(st5,opel,0,200,1) :
        s_insert(st4,ford,0,1000,1) :  o_qty_update(1,c3,st2,200) : end(1)).

[begin(1), s_insert(st5, opel, 0, 200, 1), s_insert(st4, ford, 0, 1000, 1),
owns(c3, st4, 300, 1), stopCons(rule1, 1), beginCons(rule1, 1),
o_delete(c3, st4, 300, 1), owns(c3, st2, 300, 1), o_delete(c3, st2, 300, 1),
o_insert(c3, st2, 500, 1), end(1), commit(1)]
```

[5]Of course this interpreter can still do everything that the one in Section 7.3 does.

```
More?   n.

yes.

[eclipse 3]: exec(begin(1): o_qty_update(1,c3,st2,200) :
               s_insert(st4,ford,0,1000,1) : end(1)).

[begin(1), owns(c3, st2, 300, 1), o_delete(c3, st2, 300, 1),
o_insert(c3, st2, 500, 1), s_insert(st4, ford, 0, 1000, 1),
owns(c3, st4, 300, 1), stopCons(rule1, 1), beginCons(rule1, 1),
o_delete(c3, st4, 300, 1), end(1), commit(1)]
More?   n.

yes.
```

7.5 SQL3

In the introduction of this thesis, we have pointed out the fact that the existing ADBMSs support active rules in the form of triggers. However, these systems have each their own knowledge and execution models. SQL3 has been proposed as a standard providing a common and consistent support for active rules. Figure 7.1 and Figure 7.2 indicate how SQL3 fits into the classification given in Figure 2.2 and Figure 2.3, following the description of this emerging standard in [KMC99].

In this section, we apply our framework to represent SQL3. we will not specify the whole standard. We will restrict ourselves to some of the most saillant aspects of SQL3 instead. This aspects are AFTER/BEFORE trigger activation, event source and granularity, and action interruptability. We will also see how these aspects affect the execution model of SQL3. The simplicity of SQL3, which is not due to their essence, makes them amenable to a rapid prototyping using our logical framework. In what follows, we will summarize the dimensions of active behavior of SQL3 using tables comparable to those used in Figure 2.2 and Figure 2.3.

7.5.1 Knowledge and Execution Model

Knowledge Model

The SQL3 standard captures active behavior using two kinds of constructs:

Constraints: These are predicates defined on database states. They are defined on one single relation, on attributes of a relation, or on multiple relations. The ADBMS must ensure that they are evaluated to $true$ either immediately after each SQL statement execution (immediate mode) or at the end of the user transaction, but just before this commits (deferred mode). A violation of a constraint results in a rolling back of the faulty statement in the case of immediate constraints, or the entire transaction in the case of deferred constraints. In SQL3 there are various sorts of both relation and attribute constraints which globally can be charaterized as the built-in and cardinality integrity constraints of Section 2.1 when they are assigned an immediate mode, and as generic integrity constraints when they are assigned a deferred mode.

Triggers: These are named ECA rules that are each associated with a particular database relation. Thus a trigger has four syntactical parts: a subject relation, on which it is defined, an event part, an optional condition part, and an action part. Appendix F gives the current EBNF syntax of SQL3 and Figure 7.1 gives details of the dimensions of ECA rules in SQL3 according to the classification method of Figure 2.2, and we now turn our attention to these dimensions.

- **AFTER/BEFORE Trigger Activation**. One important issue regarding the syntax of SQL3 triggers is the distinction made between BEFORE and AFTER triggers which are syntactically distinguished by the keywords ”BEFORE” and ”AFTER”. A BEFORE (AFTER) trigger is activated before (after) the database operation associated with the event mentioned in its event part. Both forms of triggers are a SQL3 implementation of the idea of the event/action link which is the connection between the action execution and the triggering action (See Section 2.1): in BEFORE triggers, actions specified in the action part of the trigger are to execute **instead of** the action associated with the event part of the trigger.

- **Event Source and Granularity**. All events are primitive in SQL3 and are one of INSERT, DELETE, or UPDATE. SQL3 allows two levels of granularity of triggers: a tuple-oriented granularity, specified by the keywords FOR EACH ROW, and a set-oriented granularity, specified by the keywords FOR EACH STATEMENT.

- **Action Interruptability**. Trigger actions may be interruptable in order for other triggered rules to be activated or not. An non-interruptable trigger action part is enclosed between keywords BEGIN ATOMIC and END.

SQL3 triggers can access the current state of a database, the old state of the database, and the old and new values of affected tuples. All of these four tables are made accessible to the trigger condition and action parts by explicitly naming them in a REFERENCING clause in the trigger definition. The old and new values are referred to by means of two transition tables identified by the key words OLD AS and NEW AS, respectively. The old state of the database is identified by the key word OLD_TABLE AS in the list of aliases that follow the key word REFERENCING. The new database state that is obtained after the modification brought about by the triggering operation is applied to the database is identified by the key word NEW_TABLE AS.

Finally, SQL3 allows an order of trigger execution that corresponds to the ascending creation times of triggers. This means that a newly created trigger has execution precedence over all the other triggers already created.

COMPONENT	DIMENSIONS	VALUES IN SQL3
EVENT	TYPE	primitive
	SOURCE	update operation
	GRANULARITY	tuple- or set-oriented
	ROLE	mandatory
CONDITION	ROLE	optional
	CONTEXT	$BIND_E$, DB_E, DB_A
ACTION	TYPE	update operation, behavior invocation, do instead
	CONTEXT	$BIND_E$, DB_E, DB_A

Figure 7.1: Overview of dimensions of the knowledge model of SQL3

Execution Model

In SQL3, there are no explicit language constructs for ascribing priorities to triggers. Triggers are prioritized according to the ascending order of their creation times instead.

Normally, the execution model of SQL3 triggers must satisfy at least two requirements:

1. Trigger actions must always be executed in consistent database states.

2. All BEFORE triggers must be entirely executed before the database operations associated with the events mentioned in their event parts are executed; and All AFTER triggers must be entirely executed after the execution of the database operations associated with their event parts.

The first requirement is fullfilled by checking constraints every time the database is updated. The second requirement is met by executing any BEFORE trigger prior to any update to the database, and by executing any AFTER trigger after all updates caused by triggers of higher priority and by the operation associated with the event mentioned in the event part of that AFTER trigger have been executed.

As of the time of their 1999 writing, the authors of [KMC99], who are involved in the proposal of the SQL3 standard, did not restrict the use of BEFORE rules. It is however usually assumed that BEFORE rules are to be used in a way such that their actions do not update the database. Such restrictions do not exist in the standard itself, but is planned for the future ([KMC99]). In consequence, this assumption ensures that the database state seen by the BEFORE triggers is a state that is guaranteed to be consistent by the first requirement.

DIMENSIONS	VALUES IN SQL3
EVENT/CONDITION COUPLING	immediate
CONDITION/ACTION COUPLING	immediate
CONSUMPTION MODE	none
NET EFFECT POLICY	no
PRIORITY	creation time
SCHEDULING	sequential
ERROR HANDLING	backtrack

Figure 7.2: Overview of dimensions of the execution model of SQL3

7.5.2 Formalization

This section formalizes the main dimensions of SQL3 described above.

Knowledge Model

Primitive Events. As in the general framework, for each database fluent F, we introduce primitive event fluents $F\_inserted(r,t,s)$ and $F\_deleted(r,t,s)$. Though no consumption mode is officially available for SQL3, an implicit local consumption mode seems to be assumed for the primitive events of SQL3. Since SQL3 does not have any complex events, we are only left with primitive events fluents $F\_inserted(r,t,s)$ and $F\_deleted(r,t,s)$ with successor state axioms of the kind given in (5.13) and (5.13):

$$\begin{aligned} F\_inserted(r,t,do(a,s)) \equiv (\exists \vec{x},t')a = F\_insert(\vec{x},t') \wedge considered(r,t,s) \vee \\ F\_inserted(r,t,s) \wedge \neg(\exists \vec{y},t')a = F\_insert(\vec{y},t'), \end{aligned} \tag{7.29}$$

$$\begin{aligned} F\_deleted(r,t,do(a,s)) \equiv (\exists \vec{x},t')a = F\_delete(\vec{x},t') \wedge considered(r',t,s) \vee \\ F\_deleted(r,t,s) \wedge \neg(\exists \vec{y},t')a = F\_delete(\vec{y},t'). \end{aligned} \tag{7.30}$$

Tuple- versus Set-oriented Event Granularity. In order to account for the tuple- versus set-oriented values of the event granularity in SQL3, we slightly modify the syntax of event fluents. For each database fluent $F(\vec{x}, t, s)$, we introduce not only primitive event fluents $F\_inserted(r, t, s)$ and $F\_deleted(r, t, s)$, but also primitive event fluents $F\_inserted(\vec{x}, r, t, s)$ and $F\_deleted(\vec{x}, r, t, s)$.[6] We use $F\_inserted(r, t, s)$ and $F\_deleted(r, t, s)$ to capture the set-oriented granularity, and $F\_inserted(\vec{x}, r, t, s)$ and $F\_deleted(\vec{x}, r, t, s)$ to capture the tuple-oriented event granularity. Indeed, for example, the successor state axiom (7.29) intuitively states that there is an event of insertion — without a specification of what has been inserted — into a database fluent F iff some tuple has been inserted into F. To capture the tuple-oriented granularity, we give the following successor state axioms for $F\_inserted(\vec{x}, r, t, s)$ and $F\_deleted(\vec{x}, r, t, s)$:

$$\begin{aligned} F\_inserted(\vec{x}, r, t, do(a, s)) \equiv (\exists t')a = F\_insert(\vec{x}, t') \wedge considered(r, t, s) \vee \\ F\_inserted(r, t, s) \wedge \neg(\exists \vec{y}, t')a = F\_insert(\vec{y}, t'), \end{aligned} \tag{7.31}$$

$$\begin{aligned} F\_deleted(\vec{x}, r, t, do(a, s)) \equiv (\exists t')a = F\_delete(\vec{x}, t') \wedge considered(r', t, s) \vee \\ F\_deleted(r, t, s) \wedge \neg(\exists \vec{y}, t')a = F\_delete(\vec{y}, t'). \end{aligned} \tag{7.32}$$

The changes from (7.29) and (7.30) to 7.31) and (7.32) are syntactically slight, but bear a big semantical difference; the successor state axiom (7.31), for example, intuitively states that there is an insertion event of a particular tuple $\vec{x}$ into a database fluent F iff that specific tuple has been inserted into F.

Interruptability of Actions. We introduce two actions $begin\_atomic(t)$ and $end\_atomic(t)$ with the intended meaning that a transaction t must begin executing actions without allowing any further (interleaved) rule processing, and stop such a noninterleaved rule processing, respectively. These actions are intended to control the truth value of a system fluent $atomic(t, s)$, which in turn controls the $Do(P, S, s')$ predicate with respect to rule processing. Whenever $atomic(t, s)$ is true, no rule processing may occur. The following successor state axiom characterizes the fluent $atomic(t, s)$:

$$atomic(t, do(a, s)) \equiv a = begin\_atomic(t) \vee atomic(t, s) \wedge a \neq end\_atomic(t, s).$$

Execution Model

Coupling Modes and Priorities. Since SQL3 uses immediate event/condition and condition/action coupling modes, we specify its execution model using the model $(1, 1)$ of Section 6.2.2 which was formulated as: **Evaluate C immediately after the ECA rule is triggered and execute A immediately after evaluating C within the triggering transaction.**

[6]Notice the difference in syntax between these new primitive event fluents and the transition tables formalized in Section 5.2.

Suppose we have a set $\mathcal{R}$ of n ECA rules $R_1, \ldots, R_n$ of the form (5.1), where the action part $\alpha(\vec{y})$ may be of the form $begin\_atomic(t); \beta; end\_atomic(t)$, with β being any ConGolog program. Suppose also that these n rules were created in the order $R_1, \ldots, R_n$. Then, we give the following prioritized ConGolog procedure to capture the immediate execution model of SQL3:

$$\begin{array}{l}
\textbf{proc } Rules(t) \\
\quad (\pi \vec{x}_1, \vec{y}_1)[\tau_1[R_1, t]? \,;\, \zeta_1(\vec{x}_1)[R_1, t]? \,;\, \alpha_1(\vec{y}_1)[R_1, t]] | \\
\quad \{\neg[(\exists \vec{x}_1).(\tau_1[R_1, t] \wedge \zeta_1(\vec{x}_1)[R_1, t]]? \,; \\
\qquad (\pi \vec{x}_2, \vec{y}_2)[\tau_2[R_1, t]? \,;\, \zeta_2(\vec{x}_1)[R_2, t]? \,;\, \alpha_2(\vec{y}_2)[R_2, t]] | \\
\qquad \{\neg[(\exists \vec{x}_2).(\tau_2[R_2, t] \wedge \zeta_2(\vec{x}_2)[R_2, t]]? \,; \\
\qquad\qquad \vdots \\
\qquad\quad \{(\pi \vec{x}_n, \vec{y}_n)[\tau_n[R_n, t]? \,;\, \zeta_n(\vec{x}_n)[R_n, t]? \,;\, \alpha_n(\vec{y}_n)[R_n, t]] | \\
\qquad\qquad \neg[(\exists \vec{x}_n).(\tau_n[R_n, t] \wedge \zeta_n(\vec{x}_n)[R_n, t])]?\} \ldots\}\} \\
\textbf{endProc} \,.
\end{array} \tag{7.33}$$

Notice that the procedure (7.33) above formalizes *how* ECA rules are processed as follows: the procedure $Rules(t)$ takes the rule R_1 and nondeterministically picks values for $\vec{x}_1$ and $\vec{y}_1$, tests if the event $\tau_1[R_1, t]$ occurred, in which case it immediately tests whether the condition $\zeta_1(\vec{x}_1)[R_1, t]$ holds, at which point the action part $\alpha_1(\vec{y}_1)$ is executed. If the condition $\zeta_1(\vec{x}_1)[R_1, t]$ becomes false or R_1 is not triggered, the rule procedure goes on to treat the remaining rules $R_2, \ldots, R_n$ in that order.

Executing BEFORE and AFTER Triggers. Suppose that the set $\mathcal{R}$ of triggers has been partitioned into two subsets $\mathcal{R}^B = \{r_1^B, \ldots, r_m^B\}$, and $\mathcal{R}^A = \{r_1^A, \ldots, r_n^A\}$, where $\mathcal{R}^B$ and $\mathcal{R}^A$ contains only BEFORE and AFTER triggers respectively. Then each of $\mathcal{R}^B$ and $\mathcal{R}^A$ may be represented using the procedure given in (7.33). We represent $\mathcal{R}^B$ by a rule procedure called $Rules^B(t)$ and $\mathcal{R}^A$ by $Rules^A(t)$, respectively. So given rule programs $Rules^B(t)$ and $Rules^A(t)$ specified as above, Figure 7.3 completes the logical characterization of the SQL3 execution model by making the appropriate changes to the predicate $Trans(\delta, s, \delta', s')$ for handling primitive actions.

Despite its length, the semantics given in Figure 7.3 has a relatively simple structure. Line (1) is as in Definition (6.23). Lines (2)–(9) treat the case where some BEFORE rules are triggered. In this case, Line (3) executes the BEFORE rules, after which Lines (4)–(6) perform an interruptable execution of primitive actions while Lines (7)–(9) perform a noninterruptable execution. Lines (10)–(16) treat the case where no BEFORE rules are triggered. Here, Lines (10)–(13) perform an interruptable execution of primitive actions, and Lines (14)–(16) perform a noninterruptable execution. It is important to notice how BEFORE rules are treated. With the subformula $Do(Rules^B(t), do(a, s), do(a, s))$, we check whether any BEFORE rule would be triggered or not in the situation $do(a, s)$. Such check is made before a is exe-

$$
\begin{array}{ll}
(1) & Trans(a,s,a',s') \equiv (\exists a^*, s'', s^*, s^{**}, t).transOf(a,t,s) \wedge Poss(a,s) \wedge a' = nil \wedge \\
(2) & \Big(\neg Do(Rules^B(t), do(a,s), do(a,s)) \supset \\
(3) & \quad \Big(Do(Rules^B(t), s, s^{**}) \wedge \\
(4) & \quad \{\neg atomic(t, s^{**}) \supset \\
(5) & \qquad \{[s'' = do(a, s^{**}) \wedge systemAct(a^*, t) \wedge Poss(a^*, s'') \wedge s' = do(a^*, s'')] \vee \\
(6) & \qquad [s^* = do(a, s^{**}) \wedge [(\forall a'', t') systemAct(a'', t') \supset \neg Poss(a', s^*)] \wedge Do(Rules^A(t), s^*, s')]\}\} \wedge \\
(7) & \quad \{atomic(t, s^{**}) \supset \\
(8) & \qquad \{[s'' = do(a, s^{**}) \wedge systemAct(a^*, t) \wedge Poss(a^*, s'') \wedge s' = do(a^*, s'')] \vee \\
(9) & \qquad s' = do(a, s^{**}) \wedge [(\forall a'', t') systemAct(a'', t') \supset \neg Poss(a', s')]\}\}\Big)\Big) \wedge \\
(10) & \Big(Do(Rules^B(t), do(a,s), do(a,s)) \supset \\
(11) & \quad \Big(\{\neg atomic(t, s) \supset \\
(12) & \qquad \{[s'' = do(a, s) \wedge systemAct(a^*, t) \wedge Poss(a^*, s'') \wedge s' = do(a^*, s'')] \vee \\
(13) & \qquad [s^* = do(a, s) \wedge [(\forall a'', t') systemAct(a'', t') \supset \neg Poss(a', s^*)] \wedge Do(Rules^A(t), s^*, s')]\}\} \wedge \\
(14) & \quad \{atomic(t, s) \supset \\
(15) & \qquad \{[s'' = do(a, s^{**}) \wedge systemAct(a^*, t) \wedge Poss(a^*, s'') \wedge s' = do(a^*, s'')] \vee \\
(16) & \qquad s' = do(a, s) \wedge [(\forall a'', t') systemAct(a'', t') \supset \neg Poss(a', s')]\}\}\Big)\Big).
\end{array}
$$

Figure 7.3: Semantics of the SQL3 execution model

cuted, which in this specification means recording a in the log. If $Do(Rules^B(t), do(a,s), do(a,s))$ is false, then some BEFORE rule is triggered, we execute the procedure $Rules^B(t)$ and the action a, after which we may or may not call the procedure $Rules^A(t)$ to execute AFTER rules depending on whether the fluent $atomic(t,s)$ is false or not. In the case $Do(Rules^B(t), do(a,s), do(a,s))$ is true, then no BEFORE rule is triggered, and we execute the action a, after which we may or may not call the procedure for AFTER rules depending on whether the fluent $atomic(t,s)$ is false or not.

We could have modularized this long formula of Figure 7.3; however, doing this, we would have lost the overview of how the main ingredients of the execution model of SQL3 are modeled.

A Lesson Learned

Any formal system specification must aim at being validated. It must also aim at discovering anything interesting that the formal semantics reveals, e.g., underspecified parts of the system and inconsistencies, to just name a few things. The question arises now as to what extent the limited scope of our specification of SQL3 has revealed us anything interesting.

We restrict ourself to one underspecified aspect of SQL3 execution model whose source of difficulties is revealed by the semantics given in Figure 7.3. We mentioned above that the SQL3 standard did not restrict the use of BEFORE rules, and that it is usually assumed that actions of BEFORE rules must not update the database. To the authors of [KMC99], this assumption — which is not in the standard — ensures that the database state seen by the BEFORE triggers is a state that is guaranteed to be consistent. This intuition, which is underspecified in the SQL3 standard, is precisely captured by Figure 7.3 in lines (3)–(6) where we can see how a database updated by a BEFORE rule may be inconsistent with subsequent updates. Assume indeed that one of the rules executed in line (3) has an event $F\_inserted$. Assume also that that same rule inserts a tuple $\vec{X}$ into the database. Then, whenever action a is executed in line (6), the situation s^{**} will correspond to a database state — that is, a set of true fluents — that is different than the corresponding database state of situation s. The action a however is supposed to take its preconditions in the database state corresponding to the situation s. All this may result in actions that are executed though their preconditions are not satisfied in the new situation s^{**}. For example, the action a may be $F\_insert(\vec{X}, T)$, for a transaction T, and the tuple $\vec{X}$ will never be inserted into the database since it is already there. This argument and similar ones may be tightened up into a general theorem, but we do not do it here.

7.6 Summary

This chapter has concretized the theoretical concepts developed throughout the thesis by showing how active relational theories can be implemented in Prolog. Building on this theoretical development, we have extended a Prolog implementation method for basic action theories presented in [Rei01] to active relational theories. The method is justified by a consequence of Clark's fundamental theorem saying that, whenever a logical program `P` obtained from a definitional theory $\mathcal{T}$ by taking the if-halves of the sentences $(\forall \vec{x})P(\vec{x}) \equiv \phi$ yields the answer "yes" on a sentence ψ, then ψ is a logical consequence of $\mathcal{T}$; and, whenever `P` yields the answer "no" on ψ, then $\mathcal{T}$ logically entails $\neg\psi$.

Particular novelties in this chapter are:

- implementation theorem for active relational theories;
- Abstract interpreter for transactional programs with active relational theories as background theories;
- semantics of SQL3 as an illustration of the framework.

Chapter 8

Conclusion

8.1 Summary

This thesis has proposed foundations of active databases using the situation calculus. Our approach allows to formally specify and reason about both the ECA rule language and the execution models for rules. The theories introduced in the thesis allow a precise definition of the properties of the main dimensions of active behavior, such as relational databases, database querying and updates, (advanced) database transaction models, events, conditions, actions, execution of ECA rules, etc.

Developing mathematical foundations for dynamical systems has attracted many research efforts since Amir Pnueli ([Pnu77]) first showed the importance of using temporal logic to specify semantics and dynamical properties of concurrent programs. Ray Reiter's book ([Rei01]) on situation calculus theories as "logical foundations for specifying and implementing dynamical systems" is mainly about foundations for autonomous, cognitive robots that perceive and act in changing environments and reason about their actions and the knowledge they accumulate about these actions. This dissertation aimed to give similar foundations, this time not to cognitive agents, but to the dynamical world of active databases. It focussed on logical theories for capturing advanced transaction models, complex events, execution models of ECA rules, and a methodology for obtaining Prolog implementations of these theories.

To start with our endeavour, we have noticed that advanced transaction models (ATMs) found in the literature are proposed in an *ad hoc* way for dealing with new applications involving long-lived, endless, and cooperative activities. Therefore, it is not obvious to compare ATMs to each other. Also it is difficult to exactly say how an ATM extends the traditional flat model of transaction, and to formulate its properties in a way that one clearly differenciates functionalities that have been added or subtracted. To address these questions, we have introduced a general and common framework within which to specify ATMs, simulate these, specify their properties, and reason about these properties. Thus far, ACTA ([Chr91],[CR94]) seems to our knowledge the only framework addressing these questions at a high level of generality. In ACTA, a first order logic-like language is used to capture the semantics of any ATM.

We address the problem of specifying database transactions at the logical level using the situation calculus. The main contributions of this thesis with respect to the specification of ATMs can be summarized as follows:

- We constructed logical theories called *basic relational theories* to formalize ATMs along the tradition set by the ACTA framework ([CR94]); basic relational theories are non-Markovian theories ([Gab00]) in which one may explicitly refer to all past states, and not only to the previous one. They provide the formal semantics of the corresponding ATMs. They are an extension of the classical relational theories of [Rei84] to the database transaction setting.

- We extended the notion of *legal database logs* introduced in [Rei95] to accommodate transactional actions such as $Begin$, $Commit$, etc. These logs are first class citizen of the logic, and properties of the ATM are expressed as formulas of the situation calculus that logically follow from the basic relational theory representing that ATM.

- Our approach goes far beyond constructing logical theories, as it provides one with an implementable specification, thus allowing one to simulate the specified ATM using an interpreter. Our implementable specifications are written in ConGolog, an extension of GOLOG that includes parallelism ([DGLL97]). We specify an interpreter for running these specifications and show that this interpreter generates only legal logs.

After dealing with ATMs, we have extended the framework for modeling ATMs to reactive and execution models of active behaviors. With respect to these, the main contributions of the thesis can succinctly be summarized as follows:

- We applied basic relational theories to account for open nested transactions that are the kind of transaction models that is suitable for providing the most flexible execution semantics for ECA rules.

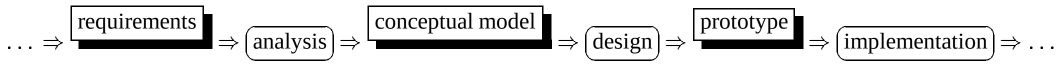
- We constructed logical theories called *active relational theories* to formalize active databases along the lines set by the framework in basic relational theories. Active relational theories too are non-Markovian theories. They provide the formal semantics of the corresponding active database model. They are an extension of the classical relational theories of [Rei84] to the transaction and active database settings.

- We specified event algebras in the situation calculus, and gave precise semantics to the following dimensions of active behavior: event consumption modes, rule priorities, and net effects. We also specified various execution models in the situation calculus, together with their coupling modes, that is, immediate, deferred, and detached execution models.

- The main result here is a set of classification theorems for the various semantics identified for important dimensions of active behavior such as consumption modes, and execution models. These theorems say roughly which semantics are equivalent and which are not.

- Our approach here too provides one with implementable specifications of reactive behaviors written in ConGolog. The semantical specification of the various execution models and database transactions readily gives us interpreters for running the specified reactive behaviors. The thesis formally justifies these implementations by extending the implementation theorem of [Rei01] to active relational theories. Also, the ISO standard for rule systems, namely SQL3, is specified using active relational theories and ConGolog.

Figure 8.1: Relational theories as conceptual models of active database management systems



We have used one single logic – the situation calculus — to accounts for virtually all features of rule languages and execution models of ADBMSs. The output of this account is a conceptual model for ADBMSs in the form of active relational theories. Thus, considering the software development cycle as depicted in Figure 8.1, an active relational theory corresponding to an ADBMS constitutes a conceptual model for that ADBMS. Since active relational theories are implementable specifications, implementing the conceptual model provides one with a rapid prototype of the specified ADBMS.

8.2 Future Work

Ideas expressed and developed in this thesis may be extended in various ways. we mention a few of them.

- **Properties of Rule programs**. Formalizing rules as ConGolog programs can be fruitful in terms of proving formal properties of active rules since proving such properties can be reduced to proving properties of programs. Here, the problems arising classically in the context of active database like confluence and termination ([WC96]) are dealt with. In Section 6.6, we gave a preliminary indication on how these properties can be formulated. We appealed to the well known distinction between *Safety* and *Progress* properties due to Manna and Pnueli ([MP91]) and argued that the classical properties such termination, confluence, and determinism that are ascribed to ECA rule programs may be fruitfully viewed in the light of the general classification of Manna and Pnueli. However, much work remains to be done on these issues.

- **Progressing Databases**. One must distinguish between our approach which is a purely logical, abstract specification in which all system properties are formulated relative to the database log, and an implementation which normally materializes the database using *progression* ([Rei01]). This is the distinguishing feature of our approach. The database log is a first class citizen of the logic, and the semantics of all transaction operations – $Commit$, $Rollback$, etc. –, primitive updates, and queries are defined with respect to this log. The main mechanism used in this respect is *regression*. However, in order to materialize the database after each update, progressing the database is the way to go. How progression can be defined for basic and active relational theories is completely open.

- **Comparing to other Approaches**. Database transaction processing is now a mature area of research. However, one needs to formally know how our formalization indeed captures any existing theory, such as ACTA, at the same level of generality. Doing so, one proves some form of correctness of our formalization, assuming that ACTA is correct. For example, we need an effective translation of our basic relational theories into ACTA axioms for a relational database and then show that the legal logs for the situation calculus basic relational theory are precisely the correct histories for its translation into a relational ACTA system.

- **Achieving Desired Properties of ATMs**. Usually, a transaction system needs a mechanism for its management. Such a mechanism enforces the desired properties of the underlying ATM. Traditionally, an algorithm using some form of locking ([BHG87]) of data items is used. A natural question that arises is: How can we accommodate a mechanism for enforcing the desired properties in our framework? Accomodating such a mechanism has one important advantage: we would obtain a way of proving the correctness of locking algorithms in logic. One way of achieving this is to embody the locking algorithms into the ternary Do predicate; however, nothing is clear yet about how this could be done.

- **Nonrelational Data Models**. Thus far, we have given axioms that accommodate a complete initial database state. This, however, is not a requirement of the theory we are presenting. Therefore our account could, for example, accommodate initial databases with null values, open initial database states, initial databases accounting for object orientation, or initial semistructured databases. These are just examples of some of the generalizations that our initial databases could admit.

- **Second Order Features**. It is important to notice that the only place where the second order nature of our framework is needed is in the proof of the properties of the transaction models that rely on a second order induction principle contained in the foundational axioms of the situation calculus. For the Markovian situation calculus, it is shown in [PR99] that the second order nature of this language is not at all needed in simulating basic action theories. It remains to show that this is also the case for the non-Markovian setting.

- **Accounting for Further ATMs**. We would like to accounting for some of the recent ATMs, for example those reported in [JK97] and open nested transactions proposed in the context of mobile computing, and reason about the specifications obtained.

- **Further Classification Theorems**. The issue of classifying execution models and consumption modes needs further study. For example, we classified consumption modes under the assumption that the scope of consumption is global. We would like to know what happens when the local scope is considered.

- **Systematic Implementation of some of the Semantics**. Being foundational, this work has been theoretical by its nature. Though we have shown how to implement the theories of this thesis, and indeed implemented some short programs as an illustration, it remains to systematically implement a full fledged system by following the guidelines laid down above.

- **Development Methodology**. Finally, we could explore ways of making the framework of this thesis part of a logic-based development methodology for active rule systems. Such a methodology would exhibit the important advantage of uniformity in many of its phases by using the single language of the situation calculus.

Bibliography

[AB98] M. Arenas and L. Bertossi. Hypothetical temporal queries in databases. In A. Borgida, V. Chaudhuri, and V. Staudt, editors, *Proceedings of the ACM SIGMOD/PODS 5th International Workshop on Knowledge Representation meets Databases (KRDB'98)*, pages 4.1–4.8, 1998. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-10/.

[Abi88] S. Abiteboul. Updates, a new frontier. In *Proceedings of the Second International conference on Database Theory*, pages 1–18, 1988.

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, Readind, MA, 1995.

[AHW95] A. Aiken, J.M. Hellerstein, and J. Widom. Static analysis techniques for predicting the behaviour of active database rules. *ACM Transaction on Database Systems*, 20:3–41, 1995.

[AV88] S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *Proceedings of the 7th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 240–250, 1988.

[AV90] S. Abiteboul and V. Vianu. Procedural languages for database queries and updates. *Journal of Computer System Sciences*, 41(2):181–229, 1990.

[BCP95] E. Baralis, S. Ceri, and S. Paraboschi. Improved rule analysis by means of triggering and activation graphs. In *Rules in Database Systems, RIDS'95*, pages 165–181, Athens, Greece, 1995.

[BCW93a] E. Baralis, S. Ceri, and J. Widom. Better termination analysis for active databases. In N.W. Paton and H. Williams, editors, *Rules in Database Systems*, pages 163–179. Springer Verlag, 1993.

[BCW93b] M. Baudinet, J. Chomicki, and P Wolper. *Temporal Deductive Databases*, chapter 13, pages 294–320. The Benjamin/Cummings Publishing Company, Redwood City, CA, 1993.

[BG85] P. Bernstein and N. Goodman. Serializability theory for replicated databases. *Journal of Comput. System Science*, 31(3):355–374, 1985.

[BGP97] C. Baral, M. Gelfond, and A. Provetti. Representing actions: Laws, observation and hypothesis. *Journal of Logic programming*, 31(1-3):201–244, 1997.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley, Reading, MA, 1987.

[BK92] A. Bonner and M. Kifer. Transaction logic programming. Technical report, University of Toronto, 1992.

[BK98] A. Bonner and M. Kifer. A logic for programming database transactions. In J. Chomicki and Saake G., editors, *Logics for Databases and Information Systems*. Kluwer Academic Publishers, 1998. Chapter 5.

[BL96] C. Baral and J. Lobo. Formal characterizations of active databases. In *International Workshop on Logic in Databases, LIDS'96*, 1996.

[BLT97] C. Baral, J. Lobo, and G. Trajcevski. Formal characterizations of active databases: Part ii. In *Proceedings of Deductive and Object-Oriented Databases, DOOD'97*, 1997.

[BM01] J. Bailey and S. Mikulás. Expressivemenss issues and decision problems for active database event queries. In *ICDT'2001*, pages 69–82, 2001.

[BPV99] L. Bertossi, J. Pinto, and R. Valdivia. Specifying database transactions and active rules in the situation calculus. In H. Levesque and F. Pirri, editors, *Logical Foundations of Cognitive Agents. Contributions in Honor of Ray Reiter*, pages 41–56, New-York, 1999. Springer Verlag.

[CC95] T. Coupaye and C. Collet. Denotational semantics for an active rule execution model. In T. Sellis, editor, *Rules in Database Systems: Proceedings of the Second International Workshop, RIDS '95*, pages 36–50. Springer Verlag, 1995.

[CF97] S. Ceri and P. Fraternali. *Designing Database Applications with Objetcs and Rules: The IDEA Methodology*. Addison Wesley, Readind, MA, 1997.

[Chr91] P.K. Chrysanthis. *ACTA, A Framework for Modeling and Reasoning about Extended Transactions*. PhD thesis, Department of Computer and Information Science, University of Massachusetts, Amherst, 1991.

[Cla78] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322, New-York, 1978. Plenum Press.

[CM91] S. Chakravarthy and D. Mishra. An event specification language (snoop) for active databases and its detection. Technical Report UF-CIS-TR-91-23, University of Florida, 1991.

[Cod70] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[CR94] P. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models. *ACM Transactions on Database Systems*, 19(3):450–491, 1994.

[CS94] R. Chandra and A. Segev. Active data bases for financial applications. In *Proceedings of the Fourth International Workshop on Research in Data Engineering (RIDE-ADS)*, pages 46–52, 1994.

[DGG95] K.R. Dittrich, S. Gatziu, and A. Geppert. The active database management system manifesto: A rulebase of adbms. In T. Sellis, editor, *Rules in Database Systems: Proceedings of the Second International Workshop, RIDS '95*, pages 3–17. Springer Verlag, 1995.

[DGLL97] G. De Giacomo, Y. Lespérance, and H.J. Levesque. Reasoning about concurrent execution, prioritized interrupts, and exogeneous actions in the situation calculus. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1221–1226, 1997.

[DGLL00] G. De Giacomo, Y. Lespérance, and H.J. Levesque. Congolog, a concurrent programming language based on the situation calculus: foundations. *Artificial Intelligence*, 121(1-2):109–169, 2000.

[Elm92] Ahmed K. Elmagarmid. *Database transaction models for advanced applications*. Morgan Kaufmann, San Mateo, CA, 1992.

[End73] H.B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, New York, 1973.

[FT95] P. Fraternali and L. Tanca. A structured approach to the definition of the semantics of active databases. *ACM Transactions on Database Systems*, 20:414–471, 1995.

[FUV83] R. Fagin, J. Ullman, and M.Y. Vardi. Updating logical databases. In *Proceedings of the second ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1983.

[FWP97] A.A.A. Fernandes, M.H. Williams, and N.W. Paton. A logic-based integration of active and deductive databases. *New Generation Computing*, 15(2):205–244, 1997.

[GA95] J. Gray and Reuter A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, CA, 1995.

[Gab00] A. Gabaldon. Non-markovian control in the situation calculus. In G. Lakemeyer, editor, *Proceedings of the Second International Cognitive Robotics Workshop*, pages 28–33, Berlin, 2000.

[Gab02a] A. Gabaldon. Non-markovian control in the situation calculus. In *Proceedings of AAAI*, Edmonton, Canada, 2002.

[Gab02b] Alfredo Gabaldon. Programming hierarchical task networks in the situation calculus. In *AIPS'02 Workshop on On-line Planning and Scheduling*, Toulouse, France, April 2002.

[GD94] S. Gatziu and K.R. Dittrich. Detecting composite events in active database systems using petri nets. In *Proceeding on Research Issues in Data Engineering, RIDE'94*, pages 2–9, 1994.

[GJS92] N.H. Gehani, H.V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model and implementation. In *Proceedings of the 18th VLDB Conference*, pages 327–338, Vancouver, 1992.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R Kowalski and K.A. Bowen, editors, *Proceedings the 5th International Conference on Logic Programming*, pages 1070–1080, Cambridge, MA, 1988. MIT Press.

[GL90] A. Guessoum and J.W. Lloyd. Updating knowledge bases. *New Generation Computing*, 8(1):71–89, 1990.

[Gra91] G. Grahne. *The Problem of Incomplete Information in Relational Databases*. Springer Verlag, Berlin, 1991.

[HD93] J.V. Harrisson and S.W. Dietrich. Integrating active and deductive rules. In *Rules in Database Systems, RIDS'93*, pages 288–305, Edinburgh, 1993. Springer-Verlag.

[HJ91] R. Hull and D. Jacobs. Language constructs for programming active databases. In *Proceedings of the 17th International Conference on VLDB*, Barcelona, 1991.

[HLM88] M. Hsu, R. Ladin, and R. McCarthy. An execution model for active database management systems. In *Proceedings of the third International Conference on Data and Knowledge Bases*, pages 171–179. Morgan Kaufmann, 1988.

[JF96] U Jaeger and J.C. Freytag. Annotated bibliography on active databases. Technical report, Humboldt-University, Berlin, 1996.

[JK97] S. Jajodia and L. Kerschberg. *Advanced Transaction Models and Architectures*. Kluwer Academic Publishers, Boston, 1997.

[Kir01a] I Kiringa. Simulation of advanced transaction models using golog. In *Proceedings of the 8th Biennial Workshop on Data Bases and Programming Languages (DBPL'01)*, 2001. Volume 2397 of Springer Lecture Notes in Computer Science, p. 318-341, Springer, 2002.

[Kir01b] I Kiringa. A theory of advanced transaction models in the situation calculus. In *LICS'01*, 2001.

[Kir01c] I. Kiringa. Towards a theory of advanced transaction models in the situation calculus (extended abstract). In *Proceedings of the VLDB 8th International Workshop on Knowledge Representation Meets Databases (KRDB'01)*, 2001.

[Kir02] I. Kiringa. Specifying event logics for active databases. In *Proceedings of the KR 9th International Workshop on Knowledge Representation Meets Databases (KRDB'02)*, pages 79–91, 2002.

[KM91] H. Katsuno and A.O. Mendelzon. On the difference between updating a knowledge base and revising it. In *Proceedings of the Second International Conference on Principles of Knowledge Representation an Reasoning*, pages 387–394, Los Altos, CA, 1991. Morgan Kaufmann Publishers.

[KMC99] K. Kulkarni, N. Mattos, and R. Cochrane. Active database features in sql-3. In N. Paton, editor, *Active Rules in Database Systems*, pages 197–219. Springer Verlag, 1999.

[Kow74] R.A. Kowalski. Predicate logic as a programming language. *Information Processing*, 74:569–574, 1974.

[KR02] I. Kiringa and R. Reiter. Specifying semantics of active databases in the situation calculus (extended abstract). Submitted, 2002.

[KS86] R.A. Kowalski and M.J. Sergot. A logic-based calculus of events. *New Generation Computing*, 4, 1986.

[Lak96] G. Lakemeyer. Only knowing in the situation calculus. In *Proc. of the 5th International Conference on Principles of Knowledge Representation and Reasoning*, pages 14–25. Morgan Kaufmann, 1996.

[LHL95] B. Ludäscher, U. Hamann, and G. Lausen. A logical framework for active rules. In *Proceedings of the Seventh International Conference on Management of Data*, Pune, 1995. Tata and McGraw-Hill.

[Lif87] V. Lifschitz. On the semantics of strips. In M.P. Georgeff and A.L. Lansky, editors, *Reasoning About Actions and Plans*, pages 1–9, Los Altos, CA, 1987. Morgan Kaufmann Publishing.

[LLL+94] Y. Lespérance, H. Levesque, F. Lin, D. Marcu, R. Reiter, and R. Scherl. A logical approach to high-level robot programming – a progress report. In *Control of the Physical World by Intelligent Systems, Working Notes of the 1994 AAAI Fall Symposium*, pages 109–119, New Orleans, November 1994.

[Llo88] J.W. Lloyd. *Foundations of Logic Programming, Second, Extended Edition*. Springer-Verlag, Berlin, 1988.

[LML96] B. Ludäscher, W. May, and G. Lausen. Nested transactions in a logical language for active rules. Technical Report Jun20-1, Technical Univ. of Munich, June 1996.

[LMWF88] N. Lynch, M.M. Merritt, W. Weihl, and A. Fekete. A theory of atomic transactions. In M. Gyssens, J. Parendaens, and D. Van Gucht, editors, *Proceedings of the Second International Conference on Database Theory*, pages 41–71, Berlin, 1988. Springer Verlag. LNCS 326.

[LMWF94] N. Lynch, M.M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann, San Mateo, 1994.

[LR94] F. Lin and R. Reiter. State constraints revisited. *J. of Logic and Computation*, 4(5):655–678, 1994.

[LRL+97] H. Levesque, R. Reiter, Y. Lespérance, Fangzhen Lin, and R.B. Scherl. Golog: A logic programming language for dynamic domains. *J. of Logic Programming, Special Issue on Actions*, 31(1-3):59–83, 1997.

[MC98] J. McCarthy and T. Costello. Combining narratives. In A.G. Cohn and L.K. Schubert, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR'98*, pages 48–59, San Francisco, CA, 1998. Morgan Kaufmann.

[McC63] J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963.

[MD89] D. McCarthy and U. Dayal. The architecture of an acctive data base management system. In *ACM-SIGMOD Conference on Management of Data*, Portland, 1989.

[MH69] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[Min96] J. Minker. Logic and databases: A 20 year retrospective. In *International Workshop on Logic in Databases, LIDS'96*, pages 3–57, 1996.

[Mor83] M. Morgenstern. Active databases as a paradigm for enhanced computing environments. In *Proceedings of the International Conference on Very Large Databases*, pages 34–42, 1983.

[Mos85] J. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing. Information Systems Series*. The MIT Press, Cambridge, MA, 1985.

[MP90] B.E. Martin and C Pedersen. Long-lived concurrent activities. Technical report, HP Laboratories, 1990. HPL-90-178.

[MP91] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer Verlag, New-York, 1991.

[NRZ92] M.H. Nodine, S. Ramaswamy, and Zdonik. A cooperative transaction model for design databases. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 53–85, San Mateo, CA, 1992. Morgan Kaufmann.

[OV99] T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, 2nd Edition*. Prentice Hall, Upper Saddle River, NJ, 1999.

[Pat99] N.W. Paton. *Active Rules in Database Systems*. Springer Verlag, New York, 1999.

[PCFW95] N.W. Paton, J. Campin, A.A.A. Fernandes, and M.H. Williams. Formal specifications of active database functionality: A survey. In T. Sellis, editor, *Rules in Database Systems: Proceedings of the Second International Workshop, RIDS '95*, pages 21–35. Springer Verlag, 1995.

[PD99] N.W. Paton and O. Diáz. Active database systems. *ACM Computing Review*, 31(1):63–103, 1999.

[PDW$^+$93] N.W. Paton, O. Diaz, M.H. Williams, J. Campin, A. Dinn, and A. Jaime. Dimensions of active behaviour. In N.W. Paton and H. Williams, editors, *Rules in Database Systems*, pages 42–57. Springer Verlag, 1993.

[Ped89] E.P.D. Pednault. Exploring the middle ground between strips and the situation calculus. In R. Brachman, H. Levesque, and R. Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 324–332, Toronto, 1989. Morgan Kaufmann Publishing.

[Pin98] J.A. Pinto. Occurrences and narratives as constraints in the branching structure of the situation calculus. *Journal of Logic and Computation*, 8:777–808, 1998.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on foundations of Computer Science*, pages 46–57. IEEE Computer Society, 1977.

[PR99] F. Pirri and R. Reiter. Some contributions to the metatheory of the situation calculus. *Journal of the ACM*, 46(3):325–364, 1999.

[Prz88] T. Przymusinski. On the declarative semantics of stratified deductive databases and logic programming. In J. Minker, editor, *Foundations of Deductive Databases and logic Programming*, pages 193–216, Los Altos, CA, 1988. Morgan Kaufmann Publishers.

[PV95] P. Picouet and V. Vianu. Semantics and expressiveness issues in active databases. In *ACM Symposium on Principles of Database Systems*, pages 126–138, San José, 1995.

[PV97] P. Picouet and V. Vianu. Expressiveness and complexity active databases. In *ICDT'97*, 1997.

[Rei78] R. Reiter. On closed world data bases. In *Logic and Data Bases, Symposium on Logic and Data Bases*, pages 55–76, New-York, 1978. Plenum Press.

[Rei84] R. Reiter. Towards a logical reconstruction of relational database theory. In M. Brodie, J. Mylopoulos, and J. Schmidt, editors, *On Conceptual Modelling*, pages 163–189, New-York, 1984. Springer Verlag.

[Rei86] R. Reiter. A sound and sometimes complete query evaluation algorithm for relational databases with null values. *Journal of the ACM*, 33:349–370, 1986.

[Rei91] R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380, San Diego, 1991. Academic Press.

[Rei95] R. Reiter. On specifying database updates. *J. of Logic Programming*, 25:25–91, 1995.

[Rei96] R. Reiter. Natural actions, concurrency and continuous time in the situation calculus. In L.C. Aiello, J. Doyle, and S.C. Shapiro, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifth International Conference (KR'96)*, pages 2–13, San Francisco, CA, 1996. Morgan Kaufmann.

[Rei01] R. Reiter. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems*. MIT Press, Cambridge, 2001.

[RPS99] S. Reddi, A. Poulovassilis, and C. Small. Pfl: an active functional dbpl. In N.W. Paton, editor, *Active Rules in Databases Systems*, pages 297–308, New-York, 1999. Springer Verlag.

[San00] M.V. Santos. Specifying and reasoning about actions in open-worlds using transaction logic. In *Proceedings of the ECAI 2000 Workshop on Cognitive Robotics*, Berlin, Germany, 2000.

[SC85] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

[Sch90] L.K. Schubert. Monotonic solution to the frame problem in the situation calculus: A efficient method for worlds with fully specified actions. In H.E. Kyberg, R.P. Louis, and G.N. Carlson, editors, *Knowledge Representation and Defeasible Reasoning*, pages 23–67, Boston, Mass., 1990. Kluwer Academic Press.

[SV87] Abiteboul S. and V. Vianu. A transaction language complete for database update and specification. In *Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 260–268, San Diego, CA, 1987.

[Ten76] R.D. Tennent. The denotational semantics of programming languages. *Communications of the ACM*, 19:437–453, 1976.

[Ter99] E. Ternovskaia. Automata theory for reasoning about automata. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 153–158, 1999.

[VEK76] M.H. Van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

[VGRS88] A. Van Gelder, K.A. Ross, and J.S. Schlipf. Unfounded sets and well-founded semantics for general programs. In *Proceedings the 7th Annual ACM Symposium on Principles of Database Systems*, pages 221–230. ACM Press, 1988.

[WC96] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[Wid92] J. Widom. A denotational semantics for the starburst production rule language. *SIGMOD RECORD*, 21(3):4–9, September 1992.

[Wid93] J. Widom. Deductive and active databases: Two paradigms or ends of a spectrum? In N.W. Paton and H. Williams, editors, *Rules in Database Systems*, pages 306–315. Springer Verlag, 1993.

[Wid94] J. Widom. Research issues in active database systems: Report from the closing panel at ride-ads'94. *ACM-SIGMOD*, 25, 1994.

[Win90] M. Winslett. *Updating Logical Databases*. Cambridge University Press, Cambridge, MA, 1990.

[WS92] G. Weikum and H.J. Schek. Concepts and applications of multilevel transactions and open nested transactions. In A.K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 516–553, San Mateo, CA, 1992. Morgan Kaufmann.

[Zan93] C. Zaniolo. A unified semantics for active and deductive databases. In N.W. Paton and H. Williams, editors, *Rules in Database Systems*, pages 271–287. Springer Verlag, 1993.

[Zan95] C. Zaniolo. Active database rules with transaction-conscious stable-model semantics. In T.W. Ling and A.O. Mendelzon, editors, *Fourth International Conference on Deductive and Object-Oriented Databases*, pages 55–72, Berlin, 1995. Springer Verlag.

[ZU01] D. Zimmer and R. Unland. On the semantics of complex events in active database managements systems. Unpublished, 2001.

Appendix A

The Regression Operator

A.1 Reiter's Regression Operator

Definition A.1 *(Markovian Regressable Formulas) A formula W of $\mathcal{L}_{sitcalc}$ is Markovian regressable iff*

1. *W is a first order formula whose terms of of sort $\mathcal{S}$ are all of the syntactic form $do([\alpha_1, \cdots, \alpha_n], S_0)$, where $n \geq 0$ and $\alpha_1, \cdots, \alpha_n$ are of sort $\mathcal{A}$.*

2. *In every atom $Poss(\alpha, \sigma)$ mentionned by W, α is of the form $A(t_1, \cdots, t_m)$, where $n \geq 0$ and A is some n-ary action function symbol of $\mathcal{L}_{sitcalc}$.*

3. *W does not quantify over situations and does not mention the predicate symbol $\sqsubset$, nor equality atoms over situation terms.*

Suppose W, W_1, W_2, and W' are regressable formulas of $\mathcal{L}_{sitcalc}$, and $\mathcal{D}$ is a basic (Markovian) action theory. Then the regression operator is recursively defined as follows ([Rei01]).

(i) Suppose W is an atom. Then

- Assume W is the equality $do([\alpha_1, \cdots, \alpha_m], S_0) = do([\alpha'_1, \cdots, \alpha'_n], S_0)$ between two terms of sort $\mathcal{S}$. Then: if $m = n = 0$, then $\mathcal{R}[W] = true$; if $m \neq n$, then $\mathcal{R}[W] = false$; if $m = n$ and $m, n \geq 1$, then $\mathcal{R}[W] = \alpha_1 = \alpha'_1 \wedge \cdots \wedge \alpha_m = \alpha'_n$.

- Assume W is the equality $do([\alpha_1, \cdots, \alpha_m], S_0) \sqsubset do([\alpha'_1, \cdots, \alpha'_n], S_0)$ between two terms of sort $\mathcal{S}$. Then: if $m = 0$ and $n \geq 1$, then $\mathcal{R}[W] = true$; if $m \geq n$, then $\mathcal{R}[W] = false$; if $1 \leq m < n$, then $\mathcal{R}[W] = \alpha_1 = \alpha'_1 \wedge \cdots \wedge \alpha_m = \alpha'_m$.

- If W is an equality involving terms of sorts $\mathcal{A}$ or $\mathcal{O}$, then $\mathcal{R}[W] = W$.

- If S_0 is the only term of sort $\mathcal{S}$ mentioned by W, then $\mathcal{R}[W] = W$.

- If W is a regressable atom $Poss(A(\vec{t}), \sigma)$, where $A(\vec{x})$ is an action with precondition axiom $Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$ in $\mathcal{D}_{pa}$, then $\mathcal{R}[W] = \mathcal{R}[\Pi_A(\vec{t}, s)]$, where all quantified variables of $\Pi_A(\vec{x}, s)$ conflicting with free variables of $Poss(A(\vec{t}), s)$ have been renamed.

- If W mentions a term of the form $f(\vec{t}, do(\alpha, \sigma))$ for some functional fluent f with successor state axiom $f(\vec{x}, do(a, s)) = y \equiv \Phi_f(\vec{x}, y, a, s)$ in $\mathcal{D}_{ss}$, then

$$\mathcal{R}[W] = \mathcal{R}[(\exists y).\Phi_f(\vec{t}, y, \alpha, \sigma) \wedge W|^{f(\vec{t}, do(\alpha, \sigma))}_{y}],$$

  where all quantified variables of $\Phi_f(\vec{x}, y, a, s)$ conflicting with free variables of $F(\vec{t}, do(\alpha, \sigma))$ have been renamed.

- If W is a fluent atom of the from $F(\vec{t}, do(\alpha, \sigma))$, where F is a Fluent with successor state axiom $F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$ in $\mathcal{D}_{ss}$, then $\mathcal{R}[W] = \mathcal{R}[\Phi_F(\vec{t}, \alpha, \sigma)]$, where all quantified variables of $\Phi_F(\vec{x}, a, s)$ conflicting with free variables of $F(\vec{t}, do(\alpha, \sigma))$ have been renamed.

(ii) Suppose W, W_1, and W_2 are a non-atomic formulas. Then $\mathcal{R}[\neg W] = \neg\mathcal{R}[W]$, $\mathcal{R}[W_1 \wedge W_2] = \mathcal{R}[W_1] \wedge \mathcal{R}[W_2]$, and $\mathcal{R}[(\exists x)W] = (\exists x)\mathcal{R}[W]$.

A.2 Regression Operator For Active Relational Theories

Definition A.2 *(Non-Markovian Regressable Formulas) A formula W of $\mathcal{L}_{sitcalc}$ is non-Markovian regressable iff*

1. *W satisfies the conditions (1)–(3) of Definition A.1.*

2. *W may quantifies over situations and mention the predicate symbol $\sqsubset$ as well as equality atoms over situation terms.*

Suppose W and W' are regressable formulas of $\mathcal{L}_{sitcalc}$, and $\mathcal{D}$ is an active relational theory. Then the regression operator is recursively defined as follows (slightly extending [Gab00]).

(i) Suppose W is an atom

- which is situation-independent,
- $do([\alpha_1, \cdots, \alpha_m], S_0) = do([\alpha'_1, \cdots, \alpha'_n], S_0)$,
- $do([\alpha_1, \cdots, \alpha_m], S_0) \sqsubset do([\alpha'_1, \cdots, \alpha'_n], S_0)$,
- $Poss(A(t_1, \cdots, t_m), \sigma)$,
- mentioning S_0 as the only term of sort $\mathcal{S}$,
- mentioning a functional term of the form $g(\vec{t}, do(\alpha, \sigma))$,

- $F(\vec{t}, do(\alpha, \sigma))$ or $F(\vec{t}, tr, do(\alpha, \sigma))$ which is a relational fluent,
- $F(r, t, do(\alpha, \sigma))$ which is a primitive event fluent,
- $F(r, \vec{t}, tr, do(\alpha, \sigma))$ which is a transition table fluent,
- $dep(t, t', do(\alpha, \sigma))$ which is a dependency predicate.

Then $\mathcal{R}[W]$ is the same as in Markovian regression.

(ii) Suppose W is of the form $(\exists s).do([\alpha_1, \cdots, \alpha_m], s) \sqsubset do([\alpha'_1, \cdots, \alpha'_n], S_0) \wedge W'$, with $m, n \geq 1$. If $m \geq n$, then $\mathcal{R}[W] = false$, otherwise

$$\begin{aligned}\mathcal{R}[W] &= \mathcal{R}[(\exists s).do([\alpha_1, \cdots, \alpha_m], s) = do([\alpha'_1, \cdots, \alpha'_{n-1}], S_0)] \wedge W'] \vee \\ &\quad \mathcal{R}[(\exists s).do([\alpha_1, \cdots, \alpha_m], s) \sqsubset do([\alpha'_1, \cdots, \alpha'_{n-1}], S_0) \wedge W'].\end{aligned}$$

(iii) Suppose W is of the form $(\exists s).do([\alpha_1, \cdots, \alpha_m], s) = do([\alpha'_1, \cdots, \alpha'_n], S_0) \wedge W'$, with $m, n \geq 1$. If $m > n$, then $\mathcal{R}[W] = false$, otherwise

$$\mathcal{R}[W] \;=\; \mathcal{R}[(\exists s).s = do([\alpha'_1, \cdots, \alpha'_{n-m}], S_0)] \wedge \alpha_1 = \alpha'_{n-m+1} \wedge \cdots \wedge \alpha_m = \alpha'_n \wedge W'].$$

(iv) Suppose W is of the form $(\exists s).do(s = do([\alpha_1, \cdots, \alpha_n], S_0) \wedge W'$, with $n \geq 1$. Then $\mathcal{R}[W] = \mathcal{R}[W'|^{s}_{do([\alpha_1, \cdots, \alpha_n], S_0)}]$.

(v) Non-atomic cases are treated as in the case of basic Markovian action theories.

Appendix B

The Revised Lloyd-Topor Rules

The implementation theorem appeals to the revised Lloyd-Topor transformation rules ([Rei01]). These are the well-known transformation rules proposed by Lloyd and Topor ([Llo88]), except that the revised rules avoid introducing new (auxilliary) predicates and clauses. We use the revised Lloyd-Topor rules for transforming if-halves of definitions into a syntactic form amenable to Prolog implementation according to the implementation theorem. The revised Lloyd-Topor rules transform a sentence of the form $W \supset A$, where W is an arbitrary first order formula and A is an atom, into a formula of the form $lt(W) \supset A$, which is an executable Prolog formula; $lt(W)$ is recursively defined as follows.

$$
\begin{aligned}
&lt(W) = W, \text{ where } W \text{ is a literal},\\
&lt(W_1 \wedge W_2) = lt(W_1) \wedge lt(W_2),\\
&lt(W_1 \vee W_2) = lt(W_1) \vee lt(W_2),\\
&lt(W_1 \supset W_2) = lt(\neg W_1 \vee W_2),\\
&lt(W_1 \equiv W_2) = lt((W_1 \supset W_2) \wedge (W_2 \supset W_1)),\\
&lt((\forall x)W) = lt(\neg(\exists x)\neg W),\\
&lt((\exists x)W) = lt(W),\\
&lt(\neg\neg W) = lt(W),\\
&lt(\neg(W_1 \wedge W_2)) = lt(\neg W_1)) \vee lt(\neg W_2),\\
&lt(\neg(W_1 \vee W_2)) = lt(\neg W_1)) \wedge lt(\neg W_2),\\
&lt(\neg(W_1 \supset W_2)) = lt(\neg(\neg W_1 \vee W_2)),\\
&lt(\neg(W_1 \equiv W_2)) = lt(\neg((W_1 \supset W_2) \wedge lt(W_2 \supset W_1))),\\
&lt(\neg(\forall x)W) = lt((\exists x)\neg W),\\
&lt(\neg(\exists x)W) = \neg lt(W).
\end{aligned}
$$

Appendix C

Semantics of ConGolog Programs

The semantics of ConGolog programms in terms of $Final(\delta, s)$, $Trans(\delta, s, \delta', s')$, and $Do(\delta, s, s')$ is a follows:

$$\begin{aligned}
&Final(nil, s) \equiv TRUE, Final(\alpha, s) \equiv FALSE, Final(\phi?, s) \equiv FALSE,\\
&Final(\delta_1; \delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s),\\
&Final(\delta_1 | \delta_2, s) \equiv Final(\delta_1, s) \vee Final(\delta_2, s),\\
&Final(\pi x\ \delta, s) \equiv (\exists x) Final(\delta, s), Final(\delta_1^*, s) \equiv TRUE,\\
&Final(\delta_1 \parallel \delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s), Final(\delta^{||}, s) \equiv TRUE.
\end{aligned}$$

$$\begin{aligned}
&Trans(nil, s, \delta, s') \equiv FALSE,\\
&Trans(\alpha, s, \delta, s') \equiv Poss(\alpha, s) \wedge \delta = nil \wedge s' = do(\alpha, s),\\
&Trans(\phi?, s, \delta, s') \equiv \phi[s] \wedge \delta = nil \wedge s' = s,\\
&Trans(\delta_1; \delta_2, s, \delta, s') \equiv Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta, s') \vee\\
&\qquad\qquad (\exists \delta').\delta = (\delta'; \delta_2) \wedge Trans(\delta_1, s, \delta', s'),\\
&Trans(\delta_1 | \delta_2, s, \delta, s') \equiv Trans(\delta_1, s, \delta, s') \vee Trans(\delta_2, s, \delta, s'),\\
&Trans(\pi x\ \delta_1, s, \delta_2, s') \equiv (\exists x)\ Trans(\delta_1, s, \delta_2, s'),\\
&Trans(\delta_1^*, s, \delta_2, s') \equiv (\exists \delta').\ \delta_2 = (\delta'; \delta_1^*) \wedge Trans(\delta_1, s, \delta', s'),\\
&Trans(\delta_1 \parallel \delta_2, s, \delta, s') \equiv (\exists \delta').\delta = (\delta' \parallel \delta_2) \wedge Trans(\delta_1, s, \delta', s') \vee\\
&\qquad\qquad \delta = (\delta_1 \parallel \delta') \wedge Trans(\delta_2, s, \delta', s'),\\
&Trans(\delta^{||}, s, \delta, s') \equiv (\exists \delta').\delta = (\delta' \parallel \delta^{||}) \wedge Trans(\delta, s, \delta', s'),\\
&Trans(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2, s, \delta, s') \equiv Trans([\phi?; \delta_1] \mid [\neg\phi?; \delta_2], s, \delta, s'),\\
&Trans(\textbf{While } \phi \textbf{ do } \delta \textbf{ endWhile }, s, \delta', s') \equiv Trans([\phi?; \delta]^*\ ;\ \neg\phi?, s, \delta, s').
\end{aligned}$$

$$Do(\delta, s, s') =_{df} (\exists \delta').Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s').$$

Here, $Trans^*$ denotes the transitive closure of $Trans$.

Appendix D

Sample Programs

D.1 The BRT for the Debit/Credit Example

```
/*  Transaction specification */

proc(dbTrans(T,Tid,Aid,Amt),
        begin(T) :
        pi(bid, pi(abal, pi(tid, ?(accounts(Aid,bid,abal,tid,T)) :
                      execDebitCredit(T,bid,Tid,Aid,Amt)))):
        end(T)).

proc(execDebitCredit(T,Bid,Tid,Aid,Amt),
        a_update(T,Aid,Amt) :
        pi(tid, pi(abal, ?(accounts(Aid,Bid,abal,tid,T)) :
                         t_update(T,Tid,Amt) :
                         b_update(T,Bid,Amt)))).

proc(a_update(T,Aid,Amt),
      pi(bid, pi(abal1, pi(tid,
               ?(accounts(Aid,bid,abal1,tid,T)) :
               pi(abal2, ?(abal2 is abal1 + Amt) :
                         a_delete(Aid,bid,abal1,tid,T) :
                         a_insert(Aid,bid,abal2,tid,T)))))).

proc(t_update(T,Tid,Amt),
          pi(tbal1, ?(tellers(Tid,tbal1,T)) :
                    pi(tbal2, ?(tbal2 is tbal1 + Amt) :
                              t_delete(Tid,tbal1,T) :
```

```
                              t_insert(Tid,tbal2,T)))).

proc(b_update(T,Bid,Amt),
         pi(bbal1, pi(bname,
               ?(branches(Bid,bbal1,bname,T)) :
               pi(bbal2, ?(bbal2 is bbal1 + Amt) :
                         b_delete(Bid,bbal1,bname,T) :
                         b_insert(Bid,bbal2,bname,T))))).

/* Declaration of primitive actions */

primitive_action(a_insert(_,_,_,_,_)). primitive_action(a_delete(_,_,_,_,_)).
primitive_action(b_insert(_,_,_,_)). primitive_action(b_delete(_,_,_,_)).
primitive_action(t_insert(_,_,_)). primitive_action(t_delete(_,_,_)).

/* Action precondition axioms for internal actions*/

poss(a_insert(Aid,Bid,Abal,Tid,T),S) :- not accounts(Aid,Bid,Abal,Tid,_,S),
  ic(do(a_insert(Aid,Bid,Abal,Tid,T),S)), running(T,S).
poss(a_delete(Aid,Bid,Abal,Tid,T),S) :- accounts(Aid,Bid,Abal,Tid,_,S),
  ic(do(a_delete(Aid,Bid,Abal,Tid,T),S)), running(T,S).
poss(b_insert(Bid,Bbal,Bname,T),S) :- not branches(Bid,Bbal,Bname,_,S),
  ic(do(b_insert(Bid,Bbal,Bname,T),S)), running(T,S).
poss(b_delete(Bid,Bbal,Bname,T),S) :- branches(Bid,Bbal,Bname,_,S),
  ic(do(b_delete(Bid,Bbal,Bname,T),S)), running(T,S).
poss(t_insert(Tid,Tbal,T),S) :- not tellers(Tid,Tbal,_,S),
  ic(do(t_insert(Tid,Tbal,T),S)), running(T,S).
poss(t_delete(Tid,Tbal,T),S) :- tellers(Tid,Tbal,_,S),
  ic(do(t_delete(Tid,Tbal,T),S)), running(T,S).

/* Integrity constraints */

ic(do(A,S)) :-
  holds((all(a, all(b, all(bal, all(t, all(b1, all(bal1, all(t1,
         accounts(a,b,bal,t,_,do(A,S)) & accounts(a,b1,bal1,t1,_,do(A,S)) =>
                          b=b1 & bal=bal1 & t=t1))))))) &
      all(b, all(bal, all(bname, all(bal1, all(bname1,
       branches(b,bal,bname,_,do(A,S)) & branches(b,bal1,bname1,_,do(A,S)) =>
                          bal=bal1 & bname=bname1))))) &
      all(t, all(bal, all(bal1,
           tellers(t,bal,_,do(A,S)) & tellers(t,bal1,_,do(A,S)) =>
```

```
                              bal=bal1)))), S,_S).

gic(S) :- holds(all(a, all(b, all(bal, all(t,
                          accounts(a,b,bal,t,_,S) => bal >= 0)))), S,_S).

/*  Succ state axioms for base tables (database fluents) of the DB.
    Notice that the axioms as presented here have undergone a simplification
    that allowed us to get rid of "writes/3", which, however, was important
    in proving some of the properties of legal logs. */

accounts(Aid,Bid,Abal,Tid,T,do(A,S)) :-
    not A = rollback(T),
    ( A=a_insert(Aid,Bid,Abal,Tid,_);
      accounts(Aid,Bid,Abal,Tid,_,S), not A=a_delete(Aid,Bid,Abal,Tid,_));
    A = rollback(T),
    holds(some(tprime, somes(sprime, do(begin(T),sprime) << S,
                               accounts(Aid,Bid,Abal,Tid,tprime,sprime))), S, _).

branches(Bid,Bbal,Bname,T,do(A,S)) :-
   not A = rollback(T),
   ( A=b_insert(Bid,Bbal,Bname,_);
     branches(Bid,Bbal,Bname,_,S), not A=b_delete(Bid,Bbal,Bname,_));
    A = rollback(T),
    holds(some(tprime, somes(sprime, do(begin(T),sprime) << S,
                                branches(Bid,Bbal,Bname,tprime,sprime))), S, _).

tellers(Tid,Tbal,T,do(A,S)) :-
   not A = rollback(T1),
   ( A=t_insert(Tid,Tbal,_);
     tellers(Tid,Tbal,_,S), not A=t_delete(Tid,Tbal,_)) ;
    A = rollback(T2),
    holds(some(tprime, somes(sprime, do(begin(T),sprime) << S,
                                tellers(Tid,Tbal,tprime,sprime))), S, _).

/* Restoring situation arguments */

restoreSitArg(accounts(W,V,X,Y,Z),S,accounts(W,V,X,Y,Z,S)).
restoreSitArg(branches(W,X,Y,Z),S,branches(W,X,Y,Z,S)).
restoreSitArg(tellers(W,X,Y),S,tellers(W,X,Y,S)).

/* Initial database */
/* Note that the Init. DB satisfies the integrity
   constraints to be enforced. */

accounts(a1,b1,1000,t1_1,_,s0). accounts(a2,b1,100,t1_1,_,s0).
accounts(a3,b1,10,t1_2,_,s0). accounts(a4,b1,0,t1_3,_,s0).
accounts(a5,b2,1000,t2_1,_,s0). accounts(a6,b2,0,t2_2,_,s0).
accounts(a7,b3,5000,t3_1,_,s0). accounts(a8,b3,500,t3_2,_,s0).
accounts(a9,b3,1000,t3_2,_,s0).
branches(b1,10000,collegeStr,_,s0). branches(b2,10000,harborStr,_,s0).
branches(b3,10000,univStr,_,s0).
tellers(t1_1,5000,_,s0). tellers(t1_2,5000,_,s0). tellers(t1_3,5000,_,s0).
tellers(t2_1,5000,_,s0). tellers(t2_2,5000,_,s0). tellers(t3_1,5000,_,s0).
tellers(t3_2,5000,_,s0).

/* Each transaction is responsible for actions that bear its name. */

responsible(T,a_insert(_,_,_,_,T),S). responsible(T,a_delete(_,_,_,_,T),S).
responsible(T,b_insert(_,_,_,T),S). responsible(T,b_delete(_,_,_,T),S).
responsible(T,t_insert(_,_,T),S). responsible(T,t_delete(_,_,T),S).
responsible(T,accounts(_,_,_,_,T),S). responsible(T,branches(_,_,_,T),S).
responsible(T,tellers(_,_,T),S).

/* Conflict table */

updConflict(a_insert(X,_,_,_,T),a_delete(X,_,_,_,T1),S) :- not T=T1.
updConflict(a_delete(X,_,_,_,T),a_insert(X,_,_,_,T1),S) :- not T=T1.
updConflict(b_insert(X,_,_,T),b_delete(X,_,_,T1),S) :- not T=T1.
updConflict(b_delete(X,_,_,T),b_insert(X,_,_,T1),S) :- not T=T1.
updConflict(t_insert(X,_,T),t_delete(X,_,T1),S) :- not T=T1.
updConflict(t_delete(X,_,T),t_insert(X,_,T1),S) :- not T=T1.
updConflict(accounts(X,_,_,_,T),a_delete(X,_,_,_,T1),S) :- not T=T1.
updConflict(accounts(X,_,_,_,T),a_insert(X,_,_,_,T1),S) :- not T=T1.
updConflict(branches(X,_,_,T),b_delete(X,_,_,T1),S) :- not T=T1.
updConflict(branches(X,_,_,T),b_insert(X,_,_,T1),S) :- not T=T1.
updConflict(tellers(X,_,T),t_delete(X,_,T1),S) :- not T=T1.
updConflict(tellers(X,_,T),t_insert(X,_,T1),S) :- not T=T1.
updConflict(a_delete(X,_,_,_,T1),accounts(X,_,_,_,T),S) :- not T=T1.
updConflict(a_insert(X,_,_,_,T1),accounts(X,_,_,_,T),S) :- not T=T1.
updConflict(b_delete(X,_,_,T1),branches(X,_,_,T),S) :- not T=T1.
updConflict(b_insert(X,_,_,T1),branches(X,_,_,T),S) :- not T=T1.
updConflict(t_delete(X,_,T1),tellers(X,_,T),S) :- not T=T1.
updConflict(t_insert(X,_,T1),tellers(X,_,T),S) :- not T=T1.

/* Miscelanous predicates */

/* Notice that we droped the tuple argument in writes/3
   to ease the simulations. */

writes(a_insert(_,_,_,_,T),accounts,T).
writes(a_delete(_,_,_,_,T),accounts,T).
writes(b_insert(_,_,_,T),branches,T).
writes(b_delete(_,_,_,T),branches,T).
writes(t_insert(_,_,T),tellers,T).
writes(t_delete(_,_,T),tellers,T).

readAct(accounts(_,_,_,_,T),accounts,T).
readAct(branches(_,_,_,T),branches,T).
readAct(tellers(_,_,T),tellers,T).

/* End of the Example */
```

D.2 The ART for the Portfolio Example

```
/* The Portfolio Example */

/*  Transaction specification */

proc(h_val_update(T,Hid,Amt),
      pi(hn, pi(c, pi(v, pi(v1,
               ?(holder(Hid,hn,c,v,T)) :
               ?(v1 is v + Amt) :
                h_delete(Hid,hn,c,v,T) :
                h_insert(Hid,hn,c,v1,T)))))).


proc(s_qty_update(T,Sid,Amt),
      pi(sn, pi(pr, pi(qty, pi(qty1,
               ?(stock(Sid,sn,pr,qty,T)) :
               ?(qty1 is qty + Amt) :
               s_delete(Sid,sn,pr,qty,T) :
               s_insert(Sid,sn,pr,qty1,T)))))).

proc(s_pr_update(T,Sid,Amt),
      pi(sn, pi(pr, pi(qty, pi(pr1,
               ?(stock(Sid,sn,pr,qty,T)) :
               ?(pr1 is pr + Amt) :
                s_delete(Sid,sn,pr,qty,T) :
                s_insert(Sid,sn,pr1,qty,T)))))).



proc(o_qty_update(T,Hid,Sid,Amt),
      pi(qty, pi(qty1,
         ?(owns(Hid,Sid,qty,T)) :
         ?(qty1 is qty + Amt) :
         o_delete(Hid,Sid,qty,T) :
         o_insert(Hid,Sid,qty1,T)))).

/* Rule procedure following the immediate E/C and C/A coupling modes */

proc(rules(T),
       ?(stock_inserted(rule1,T)) :
       pi(hid, pi(sid, pi(qty2,
           ?(some(pr, some(sn, some(qty1,
                           stock_inserted(rule1,sid,sn,pr,qty1,T) &
                           owns(hid,sid,qty2,T) & pr = 0)))) :
       stopCons(rule1,T) : beginCons(rule1,T) :
       o_delete(hid,sid,qty2,T)))) #

       ?(owns_inserted(rule2,T)) :
       pi(hid, pi(sid, pi(qty1,
                           ?(owns_inserted(rule2,hid,sid,qty1,T))))) :
       stopCons(rule2,T) : beginCons(rule2,T) :
       pi(amt, pi(sn, pi(pr, pi(qty2,
           ?(stock(sid,sn,pr,qty2,T)) :  ?(amt is pr * qty1) :
           h_val_update(T,hid,amt) : s_qty_update(T,sid,-qty1))))) #


       ?(-( (stock_inserted(rule1,T) &
             some(sid, some(sn, some(pr, some(qty1, some(hid, some(qty2,
                           stock_inserted(rule1,sid,sn,pr,qty1,T) &
                                   owns(hid,sid,qty2,T) & pr = 0))))))) v
            (owns_inserted(rule2,T) &
             some(hid, some(sid, some(qty1,
                             owns_inserted(rule2,hid,sid,qty1,T)))))))).



/* Declaration of primitive actions */

primitive_action(h_insert(_,_,_,_,_)). primitive_action(h_delete(_,_,_,_,_)).
primitive_action(s_insert(_,_,_,_,_)). primitive_action(s_delete(_,_,_,_,_)).
primitive_action(o_insert(_,_,_,_)). primitive_action(o_delete(_,_,_,_)).

/* Action precondition axioms for internal actions*/

poss(h_insert(Hid,Hname,Country,Value,T),S) :-
  not holder(Hid,Hname,Country,Value,_,S),
  ic(do(h_insert(Hid,Hname,Country,Value,T),S)), running(T,S).
poss(h_delete(Hid,Hname,Country,Value,T),S) :-
  holder(Hid,Hname,Country,Value,_,S),
  ic(do(h_delete(Hid,Hname,Country,Value,T),S)), running(T,S).
poss(s_insert(Sid,Sname,Pr,Qty,T),S) :- not stock(Sid,Sname,Pr,Qty,_,S),
  ic(do(s_insert(Sid,Sname,Pr,Qty,T),S)), running(T,S).
poss(s_delete(Sid,Sname,Pr,Qty,T),S) :- stock(Sid,Sname,Pr,Qty,_,S),
  ic(do(s_delete(Sid,Sname,Pr,Qty,T),S)), running(T,S).
poss(o_insert(Hid,Sid,Qty,T),S) :- not owns(Hid,Sid,Qty,_,S),
  ic(do(o_insert(Hid,Sid,Qty,T),S)), running(T,S).
poss(o_delete(Hid,Sid,Qty,T),S) :- owns(Hid,Sid,Qty,_,S),
  ic(do(o_delete(Hid,Sid,Qty,T),S)), running(T,S).

/* Integrity constraints */

ic(do(A,S)) :-
 holds((all(hid, all(hn, all(c, all(v, all(hn1, all(c1, all(v1,
  holder(hid,hn,c,v,_,do(A,S)) & holder(hid,hn1,c1,v1,_,do(A,S)) =>
                                          hn=hn1 & c=c1 & v=v1))))))) &
  all(sid, all(sn, all(pr, all(qty, all(sn1, all(pr1, all(qty1,
   stock(sid,sn,pr,qty,_,do(A,S)) & stock(sid,sn1,pr1,qty1,_,do(A,S)) =>
                                        bal=bal1 & bname=bname1))))))) &
  all(hid, all(sid, all(qty, all(qty1,
           owns(hid,sid,qty,_,do(A,S)) & owns(hid,sid,qty1,_,do(A,S)) =>
                                            qty=qty1)))), S,_S).

gic(S) :- holds(all(hid, all(hn, all(c, all(v,
                            holder(hid,hn,c,v,_,S) => v >= 0)))), S,_S).

/* The transition tables */

holder_inserted(Rid,Hid,Hname,Country,Value,T,do(A,S)) :- considered(Rid,T,S),
                      (A = h_insert(Hid,Hname,Country,Value,T1) ;
                             holder_inserted(Rid,Hid,Hname,Country,Value,T,S),
                             not A = h_delete(Hid,Hname,Country,Value,T1)).

holder_deleted(Rid,Hid,Hname,Country,Value,T,do(A,S)) :- considered(Rid,T,S),
                      (A = h_delete(Hid,Hname,Country,Value,T1) ;
                             holder_deleted(Rid,Hid,Hname,Country,Value,T,S),
                             not A = h_insert(Hid,Hname,Country,Value,T1)).

stock_inserted(Rid,Sid,Sname,Pr,Qty,T,do(A,S)) :- considered(Rid,T,S),
                     (A = s_insert(Sid,Sname,Pr,Qty,T1) ;
                             stock_inserted(Rid,Sid,Sname,Pr,Qty,T,S),
                             not A = s_delete(Sid,Sname,Pr,Qty,T1)).

stock_deleted(Rid,Sid,Sname,Pr,Qty,T,do(A,S)) :- considered(Rid,T,S),
                     (A = s_delete(Sid,Sname,Pr,Qty,T1) ;
                             stock_deleted(Rid,Sid,Sname,Pr,Qty,T,S),
                             not A = s_insert(Sid,Sname,Pr,Qty,T1)).

owns_inserted(Rid,Hid,Sid,Qty,T,do(A,S)) :- considered(Rid,T,S),
      (A = o_insert(Hid,Sid,Qty,T1) ; owns_inserted(Rid,Hid,Sid,Qty,T,S),
                                      not A = o_delete(Hid,Sid,Qty,T1)).

owns_deleted(Rid,Hid,Sid,Qty,T,do(A,S)) :- considered(Rid,T,S),
     (A = o_delete(Hid,Sid,Qty,T1) ; owns_deleted(Rid,Hid,Sid,Qty,T,S),
                                     not A = o_insert(Hid,Sid,Qty,T1)).


/* Primitive event fluents */

holder_inserted(Rid,T,do(A,S)) :- considered(Rid,T,S),
  (A = h_insert(Hid,Hname,Country,Value,T1) ; holder_inserted(Rid,T,S)).
```

```
holder_deleteted(Rid,T,do(A,S)) :- considered(Rid,T,S),
  (A = h_delete(Hid,Hname,Country,Value,T1) ;  holder_deleted(Rid,T,S)).

stock_inserted(Rid,T,do(A,S)) :- considered(Rid,T,S),
  (A = s_insert(Sid,Sname,Pr,Qty,T1) ; stock_inserted(Rid,T,S)).

stock_deleted(Rid,T,do(A,S)) :- considered(Rid,T,S),
  (A = s_delete(Sid,Sname,Pr,Qty,T1) ; stock_deleted(Rid,T,S)).

owns_inserted(Rid,T,do(A,S)) :- considered(Rid,T,S),
  (A = o_insert(Hid,Sid,Qty,T1) ; owns_inserted(Rid,T,S)).

owns_deleted(Rid,T,do(A,S)) :- considered(Rid,T,S),
  (A = o_delete(Hid,Sid,Qty,T1) ; owns_deleted(Rid,T,S)).

/*  Succ state axioms for base tables of the DB */

holder(Hid,Hname,Country,Value,T,do(A,S)) :-
  not A = rollback(T),
  ( A=h_insert(Hid,Hname,Country,Value,_);
    holder(Hid,Hname,Country,Value,_,S),
    not A=h_delete(Hid,Hname,Country,Value,_));
  A = rollback(T),
    holds(some(tprime, somes(sprime, do(begin(T),sprime) << S,
                  holder(Hid,Hname,Country,Value,tprime,sprime))), S, _).

stock(Sid,Sname,Pr,Qty,T,do(A,S)) :-
   not A = rollback(T),
   ( A=s_insert(Sid,Sname,Pr,Qty,_);
     stock(Sid,Sname,Pr,Qty,_,S), not A=s_delete(Sid,Sname,Pr,Qty,_));
    A = rollback(T),
    holds(some(tprime, somes(sprime, do(begin(T),sprime) << S,
                             stock(Sid,Sname,Pr,Qty,tprime,sprime))), S, _).

owns(Hid,Sid,Qty,T,do(A,S)) :-
   not A = rollback(T1),
   ( A=o_insert(Hid,Sid,Qty,_);
     owns(Hid,Sid,Qty,_,S), not A=o_delete(Hid,Sid,Qty,_)) ;
    A = rollback(T2),
    holds(some(tprime, somes(sprime, do(begin(T),sprime) << S,
                             owns(Hid,Sid,Qty,tprime,sprime))), S, _).

/* Restoring situation arguments */

restoreSitArg(holder(W,V,X,Y,Z),S,holder(W,V,X,Y,Z,S)).
restoreSitArg(stock(W,V,X,Y,Z),S,stock(W,V,X,Y,Z,S)).
restoreSitArg(owns(W,V,X,Y),S,owns(W,V,X,Y,S)).

restoreSitArg(holder_inserted(R,W,V,X,Y,Z),S,holder_inserted(R,W,V,X,Y,Z,S)).
restoreSitArg(holder_deleted(R,W,V,X,Y,Z),S,holder_deleted(R,W,V,X,Y,Z,S)).
restoreSitArg(stock_inserted(R,W,V,X,Y,Z),S,stock_inserted(R,W,V,X,Y,Z,S)).
restoreSitArg(stock_deleted(R,W,V,X,Y,Z),S,stock_deleted(R,W,V,X,Y,Z,S)).
restoreSitArg(owns_inserted(R,W,V,X,Y),S,owns_inserted(R,W,V,X,Y,S)).
restoreSitArg(owns_deleted(R,W,V,X,Y),S,owns_deleted(R,W,V,X,Y,S)).

restoreSitArg(holder_inserted(R,Z),S,holder_inserted(R,Z,S)).
restoreSitArg(holder_deleted(R,Z),S,holder_deleted(R,Z,S)).
restoreSitArg(stock_inserted(R,Z),S,stock_inserted(R,Z,S)).
restoreSitArg(stock_deleted(R,Z),S,stock_deleted(R,Z,S)).
restoreSitArg(owns_inserted(R,Z),S,owns_inserted(R,Z,S)).
restoreSitArg(owns_deleted(R,Z),S,owns_deleted(R,Z,S)).

/* Axioms for notification table */

served(Sid,do(A,S)) :- A = notify(Sid) ; served(Sid,S).
vseQuotation(Sid,Pr,do(A,S)) :- vseQuotation(Sid,Pr,S).
tseQuotation(Sid,Pr,do(A,S)) :- tseQuotation(Sid,Pr,S).

poss(notify(Sid),S).

/* Initial database */

/* Assume that all rules are considered in the initial situation. */

considered(Rid,T,s0).

/* Note that the Init. DB satisfies the integrity
   constraints to be enforced. */

holder(c1,smith,canada,30000,_,s0). holder(c2,diouf,senegal,20000,_,s0).
holder(c3,brown,canada,9000,_,s0).
```

```
stock(st1,ibm,100,5000,_,s0). stock(st2,oracle,30,1000,_,s0).
stock(st3,gm,10,5000,_,s0). % stock(st4,ford,60,1000,_,s0).
owns(c1,st1,300,_,s0). owns(c2,st1,200,_,s0). owns(c3,st2,300,_,s0).
owns(c3,st4,300,_,s0).

vseQuotation(st1,110,s0). vseQuotation(st3,10,s0).
tseQuotation(st2,50,s0). tseQuotation(st4,40,s0).

/* Each transaction is responsible for actions that bear its name. */

responsible(T,h_insert(_,_,_,_,T),S). responsible(T,h_delete(_,_,_,_,T),S).
responsible(T,s_insert(_,_,_,_,T),S). responsible(T,s_delete(_,_,_,_,T),S).
responsible(T,o_insert(_,_,_,T),S). responsible(T,o_delete(_,_,_,T),S).
responsible(T,holder(_,_,_,_,T),S). responsible(T,stock(_,_,_,_,T),S).
responsible(T,owns(_,_,_,T),S).

/* Conflict table */

updConflict(h_insert(X,_,_,_,T),h_delete(X,_,_,_,T1),S) :- not T=T1.
updConflict(h_delete(X,_,_,_,T),h_insert(X,_,_,_,T1),S) :- not T=T1.
updConflict(s_insert(X,_,_,_,T),s_delete(X,_,_,_,T1),S) :- not T=T1.
updConflict(s_delete(X,_,_,_,T),s_insert(X,_,_,_,T1),S) :- not T=T1.
updConflict(o_insert(X,_,_,T),o_delete(X,_,_,T1),S) :- not T=T1.
updConflict(o_delete(X,_,_,T),o_insert(X,_,_,T1),S) :- not T=T1.
updConflict(holder(X,_,_,_,T),h_delete(X,_,_,_,T1),S) :- not T=T1.
updConflict(holder(X,_,_,_,T),h_insert(X,_,_,_,T1),S) :- not T=T1.
updConflict(stock(X,_,_,_,T),s_delete(X,_,_,_,T1),S) :- not T=T1.
updConflict(stock(X,_,_,_,T),s_insert(X,_,_,_,T1),S) :- not T=T1.
updConflict(owns(X,_,_,T),o_delete(X,_,_,T1),S) :- not T=T1.
updConflict(owns(X,_,_,T),o_insert(X,_,_,T1),S) :- not T=T1.
updConflict(h_delete(X,_,_,_,T1),holder(X,_,_,_,T),S) :- not T=T1.
updConflict(h_insert(X,_,_,_,T1),holder(X,_,_,_,T),S) :- not T=T1.
updConflict(s_delete(X,_,_,_,T1),stock(X,_,_,_,T),S) :- not T=T1.
updConflict(s_insert(X,_,_,_,T1),stock(X,_,_,_,T),S) :- not T=T1.
updConflict(o_delete(X,_,_,T1),owns(X,_,_,T),S) :- not T=T1.
updConflict(o_insert(X,_,_,T1),owns(X,_,_,T),S) :- not T=T1.

/* Miscelanous predicates */

/* Notice that we droped the tuple argument in writes/3
   to ease the simulations. */
```

```
writes(h_insert(_,_,_,_,T),holder,T).
writes(h_delete(_,_,_,_,T),holder,T).
writes(s_insert(_,_,_,_,T),stock,T).
writes(s_delete(_,_,_,_,T),stock,T).
writes(o_insert(_,_,_,T),owns,T).
writes(o_delete(_,_,_,T),owns,T).

readAct(holder(_,_,_,_,T),holder,T).
readAct(stock(_,_,_,_,T),stock,T).
readAct(owns(_,_,_,T),owns,T).

transOf(h_insert(_,_,_,_,T),T).
transOf(h_delete(_,_,_,_,T),T).
transOf(s_insert(_,_,_,_,T),T).
transOf(s_delete(_,_,_,_,T),T).
transOf(o_insert(_,_,_,T),T).
transOf(o_delete(_,_,_,T),T).

/* Database fluents */

db_fluent(holder(_,_,_,_,_)).
db_fluent(stock(_,_,_,_,_)).
db_fluent(owns(_,_,_,_)).

/* End of the Example */
```

Appendix E

Examples

Under the delayed execution model, the two rules shown in Figure 5.1 can be compiled into the rule program shown below:

```
**proc** $Rules(trans)$
    $(\pi\ c, time, bal, price', s\_id, price, clos\_pr)$
      $[price\_inserted[Update\_stocks, trans]\ ?\ ;$
      $[\{price\_inserted(s\_id, price, time) \wedge customer(c, bal, s\_id) \wedge$
            $stock(s\_id, price', clos\_pr)\}[Update\_stocks, trans] \wedge assertionInterval(trans)]\ ?\ ;$
      $stock\_insert(s\_id, price, clos\_pr)[Update\_stocks, trans]]\ |$
    $(\pi\ new\_price, time, bal, pr, clos\_pr, c, s\_id, 100))$
      $[price\_inserted[Buy\_100shares, trans]\ ?\ ;$
      $[\{price\_inserted(s\_id, new\_price, time) \wedge customer(c, bal, s\_id) \wedge stock(s\_id, pr, clos\_pr) \wedge$
            $new\_price < 50 \wedge clos\_pr > 70\}[Update\_stocks, trans] \wedge assertionInterval(trans)]\ ?\ ;$
      $buy(c, s\_id, 100)[Update\_stocks, trans]]\ |$
      $\neg[(\exists c, time, bal, price')(price\_inserted[Update\_stocks, trans] \wedge$
        $\{price\_inserted(s\_id, price, time) \wedge customer(c, bal, s\_id) \wedge$
          $stock(s\_id, price', clos\_pr)\}[Update\_stocks, trans])] \vee$
        $(\exists new\_price, time, bal, pr, clos\_pr)(price\_inserted[Buy\_100shares, trans] \wedge$
          $\{customer(c, bal, s\_id) \wedge stock(s\_id, pr, clos\_pr) \wedge$
            $new\_price < 50 \wedge clos\_pr > 70\}[Update\_stocks, trans]) \wedge assertionInterval(trans)\}\ ?$
**endProc**.
```

Appendix F

SQL3 Syntax for Triggers

For these EBNF rules, see [KMC99] for detail.

```
<trigger definition> ::=
     CREATE TRIGGER <trigger name>
        <trigger action time>
        <trigger event> ON <relation name>
        [REFERENCING <old or new values alias list>]
        <trigger action>

<trigger action time> ::= BEFORE | AFTER

<trigger event> ::= INSERT | DELETE | UPDATE [OF <column name list>]

<old or new values alias list> ::=
        OLD [AS] <identifier> | NEW [AS] <identifier>
        | OLD_TABLE [AS] <identifier> | NEW_TABLE [AS] <identifier>

<trigger action> ::=
              [FOR EACH {ROW | STATEMENT}]
              [<trigger condition>]
              <triggered SQL statement>

<trigger condition> ::= WHEN ‘‘(’’ <condition> ‘‘)’’

<triggered SQL statement> ::=
              <SQL procedure statement>
              | BEGIN ATOMIC {<SQL procedure statement> ‘‘;’’} END
```

Appendix G

Proofs

Theorem 3.2

1. Suppose that we have a structure $\mathfrak{M}$ and a variable assignment $\mathcal{V}$ such that the following holds

$$\models_{\mathfrak{M},\mathcal{V}[a_1/A_1,a_2/A_2,s_1/S_1,s_2/S_2]} do(a_1,s_1) = do(a_2,s_2);$$

here, S_j is a ground situation term. Thus, by the rule for $=$, $\|do(A_1,S_1)\|_{\mathfrak{M},\mathcal{V}} = \|do(A_2,S_2)\|_{\mathfrak{M},\mathcal{V}}$, which amounts to $\|S_1\|_{\mathfrak{M},\mathcal{V}} \circ [\|A_1\|_{\mathfrak{M},\mathcal{V}}] = \|S_2\|_{\mathfrak{M},\mathcal{V}} \circ [\|A_2\|_{\mathfrak{M},\mathcal{V}}]$. This means that we have two lists that are the same; thus they agree both on their last element and on the rest of elements. That is, $\|A_1\|_{\mathfrak{M},\mathcal{V}} = \|A_2\|_{\mathfrak{M},\mathcal{V}}$, and $\|S_1\|_{\mathfrak{M},\mathcal{V}} = \|S_2\|_{\mathfrak{M},\mathcal{V}}$. Henceforth we have $\models_{\mathfrak{M},\mathcal{V}[a_1/A_1,a_2/A_2,s_1/S_1,s_2/S_2]} a_1 = a_2 \wedge s_1 = s_2$.

2. Let $\mathfrak{M} = (\mathfrak{U},\mathfrak{I})$ and $\mathcal{V}$ be a variable assignment. It is enough to prove that if $\models_{\mathfrak{M},\mathcal{V}[f/F]} f(S_0) \wedge (\forall a,s)[f(s) \supset f(do(a,s))]$ then $\models_{\mathfrak{M},\mathcal{V}[f/F]} (\forall s) f(s)$. Now suppose

$$\models_{\mathfrak{M},\mathcal{V}[f/F,a/A,s/S]} f(S_0) \wedge (\forall a,s)[f(s) \supset f(do(a,s)).$$

Thus, we have

$$\models_{\mathfrak{M},\mathcal{V}[f/F,a/A,s/S]} F(S_0) \text{ and} \qquad (*)$$

$$\models_{\mathfrak{M},\mathcal{V}[f/F,a/A,s/S]} F(S) \supset F(do(A,S)). \qquad (**)$$

$(**)$ is equivalent to $\models_{\mathfrak{M},\mathcal{V}[f/F,a/A,s/S]} F(S) \;\Rightarrow\; \models_{\mathfrak{M},\mathcal{V}[f/F,a/A,s/S]} F(do(A,S))$ which in turn is equivalent to $\models_{\mathfrak{M},\mathcal{V}[f/F,a/A,s/S]} F(S) \;\Rightarrow\; \Phi_F(A,S)$, where F has a successor state axiom of the form $F(do(A,s)) \equiv \Phi_F(A,s)$. Now we show that $\models_{\mathfrak{M},\mathcal{V}[f/F]} (\forall s) f(s)$ by induction over all the situations s such that $S_0 \sqsubset s$ using $(*)$ and $(**)$. The inductive step is ensured by the fact that, according to $(**)$, F is true in S and the condition for its truth in the next situation is also true.

3. Suppose that we have a structure $\mathfrak{M}$ and a variable assignment $\mathcal{V}$ such that $\models_{\mathfrak{M},\mathcal{V}[s/S]} s \sqsubset S_0$ for some S such that $||S||_{\mathfrak{M},\mathcal{V}} \in \mathfrak{U}^*_{\mathcal{A}}$. Thus, by the semantical rule for $\sqsubset$, $||S||_{\mathfrak{M},\mathcal{V}}$ is a prefix of $||S_0||_{\mathfrak{M},\mathcal{V}}$, and therefore a prefix of [ ]. This, however, contradicts the fact [ ] does not have any prefix by definition.

4. Let $\mathfrak{M}$ be a structure and $\mathcal{V}$ a variable assignment. Thus, by the semantic rule for $\sqsubset$, if $\models_{\mathfrak{M},\mathcal{V}[s/S,a/A,s'/S']}$ $s \sqsubset do(a, s')$, for some S, A, and S' such that $||S||_{\mathfrak{M},\mathcal{V}} \in \mathfrak{U}^*_{\mathcal{A}}$, $||A||_{\mathfrak{M},\mathcal{V}} \in \mathfrak{U}_{\mathcal{A}}$, and $||S'||_{\mathfrak{M},\mathcal{V}} \in \mathfrak{U}^*_{\mathcal{A}}$, then $||S||_{\mathfrak{M},\mathcal{V}}$ is a prefix of $||do(A, S')||_{\mathfrak{M},\mathcal{V}}$, and therefore a prefix of $||S'||_{\mathfrak{M},\mathcal{V}} \circ [||A||_{\mathfrak{M},\mathcal{V}}]$. Henceforth, $||S||_{\mathfrak{M},\mathcal{V}}$ is a history that is equal to a prefix of $||S'||_{\mathfrak{M},\mathcal{V}}$ up to the identity; henceforth $s \sqsubseteq s'$. ■

Lemma 4.6

We must prove two goals:

$$\mathcal{D}_f \cup \{(4.20)\} \models legal(S_0), \tag{G.1}$$

and

$$\mathcal{D}_f \cup \{(4.20)\} \models (\forall s, a)[legal(do(a, s)) \equiv legal(s) \wedge Poss(a, s) \wedge \\ (\forall a', t)[systemAct(a', t) \wedge responsible(t, a', s) \wedge Poss(a', s) \supset a = a']] \tag{G.2}$$

Goal (G.1): By Abbreviation (4.20), we must pursue two subgoals:

$$\mathcal{D}_f \cup \{(4.20)\} \models (\forall a, s^*)[do(a, s^*) \sqsubseteq S_0 \supset Poss(a, s^*)]$$

and

$$\mathcal{D}_f \cup \{(4.20)\} \models (\forall a', a'', s', t)[systemAct(a', t) \wedge responsible(t, a', s') \wedge \\ responsible(t, a'', s') \wedge Poss(a', s') \wedge do(a'', s') \sqsubset S_0 \supset a' = a''].$$

For the first subgoal, we are lead ultimately to a success in the proof of its consequent by using the foundational axiom (4.4) and the sentence $S_0 \neq do(a, s)$, a consequence of $\mathcal{D}_f$; and the second subgoal is obviously true by virtue of the fact that $S_0 \neq do(a, s)$.

Goal (G.2): The proof is by induction on s, using the induction axiom (4.3). The case $s = S_0$ is intuitive enough:

$$\mathcal{D}_f \cup \{(4.20)\} \models (\forall a)[legal(do(a, S_0)) \equiv legal(S_0) \wedge Poss(a, S_0) \wedge \\ (\forall a', t)[systemAct(a', t) \wedge responsible(t, a', S_0) \wedge \\ Poss(a', S_0) \supset a = a']];$$

its proof, though tedious, is straightforward. Now, assume the formula in (G.2) proven for $do(a, s)$. We must prove it for $do(a^*, do(a, s))$; i.e. we must prove

$$\mathcal{D}_f \cup \{(4.20)\} \models (\forall s, a, a^*)[legal(do(a^*, do(a, s))) \equiv legal(do(a, s)) \wedge \\ Poss(a^*, do(a, s)) \wedge (\forall a', t)[systemAct(a', t) \wedge responsible(t, a', do(a, s)) \wedge \\ Poss(a', do(a, s)) \supset a^* = a']]. \tag{G.3}$$

$\Longleftarrow$:
Assume for fixed a, a^*, and s $legal(do(a, s)), Poss(a^*, do(a, s)))$, and

$$(\forall a', t).systemAct(a', t) \wedge responsible(t, a', do(a, s)) \wedge Poss(a', do(a, s)) \supset a^* = a'. \qquad (\dagger)$$

By induction hypothesis, $legal(do(a, s))$ can be replaced by $legal(s)$, $Poss(a, s)$, and

$$(\forall a', t).systemAct(a', t) \wedge responsible(t, a', s) \wedge Poss(a', s) \supset a = a'. \qquad (\ddagger)$$

From $legal(s)$, $Poss(a, s)$, and $Poss(a^*, do(a, s))$ we obtain

$$(\forall s^{**}).do(a, s^{**}) \sqsubseteq do(a^*, do(a, s))) \supset Poss(a, s^{**}). \qquad (\$)$$

From $legal(s)$, ($\dagger$), and ($\ddagger$) we have

$$(\forall a_1, a_2, s', t)[systemAct(a_1, t) \wedge responsible(t, a_1, do(a, s)) \wedge Poss(a_1, s') \wedge \\ do(a_2, s') \sqsubset do(a^*, do(a, s)) \supset a_1 = a_2]. \qquad (\$\$)$$

Now, from (\$) and (\$\$) follows $legal(do(a^*, do(a, s)))$.

$\Longrightarrow$:
Assume for fixed a, a^*, and s $legal(do(a, do(a^*, s)))$. Then, by Definition (4.20),

$$(\forall a_1, s_1)[do(a_1, s_1) \sqsubseteq do(a^*, do(a, s)) \supset Poss(a_1, s_1)] \wedge \\ (\forall a_2, a_3, s_2, t)[systemAct(a_2, t) \wedge responsible(t, a_2, s_2) \wedge Poss(a_2, s_2) \wedge \\ do(a_3, s_2) \sqsubset do(a^*, do(a, s)) \supset a_2 = a_3]. \qquad (G.4)$$

Thus we must show that (G.4) implies $legal(do(a, s))$, $Poss(a^*, do(a, s))$, and

$$(\forall a', t).systemAct(a', t) \wedge responsible(t, a', do(a, s)) \wedge Poss(a', do(a, s)) \supset a^* = a'. \qquad (G.5)$$

Subgoal $Poss(a^*, do(a, s))$ follows from the first conjunct of (G.4) alone; subgoal (G.5) follows from the second conjunct of (G.4), and subgoal $legal(do(a, s))$ follows from both conjuncts of (G.4). Foundational axioms and the induction hypothesis are involved in this proof whose details are omitted here.
■

Lemma 4.7

We conduct an induction on s.

For the case $s = S_0$, we must prove $legal(S_0) \supset (\forall s')[s' \sqsubseteq S_0 \supset legal(s')]$. By Lemma 4.6, $legal(S_0)$. Therefore, $(\forall s')[s' \sqsubseteq S_0 \supset legal(s')]$ which is clearly true by the foundational axiom (4.4).

Now assume the result for s and suppose for fixed a and s that $legal(do(a,s))$ holds. Then we must prove $(\forall s')[s' \sqsubseteq do(a,s) \supset legal(s')]$. Since $legal(do(a,s))$, then, by Lemma 4.6, $Poss(a,s)$ and $legal(s)$. Assume, for fixed s', $s' \sqsubseteq do(a,s)$. Henceforth we must show that $legal(s')$. For the assumption $s' = do(a,s)$, since $legal(do(a,s))$, we have immediately $legal(s')$. For the assumption $s' \sqsubset do(a,s)$, by the foundational axiom (4.5), we get $s' \sqsubset s$; and since $legal(s)$, we obtain, by induction hypothesis, $(\forall s^*)[s^* \sqsubseteq s \supset legal(s^*)]$. Henceforth $legal(s')$. ■

Lemma 4.8

by induction on the length of the sequence of actions.
For the case $n = 1$, we must prove

$$\mathcal{D}_f \models legal(do(a,s)) \equiv \\ Poss(a,s) \wedge (\forall a',t)[systemAct(a',t) \wedge responsible(t,a',s) \wedge Poss(a',s) \supset a = a'].$$

The proof is immediate by Lemma 4.6.

Assume the result for n. We must prove that $\mathcal{D}_f$ and (4.20) entail

$$legal(do([a_1,\cdots,a_{n+1}],s)) \equiv \bigwedge_{i=1}^{n+1} \{Poss(a, do([a_1,\cdots,a_{i-1}],s)) \wedge \\ (\forall a',t)[systemAct(a',t) \wedge responsible(t,a',do([a_1,\cdots,a_{i-1}],s)) \wedge \\ Poss(a',do([a_1,\cdots,a_{i-1}],s)) \supset a_i = a']\}.$$

$\Longrightarrow$:
Assume for fixed s $legal(do([a_1,\cdots,a_{n+1}],s))$. By Lemma 4.7 and the fact, provable by induction, that $s \sqsubset do(a,s)$, we have $legal(do([a_1,\cdots,a_n],s))$. Henceforth, by induction hypothesis,

$$\bigwedge_{i=1}^{n} \{Poss(a, do([a_1,\cdots,a_{i-1}],s)) \wedge \\ (\forall a',t)[systemAct(a',t) \wedge responsible(t,a',do([a_1,\cdots,a_{i-1}],s)) \wedge \\ Poss(a',do([a_1,\cdots,a_{i-1}],s)) \supset a_i = a']\}.$$

Thus, we must now only show that $Poss(a_{n+1}, do([a_1,\cdots,a_n],s))$ and

$$(\forall a',t)[systemAct(a',t) \wedge responsible(t,a',do([a_1,\cdots,a_n],s)) \wedge \\ Poss(a',do([a_1,\cdots,a_n],s)) \supset a_{n+1} = a'].$$

Both of these claims follow from Lemma 4.6.

$\Longleftarrow$:
This part of the proof is symmetric to the previous case.

This completes the proof of the inductive case. ■

Theorem 4.9

1. Assume, for fixed s, $legal(s)$ and let $s = do([B_1, \cdots, B_m], S_0)$. Then, by Lemma 4.8,

$$\bigwedge_{i=1}^{m} Poss(a_i, do([B_1, \cdots, B_{i-1}], S_0)). \tag{G.6}$$

Now assume, for fixed a, s', s'', and t, $do(a, s') \sqsubset s$, $do(a, s'') \sqsubset s$, and $externalAct(a, t)$. Therefore we must prove $s' = s''$. From the assumption $externalAct(a, t)$ and Abbreviation 4.11, we must consider four cases.

Case $a = Begin(t)$. Assume, contrary to our goal, that $s' \neq s''$. Henceforth, either $s' \sqsubset s''$ or $s'' \sqsubset s'$. Suppose $s' \sqsubset s''$. Then, by the foundational axioms (4.2), (4.5), and the assumptions $do(Begin(t), s') \sqsubset s$ and $do(Begin(t), s'') \sqsubset s$, we have $s' \sqsubset do(Begin(t), s') \sqsubset do(Begin(t), s'')$. By (G.6), we have $Poss(Begin(t), s'')$; henceforth, by the precondition axiom (4.11) for the action $Begin(t)$, we have $\neg(\exists s^*) do(Begin(t), s^*) \sqsubseteq s''$. Now, this contradicts the fact that $do(Begin(t), s') \sqsubset do(Begin(t), s'')$. The subcase $s' \sqsubset s''$ is proven in an analog way.

Case $a = End(t)$. Assume that $s' \neq s''$. Henceforth, either $s' \sqsubset s''$ or $s'' \sqsubset s'$. Suppose $s' \sqsubset s''$. Then, similarly to the previous case, we get $s' \sqsubset do(End(t), s') \sqsubset do(End(t), s'')$. By (G.6), we have $Poss(End(t), s'')$; henceforth, by the precondition axiom (4.12) for the action $End(t)$, we have $running(t, s'')$, and, by Abbreviation 4.7, we have

$$\begin{aligned}&(\exists s^*).do(Begin(t), s^*) \sqsubseteq s'' \wedge \\ &(\forall a, s^{**})[do(Begin(t), s^*) \sqsubset do(a, s^{**}) \sqsubset s'' \supset a \neq Rollback(t) \wedge a \neq End(t)].\end{aligned} \tag{G.7}$$

Since by the previous case, the log $do(Begin(t), s^*)$ that exists must be the same for both $do(End(t), s')$ and $do(End(t), s'')$, we clearly get a contradiction between the fact that $s' \sqsubset do(End(t), s') \sqsubset do(End(t), s'')$ and (G.7). The subcase $s' \sqsubset s''$ is proven in an analog way.

Cases $a = Commit(t)$ and $a = Rollback(t)$. Both cases follow immediately from the case $a = End(t)$, as both $Commit(t)$ and $Rollback(t)$ are possible only in logs following the execution of $End(t)$.
■

Theorem 4.10

Assume, for fixed s, $legal(s)$. Suppose, for fixed t,t', and s', that $sc\_dep(t, t')$, and that $do(Commit(t'), s') \sqsubset s$. Then we must show that $(\exists s^*) do(Commit(t), s^*) \sqsubseteq s$. Since $legal(s)$ and $do(Commit(t'), s') \sqsubset s$, we have, by Lemma 4.7, $legal(do(Commit(t'), s'))$ and, by Lemma 4.6, $Poss(Commit(t'), s')$. By the action precondition axiom for $Commit(t')$, we have

$$(\forall t^*)[sc\_dep(t,t^*,s) \supset (\exists s'')do(Commit(t^*),s'') \sqsubseteq s].$$

Since $sc\_dep(t,t')$, this leads easily to what we had to prove.

The second conjunct involving $r\_dep(t,t',s)$ and $Rollback(t)$ is proven in a similar way and we omit it. ■

Theorem 4.11

Assume, for fixed s, $legal(s)$. Moreover, assume, for fixed t, a, s_1, and s_2, that

$$do(Begin(t),s_1) \sqsubset do(a,s_2) \sqsubset s, \qquad (\dagger)$$

and

$$(\exists a^*,s^*,\vec{x})[do(Begin(t),s_1) \sqsubset do(a^*,s^*) \sqsubset do(a,s_2) \wedge writes(a^*,F,\vec{x},t)]. \qquad (\ddagger)$$

We must prove that

$$a = Rollback(t) \supset ((\exists t_1)F(\vec{x},t_1,do(a,s_2)) \equiv (\exists t_2)F(\vec{x},t_2,s_1)), \qquad (\$)$$

and

$$a = Commit(t) \supset ((\exists t_1)F(\vec{x},t_1,do(a,s_2)) \equiv (\exists t_2)F(\vec{x},t_2,s_2)). \qquad (\$\$)$$

1. We first prove ($). Assume $a = Rollback(t)$; then we must show that

$$(\exists t_1)F(\vec{x},t_1,do(a,s_2)) \equiv (\exists t_2)F(\vec{x},t_2,s_1).$$

$\Longrightarrow$:

Suppose, after eliminating existentials in the conclusion, for fixed $\vec{x}$, that $F(\vec{x},t_1,do(a,s_2))$. Then, by the assumption that $a = Rollback(t)$ and that $legal(s)$ holds, Theorem 4.9 assures us that there is no other $Rollback(t)$ neither a $Commit(t)$ between $do(Begin(t),s_1)$ and $do(a,s_2)$. Furthermore, from (†), the assumption that $a = Rollback(t)$, and the axiom (4.8), we get, for fixed F,

$$\begin{aligned}&a = Rollback(t) \wedge \\ &[(\exists a^*,s^*,\vec{x}).do(Begin(t),s_1) \sqsubset do(a^*,s^*) \sqsubseteq s \wedge writes(a^*,F,\vec{x},t)] \wedge (\exists t')F(\vec{x},t',s_1) \vee \qquad (*)\\ &[(\forall a^*,s^*,\vec{x}).do(Begin(t),s_1) \sqsubset do(a^*,s^*) \sqsubseteq s \supset \neg writes(a^*,F,\vec{x},t)] \wedge (\exists t')F(\vec{x},t',s).\end{aligned}$$

Therefore, by assumption $(\ddagger)$, we have to pursue the following case:

$$\begin{aligned}&a = Rollback(t) \wedge \\ &[(\exists a^*,s^*,\vec{x}).do(Begin(t),s') \sqsubset do(a^*,s^*) \sqsubseteq s \wedge writes(a^*,F,\vec{x},t)] \wedge (\exists t')F(\vec{x},t',s').\end{aligned}$$

From this case, we get the following formulas in a straightforward way (by performing some variable renaming): $(\exists t_2)F(\vec{x}, t_2, s_1)$, $a = \mathit{Rollback}(t)$, $(\exists a^*, s^*, \vec{x}).do(Begin(t), s_1) \sqsubset do(a^*, s^*) \sqsubseteq s \wedge writes(a^*, F, \vec{x}, t)$. Henceforth we conclude that $(\exists t_2)F(\vec{x}, t_2, s_1)$.

$\Longleftarrow$:
Assume, for fixed $\vec{x}$, that $(\exists t_2)F(\vec{x}, t_2, s_1)$. Then, by (‡) and the assumption that $a = \mathit{Rollback}(t)$, we get

$$a = \mathit{Rollback}(t) \wedge (\exists t_2)F(\vec{x}, t_2, s_1) \wedge \qquad (**)$$
$$(\exists a^*, s^*, \vec{x})[do(Begin(t), s_1) \sqsubset do(a^*, s^*) \sqsubset do(a, s_2) \wedge writes(a^*, F, \vec{x}, t)].$$

From $(**)$, we get, by assumption (†), the following:

$a = \mathit{Rollback}(t) \wedge (\exists t_2)F(\vec{x}, t_2, s_1) \wedge$
$do(Begin(t), s_1) \sqsubset do(a, s_2) \wedge (\exists a^*, s^*, \vec{x})[do(Begin(t), s_1) \sqsubset do(a^*, s^*) \sqsubset do(a, s_2) \wedge writes(a^*, F, \vec{x}, t)]$.

We therefore conclude, by axiom (4.8), that $(\exists t_1)F(\vec{x}, t_1, do(a, s_2))$.

2. Now we prove ($$). Assume that $a = Commit(t)$; then we must prove that $(\exists t_1)F(\vec{x}, t_1, do(a, s_2)) \equiv (\exists t_2)F(\vec{x}, t_2, s_2))$.

$\Longrightarrow$:
Suppose, after removing all the existentials in the conclusion, for fixed $\vec{x}$, that $F(\vec{x}, t_1, do(a, s_2))$. Since $a = Commit(t)$, by axiom (4.8), we have

$$(\gamma_F^+(\vec{x}, Commit(t), \vec{t_1}, s) \vee (\exists t_1)F(\vec{x}, t_1, s_2) \wedge \neg\gamma_F^-(\vec{x}, Commit(t), \vec{t_1}, s)). \qquad (\dagger\dagger)$$

We set $\gamma_F^+(\vec{x}, Commit(t), \vec{t_1}, s) \equiv false$ and $\neg\gamma_F^-(\vec{x}, Commit(t), \vec{t_1}, s) \equiv false$.[1] Thus (††) is equivalent to $F(\vec{x}, t_2, \vec{t_1}, s_2)$, for some t_2.

$\Longleftarrow$: This case is symmetric to the first one.
Suppose, for fixed $\vec{x}$, that $(\exists t_2)F(\vec{x}, t_2, \vec{t_1}, s_2)$. Since $a = Commit(t)$, with the setting of the if-part, $F(\vec{x}, t_2, \vec{t_1}, s_2)$ is equivalent to

$$(\gamma_F^+(\vec{x}, Commit(t), \vec{t_1}, s) \vee (\exists t_2)F(\vec{x}, t_2, s_2) \wedge \neg\gamma_F^-(\vec{x}, Commit(t), s)),$$

which, by axiom (4.8), is equivalent to $F(\vec{x}, t_1, do(a, s_2))$, for some t_1. ■

Theorem 4.12

[1]In general, whenever an update a does not have any influence on the truth value of a fluent F, we set $\gamma_F^+(\vec{x}, a, \vec{t_1}, s) \equiv false$ and $\neg\gamma_F^-(\vec{x}, a, \vec{t_1}, s) \equiv false$.

By assuming, for fixed s, s', and t $legal(s)$ and $do(Commit(t), s') \sqsubseteq s$, we must prove the claim

$$\bigwedge_{IC \in \mathcal{D}_{IC_e}} IC(do(Commit(t), s')) \wedge \bigwedge_{IC \in \mathcal{D}_{IC^v}} IC(do(Commit(t), s')).$$

A. Assume, indeed, that $legal(s)$ and $do(Commit(t), s') \sqsubseteq s$. Then, by Lemma 4.7, $legal(do(Commit(t), s'))$. Henceforth, by Lemma 4.6, $Poss(Commit(t), s')$. Now, by the action precondition axiom for $Commit(t)$, $\bigwedge_{IC \in \mathcal{D}_{IC^v}} IC(s')$. Since $\gamma_F^+(\vec{x}, Commit, \vec{t_1}, s) \equiv false$ and $\gamma_F^-(\vec{x}, Commit, \vec{t_1}, s) \equiv false$, then, by axiom (4.8), we have $F(\vec{x}, t^*, s') \equiv F(\vec{x}, t, do(Commit(t), s'))$, for any t^*. Therefore, $\bigwedge_{IC \in \mathcal{D}_{IC^v}} IC(do(Commit, s'))$.

B. Suppose $s = do(T, s^*)$, where T is a ground transaction. Since $legal(do(T, s^*))$ and suppose T is the sequence $[A_1, \cdots, A_m]$, then, by Lemma 4.8, we have

$$\bigwedge_{i=1}^{m} Poss(A_i, do([A_1, \cdots, A_{i-1}], s^*)).$$

Since $do(Commit, s') \sqsubseteq do(T, s^*)$, by repeatedly applying axiom (4.5), we find out that $s^* \sqsubseteq do(Commit, s') \sqsubseteq do(T, s^*)$.

Suppose there are n actions before $Commit(t)$ in T. Thus, by the action precondition axioms for updates,

$$\bigwedge_{i=1}^{n} \bigwedge_{IC \in \mathcal{D}_{IC_e}} IC(do([A_1, \cdots, A_{i-1}], s^*)).$$

Henceforth, $\bigwedge_{IC \in \mathcal{D}_{IC_e}} IC(s')$. From this point, we obtain $\bigwedge_{IC \in \mathcal{D}_{IC_e}} IC(do(Commit, s'))$ by a reasoning similar to part A.

By combining A and B, we conclude that the claim holds. ■

Theorem 4.13

We use the relative satisfiability theorem for non-Markovian basic action theories ([Gab00]) stating that a basic action theory $\mathcal{D}$ is satisfiable iff $\mathcal{D}_{una} \cup \mathcal{D}_{S_0}$ is satisfiable. Since the relative satisfiability theorem deals with the general case of first order initial databases, take the initial database as being $\mathcal{D}_{S_0} \cup \mathcal{D}_{una} \cup \mathcal{D}_{IC}[S_0]$. Therefore, we obtain as immediate consequence that $\mathcal{D}$ is satisfiable. ■

Theorem 4.14

Suppose we fix s, s', t, and a', and assume $legal(s)$ and $do(Rollback(t), s') \sqsubseteq s$. Thus we must prove

$$committed(a, s') \equiv committed(a, do(Rollback(t), s')), \qquad (*)$$

and

$$rolledBack(a, s') \equiv rolledBack(a, do(Rollback(t), s')). \qquad (**)$$

1. First prove $(*)$.

   $\Longrightarrow$:
   Assume that $committed(a, s')$. Then, by Abbreviation (4.21), we have

   $$(\exists t^*, s^*).responsible(t^*, a, s') \wedge do(Commit(t^*), s^*) \sqsubseteq s'.$$

   Since $do(Rollback(t), s') \sqsubseteq s$, this implies that

   $$(\exists t^*, s^*).responsible(t^*, a, s') \wedge do(Commit(t^*), s^*) \sqsubseteq s' \wedge do(Rollback(t), s') \sqsubseteq s,$$

   which, by the foundational axioms, and the transitivity of $\sqsubseteq$, implies that

   $$(\exists t^*, s^*).responsible(t^*, a, s') \wedge do(Commit(t^*), s^*) \sqsubseteq s' \sqsubseteq do(Rollback(t), s') \sqsubseteq s.$$

   By (4.21), the later clearly implies that $committed(a, do(Rollback(t), s'))$.

   $\Longleftarrow$:
   Assume that $committed(a, do(Rollback(t), s'))$. Then, by Abbreviation (4.21), we have

   $$(\exists t^*, s^*).responsible(t^*, a, do(Rollback(t), s')) \wedge do(Commit(t^*), s^*) \sqsubseteq do(Rollback(t), s').$$

   Since clearly $responsible(t^*, a, do(Rollback(t), s'))$ implies that $responsible(t^*, a, s')$, and, by assumption, $do(Rollback(t), s') \sqsubseteq s$ holds, we conclude that

   $$(\exists t^*, s^*).responsible(t^*, a, s') \wedge do(Commit(t^*), s^*) \sqsubseteq do(Rollback(t), s') \sqsubseteq s,$$

   which, by the foundational axioms, implies that

   $$(\exists t^*, s^*).responsible(t^*, a, s') \wedge do(Commit(t^*), s^*) \sqsubseteq s' \wedge do(Rollback(t), s') \sqsubseteq s.$$

   Again, by (4.21), the later implies that $committed(a, s')$.

2. The case $(**)$ is proven in a similar way. ■

Lemma 4.15

Assuming, for fixed s, $legal(s)$, and, for fixed t, s_1, a, s_2, s', and F that

$$do(Begin(t), s_1) \sqsubset do(a, s_2) \sqsubset s, \qquad \text{(a)}$$

$$(\exists a^*, s^*, \vec{x})[do(Begin(t), s_1) \sqsubset do(a^*, s^*) \sqsubset do(a, s_2) \wedge writes(a^*, F, \vec{x}, t)], \qquad \text{(b)}$$

and

$$(\exists n).a = Rollback(t, n) \wedge sitAtSavePoint(t, n) = s', \qquad \text{(c)}$$

we must prove that

$$(\exists t_1)F(\vec{x}, t_1, do(a, s_2)) \equiv (\exists t_2)F(\vec{x}, t_2, s').$$

By skolemizing the existentials in the antecedents and some logical manipulation, we get the following set of assumptions

$$do(Begin(t), s_1) \sqsubset do(Rollback(t, N), s_2) \sqsubset s, \tag{a'}$$

$$do(Begin(t), s_1) \sqsubset do(a^*, s^*) \sqsubset do(Rollback(t, N), s_2) \wedge writes(a^*, F, \vec{x}, t)], \tag{b'}$$

$$sitAtSavePoint(t, N) = s', \tag{c'}$$

With (a')-(c'), we must show that

$$(\exists t_1)F(\vec{x}, t_1, do(Rollback(t, N), s_2)) \equiv (\exists t_2)F(\vec{x}, t_2, sitAtSavePoint(t, N)).$$

$\Longrightarrow$:
Suppose that $(\exists t_1)F(\vec{x}, t_1, do(Rollback(t, N), s_2))$. Then from (4.29), we get

$$restoreSavePoint(F, \vec{x}, N, t, s),$$

and, by Abbreviation 4.30, we have $F(\vec{x}, t, sitAtSavePoint(t, N))$; henceforth

$$(\exists t_2)F(\vec{x}, t_2, sitAtSavePoint(t, N)).$$

$\Longleftarrow$:
Suppose that $(\exists t_2)F(\vec{x}, t_2, sitAtSavePoint(t, N))$. Then we have

$$(\exists s^*)sitAtSavePoint(t, N) = s^* \wedge F(\vec{x}, t_2, s^*).$$

Since $legal(s)$, by assumption (a'), we have $Poss(Rollback(t, N), s_2)$; thus, by Axiom (4.25), we obtain $sitAtSavePoint(t, N) \sqsubset s$. Therefore, by Abbreviation 4.30, we conclude that

$$restoreSavePoint(F, \vec{x}, N, t, s).$$

So, with the fact that $Poss(Rollback(t, N), s_2)$, we get, by Axiom (4.29), $(\exists t_1)F(\vec{x}, t_1, do(Rollback(t, N), s_2))$.
■

Corollary 4.16

This follows from Theorem 4.11, which continues to hold for flat transactions with savepoints, and Lemma 4.15, by using the fact that $[(P \supset Q) \wedge (P \supset R \wedge S)] \supset [P \supset (Q \wedge R \wedge S)]$. ■

Theorem 4.17

Asume, for fixed s, t, n, and s', that

$$legal(s), \tag{a}$$

$$do(Rollback(t,n),s') \sqsubset s. \tag{b}$$

Now assume by contradiction that

$$(\exists n^*, s^*).do(Rollback(t,n),s') \sqsubset do(Rollback(n^*),s^*) \sqsubset s \wedge \tag{\$}$$
$$sitAtSavePoint(n) \sqsubseteq sitAtSavePoint(n^*) \sqsubset do(Rollback(t,n),s').$$

By Lemma 4.7, and assumptions (a), (b), and ($), we have (after skolemizing the existentials in ($)), $legal(Rollback(t,n),s')$ and $legal(Rollback(t,N^*),S^*)$. Therefore, by Lemma 4.8, we conclude that $Poss(Rollback(t,n),s')$ and $Poss(Rollback(t,N^*),S^*)$. Now, by the action precondition axiom (4.25), from the fact that $Poss(Rollback(t,N),S^*)$ we get

$$(\exists s_1).s_1 = sitAtSavePoint(t,N) \wedge s_1 \sqsubset S^* \wedge \tag{c}$$
$$\neg(\exists s_2,s_3).s_2 \sqsubseteq s_1 \sqsubseteq s_3 \wedge Ignore(t,s_2,s_3),$$

which is equivalent to

$$sitAtSavePoint(t,N) \sqsubset S^* \wedge \tag{d}$$
$$\neg(\exists s_2,s_3).s_2 \sqsubseteq sitAtSavePoint(t,N) \sqsubseteq s_3 \wedge Ignore(t,s_2,s_3).$$

Now notice that ($) also implies the following formula:

$$(\exists s_4)s_4 = sitAtSavePoint(t,n) \wedge \tag{e}$$
$$sitAtSavePoint(t,n) \sqsubseteq do(Rollback(t,n),s').$$

Since $sitAtSavePoint(t,n) \sqsubset do(Rollback(t,n),s')$, by Axiom (4.26) formula (e) is equivalent to

$$Ignore(t, sitAtSavePoint(t,n), do(Rollback(t,n),s')). \tag{f}$$

However, recall that we have

$$sitAtSavePoint(t,n) \sqsubset sitAtSavePoint(t,N^*) \sqsubset do(Rollback(t,n),s')).$$

This fact, combined with (f) gives

$$(\exists s_7,s_8).s_7 = sitAtSavePoint(t,N^*)s_8 \wedge Ignore(t,s_7,s_8)), \tag{g}$$

which by Axiom (4.25) would make $Poss(Rollback(t,N^*),S^*)$ false, thus leading to a contradiction. ■

Theorem 4.18

Assume, for fixed s, s', s'', and t that

$$do(Chain(t), s') \sqsubset do(Rollback(t), s'') \sqsubseteq s, \tag{a}$$

$$(\exists a^*, s^*, \vec{x})[do(Chain(t), s') \sqsubset do(a^*, s^*) \sqsubset do(Rollback(t), s'') \wedge writes(a^*, F, \vec{x}, t)], \tag{b}$$

$$\neg(\exists s^*) do(Chain(t), s') \sqsubset do(Chain(t), s^*) \sqsubset do(Rollback(t), s''). \tag{c}$$

We must prove that

$$(\exists t')F(\vec{x}, t', do(Rollback(t), s'')) \equiv (\exists t'')F(\vec{x}, t'', do(Chain(t), s')). \tag{\$\$}$$

$\Longrightarrow$:
Suppose, for fixed $\vec{x}$, that $(\exists t')F(\vec{x}, t', do(Rollback(t), s''))$. Then, by the assumption (a), Axiom 4.8, and Abbreviation 4.13 that

$$(\exists s_1).(\forall a^*, s^*)[s_1 \sqsubset do(a^*, s^*) \sqsubset s'' \supset a^* \neq Chain(t) \wedge a^* \neq Begin(t)] \wedge$$
$$\{[(\exists a^*, s^*, t^*, \vec{x}).do(Chain(t), s_1) \sqsubset do(a^*, s^*) \sqsubseteq s'' \wedge writes(a^*, F, \vec{x}, t) \wedge F(\vec{x}, t^*, s_1)] \vee$$
$$[(\forall a^*, s^*, \vec{x}).do(Chain(t), s_1) \sqsubset do(a^*, s^*) \sqsubseteq s'' \supset \neg writes(a^*, F, \vec{x}, t)] \wedge (existst^*)F(\vec{x}, t^*, s'')\}.$$

Therefore, by assumption b and (d), we conclude that

$$(\exists s_1).(\forall a^*, s^*)[s_1 \sqsubset do(a^*, s^*) \sqsubset s'' \supset a^* \neq Chain(t) \wedge a^* \neq Begin(t)] \wedge$$
$$[(\exists a^*, s^*, t^*, \vec{x}).do(Chain(t), s_1) \sqsubset do(a^*, s^*) \sqsubseteq s'' \wedge writes(a^*, F, \vec{x}, t) \wedge F(\vec{x}, t^*, s_1)].$$

This entails

$$(\exists s_1, a^*, s^*, t^*, \vec{x}).do(Chain(t), s_1) \sqsubset do(a^*, s^*) \sqsubseteq s'' \wedge writes(a^*, F, \vec{x}, t) \wedge F(\vec{x}, t^*, s_1)],$$

which in turn, by assumption (a) and (b), entails $(\exists t^*)F(\vec{x}, t^*, s')$.

$\Longleftarrow$:
Suppose for fixed $\vec{x}$ and F that $(\exists t'')F(\vec{x}, t'', do(Chain(t), s'))$. Then, by conjoining this with assumptions (b) and (a), we can conclude by Axiom 4.8 and Abbreviation 4.13 that $(\exists t')F(\vec{x}, t', do(Rollback(t), s''))$.
■

Theorem 4.19

The proof is similar to that of Theorem 4.11. The major difference lies in the fact that in addition to the assumptions (†) and (‡) made in the proof of Theorem 4.11, we must also draw the consequences of assuming, for fixed s, that $legal(s)$ holds, and, for fixed t, a, s_1, and s_2, that

$$do(Spawn(t, t'), s_1) \sqsubset do(a, s_2) \sqsubset s, \tag{†'}$$

and

$$(\exists a^*, s^*, \vec{x})[do(Spawn(t,t'), s_1) \sqsubset do(a^*, s^*) \sqsubset do(a, s_2) \wedge writes(a^*, F, \vec{x}, t)]. \qquad (\ddagger')$$

The rest of the proof is as for Theorem 4.11, but with the successor state axiom 4.49 for closed nested transactions to be used instead of axiom 4.8. ■

Lemma G.1 *Suppose $\mathcal{D}$ is a basic relational theory for closed nested transactions. Then any legal log satisfies the weak rollback and commit dependency properties; i.e.,*

$$\begin{array}{l}
\mathcal{D} \models legal(s) \supset \\
\quad (\forall t, t').\{wr\_dep(t,t',s) \supset [do(Rollback(t'), s') \sqsubset s \supset \\
\qquad [(\forall s^*)[s^* \sqsubset s \wedge do(Commit(t), s^*) \not\sqsubseteq do(Rollback(t'), s')] \supset \\
\qquad\qquad (\exists s'')do(Rollback(t), s'') \sqsubseteq s]]\} \wedge \\
\quad \{c\_dep(t,t',s) \supset [do(Commit(t), s^*) \sqsubset s \supset \\
\qquad [do(Commit(t'), s') \sqsubseteq s \supset [do(Commit(t'), s') \sqsubset do(Commit(t), s^*)]]]\}.
\end{array}$$

Proof: This is provable in a similar way as for Theorem 4.10 and we omit the proof here. ■

Theorem 4.20

By Abbreviation 4.3, we must establish two entailments:

1. $\mathcal{D} \models legal(s) \supset$
$$\{parent(t,t',s) \wedge do(Commit(t'), s') \not\sqsubseteq do(Commit(t), s'') \sqsubset s \supset (\exists s^*)do(Rollback(t'), s^*) \sqsubset s\}.$$

and

2. $\mathcal{D} \models legal(s) \supset$
$$\{parent(t,t',s) \wedge do(Commit(t'), s') \not\sqsubseteq do(Rollback(t), s'') \sqsubset s \supset (\exists s^*)do(Rollback(t'), s^*) \sqsubset s\}.$$

1. Assume, for fixed s, t, t', a, s', and s'' that

$$legal(s), \qquad (a)$$

$$parent(t,t',s), \qquad (b)$$

$$do(Commit(t'), s') \not\sqsubseteq do(Commit(t), s'') \sqsubset s. \qquad (c)$$

We must prove that $(\exists s^*)do(Rollback(t'), s^*) \sqsubset s$.

By Axiom (4.36), and the assumptions (b) and (c), we conclude that

$$(\exists s_1)do(Spawn(t,t'), s_1) \sqsubset do(Commit(t), s'') \sqsubset s).$$

Therefore, by the dependency axiom (4.47), we have $c\_dep(t, t', s'')$.

Since $c\_dep(t, t', s'')$, by Lemma G.1 and assumption (a), we obtain

$$\begin{aligned} & do(Commit(t), s_1) \sqsubset s \supset \\ & \quad [do(Commit(t'), s_2) \sqsubseteq s \supset [do(Commit(t'), s_2) \sqsubset do(Commit(t), s_1)]], \end{aligned}$$

which is logically equivalent to

$$\begin{aligned} & do(Commit(t), s_1) \not\sqsubset s \vee do(Commit(t'), s_2) \not\sqsubseteq s \vee \\ & do(Commit(t'), s_2) \sqsubset do(Commit(t), s_1), \end{aligned}$$

which, in turn, is equivalent to

$$\begin{aligned} do(Commit(t), s_1) \sqsubset s \wedge do(Commit(t'), s_2) \not\sqsubset do(Commit(t), s_1) \supset & \\ do(Commit(t'), s_2) \not\sqsubseteq s. & \end{aligned} \tag{d}$$

By assumption (c), appropriate unification, (d), and Modus Ponens, we get

$$(\forall s_2) do(Commit(t'), s_2) \not\sqsubseteq s.$$

Finally, by Theorem 4.9, we conclude that $(\exists s^*) do(Commit(t'), s^*) \sqsubseteq s$, which implies QED.

2. We make the same assumptions as in Part 1 of the proof, except that the following:

$$do(Commit(t'), s') \not\sqsubseteq do(a, s'') \sqsubset s \tag{c'}$$

replaces (c). We must prove that $(\exists s^*) do(Rollback(t'), s^*) \sqsubset s$.

By Axiom (4.36), and assumptions (b) and (c'), we get

$$(\exists s_1) do(Spawn(t, t'), s_1) \sqsubset do(Rollback(t), s'') \sqsubset s).$$

Thus, by the dependency axiom (4.48), we conclude that $wr\_dep(t', t, s'')$.

Since $c\_dep(t', t, s'')$, by Lemma G.1 and assumption (a), we obtain

$$\begin{aligned} [do(Rollback(t), s_1) \sqsubset s \supset [(\forall s^*)[s^* \sqsubset s \wedge do(Commit(t), s^*) \not\sqsubseteq do(Rollback(t), s_1)] \supset & \\ (\exists s'') do(Rollback(t), s'') \sqsubseteq s]], & \end{aligned}$$

which is logically implies that

$$\begin{aligned} do(Rollback(t), s_1) \sqsubset s \wedge do(Commit(t'), s^*) \not\sqsubseteq do(Rollback(t), s_1)] \supset & \\ (\exists s'') do(Rollback(t'), s'') \sqsubseteq s. & \end{aligned} \tag{d'}$$

By assumption (c), appropriate unification, (d'), and Modus Ponens, we conclude that

$$(\exists s^*) do(Commit(t'), s^*) \sqsubseteq s,$$

which implies QED. ■

Proposition 5.6

It is sufficient to give a formula of the past temporal fragment of the situation calculus for each of the consumption modes involved.

1. **First**. The formula in (5.19), i.e.,

$$(\exists r')e_2[r',t,s] \wedge (\exists s',r'')[s' \sqsubset s \wedge e_1[r'',t,s'] \wedge$$
$$(\forall s^*).s' \sqsubset s^* \sqsubset s \supset ((\exists r_1)e_1[r_1,t,s^*] \vee \neg(\exists r_2)e_2[r_2,t,s^*])]$$

   is expressible by the following past temporal formula:

$$(\exists r')e_2[r',t] \wedge since((\exists r'')e_1[r'',t], (\exists r_1)e_1[r_1,t] \vee \neg(\exists r_2)e_2[r_2,t]).$$

2. **Consumed Last**. The formula in (5.21), i.e.,

$$(\exists r')e_2[r',t,s] \wedge (\exists s',r'')[s' \sqsubset s \wedge e_1[r'',t,s'] \wedge$$
$$(\forall s^*).s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)e_1[r_1,t,s^*] \wedge \neg(\exists r_2)e_2[r_2,t,s^*]]$$

   is expressible by the following past temporal formula:

$$(\exists r')e_2[r',t] \wedge since((\exists r'')e_1[r'',t], \neg(\exists r_1)e_1[r_1,t] \wedge \neg(\exists r_2)e_2[r_2,t]).$$

3. **Non-Consumed Last**. The formula in (5.23), i.e.,

$$(\exists r')e_2[r',t,s] \wedge (\exists s',r'')[s' \sqsubset s \wedge e_1[r'',t,s'] \wedge$$
$$(\forall s^*).s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)e_1[r_1,t,s^*]]$$

   is expressible by the following past temporal formula:

$$(\exists r')e_2[r',t] \wedge since((\exists r'')e_1[r'',t], \neg(\exists r_1)e_1[r_1,t]).$$

4. **Cumulative**. The formula in (5.25), i.e.,

$$(\exists r')e_2[r',t,s] \wedge (\exists s',r'')[s' \sqsubset s \wedge e_1[r'',t,s'] \wedge$$
$$(\forall s^*).s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_2)e_2[r_2,t,s^*]]$$

   is expressible by the following past temporal formula:

$$(\exists r')e_2[r',t] \wedge since((\exists r'')e_1[r'',t], \neg(\exists r_2)e_2[r_2,t]).$$ ■

Theorem 5.9

Suppose an event logic $\mathcal{E} = (E, C, \mathcal{L})$ given by:

- $E = \{F\_inserted(r,t,s), F\_inserted(r,t,s), seq\_ev^{CM}(r,t,e_1,e_2,s), simult\_ev(r,t,e_1,e_2,s),$ $conj\_ev(r,t,e_1,e_2,s), disj\_ev(r,t,e_1,e_2,s), neg\_ev(r,t,e,s)\}$,
  with $CM \in \{F, CL, NL, CUMUL\}$;
- $C = \{\neg, \wedge, \sqsubset\}$;
- $\mathcal{L}$ is the past temporal fragment of the situation calculus.

Suppose further that $\mathcal{D}$ is a set of situation calculus formula that specify the semantics of events according to the event logic $\mathcal{E}$, and that $e[r,t,s]$ and $e'[r,t,s]$ are two events of the event logic $\mathcal{E}$ such that we want to establish, for given R and T, that $\mathcal{D} \models (\forall s).e[R,T,s] \supset e'[R,T,s]$. Assume that S_n is the actual situation. Since we deal with the past temporal fragment of the situation calculus, the implication problem is reducible to the problem of checking whether $\mathcal{D}$ logically implies

$$\begin{aligned}\neg e[R,T,S_0] \vee e'[R,T,S_0] \wedge \neg e[R,T,S_1] \vee e'[R,T,S_1] \wedge \cdots \wedge \\ \neg e[R,T,S_n] \vee e'[R,T,S_n],\end{aligned} \tag{G.8}$$

with $S_0 \sqsubseteq S_1 \sqsubseteq \cdots \sqsubseteq S_n$, where $S_1, S_2, \cdots S_{n-1}$ are all the successive intermediate situations between S_0 and S_n. Notice that (G.8) mentions only atoms (all of which are from the set E) and its only connectors are from the set C.

Using Proposition 5.6, we can transform (G.8) into a formula of the past temporal logic fragment of the situation calculus (and vice-versa). Since this past temporal formula will only mention atoms, we use a straightforward encoding to transform each of its atoms into a proposition. By Theorem 4.1 of [SC85], stating that propositional linear temporal logic is PSPACE-complete, we conclude that the implication problem is PSPACE-hard. The proof for the equivalence problem follows easily from the implication case. ■

Theorem 5.12

1. From (5.18), (5.20), (5.24), it is sufficient to establish the following three entailments:[2]

$$\begin{aligned}\mathcal{D} \models (\exists s')(\forall s^*)\{s' \sqsubset s^* \sqsubset s \supset ((\exists r_1)e_1[r_1,t,s^*] \vee \neg(\exists r_2)e_2[r_2,t,s^*])\} \supset \\ (\exists s'')(\forall s^{**})\{s'' \sqsubset s^{**} \sqsubset s \supset \neg(\exists r_1)e_1[r_1,t,s^{**}] \wedge \neg(\exists r_2)e_2[r_2,t,s^{**}]\},\end{aligned} \tag{G.9}$$

$$\begin{aligned}\mathcal{D} \models (\exists s')(\forall s^*)\{s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)e_1[r_1,t,s^*] \wedge \neg(\exists r_2)e_2[r_2,t,s^*]]\} \supset \\ (\exists s'')(\forall s**)\{s' \sqsubset s^{**} \sqsubset s \supset \neg(\exists r_2)e_2[r_2,t,s^{**}]]\},\end{aligned} \tag{G.10}$$

$$\begin{aligned}\mathcal{D} \models (\exists s')(\forall s^*)\{s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_2)e_2[r_2,t,s^*]]\} \supset \\ (\exists s')(\forall s^*)\{s' \sqsubset s^{**} \sqsubset s \supset ((\exists r_1)e_1[r_1,t,s^{**}] \vee \neg(\exists r_2)e_2[r_2,t,s^{**}])]\}.\end{aligned} \tag{G.11}$$

[2]Without loss of generality, we assume that e_1 and e_2 below are simple event fluents.

Let us establish the entailment (G.9). We must prove that the antecedent of the involved formula taken as premise entails the existence of a situation s'' such that

$$(\forall s^{**})\{s'' \sqsubset s^{**} \sqsubset s \supset \neg(\exists r_1)e_1[r_1, t, s^{**}] \wedge \neg(\exists r_2)e_2[r_2, t, s^{**}]\}.$$

We set $s = do(A, s'')$, for some A. Then, obviously, for all s^{**} such that $s'' \sqsubset s^{**} \sqsubset s$, we obtain vacuously $\neg(\exists r_1)e_1[r_1, t, s^{**}] \wedge \neg(\exists r_2)e_2[r_2, t, s^{**}]$. Note that the fact that $\neg(\exists r_1)e_1[r_1, t, s^{**}]$ holds will never contradict the assumptions, since there is no situation between s'' and s in which $(\exists r_1)e_1[r_1, t, s^{**}]$ could hold.

To establish the entailment (G.10), it suffice to notice that, by applying first order proof rules, we obtain the following goal to prove:

$$\begin{aligned}
&e_2[sk_1(t,s), t, s],\ sk_2(t,s) \sqsubset s,\ e_1[sk_3(t,s), t, sk_2(t,s)],\\
&sk_2(t,s) \sqsubset s^* \sqsubset s \supset (\neg(\exists r_1)e_1[r_1, t, s^*] \wedge \neg(\exists r_2)e_2[r_2, t, s^*]),\\
&s' \sqsubset S^{**} \sqsubset S, (\neg(\exists r_1)e_1[r_1, t, S^{**}] \wedge \neg(\exists r_2)e_2[r_2, t, S^{**}])\\
&\Longrightarrow\\
&\neg(\exists r_2)e_2[r_2, T, S^{**}].
\end{aligned} \tag{G.12}$$

Finally, to establish the entailments (G.11), we have to prove:

$$\begin{aligned}
&e_2[sk_1(t,s), t, s],\ sk_2(t,s) \sqsubset s,\ e_1[sk_3(t,s), t, sk_2(t,s)],\\
&sk_2(t,s) \sqsubset s^* \sqsubset s \supset \neg(\exists r_2)e_2[r_2, t, s^*],\\
&s' \sqsubset S^{**} \sqsubset S, \neg(\exists r_2)e_2[r_2, t, S^{**}]\\
&\Longrightarrow\\
&(\exists r_1)e_1[r_1, T, S^{**}] \vee \neg(\exists r_2)e_2[r_2, T, S^{**}].
\end{aligned} \tag{G.13}$$

Both entailments (G.12) and (G.13) are obvious.

2. Here, it is sufficient to establish the following three entailments:

$$\mathcal{D} \models (\exists s')(\forall s^*)\{s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)e_1[r_1, t, s^*]\} \supset$$
$$(\exists s'')\{(\forall s^{**})[s^* \sqsubset s \supset \neg((\exists r_1)e_1[r_1, t, s^{**}] \wedge \mathit{seq\_ev}(r, t, e_1, e_2, s^{**}))] \wedge$$
$$(\forall s^{**})[s^{**} \sqsubset s'' \sqsubset s \supset [(\exists r_1)e_1[r_1, t, s^{**}] \supset (\exists s_1).s_1 \sqsubset s \wedge \mathit{seq\_ev}(r, t, e_1, e_2, s_1)]]\},$$

$$\mathcal{D} \models (\exists s')\{(\forall s^{**})[s^* \sqsubset s \supset \neg((\exists r_1)e_1[r_1, t, s^{**}] \wedge \mathit{seq\_ev}(r, t, e_1, e_2, s^{**}))] \wedge$$
$$(\forall s^{**})[s^{**} \sqsubset s' \sqsubset s \supset [(\exists r_1)e_1[r_1, t, s^{**}] \supset (\exists s_1).s_1 \sqsubset s \wedge \mathit{seq\_ev}(r, t, e_1, e_2, s_1)]]\} \supset$$
$$(\exists s'')(\forall s^*)[s^* \sqsubset s \supset \neg((\exists r_1)e_1[r_1, t, s^*] \wedge \mathit{seq\_ev}(r, t, e_1, e_2, s^*))] \wedge$$
$$(\forall s^*)[s'' \sqsubset s^* \sqsubset s \supset [(\exists r_1)e_1[r_1, t, s^*] \supset (\exists s_1).s_1 \sqsubset s \wedge \mathit{seq\_ev}(r, t, e_1, e_2, s_{)}]],$$

$$\mathcal{D} \models (\exists s')(\forall s^*)[s^* \sqsubset s \supset \neg((\exists r_1)e_1[r_1, t, s^*] \wedge \mathit{seq\_ev}(r, t, e_1, e_2, s^*))] \wedge$$
$$(\forall s^*)[s' \sqsubset s^* \sqsubset s \supset [(\exists r_1)e_1[r_1, t, s^*] \supset (\exists s_1).s_1 \sqsubset s \wedge \mathit{seq\_ev}(r, t, e_1, e_2, s_1)]] \supset$$
$$(\exists s'')(\forall s^*)\{s'' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)e_1[r_1, t, s^*]\}.$$

The proofs are straightforward though tedious. Let us prove the first of these entailments. After some skolemizations and quantifier eliminations, using $\mathcal{D}$, we have, for fixed r, s, t, e_1, and e_2 (call them R, S, T, E_1 and E_2, respectively), the following to establish:

$$S' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)E_1[r_1, T, s^*]$$
$$\Longrightarrow$$
$$\{(\forall s^{**})[s^{**} \sqsubset s \supset \neg((\exists r_1)E_1[r_1, T, s^{**}] \wedge \mathit{seq\_ev}(R, T, E_1, E_2, s^{**}))] \wedge$$
$$(\forall s^{**})[s^{**} \sqsubset s'' \sqsubset s \supset [(\exists r_1)E_1[r_1, t, s^{**}] \supset (\exists s_1).s_1 \sqsubset s \wedge \mathit{seq\_ev}(R, T, E_1, E_2, s_1)]]\},$$

We first show that

$$S' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)E_1[r_1, T, s^*]$$
$$\Longrightarrow$$
$$(\forall s^{**})[s^{**} \sqsubset s \supset \neg((\exists r_1)E_1[r_1, T, s^{**}] \wedge \mathit{seq\_ev}(R, T, E_1, E_2, s^{**}))].$$

This task is equivalent (after some logical transformations) to:

$$S' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)E_1[r_1, T, s^*], S^{**} \sqsubset s,$$
$$\Longrightarrow$$
$$\mathit{seq\_ev}(R, T, E_1, E_2, S^{**}) \supset \neg(\exists r_1)E_1[r_1, T, S^{**}].$$

By moving $\mathit{seq\_ev}(R, T, E_1, E_2, S^{**})$ to the antecedents, and Modus Ponens in the antecedents and unification, we get $(\exists r_1)E_1[r_1, T, S^{**}]$.

Now we show that

$$S' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)E_1[r_1, T, s^*]$$
$$\Longrightarrow$$
$$(\forall s^{**})[s^{**} \sqsubset s'' \sqsubset s \supset [(\exists r_1)E_1[r_1, t, s^{**}] \supset (\exists s_1).s_1 \sqsubset s \wedge \mathit{seq\_ev}(R, T, E_1, E_2, s_1)]].$$

First skolemize the universal $(\forall s^{**})$, and then move $s^{**} \sqsubset s'' \sqsubset s$ and $(\exists r_1)E_1[r_1, t, s^{**}]$ (with appropriate skolem functions instead of s^{**}) before $\Longrightarrow$. After that, it is easy to see that the definition of the LIFO consumption mode given in (5.28) applies. ■

Theorem 6.10

The proof is by induction over the structure of the well fromed program T. We conduct the proof only for CNTs. Assume for fixed s that $Do(T, S_0, s)$.

Case: T is a primitive database update.
By Definition (6.3), we have

$$(\exists \delta').Trans^*(T, S_0, \delta', s) \wedge Final(\delta', s).$$

By Definition (6.1), this implies that

$$\begin{aligned}&(\exists \delta').Poss(T, S_0) \wedge \delta' = nil \wedge \\ &\quad \{(\exists a'', s'', t)[s'' = do(T, S_0) \wedge systemAct(a'', t) \wedge Poss(a'', s'') \wedge s = do(a'', s'')] \vee \\ &\quad s = do(T, S_0) \wedge [(\forall a'', t)systemAct(a'', t) \supset \neg Poss(a'', S_0)]\} \wedge \\ &\quad Final(\delta', S_0),\end{aligned} \tag{G.14}$$

which, by the semantics of $Final$ (Appendix C) and minor transformations, is equivalent to

$$\begin{aligned}&Poss(T, S_0) \wedge \\ &\{(\exists a'', s'', t)[s'' = do(T, S_0) \wedge systemAct(a'', t) \wedge Poss(a'', s'') \wedge s = do(a'', s'')] \vee \\ &\; s = do(T, S_0) \wedge [(\forall a'', t)systemAct(a'', t) \supset \neg Poss(a'', S_0)]\}.\end{aligned} \tag{G.15}$$

Now we must show that the following two formulas hold:

$$\begin{aligned}&Poss(T, S_0) \wedge (\exists a'', s'', t)[s'' = do(T, S_0) \wedge systemAct(a'', t) \wedge \\ &\qquad\qquad Poss(a'', s'') \wedge s = do(a'', s'')] \supset legal(s),\end{aligned} \tag{G.16}$$

$$Poss(T, S_0) \wedge s = do(T, S_0) \wedge [(\forall a'', t)systemAct(a'', t) \supset \neg Poss(a'', S_0)] \supset legal(s). \tag{G.17}$$

Proof of (G.16): By Lemma 4.6, we have $legal(S_0)$. Notice that (G.16) is equivalent to

$$\begin{aligned}&Poss(T, S_0) \wedge (\exists a'', t)[systemAct(a'', t) \wedge \\ &\qquad\qquad Poss(a'', do(T, S_0)) \wedge s = do(a'', do(T, S_0))] \supset legal(s).\end{aligned} \tag{G.18}$$

By assuming the antecedent of (G.18), we conclude by Lemma 4.6 and the fact that $legal(S_0)$ that $legal(s)$.
Proof of (G.17): Notice that the formula (G.17) is equivalent to

$$Poss(T, S_0) \wedge (\exists a'', t)[systemAct(a'', t) \supset \neg Poss(a'', S_0)] \supset legal(s). \tag{G.19}$$

The proof of this is immediate, using Lemma 4.6.

Case: T is a test action of the form $\Phi?$.
By Definition (6.3), we have

$$(\exists a').Holds(\Phi, S_0, s) \wedge Final(nil, s).$$

Therefore, by the semantics of $Final$, we have $Holds(\Phi, S_0, s)$. By unwinding $Holds(\Phi, S_0, s)$ and using Definition 6.8, whenever we reach a fluent literal, we record that literal into the log. Therefore, since $legal(S_0)$, by the UPA for test actions, we get by Lemma 4.6 $legal(s)$, with $s = do([\phi_1, \cdots, \phi_n], S_0)$. The ϕ_i are fluents introduced into the log by test actions generated through the unwinding of $Holds(\Phi, S_0, s)$.

Cases: T neither a primitive database update, nor a test action.
All these cases reduce to the base cases treated above by unwinding the program T using the definitions in Appendix C. ■

Theorem 6.11

As in Theorem 6.10, the proof is by induction on the structure of T. All cases to consider are like there, except for primitive database updates. Hence, we deal with this case. Assume for fixed s that $Do(T, S_0, s)$. Then, as in Theorem 6.10, by Definition (6.3), we have

$$(\exists \delta').Trans^*(T, S_0, \delta', s) \wedge Final(\delta', s).$$

By Definition (6.23), this implies that

$$\begin{aligned}&(\exists \delta', s'', a'', s^*, t).TransOf(T, t, S_0) \wedge Poss(T, S_0) \wedge \delta' = nil \wedge \\ &\quad \{[s'' = do(T, S_0) \wedge systemAct(a'', t) \wedge Poss(a'', s'') \wedge s = do(a'', s'')] \vee \\ &\quad\; s^* = do(T, S_0) \wedge [(\forall a'', t).systemAct(a'', t) \supset \neg Poss(a'', S_0)] \wedge Do(Rules(t), s^*, s)\} \wedge \\ &\quad Final(\delta', S_0),\end{aligned} \tag{G.20}$$

which, by the semantics of $Final$ (Appendix C) and minor transformations, is equivalent to

$$\begin{aligned}&(\exists s'', a'', s^*, t).TransOf(T, t, S_0) \wedge Poss(T, S_0) \wedge \\ &\quad \{[s'' = do(T, S_0) \wedge systemAct(a'', t) \wedge Poss(a'', s'') \wedge s = do(a'', s'')] \vee \\ &\quad\; s^* = do(T, S_0) \wedge [(\forall a'', t).systemAct(a'', t) \supset \neg Poss(a'', S_0)] \wedge Do(Rules(t), s^*, s)\}.\end{aligned} \tag{G.21}$$

Now we must show that the following two formulas hold:

$$(\exists s'', a'', s^*, t).TransOf(T, t, S_0) \wedge Poss(T, S_0) \wedge \\ [s'' = do(T, S_0) \wedge systemAct(a'', t) \wedge Poss(a'', s'') \wedge s = do(a'', s'')] \supset legal(s), \tag{G.22}$$

$$(\exists s'', a'', s'', s^*, t).TransOf(T, t, S_0) \wedge Poss(T, S_0) \wedge s^* = do(T, S_0) \wedge \\ [(\forall a'', t).systemAct(a'', t) \supset \neg Poss(a'', S_0)] \wedge \\ Do(Rules(t), s^*, s) \supset legal(s). \tag{G.23}$$

Proof of (G.22): By Lemma 4.6, we have $legal(S_0)$. Notice that (G.16) is equivalent to

$$(\exists a'', s^*, t).TransOf(T, t, S_0) \wedge Poss(T, S_0) \wedge \\ [systemAct(a'', t) \wedge Poss(a'', do(T, S_0)) \wedge s = do(a'', do(T, S_0))] \supset legal(s). \tag{G.24}$$

From the antecedent of (G.24), we get, by Lemma 4.6 and the fact that $legal(S_0)$, that

$$(\exists s^*, t).TransOf(T, t, S_0) \wedge S_0 \sqsubset s \wedge legal(s). \tag{G.25}$$

Therefore, we conclude that $legal(s)$.

Proof of (G.23): Through an argument similar to the proof of (G.23), we find that there is a situation s^* that is legal. Now, to execute any one of the rule programs (6.7) – (6.10) in some legal situation s^* to reach situation s, all the actions involved must have been possible according to the semantics of ConGolog programs (Appendix C). Therefore the outcome s must be legal, i.e. $legal(s)$ holds. ■

Theorem 6.15

By Definition 6.13, we must prove that, whenever Q is a database query, we have

$$(\forall t, s).[(\exists s').Do(Rules^{(2,2)}(t), s, s') \wedge Q[s']] \supset \\ [(\exists s'').Do(Rules^{(1,1)}(t), s, s'') \wedge Q[s'']]. \tag{G.26}$$

Therefore, by the definitions of $Rules^{(1,1)}(t)$ and $Rules^{(2,2)}(t)$, we need to prove that

$$
\begin{aligned}
&(\forall t, s).\\
&\{(\exists s').Do(\{(\pi \vec{x}_1, \vec{y}_1)[\tau_1[R_1, t]? \ ; \ (\zeta_1(\vec{x}_1)[R_1, t] \wedge assertionInterval(t))? \ ; \ \alpha_1(\vec{y}_1)]|\\
&\qquad\qquad\vdots\\
&\quad(\pi \vec{x}_n, \vec{y}_n)[\tau_n[R_n, t]? \ ; \ (\zeta_n(\vec{x}_n)[r_n, t] \wedge assertionInterval(t))? \ ; \ \alpha_n(\vec{y}_n)]|\\
&\quad\neg\{[(\exists \vec{x}_1)(\tau_1[R_1, t] \wedge \zeta_1(\vec{x})[R_1, t]) \vee \ldots\\
&\qquad \vee (\exists \vec{x}_n)(\tau_n[R_n, t] \wedge \zeta_n(\vec{x}_n)[R_n, t])] \wedge\\
&\qquad\qquad assertionInterval(t)\} \ ?\}, \ s, \ s') \wedge Q[s']\} \supset\\
&\{(\exists s'').Do(\{(\pi \vec{x}_1, \vec{y}_1)[\tau_1[R_1, t]? \ ; \ \zeta_1(\vec{x}_1)[R_1, t]? \ ; \ \alpha_1(\vec{y}_1)[R_1, t]]|\\
&\qquad\qquad\vdots\\
&\quad(\pi \vec{x}_n, \vec{y}_n)[\tau_n[R_n, t]? \ ; \ \zeta_n(\vec{x}_n)[R_n, t]? \ ; \ \alpha_n(\vec{y}_n)[R_n, t]]|\\
&\quad\neg[(\exists \vec{x}_1)(\tau_1[R_1, t] \wedge \zeta_1(\vec{x})[R_1, t]) \vee \ldots \vee (\exists \vec{x}_n)(\tau_n[R_n, t] \wedge \zeta_n(\vec{x}_n)[R_n, t])] \ ?\}, \ s, \ s'') \wedge Q[s'']\}.
\end{aligned}
\tag{G.27}
$$

By the definition (6.3) of Do and the semantics of ConGolog (Appendix C), we must prove:

$$
\begin{aligned}
&(\forall t, s).\\
&\{(\exists s').Trans^*(\{(\pi \vec{x}_1, \vec{y}_1)[\tau_1[R_1, t]? \ ; \ (\zeta_1(\vec{x}_1)[R_1, t] \wedge assertionInterval(t))? \ ; \ \alpha_1(\vec{y}_1)]|\\
&\qquad\qquad\vdots\\
&\quad(\pi \vec{x}_n, \vec{y}_n)[\tau_n[R_n, t]? \ ; \ (\zeta_n(\vec{x}_n)[r_n, t] \wedge assertionInterval(t))? \ ; \ \alpha_n(\vec{y}_n)]|\\
&\quad\neg\{[(\exists \vec{x}_1)(\tau_1[R_1, t] \wedge \zeta_1(\vec{x})[R_1, t]) \vee \ldots\\
&\qquad \vee (\exists \vec{x}_n)(\tau_n[R_n, t] \wedge \zeta_n(\vec{x}_n)[R_n, t])] \wedge\\
&\qquad\qquad assertionInterval(t)\} \ ?\}, \ s, \ nil, \ s') \wedge Q[s']\} \supset\\
&\{(\exists s'').Trans^*(\{(\pi \vec{x}_1, \vec{y}_1)[\tau_1[R_1, t]? \ ; \ \zeta_1(\vec{x}_1)[R_1, t]? \ ; \ \alpha_1(\vec{y}_1)[R_1, t]]|\\
&\qquad\qquad\vdots\\
&\quad(\pi \vec{x}_n, \vec{y}_n)[\tau_n[R_n, t]? \ ; \ \zeta_n(\vec{x}_n)[R_n, t]? \ ; \ \alpha_n(\vec{y}_n)[R_n, t]]|\\
&\quad\neg[(\exists \vec{x}_1)(\tau_1[R_1, t] \wedge \zeta_1(\vec{x})[R_1, t]) \vee \ldots \vee (\exists \vec{x}_n)(\tau_n[R_n, t] \wedge \zeta_n(\vec{x}_n)[R_n, t])] \ ?\}, \ s, \ nil, \ s'') \wedge Q[s'']\}.
\end{aligned}
\tag{7.28}
$$

By the semantics of $Trans$, we may unwind the $Trans^*$ predicate in the antecedent of (7.28) to obtain, in each step, a formula which is a big disjunction of the form

$$
\begin{aligned}
(\exists s').[(\phi_1^1 \wedge \phi_2^1 \wedge assertionInterval(t) \wedge \phi_3^1) \vee \cdots\\
\vee (\phi_1^n \wedge \phi_2^n \wedge assertionInterval(t) \wedge \phi_3^n) \vee \Phi] \wedge Q[s'],
\end{aligned}
\tag{7.29}
$$

where ϕ_1^i represents the formula $\tau_i[R_i, t]$, ϕ_2^i represents $\zeta_i(\vec{x}_i)[R_i, t]$, and ϕ_3^i represents the situation calculus formula generated from $\alpha_i(\vec{y}_i)$, with $i = 1, \cdots, n$; Φ represents the formula in the last test action of $Rules^{(2,2)}(t)$. Similarly, we may unwind the $Trans^*$ predicate in the consequent of (7.28) to obtain, in each step, a formula which is a big disjunction of the form

$$(\exists s'').[(\phi_1^1 \wedge \phi_2^1 \wedge \phi_3^1) \vee \cdots \vee (\phi_1^n \wedge \phi_2^n \wedge \phi_3^n) \vee \Phi'] \wedge Q[s''], \tag{7.30}$$

where ϕ_1^i, ϕ_2^i, and ϕ_3^i are to interpret as above, and Φ' represents the formula in the last test action of $Rules^{(1,1)}(t)$. Φ' differs from Φ only through the fact that Φ is a conjuction with one more conjunct which is $assertionInterval(t)$. Also, since no nested transaction is involved, and since both rule programs involved are confluent, we may set $s' = s''$. Therefore, clearly (7.29) implies (7.30). ■

Theorem 6.16

This proof is similar to that of Theorem 6.15, so we omit it. ■

Corollary 6.17

The proof is immediate from Theorems 6.15 and 6.16. ■

Theorem 6.21

The proofs are very much similar to that of Theorem 6.15, so there is no need of repeating them here. The key idea is: whenever a situation exists that satisfies a database query Q for some rule program, we might obtain the corresponding situation for the model $(1, 1)$ by just droping the nested transaction actions such as $Begin(t, t', CLOSED, NONCOMP)$, $End(t)$, etc. ■

Theorem 7.11

This is straightforward. ■

Corollary 7.12

Withouth taking the details of quantifiers into account, this is easy by

$$\begin{aligned}&(A_1 \vee A_2) \supset B \equiv \neg(A_1 \vee A_2) \vee B \equiv (\neg A_1 \wedge \neg A_2) \vee B \equiv \\ &(\neg A_1 \vee B) \wedge (\neg A_2 \vee B) \equiv (A_1 \supset B) \wedge (A_2 \supset B).\end{aligned}$$ ■

Theorem 7.13

Straightforward and similar to the proof of Theorem 7.12 ■

Corollary 7.14

The proof is immediate from Theorem 7.13 ■

Theorem 7.19

The proof is by induction on the structure of the regressable sentence W. For the purpose of carrying out this induction, we use a well-founded ordering relation $\prec$ defined in [PR99] for Markovian regressable sentences and adapted to non-Markovian structures in [Gab02a].

Let us recall this index structure. Suppose W is a non-Markovian regressable sentence of a given relational language $\mathfrak{R}$. A *maximal* situation term σ of W is a situation term that is such that there is no other situation term σ' of W with $\sigma \sqsubset \sigma'$. Define $index(W)$ as the tuple $((C, E, I, \lambda_1, \lambda_2, \cdots), P)$, where

- C is the number of logical symblols (i.e. connectives and quantifiers) mentioned by W.
- E is the number of equality atoms on situation terms mentioned by W.
- I is the number of $\sqsubset$-atoms mentioned by W.
- λ_i, $m \geq 1$, is the number of bounded maximal situations mentioned by W.
- P is the number of $Poss$ predicates mentioned by W.

Base Case: $index(W) = ((0, 0, 0, \cdots), 0)$. That is, W does not mention any of $Poss$, $\sqsubset$, or equality between situation terms. Moreover, W may mention S_0 as its only situation term. Clearly, this is the case when W is a fluent uniform in S_0 or a situation independent predicate. If W is a fluent atom (we restrict ourself to relational fluents), the only subsets of $\mathcal{D}$ and $\mathcal{D}^{\Delta}$ that can entail such a formula are $\mathcal{D}_{S_0}$ and $\mathcal{D}^{\Delta}_{ss}$, or $\mathcal{D}_{S_0}$ and $\mathcal{D}^{\Delta}_{dep}$. It is easy to verify that $\mathcal{D}_{S_0} \models W$ iff $\mathcal{D}_{ss} \models W$, and $\mathcal{D}_{S_0} \not\models W$ iff $\mathcal{D}_{dep} \not\models W$. If W is a situation independent atom, then, whenever W was in $\mathcal{D}_{S_0}$, it will also appear in $\mathcal{D}^{\Delta}_{S_0}$, so that, trivially, $\mathcal{D} \models W$ iff $\mathcal{D}^{\Delta} \models W$.

Inductive Case: Suppose that the theorem holds for all regressable formulas W' such that $index(W') \prec index(W)$. The proof now proceeds by structural induction on the form of bounded formulas. In particular, it must deal with the case where W is of the form $Poss(A(\vec{t}), \sigma)$, a relational fluent, or one of the definitions in Abbreviation 7.1. The two first cases are teated like in the proof of Theorem 2 in [PR99]. The case of the definitions in Abbreviation 7.1 is particular to the non-Markovian context. To treat just one of these definitions, suppose W is of the form $(\exists s\ :\ s = do([\alpha_1, \cdots, \alpha_n], S_0))W'$.

1. If W' is empty, then the theorem follows, since both $\mathcal{D}$ and $\mathcal{D}^{\Delta}$ contain all the foundational axioms Σ of the situation calculus which are all what is needed to establish an equality between situation terms.

2. If W' is not empty, then $W'' = W'|^{s}_{do([\alpha_1,\cdots,\alpha_n],S_0)}$ is bounded by a situation term rooted in s. So W'' mentions some s' such that $s \sqsubset s'$. It must be shown that $index(W'') \prec index(W)$. The values of P and λ_i, $m \geq 1$, are clearly the same in both W and W'', but the values for C and E are

clearly different. Hence $index(W'') \prec index(W)$. Since W and W'' are equivalent, the theorem follows by induction hypothesis. ■

Theorem 7.20

Assume the conditions of this corollary. That is, assume $\mathcal{D}^{\Delta}$ as described in the conditions of Theorem 7.19. Then, by Theorem Theorem 7.19, for any given regressable sentence G, we have $\mathcal{D} \models G$ iff $\mathcal{D}^{\Delta} \models G$. Since $\mathcal{D}$ fullfils the conditions of Clark's generalized Theorem 7.9, the conclusion of this theorem follows. ■

Theorem 7.21

The proof is straightforward and similar to that of Theorem 7.15 ■

Corollary 7.22

The proof is easy from Theorem 7.21 ■

Theorem 7.24

This is much like in Theorem 7.13 and Corollary 7.14 ■