On the Efficient Maintenance of Temporal Integrity
in Knowledge Bases

by

Dimitrios Plexousakis

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

© Copyright by Dimitrios Plexousakis 1996
Abstract

The maintenance of semantic integrity has been recognized as a cornerstone issue for the development of databases and knowledge bases alike. Despite the extensive research conducted during the last two decades, semantic integrity maintenance has yet to become a practical technology. Furthermore, the need for modeling evolving domains has given rise to challenging research issues relating to the incorporation of time in knowledge bases. In this thesis, we study the problem of maintaining the integrity of temporal deductive knowledge bases. We argue that existing approaches in either temporal or deductive databases do not address the problem in a satisfactory manner, nor do they deal with all the issues involved in a unified framework. At first, we propose an assertion language that permits us to express different types of temporal assertions that are not expressible in other formalisms. We define the notion of temporal constraint satisfaction in a bitemporal context. We then follow two orthogonal approaches to integrity maintenance, both using a combination of compile-time and run-time optimization steps. The former, termed temporal integrity monitoring, operates in two phases: compilation and evaluation. The result of the compilation phase is the derivation of simplified forms of constraints that suffice to be evaluated at run-time. The evaluation phase performs additional simplifications with respect to the transactions taking place at run-time, by combining the previously generated simplified forms. The latter of the proposals, termed transaction modification, delegates the task of integrity maintenance to determinate transactions specified in terms of precondition/postcondition pairs. It suggests additions to the transaction’s postcondition whose effect is to maintain the constraints. Both approaches demonstrate that considerable savings can be incurred by performing compile-time simplifications. The latter approach, in particular, establishes the theoretical foundations for the development of a tool that assists the database designer by providing valuable feedback concerning the safety of transactions. We show how this technique can also be applied to business process analysis and redesign. Our results show that these tasks can be assisted by analysis techniques that possess a firm theoretical basis and are aimed at proving the design correct according to the designer’s intention.
Contents

1 Introduction ........................................ 1
   1.1 Motivation ........................................ 1
   1.2 Research Issues and Existing Approaches ............. 3
   1.3 Examples ......................................... 6
   1.4 Problem Statement ................................ 8
   1.5 Organization of the Thesis ........................ 9

2 Related Work ..................................... 11
   2.1 Languages for the Specification of Temporal Constraints .......... 11
       2.1.1 Temporal Logic ................................ 11
       2.1.2 Situation Calculus ............................ 13
       2.1.3 Deontic Logic ................................ 14
       2.1.4 Event-based Formalisms ....................... 16
       2.1.5 Constraints as Epistemic Queries ............. 18
   2.2 Integrity Maintenance Methods ..................... 19
       2.2.1 Integrity Monitoring .......................... 20
       2.2.2 Integrity Maintenance by Transactions .......... 30

3 A Temporal Assertion Language .................. 37
   3.1 Overview of Telos ................................ 37
   3.2 Temporal Knowledge ................................ 39
   3.3 The Assertional Component ........................ 43
       3.3.1 Reifying the Assertion Language ............ 47
       3.3.2 Syntactic Classification of Constraints .... 48
   3.4 Constraint Satisfaction .......................... 50
3.4.1 Expressive Power ........................................... 53
3.4.2 Complexity Issues ......................................... 57

4 Temporal Integrity Monitoring .............................. 58
  4.1 Compilation and Simplification ......................... 59
    4.1.1 Compile-time Simplification ....................... 63
    4.1.2 Temporal Simplification ........................... 65
    4.1.3 Soundness and Completeness ...................... 68
  4.2 Dependence Graphs ....................................... 70
    4.2.1 Graph Construction and Properties ............... 70
  4.3 Optimizations ............................................. 79
  4.4 Evaluation Phase ......................................... 81
    4.4.1 Using Dependence Graphs for Constraint Checking 81
    4.4.2 Generating Simplified Forms for Transactions .... 84
    4.4.3 Updates of Integrity Constraints and Deductive Rules 86

5 Transaction Modification ................................ 100
  5.1 Ramifications ............................................. 103
    5.1.1 Background ........................................... 103
    5.1.2 The Frame and Ramification Problems .............. 107
  5.2 Integrity Maintenance by Transaction Modification .... 111
    5.2.1 Ramifications and Integrity Maintenance .......... 111
    5.2.2 Dynamic Integrity Constraints .................... 116
  5.3 Extensions ................................................ 120
    5.3.1 Multiple Transactions ............................... 120
    5.3.2 Multiple Constraints ................................. 123
    5.3.3 Conjoining Transaction Specifications ............ 123
    5.3.4 Inheritance of Transaction Specifications ......... 124
  5.4 Discussion ............................................... 125

6 An Application: Analysis of Business Processes .......... 127
  6.1 Business Process Analysis ................................ 127
  6.2 Situation Calculus and GOLOG ......................... 131
6.2.1 The Frame and Ramification Problems ........................................... 133
6.3 Process Simulation ........................................................................... 136
6.4 Process Analysis .............................................................................. 139
  6.4.1 Multiple Actions and Multiple Constraints ................................... 142
  6.4.2 Conjoining Action Specifications ................................................ 143
  6.4.3 Inheritance of Action Specifications ............................................. 144
6.5 Summary .......................................................................................... 144

7 Concluding Remarks ......................................................................... 146
  7.1 Summary ....................................................................................... 146
  7.2 Outlook ......................................................................................... 148

A Example: A Simple Elevator Controller ........................................... 173
List of Figures

3.1 An example Telos knowledge base ........................................... 38
3.2 Temporal relations ............................................................... 40
4.1 Temporal Simplification Table .............................................. 67
4.2 A Dependence Graph ............................................................ 71
4.3 General form of a dependence graph ....................................... 71
4.4 Compiling recursive rules ....................................................... 72
4.5 Mutually recursive rules ......................................................... 73
4.6 The $\delta$-wavefront Algorithm ............................................. 78
4.7 Interval Minimization ............................................................ 79
4.8 Evaluation phase algorithm .................................................... 83
4.9 Integrity Constraint Insertion ............................................... 88
4.10 Integrity Constraint Deletion ............................................... 89
4.11 Deductive Rule Insertion ..................................................... 91
4.12 Deductive Rule Deletion ..................................................... 93
4.13 Edge Insertion ................................................................. 95
4.14 Edge Deletion ................................................................. 97
4.15 Performance of Incremental T.C. Computation (Edge Insertion) .... 98
4.16 Performance of Incremental T.C. Computation (Edge Deletion) .... 99
5.1 Resolution-based derivation ................................................... 106
6.1 Primitive action specifications in the library management system ...... 130
6.2 Specification of the library management system (cont’d) ............... 136
6.3 Process specification in GOLOG .............................................. 137
6.4 Process specification in GOLOG (cont’d) .................................. 137
6.5 Event and axiom specifications for the library management example .... 138

A.1 Specification of an Elevator Controller ....................... 174
A.2 Strengthened Specification of an Elevator Controller .......... 175
Chapter 1

Introduction

1.1 Motivation

The maintenance of semantic integrity has been recognized as a cornerstone issue for the development of data bases and knowledge bases alike [Flo74], [EC75], [Ull88], [MCPT91], [Mat91], [LNW91]. Aimed at modeling a multitude of application domains, knowledge base management systems (hereafter KBMSs) will be required to represent consistently and reason efficiently with a large amount and a wide range of knowledge. Integrity constraints express application dependent semantics that are not built into the semantics of the knowledge representation language or the data structures used. Consequently, they constitute a means for controlling the quality of information stored in knowledge bases.

Despite the extensive research conducted during the last decade, semantic integrity enforcement has yet to become a practical technology. This is due to the lack of efficient methods for checking the satisfaction of general integrity constraints. Commercial database management systems provide automatic enforcement of limited types of integrity constraints, such as keys and referential integrity constraints, if at all. The maintenance of semantic integrity is left as a task that user-specified transactions must accomplish. This imposes an additional burden on the transaction designer. Transaction specification and implementation becomes an error-prone process, since certain constraints may escape the designer’s attention. Hence, one of the foremost goals of research in integrity constraint maintenance is to relegate the responsibility of ensuring that constraints do not become violated out of application programs and into the database management system itself. The vast majority of research results on semantic integrity maintenance concern static integrity
constraints, whereas only a few papers have dealt with the enforcement of *transition* or *dynamic* integrity constraints.

Furthermore, the need for modeling evolving domains has given rise to challenging research issues relating to the incorporation of time in knowledge bases. The, by now well-established, notions of *static* and *transitional integrity* [EC75], [Nic82], [ELG84] must be generalized to that of *temporal integrity*. The problem of ensuring that the correctness criteria expressed by integrity constraints will not be violated in a dynamic environment now has an additional dimension, namely that of monitoring time-dependent properties. Such properties arise naturally in dynamic domains and the notion of time contained in such properties can be relative or absolute. For instance, a common task of power plant monitoring systems is to enforce the requirement that values of certain parameters fluctuate in limited ways in certain time periods. Financial and trading applications involve complex operations on time-series, e.g. stock prices over time, and need to preserve constraints on the time-varying characteristics of the objects involved [CS93]. In a similar vein, Business Process Management Systems [Kar94] need to model processes enacted by humans and in which time is inherent. Properties that must be maintained by the execution of processes are naturally expressed as temporal constraints. Other applications include telecommunication systems [GT94], medical information systems, geographical [AO93], statistical and scientific databases [Mic90].

Temporal databases [Cho93], [Tan93] and their extensions, referred to as knowledge bases, [Sri88], [CI88], [Org93], [BCW93], [Ple93a], [Orc95] need robust mechanisms for ensuring that time-dependent properties do not become violated due to the evolution of the knowledge base or the passage of time. The properties that need to be ensured may refer to arbitrarily many states, past or future, of the knowledge base. Thus, the verification of the properties expressed by integrity constraints may involve reasoning in multiple states which must be available at all times. The complexity of verifying temporal integrity constraints is substantially higher than that of verifying properties that refer to a single state only or to pairs of consecutive states. In this thesis we propose techniques that simplify the verification process by performing optimization steps at times when system performance is not as critical.
1.2 Research Issues and Existing Approaches

So far, the research community has dealt with the problem of maintaining semantic integrity in contexts such as relational, deductive or object-oriented databases [Sto75], [Bry87], [CW90], [JJ91]. A detailed overview of existing methods can be found in [Ple91]. A considerable amount of work exists also on the view update problem. The integrity maintenance problem is related to the view update problem: constraints are regarded as views that must be kept empty. Thus, enforcing constraints is analogous to deleting entries from a view. View update techniques however do not lend themselves readily to providing an efficient solution to the integrity maintenance problem since they do not exploit the basic assumptions of incremental integrity checking.

On the other hand, research in temporal databases has almost exclusively adopted a relational model [Kun85b], [LS87], [Cho92a], [Cho92b], [GL93], [SW95a]. Work on temporal deductive databases has mainly dealt with the problem of finitely representing infinite temporal properties [C188],[KSW90] and the evaluation of temporal logic programs [AM87], [Cho93], [BCW93]. This thesis constitutes a proposal towards devising efficient methods for enforcing temporal integrity constraints in a structurally object-oriented framework and in the presence of temporal deductive rules. To the best of our knowledge, the research reported in [Ple93a] is the first attempt in addressing this problem.

The incorporation of time in knowledge bases for the purpose of maintaining histories of generic events and activities, or for modeling evolving domains (real-time or otherwise), gives rise to several semantical and ontological problems. Time, as a domain of possible values, is infinite by nature and, thus, temporal formulae may refer to arbitrary time points in the past or the future. However, temporal integrity constraints must be interpreted over finite structures (knowledge bases). In addition, objects may begin or cease to exist during the lifespan of the modeling effort. A preliminary effort to deal with these problems in the context of Telos [MBJK90], a knowledge representation language incorporating a dual notion of time, can be found in [Ple93b], [Ple90]. The problem of changing ontologies has also been examined in [HS91]. Different notions of constraint satisfaction in finite or infinite histories were first proposed in [LS87]. In the present document, we propose to extend the notion of constraint satisfaction over finite histories to account for the dual presence of time (as history and belief time) in our formalism.
Temporal integrity maintenance encompasses several challenging research issues. A first such issue is the specification of constraints. We will argue for a declarative, as opposed to procedural, specification of integrity constraints. Constraints specify what properties must hold in valid knowledge base states and transitions, and not how these properties are to be enforced. Procedural constraint specification ties constraint enforcement with transaction specification and leads to expensive run-time integrity checks. Declarative specification of constraints permits their treatment as first-class citizens of the knowledge base and allows automating their optimization process. First-Order Temporal Logic [Eme90] and several of its variants or subsets have been the most popular formalisms for expressing temporal constraints. Temporal logic is natural in describing occurrences of events rather than the events themselves. Transactions for instance, are not expressible in temporal logic. Despite its popularity, the computational properties of first-order temporal logic are not particularly attractive. It has no effective complete proof methods and automated theorem-proving techniques are not practical in an industrial setting. Deontic variants of Dynamic Logic [WMW89] have also been proposed for the expression of dynamic and deontic properties. In this thesis, we propose a reified temporal logic based on time intervals for the expression of integrity constraints and deductive rules. This language is a variation of the assertion language of Telos. Constraints will be characterized along several dimensions (e.g. relative time versus real-time, historical versus epistemic, etc.) and syntactic criteria for the classification of assertions will be given. The classification into different syntactic forms, allows us to use appropriate enforcement algorithms for the individual types of constraints.

Integrity constraint verification consists of determining whether all integrity constraints are satisfied in the state resulting after an update. The expressive power of the assertion language, the anticipated large numbers of integrity constraints and deductive rules, and the inherent complexity of deduction in first-order (temporal) logic constitute major impediments to constraint verification. Constraint simplification methods attempt to derive simpler forms of the integrity constraints that have to be verified when an update occurs. Simplification methods can be classified as run-time or compile-time depending on whether simplification takes place at update time or at knowledge base definition time. Naive constraint verification methods check all integrity constraints after every update. Incremental methods, on the other hand, focus attention to only a subset of all constraints, namely those affected by a particular update type. Incremental methods are based on the premise
that the knowledge base is known not to violate any constraint in the state in which an update takes place. This knowledge can be exploited for the simplification of the formulae expressing the integrity constraints as well as for the minimization of the amount of historical information that needs to be examined for constraint verification. This thesis proposes incremental compile-time simplification methods for temporal integrity constraints and deductive rules. Two orthogonal lines of research are followed, namely integrity monitoring, and integrity maintenance by transactions.

Other issues related to semantic integrity maintenance and not addressed in this thesis include constraint satisfiability, i.e., determining whether the set of integrity constraints is a satisfiable set, and integrity recovery, that is, the undertaking of actions for restoring the consistency of a knowledge base after this has been violated. Last, but not least, there is a close relationship between the notions of constraint satisfaction and rule firing, as the latter notion is used in production systems [Nil80], [For82], [Mir87], [LNR87] in Artificial Intelligence and in active databases [Day88], [IEE94], [DGG95]: a rule is fired if its condition (the negation of a constraint) is found to be violated. This suggests that implementation of temporal integrity constraint enforcement can be supported by an active database management system. Existing approaches, such as [CT94], [Ger94] do not take into account the interaction of integrity constraints with deductive rules. The problem of constraint satisfiability is a major research issue of its own right and won’t be dealt with in this dissertation. Given an initial set of constraints, finite satisfiability can be performed once and for all for all possible extensions of the knowledge base. When the knowledge base is augmented though by the addition of new constraints it is the user’s responsibility to verify that the new constraint set is satisfiable; otherwise, techniques for finite satisfiability (e.g., by model generation) must be employed. The use of techniques for testing finite satisfiability can also be adapted into the integrity recovery mechanisms for yielding models satisfying the constraints. Mainly due to the complexity of constraint satisfiability, existing results are limited to special cases only of integrity constraints [BM87], [BDM88],[Man90] and for the case of finite models. Integrity recovery on the other hand has received a fair amount of attention within the last five years. With very few exceptions [ML91], [CW91], [Wut93] integrity recovery methods are tied to the relational model and employ active rules for the support of integrity restoration actions [CW90], [CFPT92], [CFPT94], [Ger94], [CT94].

Finally, before we formally define the issues discussed in the previous paragraphs, we
would like to establish the relationship between the problem of integrity maintenance and the analysis of business processes. We argue that the proposed methods provide the machinery for performing analysis of business processes and for supporting process redesign [Ham94]. This aspect of business process reengineering is lacking tools that, based on a solid theoretical framework, can support the analysis of process specifications and different forms of reasoning about properties of processes. Preliminary results on the development of such tools were reported in [Ple95b].

1.3 Examples

Several examples of different types of temporal integrity constraints are presented in this section. The constraints are formulated in natural language and are used to expose the different dimensions along which integrity constraints may be classified. These refer to the occurrence of time in the expression of constraints as relative or real time, the knowledge base states to which the constraints refer, the periodicity of the properties expressed and thus the requirement of repeated tests for verifying the constraints, the nature of time as historical or belief time, and to the genericity of objects to which constraints refer to (object- or meta-level constraints). Note that the standard “static” constraints are only special cases of temporal constraints. The characterization “state” was chosen over “static” to refer to constraints that specify properties that must hold in any state of the domain since, even non-temporal formulae become temporal when expressed in the proposed assertion language. The examples that follow intend to give a flavor of the different requirements the various types of constraints impose on the problem of integrity maintenance.

- Relative-time vs. Real-time Constraints

  **Relative-time**: “A student can register in CSC434 only if she has taken CSC238”

  **Real-time**: “A book borrowed from the library must be returned within 10 days”

- Knowledge Base States

  **State**: “An employee’s salary must be less than her manager’s”

  **Dynamic**: “An employee’s salary must never decrease”

- Periodicity
Periodic: “At the end of each day, every account balance must be greater than zero”

N-times aperiodic: “A student registering to the Ph.D. programme must graduate within 6 years”

- History and Belief Time

  Historical: “An employee cannot get a raise more than twice in the same year”

  Epistemic: “The knowledge base cannot stop believing in a class definition”

- Object-Level vs. Meta-Level Constraints

  Object-level: “An employee cannot become a manager unless she has completed 3 years of work with the same company”

  Meta-level: “All classes should have a non-empty extension at all times”

A few comments about the examples are in order. As far as the occurrence of time in constraint expressions is concerned, the first example refers to events whose occurrences stand only in some relative order in time, whereas the second relates event occurrences by an interval expressed in absolute time units. In principle, the first constraint can be verified by looking up the history of the database. A single state satisfying the required property suffices for verifying the constraint, but, in general, the entire history of the knowledge base up to the current state needs to be examined. A notion of current-time must be incorporated in a language suitable for expressing real-time constraints such as the one in the second example. Exogenous events, such as the passage of time, may be the cause of violations of real-time constraints. Hence, the verification of real-time constraints must also take into account the change in current time in addition to the changes in the knowledge base. The above types of constraints can be treated in a uniform manner if we assume that the passage of time takes place by the occurrence of a transaction whose only effect is to increment the current time by one unit of time.

The satisfaction of constraints expressing a property that must always hold should be tested at every state. Such constraints - called “state constraints” in the above classification - refer to a single state of the knowledge base. Dynamic constraints on the other hand, express properties referring to any number of consecutive states, and their verification
requires the maintenance of the history of the knowledge base state transitions. Dynamic constraints are more general than the traditional transition constraints which refer to pairs of consecutive states. When a temporal logic is used to express all of the above types of constraints, state constraints also become temporal formulae and can thus be treated as special cases of more general temporal constraints.

Periodic constraints express properties that must hold at points in time separated by a fixed interval. The periodicity of the expressed properties imposes a requirement to repeatedly verify the same formulae which may also refer to an arbitrary number of knowledge base states. The verification of aperiodic properties may also require repeated checks, but in this case, the times at which the formulae must be tested do not follow a canonical pattern as in the case of periodic constraints.

We wish to examine the problem of maintaining integrity constraints in a bitemporal context, where both the history of the domain and that of the knowledge base is modeled. Thus, temporal constraints may refer to the history of the domain (historical) or to the system's knowledge of this history (epistemic). The two temporal dimensions have different properties and give rise to distinct requirements for verifying the respective types of constraints.

Last, but not least, we wish to be able to express properties referring to generic objects and properties about the domain model and its evolution. Constraints on generic objects must be satisfied by each one of their instances. Moreover, constraints on the allowable changes in the domain model require that the history of the knowledge base itself is known and available for verifying such properties.

As the above examples show, the properties used to distinguish among the different types of constraints are not necessarily mutually exclusive.

1.4 Problem Statement

In the second section of this introduction, we gave a fairly superficial description of the problem of maintaining temporal integrity constraints. This section summarizes the research issues involved in the general problem examined in this dissertation, as well as the solutions proposed for several of these issues.

The problem of temporal integrity constraint maintenance can be broken down to the
following subproblems:

1. Definition of a language for expressing temporal assertions.

2. Formalization of the notion of constraint satisfaction.

3. Development of algorithms for efficiently verifying the satisfaction and satisfiability of constraints.

4. Devising integrity recovery mechanisms.

5. Integration of integrity maintenance in a knowledge base management system.

The initial focus of this research has been on the development of an integrity monitoring method employing compile-time simplification algorithms for temporal integrity constraints and deductive rules, defined using a variation of the assertion language of Teles [Ple93a], [Ple95a], [Ple94]. As far as integrity enforcement by transactions is concerned, we propose an incremental transaction modification technique for deterministic transactions [PM96], based on results from [BMR93],[LR92] and [Pin94].

The problems of constraint satisfiability and integrity recovery will not be dealt with in this dissertation since they constitute major research problems in their own right. Results on the integration of integrity checking in a knowledge base management system appear in [MCPT92] and [MCP+96]. We discuss the integration of the particular techniques along with the presentation of the techniques themselves. We compare our results with other approaches to knowledge base management such as, for example, [Mat91] and [LNW91].

1.5 Organization of the Thesis

In the rest of this document, we elaborate on the approach taken and on results obtained on the problem of temporal integrity maintenance. Chapter 2 reviews related work from relational, deductive, object-oriented, active and temporal databases. Chapter 3 introduces a language for specifying temporal assertions and defines the notion of constraint satisfaction in a bitemporal context. Chapter 4 presents a two-phase simplification method for temporal integrity monitoring, whereas chapter 5 presents a dual approach, namely constraint maintenance by transactions. Chapter 6 shows the application of the latter technique for constraint maintenance to business process reengineering and, in particular, in the support
of the process analysis and redesign phase. Chapter 7 concludes with a summary of the contributions of this thesis and an outline of directions for further research.
Chapter 2

Related Work

In this chapter we present a survey of research related to the main part of this dissertation. We concentrate on the following research areas: (i) languages proposed for the expression of dynamic (temporal) integrity constraints, (ii) integrity maintenance methods proposed for relational, deductive, object-oriented and temporal databases. The survey presented here is by no means complete. We discuss into some detail only (what we consider to be) the most influential proposals; the rest are mentioned briefly. The interested reader is referred to [Ple91], for a more detailed survey, and to the published papers.

We begin this chapter by reviewing different formalisms for the expression of integrity constraints (section 2.1). It has to be noted here that a considerable amount of work exists in areas such as philosophical logic, artificial intelligence and concurrent programming, that relates to the problem studied in this dissertation in the sense that common formalisms are used. We are interested in the ways these formalisms have been used in database theory and for the expression of integrity constraints in particular, and, thus, we will not devote but mere citations to these works. In section 2.2 we study some important proposals for the problem of the maintenance of integrity constraints, following integrity monitoring or maintenance by transactions.

2.1 Languages for the Specification of Temporal Constraints

2.1.1 Temporal Logic

Propositional Temporal Logic (PTL) and First-Order Temporal Logic [Eme90], [AH90],
and several of their variants have been the most popular languages chosen for the specification of temporal integrity constraints. The papers [CF84], [ELG84], [Kun84], [Kun85b], [LS87], [SL88b], [SL88a], [HS90], [FS88], [Lip90], [HS91], [Cho92a], [Cho92b], [CN93], [SHS94], [GL95], [Cho95], [SW95a] make up only a partial list of the approaches that have appeared in the literature of the recent years and in which the language used is a fragment of PTL or FOTL. For the remainder, of this section we will focus our discussion on FOTL rather than PTL, since the former includes the latter.

FOTL extends First-Order Logic [Smu68], [End72] by the introduction of temporal connectives. The temporal connectives $\circ$ and until are called future connectives, whereas $\bullet$ and since are called past connectives. For a FOL formula $\phi$, $\circ\phi$ is read as “next time $\phi$” and $\bullet\phi$ is read as “previous time $\phi$”. Based on these, other temporal connectives can be defined: $\Box$, $\Diamond$, $\blacksquare$ and $\blacklozenge$ with the meaning “always in the future”, “sometime in the future”, “always in the past” and “sometime in the past” respectively. For example, consider the integrity constraint specifying the property that an employee’s salary should never decrease. This constraint is expressed in the future fragment of FOTL as the formula

$$\forall x, s_1, s_2 \Box(\text{employee}(x) \land \text{salary}(x, s_1) \land \Diamond \text{salary}(x, s_2) \Rightarrow (s_2 \geq s_1))$$

The same constraint can be expressed in the past fragment of FOTL as the formula

$$\forall x, s_1, s_2 \blacksquare(\text{employee}(x) \land \text{salary}(x, s_2) \land \blacklozenge \text{salary}(x, s_1) \Rightarrow (s_2 \geq s_1))$$

Similar connectives have been used in tense logics [RU71]. For instance, the unary connectives $\mathcal{F}, \mathcal{G}, \mathcal{P}, \mathcal{H}$ have the meaning “it will be the case that ...”, “it will always be the case that ...”, “it has been the case that ...” and “it has always been the case that ...” respectively. The connectives $\mathcal{F}$ and $\mathcal{G}$ ($\mathcal{P}, \mathcal{H}$ resp.) are not the exact analogues of $\Diamond$ and $\Box$ ($\blacklozenge$ and $\blacksquare$ resp.). The following equivalences hold: $\Diamond A \equiv A \lor \mathcal{F} A$, $\Box A \equiv A \land \mathcal{G} A$, $\blacklozenge A \equiv A \lor \mathcal{P} A$, $\blacksquare A \equiv A \land \mathcal{H} A$.

Despite the popularity of FOTL as a language for expressing temporal constraints and queries, its computational properties are not as attractive. It has been established [Kam68] that PTL is equally expressive as monadic first-order logic. The exact characterization of the expressive power of FOTL as a query language is still an open problem. It can be easily seen though that two-sorted first-order logic in which one of the sorts is the set of integers with the standard ordering and every predicate is extended with an additional temporal argument, is at least as expressive as FOTL [Cho95]. Moreover it has been shown
that the satisfiability of restricted classes of FOTL formulas is an undecidable problem [CN93]. Specifically, the problem becomes decidable for the class of universal formulae, which are formulae in which no quantifier occurs in the scope of a temporal connective and all quantifiers are universal. The problem becomes undecidable if a single quantifier occurs in the scope of a temporal connective.

It has also been argued [CN93] that temporal integrity constraints should express safety properties [Sis85]. Intuitively, safety properties assert that an unwanted situation never arises. Formally, a temporal logic formula $\phi$ expresses a safety property if, $\phi$ holds in a sequence of states if and only if every prefix of this sequence can be extended to satisfy $\phi$. Under that restriction, a formula of the form $\Diamond \exists p(x)$ would be disallowed as an integrity constraint. In fact, the violation of such a formula cannot be determined using a finite structure. Moreover, if $p(x)$ is not tautologically false, this formula is always potentially satisfiable in an extension of the current history\(^1\) to the future. Such a formula defines what is called a liveness property, i.e., one that expresses that a desired situation will eventually arise.

### 2.1.2 Situation Calculus

The situation calculus [McC69] is a first-order language for representing dynamically evolving domains. Changes are brought to being in states of the world, situations, as the results of actions performed by an agent. A situation calculus structure thus contains a set $A$ of actions and a set $S$ of situations. For an action $\alpha \in A$ and a situation $s \in S$, the term $do(\alpha, s)$ denotes the situation that results from the execution of action $\alpha$ in situation $s$. Relations whose truth values may differ from one situation to another are called fluents. They are denoted by predicate symbols having a situation term as their last argument. Similarly, the term functional fluent is used to denote functions whose denotation varies from one situation to another. All actions in the situation calculus are assumed to be primitive and determinate.

The situation calculus has been traditionally used for hypothetical reasoning in Artificial Intelligence. Reiter proposed its use in the specification of database updates [Rei95] and the specification of state constraints [LR92], [LR94]. One of the main drawbacks of the situation calculus is its inability to constrain the occurrences of events. Unlike linear temporal logics,

\(^1\)FOTL formulae are interpreted over a history of domain states.
it can be viewed as defining a branching-time structure. The situation calculus is extended
to deal with a wider range of representational and reasoning issues, temporal reasoning
in particular, in [PR93], [Pin94]. A time line is embedded in the situation calculus by
associating with actual situations a starting time and an ending time. Both are time points
taken from a dense domain. Actual situations correspond to those that result from actions
that have taken place in the evolution of the domain being modeled. Hence, every actual
situation can be seen as a homogeneous interval during which the truth values of fluents
persist. Actions take place at the end of situations. It is also shown that most of the
representational features of other popular temporal logics, such as FOTL, Allen’s interval-
based temporal logic [All83] and the Event Calculus [KS86], can be realized in the extended
situation calculus.

Of particular interest to the present work is the formulation of database transactions in
the situation calculus as this appears in [Rei95]. Transactions are formulated as actions are
in planning domains. Reiter also provides a solution to the frame problem as it arises in
the context of database transactions. This solution is employed for proving the satisfaction
of static and dynamic constraints using a form of mathematical induction. Our canonical
example of an integrity constraint is formulated as the situation calculus formula

\[ \forall s, s', \forall p, sal_1, sal_2 \ S_0 \leq s \land s \leq s' \land sal(p, s, sal_1) \land sal(p, s, sal_2) \Rightarrow sal_1 \leq sal_2 \]

where, \( s, s' \) are situations and \( S_0 \) is the initial situation. Constraints such as the above can
be proved by using induction on the database states. Theorem-proving is performed only
with respect to the initial database state. Despite this attractive property, this approach
does not take advantage of the knowledge that constraints were known to be satisfied in
the history of states before a transaction takes place.

### 2.1.3 Deontic Logic

The use of Deontic Logic (DL) for the specification of dynamic constraints is proposed in
[MWW89], [WMW89] and [WWMD91]. DL includes first-order and dynamic logic [SWM93]
as special cases. It provides modal operators that express obligation \( O \), prohibition \( F \)
or permission \( P \) and which are used to qualify actions. Actions can be combined using
operators for non-deterministic choice, sequential and parallel execution. For an action \( a \)
and a DL formula \( \phi \), \([a] \phi \) is a formula that is true in all states in which the execution of
\(\alpha\) necessarily leads to a state where \(\phi\) holds. DL is compositional in the sense that the semantics of its formulae can be defined in terms of the composition of actions that lead to the state where the formulae are true. This is a property that the situation calculus also possesses, but FOTL doesn’t. All deontic concepts can be reduced to that of prohibition. Thus, the violation of constraints can be specified explicitly. For instance, the constraint that specifies the property that an employee’s salary should never decrease can be specified as the DL formula

\[
\forall e, s_1, s_2 [\text{employee}(e) \land \text{salary}(e, s_1) \land (s_2 < s_1) \Rightarrow [\text{change} - \text{salary}(e, s_2)] V : \\
\text{salary} - \text{change}(e, s_2)]
\]

The above formula says that a violation arises from the execution of action \(\text{salary} - \text{change}\) if the value of the new salary is less than that of the old. It is also required that correction actions be specified explicitly in order to bring the knowledge base to a consistent state. We feel that such a formulation of integrity constraints should be the result of analysis that is performed after an initial specification of the constraints. Especially the formulation of repairing axioms puts additional burden to the specifier. It is not discussed whether the derivation of the violation conditions and repairing actions can be performed automatically.

We find the main contribution of this work to be the distinction between constraints that are necessary truths of the universe of discourse and deontic constraints, which express properties that constrain the universe of discourse. An example of the former type would be the constraint specifying that a person’s age should be a non-negative integer, whereas as example of the latter type would be the constraint specifying that a book borrowed from the library should be returned within three weeks from the date it was borrowed. The different types of constraints are enforceable in distinct ways. Specifically, deontic constraints cannot simply be enforced by the rejection of an updating transaction. In fact, for a deontic constraint such as the one presented above, there is no update (except the passage of time) that may lead to the violation of the constraint. Once the constraint is discovered to be violated, the violation can be corrected by the update that consists of the borrower returning the book to the library. Hence, the distinction is practical from the point of view of adopting the necessary strategy for constraint enforcement.

The problem of the inheritance of dynamic and deontic integrity constraints in taxonomies of types is also studied in [WWMD91]. The authors show the pitfalls that exist as
far as inheritance is concerned. However, the logic is not well suited for specifying temporal
constraints as time is not explicit. In a nutshell, the merits of the approach can be found
at its conciseness and accuracy in expressing different types of intentions. However, the
problem of checking the constraints is not discussed. Our approach can also deal with the
problem of inheritance using the concept of the concerned class in Telos and as shown in
chapter 5.

2.1.4 Event-based Formalisms

Event Calculus

In [KS86], Kowalski and Sergot proposed the event calculus as a formalism for reasoning
about persistence in a logic programming [Llo87] framework. The intended applications
of such a formalism were database updates and narrative understanding. The non-monotonic
behaviour associated with reasoning about time and persistence is realized through negation
as failure [Cla78].

The primitive notion of this formalization is that of an event. Time points and periods
associated with the validity of facts in the application domain are derived from the spec-
ification of event occurrences. Both events and times are represented by means of Horn
clauses. A main difference between event calculus and the situation calculus is that the
latter deals with global states of the universe of discourse, whereas the former deals with
local events. Updates are realized by the addition of new knowledge to the knowledge base
and no explicit deletion occurs. New knowledge about the end of periods of time for which
the information to be deleted holds is added.\(^2\) This is a divergence from the standard
treatment of deletion in conventional databases. Past and future are treated symmetrically
and this permits the treatment of retroactive and proactive updates. Integrity constraints
are not treated explicitly, but the machinery for expressing them is provided. The event
calculus was developed as to avoid the frame problem. Extensions to the original proposal
were made in order to deal with the ramification problem as well [KS94].

Most interesting, and closer to the problem examined in this dissertation, are the ex-
tensions of the event calculus for reasoning not only about the history of the universe of
discourse, but also about the knowledge that the knowledge base possesses about this his-

\(^2\)The treatment of deletion in Telos is similar.
tory. This extension gave rise to a logical framework for temporal deductive databases [Sri88], [Sri91]. The formalization of historical and belief time is carried out by means of a refined with respect to time version of the event calculus. A query evaluation procedure is proposed that may be employed for constraint verification in a bitemporal context. The procedure lies entirely within a logic programming framework. We feel that the method we propose in this dissertation can benefit from the adaptation of similar techniques for the run-time evaluation of simplified integrity constraints.

To conclude this section we present the salary-change constraint formulated in the extended event calculus. The expression of the constraint uses the meta-predicate \text{HoldsAt} that qualifies an atomic proposition with the point in time in which the proposition is true.

\[
\forall x, s_1, s_2 \ \text{HoldsAt}(\text{salary}(x, s_1), T_1) \land \text{HoldsAt}(\text{salary}(x, s_2), T_2) \land T_1 < T_2 \Rightarrow s_1 \leq s_2
\]

Reasoning is then performed by instantiating meta-level clauses defining axioms of the knowledge base [Sri93]. The times at which propositions are taken to be true are obtained from the specification of event occurrences.

**Active Rules**

The emergence of active database systems [Mor83], [Day88] made possible the specification of reactive behaviour in databases. Integrity constraints can be specified and enforced by means of rules whose execution is triggered by the occurrence of events. \textit{Production} rules [CFPT92], [CFPT94], [FP92], or \textit{Event-Condition-Action} (ECA) rules [Kot88], [MD89], [UrP92], [EGS92], [Bay93] are used for specifying the system behaviour in the occurrence of events. The actions performed may be of the same type as the actions that caused the event occurrence. Such a mechanism permits the support of database functions such as integrity enforcement [CW90], [FPT92] and view maintenance [CW91], [Bay92]. Our example constraint may be specified as the following ECA rule:

\text{ON Update(salary(x, new) IF } \exists \text{old salary}(x, \text{old}) \land (\text{old} > \text{new}) \text{ THEN reject;}

Many approaches advocating the use of active rules for the specification and enforcement of integrity constraints have appeared in the literature of the recent years (e.g., [RCBB89], [Cas89], [ELW90], [UKN92], [FPT92], [GJS92], [GJ91], [GL93], [Ger94], [CT94]). Several research prototypes provide for active rule specification and execution (e.g., ODE [GJ91],
HIPAC [DBB88], POSTGRES [SJGP90], STARBURST [LLPS91], [WCL91], CHIMERA
[CFP94], RDL1 [SKdM92], REFLEX [N93], LAURE [Cas91b]. In the majority of the
proposals the underlying data model is relational and the types of constraints specifiable
are limited to static ones. A few researchers have dealt with the problem of maintain-
ing integrity constraints via active rules in deductive ([ML91], [Cas91a], [SKdM92]) and
object-oriented databases ([Cas89], [DUHK92], [KU93]). Transition and general dynamic
integrity constraints are supported by ECA rule languages that include temporal connect-
tives [SW95a], [SW95b].

The main advantage of constraint specification using the active rule paradigm is the
ability to specify, in a unified framework, the conditions to be checked as well as the en-
forcement strategy to be followed. Efficiency is another advantage when rules are coupled
with higher-level optimization strategies. The drawback is the lack of declarativeness, since
the performance of the action part of rules takes place in the same way as other database
transactions do. Although rules may be specified declaratively [Cer92], they do not always
have a declarative semantics. Properties that must be guaranteed are termination, i.e., the
avoidance of infinite executions, and confluence, i.e., ensuring that all possible executions
reach the same final state. Constraint analysis techniques [UD90], [CFPT94] have been
developed to assist in the design of a rule system with the desired behaviour.

2.1.5 Constraints as Epistemic Queries

Reiter’s view of integrity constraints [Rei88], [Rei90] builds upon the argument that queries
should be allowed to address both the aspects of the world modeled by the knowledge
base and the knowledge base’s knowledge about the world. A subset of this type of queries,
formulated as modal formulae in the epistemic logic $KTOPC$ [Lev84], is used for expressing
integrity constraints. Reiter proposes a Prolog-like query evaluator that incorporates the
negation-as-failure rule and a left-to-right evaluation rule. It is shown to be sound for a
subclass of $KTOPC$ formulae, namely the admissible formulae. A $KTOPC$ formula is
admissible if and only if it is safe, rectified and the scope of every existential quantifier or
negation sign is either a first-order formula or a subjective formula. Safe $KTOPC$ formulae
have the property that no negative subgoal contains a free variable which does not occur in
a positive subgoal (when the formula is expressed as a conjunction). As their name implies,
subjective formulae refer only to the database’s epistemic state. Any subjective formula
has the property that either the formula or its negation is a theorem of a first-order theory. Integrity constraints are subjective formulae\(^3\). The proposed query evaluator assumes the existence of a sound and complete first-order theorem prover, while no assumption is made about the first-order theory. It is shown that the query evaluator is sound for admissible constraints for any satisfiable first-order database. This result allows for the evaluation of queries and constraints that are more expressive than simple conjunctions of literals [GU92]. Completeness is also demonstrated for a more restricted class of queries and first-order theories: elementary theories extend deductive databases consisting of normal clauses by allowing disjunction and existential quantification. The evaluator is complete for queries with finitely many answers with respect to an elementary theory. Finally, it is shown that, under the Closed-World Assumption [Rei78], testing integrity satisfaction reduces to first-order entailment.

### 2.2 Integrity Maintenance Methods

This section briefly reviews some of the most prominent methods for constraint maintenance in relational, deductive and object-oriented database systems. Although this work focuses on temporal constraints in knowledge bases, methods that have been developed for enforcing constraints in relational databases, were extended to, or constitute the basis for, methods for enforcing constraints in deductive databases and knowledge bases. Adopting the generally acceptable view that the incorporation of general constraints in knowledge bases is not only desirable but also sanctioned by pragmatic criteria, we will consider methods that deal with general constraints. Enforcement methods for special types of functional dependencies [Mai83] and other simple forms of constraints will not be considered here.

The methods reviewed are classified under integrity monitoring or transaction modification, depending on whether the maintenance of constraints is undertaken by a centralized application-independent component or updating transactions are modified to become integrity preserving.

The naive approach of checking all integrity constraints after each update is highly impractical. The sources of the impracticality are identified with the high cost of reversing the update in case constraints are violated and the need to check too many elements at run-time.

\(^3\)Unfortunately, this view cannot be adopted for certain types of temporal constraints.
The idea of *incremental checking* of constraints considerably simplifies the task of constraint checking. In the remainder of this section we will only review incremental methods. Most of the constraint enforcement methods in relational, deductive databases and knowledge bases, share a set of common principles, which express the concept of incremental checking. Three such principles are listed below:

- the database state is known to be consistent prior to the update
- knowledge about updates and constraints is exploited
- updates are embedded into the original constraints, so they can be evaluated in the current state, or
- constraints are embedded into the updating transactions, so that the latter become safe.

### 2.2.1 Integrity Monitoring

Integrity monitoring relies on the presence of a centralized subsystem for the maintenance of integrity constraints. Updating transactions are embedded into constraints which then can be evaluated for verifying their satisfaction. This approach has the advantage that knowledge about the consistency of the knowledge base is maintained by the monitoring system and constraint enforcement can be carried out in an application-independent manner.

**Relational Databases**

In this section, methods that transform constraints into simplified forms that are easier to evaluate are presented. A most prominent method, extended by many researchers to methods applicable in deductive databases, is Nicolas' method for constraint simplification [Nic82]. Nicolas was among the first to observe that considerable savings for the task of checking constraint satisfaction can be incurred by taking advantage of the fact that, prior to an update, a database is known to satisfy its constraints: attention should be restricted to a subset only of the constraints imposed on the database, namely those that are affected by or related to a specific update. The method produces simplified forms equivalent to the original integrity constraints and which are sufficient to evaluate in the state resulting after the update. The generated forms depend on the nature of the updating operation. The
formulae expressing the integrity constraints are function-free first-order predicate calculus formulae in prenex conjunctive normal form (PCNF). They are also assumed to satisfy the \textit{range-restricted property}. Simplified forms are obtained by substituting for variables in the constraints constants appearing in the added, deleted or updated tuple. The method operates after the actual updates have been specified and is only applicable to single updates.

Bernstein, Blaustein and Clarke [BBC80], [BB82] proposed the use of aggregate data for enforcing a class of assertions expressed in a restricted version of relational calculus. The method is based on the automatic maintenance of redundant aggregate data. The presence of redundant data reduces the cost of checking the consistency of the database state resulting from an update. Moreover, the data are such that their maintenance is not prohibitively expensive. Examples of such redundant data are, for instance, the least upper bound or the greatest lower bound of a set. The aggregate data are then used to produce run-time tests sufficient for the maintenance of a limited class of constraints. Consistency tests are constructed and performed before the update is actually executed. The limited form of assertions is a drawback compared to other methods.

In [HI85], a generalization of the methods of [Nic82] and [BBC80] is proposed. The method deals with multiple updates, where the term “multiple” means that updates are neither restricted to single tuples nor to single ranges, as is the case of [BBC80]. A simplification algorithm for this type of updates is presented. Constraints are expressed in tuple relational calculus and in prenex form. Arbitrary transactions are considered. The algorithm yields a conjunction or disjunction of simpler constraints. It does not assume any knowledge of the exact structure of the update, as, for example, in [Nic82]. However, because of lack of specific knowledge this method can only constitute a preliminary constraint check.

A theorem-proving approach to constraint satisfaction is proposed in [HMN84]. The proposed method consists of a compilation phase which, at database design time, generates tests sufficient to guarantee the satisfaction of constraints in the updated database. It does not assume an exact knowledge of the update. It employs a theorem prover accepting as input clauses representing the original constraint, assumed to be satisfied prior to the update, a set of transition axioms which define updated relations, and the negated constraint expressed in terms of the updated relations. The theorem prover attempts to construct a refutation of the negated constraint using set-of-support resolution and clause elimination
by subsumption [GN87]. If such a refutation is obtained, then the constraint is satisfied
in the new state. Otherwise, any subset of the set of clauses obtained after resolution
and elimination constitutes a validity test. Unlike Nicolas’ method, this method is not
complete. If the validity test fails, consistency is not necessarily violated. Unlike the same
method also, compilation can be carried out before the updates are specified. At update-
time the generated tests, if any, can be instantiated according to the updates. The method
presupposes that constraints are domain-independent formulæ.

Last, but not least, Qian [QW86], [QS87] proposed an automated transformational
mechanism which exploits knowledge about the application domain and the organization
of the database, in order to reformulate integrity constraints into ones that enforce the
same conditions, but from which efficient code can be generated. The proposed method
dynamically adapts the formulation of constraints to the current application semantics and
database organization. The general task of constraint reformulation is expressed as follows:
given knowledge \( A \) and constraints \( C \), find simplified constraints \( S \) so that \( A \land S \rightarrow C \).
Reformulation of constraints is performed at constraint specification time. However, since
the existing knowledge is subject to invalidation itself, reformulation has to be reapplied
when the antecedent knowledge \( A \) becomes invalidated. Methods used for the reformulation
of constraints include finite differencing [KP81], which replaces costly computations on sets
of objects by incremental updates. From this discussion, it can be seen that, for the method
to be applicable, knowledge has to be extracted from the schema, the constraints and from
the monitoring of the database. In fact, finite differencing is not effective unless it takes
advantage of database-related information. The amount of knowledge maintained must be
relatively small so that its maintenance does not create additional burden. The knowledge
maintained must be generic knowledge about classes of objects rather than individual object
instances.

**Deductive Databases**

Constraint maintenance methods in deductive databases are classified as run-time and sim-
plication methods. The former can be carried out after the update has taken place and
follow an interpretive approach, whereas simplification methods involve a phase during
which simplified forms of constraints are generated. This phase can, in most cases, be car-
rried out at compile-time, without accessing the database, and aims at simplifying the task
of run-time checking for integrity constraints. The computation of implicit updates, i.e.,
the computation of the changes to the set of logical consequences of the database, constit-
tutes a main point of comparison between the different proposals. Another such point will
be whether reasoning with both the states before and after the update is required for the
calculation of implicit changes or for the verification of constraints.

An early treatment of integrity maintenance in deductive databases is presented in
[NY78]. The method is applicable at run time, when the update is specified. No simpli-
lication is performed on the constraints affected by the updates. Special attention is paid
though to integrity recovery. Two types of actions are proposed, namely rejecting the up-
date when a constraint is violated or automatically generating other operations so that the
constraint is satisfied thereafter. The choice between the two candidate actions depends on
whether the action will result in invalidating other constraints or in deducing redundant
information. The use of a meta-language for specifying the actions to be taken in case of
consistency violation is proposed, as opposed to the use of triggers, which would involve
having to keep multiple expressions of a constraint, each suitable for a particular type
of update. The presence of deductive rules is exploited for deriving implicit information
(query rules) and realizing derived information (generation rules). For integrity recovery
in particular, the latter type of rules are used for generating information that revalidates
the violated constraint. Transition constraints are handled by defining action relations, one
for each relation and each type of update. Their extensions contain the tuple that is to be
inserted/deleted/updated when the operation takes place and are empty otherwise. The
constraint is the consequent of a conjunct of clauses containing the action relations and
variables on both the old and new states.

The technique proposed in [SK87], [KSS87] employs an extension of SLDNF-resolution
[Llo87] for reasoning forward from updates in the updated database. The method is ap-
licable to deductive databases consisting of a set of range-restricted deductive rules and
integrity constraints. The assumption that the database satisfies its constraints prior to the
update is exploited by considering updates as top-clauses, since any violation of the con-
strains in the updated database must involve at least one of the updates in the transaction.
The updates that are considered are additions and deletions of facts and deductive rules,
as well as additions and deletion of integrity constraints. SLDNF-resolution is extended so
that any rule, constraint or negated fact can be used as top clause. Transactions are han-
dled by using every update in the transaction as top clause. The method has been shown to be sound and complete for definite\(^4\) databases.

Decker's simplification method [Dec86] involves the representation of integrity constraints in range form and the generation of simplified forms of constraints at compile-time. Upon transactions, the relevant update constraints are selected and evaluated in the updated database. The method is applicable to range-restricted constraints and deductive rules. Range-restricted formulae possess the property that they have an equivalent representation in range form. Roughly, a range expression for a variable of a formula in disjunctive normal form is a disjunction of atoms in which the variable occurs. The range forms of constraints have the property that the embedded range expressions provide instantiations that allow for the efficient evaluation of the constraints.

A simplification theorem for integrity constraint checking in stratified databases is the main result of [LST86]. It extends the simplification method presented in [LT85a] which is only applicable to definite databases. Stratification is a sufficient condition for consistency of the completion of the database [IS87]. The method is applicable to allowed databases [LT85b]. Allowedness is a synonym of range-restrictedness, as the latter is defined in [Dec86]. The simplification method deals with transactions consisting of finite sequences of non-opposing additions and deletions of clauses, and involves the computation of sets of atoms that are, explicitly or implicitly, added to or removed from the completion of the database as a result of a transaction. The computation is carried out in a finite number of steps using appropriate termination conditions and employs only the deductive rules present in the database. Hence, implicit update generation can take place without accessing the database and by only knowing the type of updates.

The Compilation method [BDM88] is an enhancement of the efficiency of simplification methods. Efficiency stems from the separation of the constraint maintenance in two phases. The first one is a compilation phase in which potential updates induced by the anticipated updates are generated. Update constraints, which are simplified forms of the integrity constraint sufficient to check in the updated database, are generated from constraints and potential updates. The compilation phase does not access the facts which may reside in secondary storage, hence it may be performed independently of the database state. The second phase is purely an evaluation phase in the updated state. The update is not

\(^4\)A deductive database is definite if all its clauses' bodies consist of positive literals [LT85a].
actually performed, but the updated state is simulated via a meta-interpreter. This simulation avoids having to restore the original state in case of integrity violation. The method is applicable to transactions in range-restricted stratified databases. Constraints are closed formulae with restricted quantification (rq-formulae), i.e., of one of the forms:

\[
\exists x_1 \ldots x_n [A_1 \land \ldots \land A_m \land Q] \text{ or } \forall x_1 \ldots x_n [\neg A_1 \lor \ldots \lor \neg A_m \lor Q],
\]

where each \( x_i \) occurs in at least one \( A_j \), and \( Q \) is a logical constant or a formula in which the \( x_i \)'s occur free, if they occur. Rq-formulae are less restrictive than fully-typed formulae [Rei81], [LT85a], and their expressive power is at least that of relational calculus. Rq-formulae are a subset of domain-independent formulae. The drawbacks of the methods of [SK87] and [Dec86] in the generation of induced updates are avoided here. These methods interleave the generation of induced updates with constraint evaluation. Moreover, evaluation of each constraint independently of that of any other, does not permit the use of “global” evaluation techniques, such as avoiding re-evaluating the same subformulae. The method is also applicable to updates of deductive rules and constraints. Dynamic constraints are restricted to refer to two consecutive states only.

In [BMM90] the compilation method is extended to treating uniformly set, conditional and sequential updates. Set updates consist of sets of atomic updates. Conditional updates are specified by a query on the knowledge base. Finally, sequential updates are specified by rules which involve a meta-predicate after which represents the state resulting from an update. Dynamic constraints, restricted to refer to two consecutive states, can thus be expressed. The main advantage of the compilation method over the rest of the approaches is the uniform treatment of more general constraints and its independence from a particular reasoning system or prover.

A different approach is followed in [Kuc91], where constraints are translated into rules, and the problem of integrity checking is reduced to that of semantic change computation for a given update. The main contribution of this approach is the optimized computation of implicit changes. The explicit updates are used in multiple steps of the computation and goals are simplified as often as possible. The method is also applicable to updates of rules and constraints but not to transition constraints.
Object-Oriented Databases

Object-orientation has to offer additional advantages to constraint maintenance: finer granularity in update specification is provided, which, in turn, results in a reduction of search space. The aggregation abstraction of object-oriented databases permits the association of constraint checking with individual attributes. This provides more precision in integrity maintenance and overcomes the limitation of relational databases of associating constraints to tuples and thus having to perform redundant evaluations in case of tuple modifications that do not affect the validity of the constraints. Moreover, code generated for integrity maintenance can itself be part of the knowledge base and maintained incrementally.

In [UD90] constraint analysis is presented as a tool for supporting the design of object-oriented databases. Being an analysis rather than an enforcement process, constraint analysis transforms first-order constraints into Horn clauses and compiles the clauses into constraint graphs. The clauses representing a constraint constitute the nodes of the graph. There exists an edge from clause $c_1$ to clause $c_2$ if the set of predicates in the body of $c_1$ is a subset of those in the body of $c_2$. Hence, a constraint graph reflects the logical ordering of constraints: the satisfaction of leaf nodes’ conditions is a necessary property of the satisfaction of their parent nodes’ conditions. The analysis is based on identifying the constraints affected by an update using the same criteria as in [Nic82]. Then, starting from leaf nodes, all clauses in a path to a most general clause (one with no parent nodes) are presented along with an explanation of the sufficient conditions for their satisfaction to the database designer, who can, then, choose the appropriate actions for enforcing the constraints. Update propagation is made explicit and the consequences are presented. Although the method provides a detailed analysis of update consequences and helps identifying the appropriate actions for enforcement, it does not consider deductive rules. Moreover, it is not applicable to multiple constraints.

Knowledge Bases

The compilation method for deductive databases [BDM88] has been extended to an object-oriented framework in [JK90], [JJ91], and has been implemented in the structurally object-oriented knowledge base management system ConceptBase [EJJ+89]. The method, based on the same principles as the original proposal of [BDM88], advances the latter in the
following respect: triggers can be specified for and attached to the smallest type of objects (attributes); triggered procedures are represented directly in the knowledge base; the generation of triggered procedures is done automatically from declarative specifications of integrity constraints and deductive rules. The compilation of constraints and rules into attached procedures takes place at constraint specification time independently of the database state and the generated procedures are triggered for checking constraint satisfaction at evaluation time. A main difference is that deductive rules are compiled into simplified forms. The compilation phase generates parameterized simplified forms of constraints and rules. These forms are instantiated appropriately when transactions take place. The insertion or deletion of rules and constraints is treated as a normal update. In this dissertation, we advance the method of [JJ91] by the treatment of temporal constraints and the optimization of the computation of implicit updates.

Here we should also note that work on the knowledge base update problem [FKUV86], [Dal88], [LLo89], [KM90], [Wut93] is relevant to the integrity maintenance problem, since, in the former, a knowledge base has to be changed to reflect the updates and maintain the constraints at the same time. However, most approaches to knowledge base updates do not distinguish between the different types of formulas that make up the knowledge base (facts, rules, constraints) and they do not exploit the knowledge that the knowledge base is known to be consistent prior to an update. Primacy is given to the update rather than the existing knowledge in the knowledge base. Inconsistency repairing can also be formulated as an update problem: a violated constraint may be considered as an update; thus, the knowledge base has to be changed to remove the inconsistency.

The generality of the approaches to the knowledge base update technique does not make them suitable for efficient constraint checking. However, the techniques can be employed for performing certain tasks related to constraint maintenance. For instance, we can employ the techniques proposed in [Wut93] for integrity maintenance in the case of updates of integrity constraints and rules. Our currently adopted strategy of accepting the constraint only if it is not violated in the current state of the knowledge base, may be refined by following the update approach and change the knowledge base in order to reflect the addition of the constraint.
Temporal Databases

A method for checking the consistency of a database specification consisting of static and temporal constraints and operation descriptions is proposed in [Kun84], [Kun85b]. Each operation description consists in turn of a precondition, a postcondition and a temporal assertion on the history of the database for applying the operation. The proposed method checks consistency of static constraints by either resolution or a tableaux method [Kun85a]. Operation descriptions are analyzed to check whether operations can ever be executed, i.e., if there exists a legal state in which the operation precondition is true and the application of the operation yields a legal state. The result of the analysis is a state transition diagram. Each state of this diagram is interpreted as an abstract state in which all static constraints are satisfied. Transitions in the diagram represent transitions from one legal state to another under the execution of an operation.

The consistency of the temporal constraints is tested as follows: each transition diagram is transformed into a family of pushdown automata by taking into account the temporal assertions of the operations. One pushdown automaton corresponds to each state of the transition diagram. Pushdown automata are used to maintain the knowledge about the history of the database. When a transition is made, the history is augmented with the precondition of the operation that caused the transition. To decide whether the set of constraints is satisfiable, test sequences are generated and their acceptance is tested by the automata. A sequence of operations is allowed to be executed if it is accepted by one of the automata. Consistency of the temporal constraints is proved if each accepted test sequence satisfies all temporal constraints. The main contribution of this work is in proving consistency of temporal constraints with transaction specifications, but the problem of integrity checking in the presence of arbitrary updates is not studied.

Lipeck, Saake and their students have proposed the use of transition graphs [LS87], [SL88a], [LF88], [HS90], [HS91], [LZ91], [SS92], [SHS94], [GL95] for the monitoring of temporal integrity constraints. Constraints are specified in Propositional Temporal Logic. No explicit time attribute is associated with database states in a database history. Transition graphs translate dynamic constraints on sequences of database states into conditions on single state transitions. Transition graphs form a basis for run-time maintenance of dynamic constraints during database evolution. They decompose the task of dynamic in-
tegrity checking over sequences of states into a sequence of multiple static checks applied to individual state transitions. Each node in a transition graph represents an object at a particular state. Edges define possible sequences on the history of the database by stating which static constraints have to be satisfied. This permits the use of techniques used for checking static constraints. At update-time, a universal monitor interprets the transition graphs. In order for the update to be acceptable, it has to lead to a state that makes the current state sequence admissible. Hence, it has to follow a corresponding path through the graph, checking at each step whether the labels of the edges leading to new states are satisfied. Optimization methods have been proposed for transition graphs. For example, paths for which it is known that they will lead to integrity violation in a future state can be deleted. Moreover, historical minimization techniques for reducing the number of past states over which constraints have to be verified appear in [HS90] and [HS91].

The underlying database is also assumed to be atemporal in the early proposals using transition graphs. The method is extended to deal with temporal constraints in temporal databases in [GL95]. It does not take into account implicit updates and considers valid time only. It is applicable to constraints that are expressible using transition graphs. The proposed method is expensive for large databases containing large numbers of objects: a transition graph must be maintained for each class instance and every relevant constraint. The absence of examples in the paper does not help convince the reader about the efficiency of the method. There is no analysis of the performance of the algorithm. The method exploits a number of properties of transition graphs including iteration invariance. This permits focusing attention only on the parts of the graph that may have changed as a result of an update. The method also handles retroactive and proactive updates although the semantics of those is not discussed.

The history-less checking of constraints specified in the past fragment of temporal logic is proposed by Chomicki in [Cho92a] and [Cho95]. The proposed method is termed “history-less” since the past states of the database are not used for determining the satisfaction of integrity constraints. Instead, each database state is augmented by auxiliary relations that encode the historical information that is necessary for the verification of integrity constraints in each state. These relations are automatically derived from constraint definitions. In this manner, checking constraints after the occurrence of updates can be done using only the current extended database state. The auxiliary relations are updated by evaluating
relational calculus queries that are derived from the constraints. Real-time constraints are
dealt-with in an extension of past temporal logic to past metric temporal logic [Cho92b].

The encoding of the history of the database has the drawback that knowledge may be lost.
Hence, there exists a trade-off between the amount of historical information that can be
stored and be available at all times and the ability to detect all violations of constraints
that may refer to an arbitrary number of past database states. The method is tied to the
relational model and it is not apparent how it can be extended to a temporal deductive
setting. Moreover, the constraint set is assumed to be fixed throughout the lifetime of the
database. However, the ideas behind historical minimization can be adapted in techniques
such as the integrity monitoring technique we propose in this dissertation.

2.2.2 Integrity Maintenance by Transactions

In this section we use a similar classification as in section 2.2.1, and comment on some
of the most influential approaches into integrity maintenance by transactions. As it will
be noticed, there exists a much larger body of work on integrity monitoring rather than
integrity maintenance by transactions.

Relational Databases

The first proposal along the lines of transaction modification was put forth by Stonebraker
in [Sto75]. The method proposed is applicable to constraints expressible in QUEL, i.e., uni-
versally quantified assertions. Each update issued is transformed into a new one into whose
qualification, the qualifications of all relevant constraints are embedded. In this manner, the
new update guarantees that constraints won’t be violated. Including the conditions of all
constraints into the update is in effect equivalent to checking all tuples the original update
specifies one by one. Although consistency preservation is guaranteed, no simplification of
constraints takes place. The constraints have to be validated as stated. In addition, the
fact that constraints are satisfied in the state prior to the update is not exploited and no
knowledge about the application domain or the types of transactions is used. The modi-
fication must take place during query processing. This method was integrated to a more
general strategy for constraint checking for the database system Ingres [CD83]. Simple
constraints, e.g., single-value dependencies, are enforced by query modification, whereas
more general constraints are checked after updates have committed. In case of a constraint
violation, user-defined procedures can be triggered or the database has to be rolled-back to its previous state.

A set-oriented language for transaction specification is used in [LTW93]. For each update and each constraint, a weakest precondition ($wp$) is derived so that, if $wp$ is true in the state prior to the update, then the constraint is guaranteed to be true in the state resulting from the update. Although the derived weakest preconditions are frequently amenable to simplification, there is no systematic treatment of precondition optimization. The method is applicable to a limited class of static constraints only and there is no mention of derivations of weakest preconditions when multiple constraints are relevant to an update. Finally, although the authors claim that it is trivial to incorporate implicit updates, the machinery provided does not account for them.

In [SS89], a general-purpose theorem prover employing heuristic rewrite rules is used for proving safety of transactions with respect to a set of static constraints. Implicit updates and dynamic constraints are not considered. Although safety of transactions is proved at compile-time, the theorem prover could take advantage of the knowledge of the updates taking part in the transaction in order to simplify the proof procedure and possibly suggest changes to the transaction specification. Although not discussed, the frame problem is implicitly dealt with by restricting attention to the predicates changed by transactions and by eliminating inertial terms from the theorems that have to be proven in order to verify integrity. The concept of constraint protectors discussed in [SMS87] resembles that of constraint ramifications [LR92], [PM96], and on which we expand on in chapter 5. The generation however of constraint protectors assumes the existence of a fairly general theory of lemmas that is independent of the transactions.

Transaction Logic [BK95] alleviates the frame and ramification problems by the use of both procedural and declarative knowledge in the specification of direct and indirect effects of actions respectively. However, in Transaction Logic, the specification of transaction expressions already includes the constraints that need to be enforced. In our approach, purely declarative transaction specifications are modified so that, every implementation meeting the new specifications provably guarantees the maintenance of integrity constraints. Constraints are considered external to the specification of transactions. Furthermore, the proof of constraint satisfaction in Transaction Logic is tied to the execution of transactions. Hence, constraint checking is purely a run-time notion.
Deductive and Active Databases

Wallace [Wal91] advocates the maintenance of semantic integrity by compiling constraints into update procedures. The idea is to apply specialization to the update procedures with respect to the defined constraints. This involves compiling integrity checking into update procedures so that further checking is avoided for the constraints for which compilation guarantees satisfaction (by design of the update procedure). A procedural language with a denotational semantics is proposed for specifying set, conditional or sequential deterministic, non-recursively defined updates. The knowledge base is augmented with semantic rules expressing the axiomatization of updates, rules and constraints. The semantic rules are partially evaluated and logically optimized. Partial evaluation involves expressing conditions on multiple knowledge base states (sequential updates) as conditions on the current state and optimization involves eliminating redundancies or inconsistencies in conditions. The simplified conditions express necessary and sufficient conditions for guaranteeing that updates using these procedures preserve integrity. They are translated back to a query that is used as a precondition of the update procedure, referring only to the state prior to the update.

In [CW90], integrity constraints are expressed as allowed formulae [VGT91]. They are semi-automatically translated into set-oriented production rules activated at the commit point of transactions. The focus of this work is on automatic integrity repair rather than optimization of constraint checking. Extensions of the method to fully automate the translation process and to guarantee termination and confluence of rule execution appear in [CFPT94]. Additional optimizations that simplify the condition part of the production rules should complement the enforcement strategy.

Procedural attachment has been used for the enforcement of specific forms of constraints in active database systems such as POSTGRES [SHP88] and HIPAC [Cha89]. In these systems hand-coded or automatically generated procedures can be attached to objects for the purpose of enforcing integrity.

Object-Oriented Databases

In [FCT88] the idea of exploiting the encapsulation principle of object-oriented database design is put forth for consistency maintenance. By collecting structures, constraints and
operations that are closely related into separate modules, the logical consistency of the database can be guaranteed if the only way to update the database is via the integrity preserving operations encapsulated along with objects and constraints.

A formal system for generating demons from deductive rules and constraints in object-oriented databases is presented in [Cas89]. Rules and constraints specified in a subset of predicate calculus are transformed into a relational algebra. A compiler generates low-level procedures (demons) that may be used to perform logic resolution. The system produces efficient code that can be used for update propagation and constraint enforcement. However, the only operations supported are insertion in a relation and retrieval from a relation. An advantage is that the produced demons have a denotational semantics since they are equivalent to their relational algebra expressions.

A transaction modification technique for object-oriented databases applicable to both static and dynamic constraints is presented in [STSW92], [STW93]. Transactions are rewritten into their greatest consistent specializations (GCSs), i.e., transactions augmented to be compliant to a single relevant constraint. The GCS of an update operation represents a conditional update that guarantees consistency. The drawback of the approach is that GCSs must be built manually. The generation of GCSs does not scale up trivially: it is not possible to build GCSs of arbitrary transactions by assembling GCSs of elementary operations. Building GCSs for multiple constraints is also a problem since the GCS of an operation for a single constraint is usually a non-elementary operation.

Knowledge Bases

The problem of integrity maintenance by transaction modification has received even less attention in knowledge bases. In Taxis [CRZNM88] assertions are specified as invariants of data classes or processes. Integrity constraints are interpreted as preconditions on transactions. Expressiveness restrictions are imposed on the assertion language for the purpose of achieving efficient enforcement of integrity assertions. Specifically, no explicit quantification is allowed and the attributes participating in selection paths are restricted not to belong to metaclasses whose extensions are expected to be large. An assertion compiler generates a set of run-time procedures for each attribute and operation. Depending on the type of assertion, the compiler generates static conditions which, if satisfied at run-time, guarantee the satisfaction of the assertions, and, in general, efficient code that optimizes
assertion checking. A triggering mechanism is used for “attaching” code to the concerned objects. Run-time optimization involves ordering of assertions according to the cardinality of the involved classes. Dynamic constraints involve references to values of attributes in the states before and after the updates or relative to time. Such constraints are compiled into procedures involving tests to be performed before and after the update. A list containing all temporal assertions and indicating when each assertion is to be triggered is maintained.

In a similar vein, the knowledge base management system KRISYS [Mat91], [Des90], maintains semantic integrity by means of demons or rules. Attention is paid to ensuring that the model-inherent constraints relating to the structuring principles of generalization, classification and aggregation are preserved, as well as domain and cardinality constraints. Ad-hoc constraints can be embedded in the high-level language provided for the definition of demons. Demons are automatically executed when events affecting the objects to which they are attached take place. No optimization takes place and the correct definition or behaviour of demons remains the responsibility of the designer. The problem of ensuring schema consistency is only dealt with in the knowledge base management system described in [LNW91]. There is no provision for the definition of general external constraints.

Temporal Databases

An early proposal on the use of triggers for the maintenance of temporal integrity constraints appears in [ELG84]. Constraints are specifies in a temporal logic using only quantifiers □ and ◇ and in the form φ ⇒ □ψ or φ ⇒ ◇ψ, with φ and ψ being non-temporal formulae. Then they are transformed into program statements of the form on φ do op, where op is a program operation. Other methods were developed subsequently based on the idea of producing triggers from temporal constraints. In [GL93], triggers are derived from the transition graphs that are built to represent the acceptable lifecycles of objects with respect to the integrity constraints. Constraint analysis is used in order to derive the integrity maintaining triggers for arbitrary transactions. The proposed method exploits the knowledge that certain preconditions of dynamic constraints are satisfied in the state prior to a transaction’s execution. This knowledge can be used to produce simplified versions of the conditions that remain to be verified. It is shown how triggers can be systematically generated using the transition graphs. Constraints are associated with existence intervals of objects, thus restricting the states that need to be examined in order to verify the constraints.
to those in which the objects exist. The drawback of this approach is that a large number of transition graphs has to be maintained and that the generated triggers are unoptimized. Implicit updates are not taken into account.

The maintenance of temporal constraints through transaction modification is examined in [Lip90]. Dynamic constraints specified in propositional temporal logic are transformed into transition graphs [LS87], [LF88], [SS92]. Paths in these graphs correspond to partially admissible state sequences. A state sequence is called partially admissible if it is correct with respect to the constraints up to the current state. Transition graphs are then transformed into refinements of transaction specifications, so that executable state sequences become admissible. Transactions are specified in terms of pre/post-condition pairs. Additionally, transaction specifications include an applicability condition and frame conditions. All conditions are expressed as formulae of predicate logic. The transaction specifier has to supply all frame conditions explicitly. A semantic frame rule stating that states are minimally changed to their subsequent states by transactions is also imposed. Transaction specifications are modified with respect to transition graphs. These graphs are built from dynamic constraints and describe life-cycles of database objects with respect to the constraints. The main transformation step consists of incorporating into transaction specifications conditions that represent which part of the constraint is satisfied in the history up to the current state and which part remains to be checked. However, this transformation introduces conditions that refer to the transition graphs instead of the predicates occurring into the constraints. It becomes unclear what the specification of the transaction means with respect to the database state. Simplifications to the refined specifications are only performed with respect to the subformulae of the postconditions that are implied by the precondition and the fact that, to reach the particular state of the graph, one incoming edge would have to be true. Moreover, implicit updates are not treated.

In [SW95a], [SW95b] temporal triggers specified in Future Temporal Logic (FTL) are used for the enforcement of integrity constraints. An incremental method for evaluating trigger conditions (FTL formulae) in parallel is proposed. The logic allows for the specification of real-time properties. Formulae are interpreted over database histories which are modeled as sequences of time-stamped states. Each successive state is obtained from the previous one by an update. The time-stamp of each state denotes the time the update took place. Checking the satisfaction of trigger conditions involves the evaluation of every trig-

35
ger condition over the history starting with the state in which the trigger comes into effect and ends in the current state. The database queries appearing in the trigger condition are presented as materialized views to the system. Requirement sets are maintained for each trigger: these contain conditions which, if satisfied by the future database history, will cause the trigger to be satisfied. At least one requirement set has to be satisfied. The algorithm incrementally replaces conditions in requirement sets by sub-conditions so that, part of the condition is verifiable in the current state and the rest is verifiable in the future history. All conditions in a requirement set have to be satisfied in order for the requirement set to satisfy the trigger condition. Optimization steps include the removal of requirements that are unsatisfiable. The correctness of the replacement step is shown and the implementation in a parallel architecture is discussed. The method works with future temporal operators but it's extension to apply to past operators is non-obvious.

Last, but not least, the implementation of integrity constraints via active rules is described in [CT95], [CT94]. This work proposes a general architecture that may be adapted to different specification and rule languages. In the case study presented, constraints specified in past temporal logic are compiled into a set of Starburst [LLPS91] rules. The compiler uses a translation into a set of SQL statements that include all the necessary rules needed for the enforcement of constraints. Additional optimization steps on the rule conditions are performed, including historical minimization [Cho92a], and magic-set transformations [BMSU86].
Chapter 3

A Temporal Assertion Language

In this chapter we propose a non-reified\footnote{A logic is called reified when its formulae are added as individuals in the language [AI84], [BTK89]. Later on, a reification of the language will take place in an attempt to provide more easily readable formulas.} temporal logic based on time intervals, as the language for expressing integrity constraints and deductive rules. The proposed language is an extension of the assertion language of Telos [MBJK90], a many-sorted first-order language with equality. A formal account of the semantics of the language based on a possible-worlds model can be found in [Ple93b]. Section 3.1 contains a brief overview of the structural component of Telos. Sections 3.2 and 3.3 present the temporal and assertional components of the language respectively. We characterize the expressive power of the language and present a working example to be used throughout the dissertation for demonstrating the applicability of our methods. In the remainder of the chapter we define the notion of constraint satisfaction in the bitemporal context of Telos and provide a syntactic classification of the different types of constraints expressible in our formalism.

3.1 Overview of Telos

The representational framework of Telos [MBJK90] constitutes a generalization of graph-theoretic data structures used in semantic networks, semantic data models and structurally object-oriented representations. Among the distinguishing aspects of Telos are the novel treatment of attributes as first-class citizens, the extensibility provided by the metaclassing mechanism and the special representational and inferential capabilities for temporal knowledge. Telos has a well defined semantics based on a possible-worlds model [Ple93b]. Telos
knowledge bases are collections of *propositions*. A *proposition*, the single representational unit provided, is formally defined as a quadruple with components *from*, *label*, *to* and *when*. These denote the *source*, *label*, *destination* and *duration* of the proposition respectively, and are propositions themselves. Referential integrity requires that any component of a proposition must exist in the knowledge base. *Telos* propositions are divided into two disjoint categories: *individuals*, representing entities\(^2\) (concrete ones such as *John* or abstract ones such as *Person*), and *attributes*, representing relationships between entities or relationships. Note that an attribute may also represent an abstract relationship. For instance the attribute *address* represents an abstract relationship between persons and geographical locations. Individuals and attributes are treated uniformly and are the building blocks of the structured objects that comprise a *Telos* knowledge base.

Propositions are organized along the structuring dimensions of *instantiation*, *generalization* and *aggregation* [BMW84]. Each proposition is an instance of one or more generic propositions called *classes*—thus giving rise to an instantiation hierarchy. Propositions are classified into *tokens*—propositions having no instances and intended to represent concrete entities in the domain of discourse, *simple classes*—propositions having only tokens as instances, *meta classes*—having only simple classes as instances, *metametaclasses*, and so on. *Telos* offers a number of built-in classes at all levels of the infinite instantiation hierarchy. Classes having instances into more than one levels of the hierarchy are termed ω-*classes*. For instance, the ω-*class* *Proposition* has all propositions as instances, whereas *Class* has all classes as instances.

Orthogonal to the classification dimension, classes can be organized in terms of *gen-

\(^2\)A more refined ontology of objects representable as *Telos* propositions can be found in [Plc90].
eralization or isA hierarchies. A class may have incomparable generalizations leading to hierarchies that are directed acyclic graphs rather than trees. The attribute mechanism is also used for attaching assertions (deductive rules and integrity constraints) to Telos objects. Inheritance of attributes and assertions with respect to generalization is assumed to be strict, in the sense that a class definition cannot override inherited attributes and assertions. Instantiation and specialization relationships are grouped by the built-in classes InstanceOf and IsA.

Figure 3.1 depicts an example Telos knowledge base in the form of a labeled directed graph. The knowledge base is intended to model an imaginary scientific conference domain. Dashed lines represent a specialization relationship (is-a) between generic entities (classes), shown in bold font, whereas solid lines represent binary relationships among entities (attributes). Integrity constraints and deductive rules are attached to classes through the attribution mechanism.

3.2 Temporal Knowledge

Telos emphasizes the use of time for representing histories of generic events and activities as well as the system’s knowledge of these histories, by providing two temporal dimensions for historical and belief time³. Historical time captures the evolution of the domain of discourse, whereas belief time records the history of the knowledge base itself, i.e., the time that facts about the domain of discourse become known. Telos adopts Allen’s interval-based time model [All83] for representing historical information about an application domain.

In our model, time is linear, discrete and unbounded. It is assumed to be isomorphic to the set Z of integer numbers. The primitive notion of the time model is that of an interval. An interval with known endpoints is represented as an ordered pair \( (a, b) \) of time points. An interval whose endpoints coincide is called a unit interval. Seven temporal relationships are used, along with their inverses, to characterize the relative position of two time intervals. These are the relationships meets, during, overlaps, before, starts, finishes and equals and their inverses met-by, contains, overlapped-by, after, started-by, finished-by. A pictorial representation of the thirteen relations is

³The terms historical and belief time correspond to the more popular in the area of temporal databases terms of valid and transaction time respectively [JCG+92].
given in figure 3.2. The temporal relations are interpreted as being mutually exclusive. A temporal element can then be defined as a finite set of time intervals. If \( i_1, i_2, \ldots, i_k \) are time intervals, then \( \{i_1, \ldots, i_k\} \) is the temporal element comprising intervals \( i_1, i_2, \ldots, i_k \). Note that intervals are homogeneous whereas elements may not be. An element comprising a single interval is identified with the interval itself.

The model also includes temporal constants (dates and times), semi-infinite time intervals, the special interval Alltime, the special interval variable Now denoting the current clock time, and the result of applying temporal operations on intervals.

Operations on intervals include intersection (*), difference (-), composition (+) and transposition (×). Intuitively, the intersection of intervals returns their common subinterval. The composition of two intervals returns the shortest interval or element that contains the intervals. The difference of two intervals resembles the set-theoretic notion of difference: it returns the subinterval(s) of the first of the operands, that is (are) not contained in the second. Finally, transposition returns an interval transposed along the time line. The
transposition operator is needed for the expression of periodic properties. Formal definitions of the operations on intervals are given below. For an interval \( i \), \( i^- \) and \( i^+ \) denote its left and right endpoint respectively. The symbol \( \epsilon \) denotes the empty interval.

**Definition 3.2.1 (Intersection)**

The intersection of intervals \( i_1 = (i_1^-, i_1^+) \) and \( i_2 = (i_2^-, i_2^+) \) is defined as follows:

\[
i_1 \cap i_2 = \begin{cases} 
    i_1, & \text{if } i_1 \left( \begin{array}{c}
        \text{during} \\
        \text{starts} \\
        \text{finishes} \\
        \text{equal}
    \end{array} \right) i_2 \\
    (i_2^-, i_1^+), & \text{if } i_1 \text{ overlaps } i_2 \\
    \epsilon, & \text{if } i_1 \left( \begin{array}{c}
        \text{before} \\
        \text{meets} \\
        \text{after} \\
        \text{met by}
    \end{array} \right) i_2 \\
    (i_1^-, i_2^+), & \text{if } i_2 \text{ overlaps } i_1 \\
    i_2, & \text{if } i_2 \left( \begin{array}{c}
        \text{during} \\
        \text{starts} \\
        \text{finishes} \\
        \text{equal}
    \end{array} \right) i_1
\end{cases}
\]

**Definition 3.2.2 (Transposition)**

The result of \( k \times i \) for some integer \( k \), is the interval \( (k \times i^-, k \times i^+) \).

**Definition 3.2.3 (Composition)**

The composition of intervals \( i_1 = (i_1^-, i_1^+) \) and \( i_1 = (i_2^-, i_2^+) \) is defined as follows:
\[
\begin{cases}
(i_1-, i_2+), \text{ if } i_1 \begin{cases} 
\text{meets} \\
\text{starts} \\
\text{overlaps}
\end{cases} i_2 \\
[i_1, i_2], \text{ if } i_1 \text{ before } i_2 \\
(i_2-, i_1+), \text{ if } i_1 \begin{cases} 
\text{overlapped - by} \\
\text{started - by} \\
\text{met - by}
\end{cases} i_2 \\
i_1 + i_2 = \begin{cases} 
[i_2, i_1], \text{ if } i_1 \text{ after } i_2 \\
i_1, \text{ if } i_1 \begin{cases} 
\text{equal} \\
\text{contains} \\
\text{finished - by}
\end{cases} i_2 \\
i_2, \text{ if } i_1 \begin{cases} 
\text{during} \\
\text{finishes}
\end{cases} i_2
\end{cases}
\]

Definition 3.2.4 (Difference)

The difference of intervals \(i_1 = (i_1-, i_1+)\) and \(i_1 = (i_2-, i_2+)\) is defined as follows:

\[
i_1 - i_2 = \begin{cases} 
(i_2-, i_1+), \text{ if } i_1 \text{ overlaps } i_2 \\
(i_2+, i_1-), \text{ if } i_1 \text{ overlapped - by } i_2 \\
(i_2+, i_1+), \text{ if } i_1 \text{ started - by } i_2 \\
(i_1-, i_2-), \text{ if } i_1 \text{ finished - by } i_2 \\
[(i_1-, i_2-), (i_2+, i_1+)] \text{ if } i_1 \text{ contains } i_2 \\
i_1, \text{ if } i_1 \begin{cases} 
\text{meets} \\
\text{met - by}
\end{cases} i_2
\end{cases}
\]

The corresponding operations on elements are defined using the set-theoretic notions of intersection, union and difference. The transposition of an element is defined as the element comprising the intervals resulting from applying the transposition operation to each interval of the element.
The following proposition expresses properties of the composition and intersection operations for intervals and elements.

**Proposition 3.2.1** *(Commutativity and Associativity)*

The composition and intersection operations on intervals and elements are commutative and associative.

**Proof:** Follows easily from the definition of the operations on intervals. It can be easily seen that interval and element difference are neither commutative nor associative.

### 3.3 The Assertional Component

This section presents an assertion language that will be employed for the expression of deductive rules and integrity constraints. A detailed exposition of the language can be found in [MBJK90], whereas a formal account of the semantics of the language can be found in [Ple90] and [Ple93b].

Telos provides an assertion language (AL) for the expression of deductive rules and integrity constraints. AL is a many-sorted first-order language with equality, whose terms are variables, constants, the result of applying the functions `from()`, `label()`, `to()` and `when()` to terms, as well as explicitly enumerated sets produced by set-valued functions. The atomic formulae of the language include the predicates: `prop(p,x,y,z,t)`, meaning: `p` is a proposition with components `x,y,z` and `t`, `instanceOf(x,y,t1,t2)`: `x` is an instance of `y` for the time period `t1` and is believed by the system for the time period `t2`, `isA(x,y,t1,t2)`: `x` is a specialization of `y` for the time `t1` and is believed by the system for time `t2`, `att(x,y,t1,t2)`: `y` is a value of the attribute `att` of `x` for `t1` and is believed for `t2`.

For any terms `x` and `y` in AL and every temporal or evaluable predicate `θ`, `x θ y` is an atomic formula with the obvious meaning. The well-formed formulae (wffs) of AL are built recursively from atomic formulae in the usual manner. Assertional knowledge is also organized along the structuring dimensions of Telos. Specifically, assertions - integrity constraints and deductive rules - can be attached as attributes to classes through the aggregation mechanism of Telos.

---

The belief time component of these predicates is omitted when they appear in the head of deductive rules.
Integrity constraints and deductive rules are expressed as rectified\textsuperscript{5} closed wffs of AL. An integrity constraint can be in one of the two forms

\[ I \equiv \forall x_1/C_1 \ldots \forall x_k/C_k \; F \quad \text{or} \quad I \equiv \exists x_1/C_1 \ldots \exists x_k/C_k \; F \]

where, \( F \) is any wff of AL whose quantified subformulae are of the above forms and in which the variables \( x_1, \ldots, x_k \) occur free, if at all. Each \( C_i \) is a Telos class. The meaning of each restricted quantification is that the variable bound by the quantifier ranges over the extension of the class instead of the entire domain. Any constraint in this form is range-restricted [Dec86], [VGT91]. This class of constraints is equivalent to both the restricted quantification form of [BDM88] and the range form of [JJ91]. The latter is obtained if a “typed” constraint of one of the above forms is transformed to its untyped form, whereas the restricted quantification form is obtained from the range form by imposing the restriction that \( F \) is in miniscope negation normal form. According to the standard transformation of typed quantified formulae to untyped ones [Llo87] the following equivalences hold:

\[ \forall x/T \; F \equiv \forall x \; T(x) \Rightarrow F \quad \text{and} \quad \exists x/T \; F \equiv \exists x \; T(x) \land F \]

The typed quantifications \( \forall x/C \; F \) and \( \exists x/C \; F \) are short forms for the formulae:

\[ \forall x \forall t \; \text{instanceOf}(t, \text{TimeInterval, Alltime}) \land \text{instanceOf}(x, C, t) \Rightarrow F \quad \text{and} \]

\[ \exists x \exists t \; \text{instanceOf}(t, \text{TimeInterval, Alltime}) \land \text{instanceOf}(x, C, t) \land F \]

The transformation preserves the range-restriction property despite the introduction of new universally quantified temporal variables, since these are restricted by the \( \text{instanceOf} \) literals. The introduction of temporal variables and their restricting literals is necessary since all atomic formulae of AL have a temporal component. For simplicity and since explicit temporal quantification appears in assertions, the introduced variable(s) may be identified with ones already appearing in the formula.

Syntactically, deductive rules are considered to be special cases of integrity constraints. Their general form is

\[ DR \equiv \forall x_1/C_1 \ldots \forall x_n/C_n \; (F \Rightarrow A) \]

\textsuperscript{5}A formula is rectified if no two quantifiers introduce the same variable [BDM88].
where $F$ and the variables $x_i$ are subject to the same restrictions as in the case of constraints. Atom $A$ may not contain belief time, since belief time is set by the system, and variables other than $x_1, \ldots, x_n$. Deductive rules in this form are also range-restricted. Moreover, deductive rules are assumed to be \textit{stratified} [Llo87].

Integrity constraints and deductive rules are objects of their own right and are, as such, associated with history and belief time intervals. The former denote the intervals over which the constraints are considered to be applicable to the domain of discourse, whereas the latter denote the intervals over which the constraints and rules are believed by the knowledge base to be applicable to the domain of discourse. If no such association appears explicitly with their definition, both intervals are assumed to be equal to the right-infinite interval (Now..*). These intervals are used by our optimization methods in order to restrict the set of updates that can possibly affect the constraints or rules to the set of updates associated with history and belief time intervals contained in or overlapping the respective intervals of the constraints or rules.

An advantageous consequence of the ability to explicitly refer to both the history of the domain of discourse and to the system’s knowledge about that history, is the ability of expressing a number of different types of constraints not expressible in formalisms with no or a single only notion of time. AL provides the ability to express constraints referring to the epistemic state of the knowledge base. To provide a better characterization of the types of constraints expressible in AL, we will refer to \textit{state}, \textit{dynamic} and \textit{dynamic epistemic} constraints. \textit{State} constraints express properties referring to individual states of the domain. \textit{Dynamic} constraints specify properties dependent on two or more domain states. Finally, \textit{dynamic epistemic} constraints refer to two or more epistemic states of the knowledge base in addition to multiple domain states. From the above definitions, it is expected that in the expression of dynamic constraints in AL, the history time variables will be constrained by explicit temporal relationships, whereas in the case of dynamic epistemic constraints, belief time variables will be constrained as well. For facilitating temporal reasoning, disjunction of temporal relationships is disallowed in AL.

Before we formally define the semantics of the above types of constraints, we present a few examples of constraints and rules that will be used throughout the dissertation.

\textbf{Example 3.3.1} A number of integrity constraints and deductive rules have been defined, for the sake of depicting the application of our constraint maintenance methods, and at-
tached to classes of figure 3.1. Constraints IC1 and IC2 are state constraints expressing the properties that “no author of a paper can be a referee for it” and “an author cannot submit a paper to a conference organized by the department she works in” respectively. Dynamic constraint IC3 enforces the property that “an employee’s salary can never decrease”. Deductive rules DR1 and DR2 express the rules that “A university affiliate works in the department that has the same address as she does” and “A university department’s address is the same as the university’s location”. IC4 is an example of a dynamic epistemic (meta-) constraint expressing the property that “the system cannot stop believing a class definition”.

They are expressed as follows:

IC1: \( \forall c/\text{ConfPaper} \ \forall r/\text{Referee} \ \forall a/\text{Author} \ \forall t_1,t_2/\text{TimeInterval} \)

\[ (\text{ref}(c,r,t_1,t_2) \land \text{author}(c,a,t_1,t_2) \Rightarrow (r \neq a)) \text{ (at 1988..*)} \]

IC2: \( \forall c/\text{Conference} \ \forall p/\text{ConfPaper} \ \forall a/\text{Author} \ \forall d/\text{Department} \ \forall t_1,t_2/\text{TimeInterval} \)

\[ (\text{submitted}(p,c,t_1,t_2) \land \text{organized}(c,d,t_1,t_2) \land \text{author}(p,a,t_1,t_2) \land \text{works}\_\text{in}(a,d,t_1,t_2) \Rightarrow \text{False}) \]

IC3: \( \forall p/\text{Employee} \ \forall s,s'/\text{Integer} \ \forall t_1,t_2,t_3/\text{TimeInterval} \)

\[ (\text{salaray}(p,s,t_1,t_2) \land \text{salaray}(p,s',t_3,t_2) \land \text{before}(t_1,t_3) \Rightarrow (s \leq s')) \]

IC4: \( \forall p,c,l/\text{Proposition} \ \forall t,t'/\text{TimeInterval} \)

\[ \text{prop}(p,c,l,t) \land \text{instanceOf}(p,\text{Class},t,t') \Rightarrow (\forall T,T'/\text{TimeInterval} \land \text{overlaps}(t,T) \land \text{overlaps}(t',T') \Rightarrow \text{instanceOf}(p,\text{Class},T,T')) \]

DR1: \( \forall u/\text{UnivAffiliate} \ \forall d/\text{Department} \ \forall s,s'/\text{String} \ \forall t_1,t_2/\text{TimeInterval} \)

\[ \text{address}(u,s,t_1,t_2) \land D\_\text{addr}(d,s',t_1,t_2) \land (s = s') \Rightarrow \text{works}\_\text{in}(u,d,t_1)) \]

DR2: \( \forall d/\text{Department} \ \forall u/\text{University} \ \forall s/\text{String} \ \forall t_1,t_2/\text{TimeInterval} \)

\[ \text{univ}(d,u,t_1,t_2) \land \text{location}(u,s,t_1,t_2) \Rightarrow D\_\text{addr}(d,s,t_1)) \]

\[ \square \]

The semantics of the above types of integrity constraints requires that a state constraint is satisfied in any state that is accessible from the current state and for the knowledge base epistemic state corresponding to the constraint’s belief time interval. A dynamic constraint must be satisfied in any sequence of states that satisfy the temporal predicates explicit in the constraint expression, for the belief time interval corresponding to the constraint’s belief time. Finally, a dynamic epistemic constraint must be true in all domain and epistemic states that satisfy the historical and belief time relationships in the constraint. These notions are formalized in a subsequent section.
3.3.1 Reifying the Assertion Language

In this section we present a reified version, RAL, of the assertion language. Reification facilitates the expression of assertions by producing expressions that are more concise than their equivalent non-reified formulae.

Atomic formulae of RAL include temporal and evaluable predicates, as well as history-time assertions and belief-time assertions. Well-formed formulae are then built inductively using logical connectives.

**Definition 3.3.1 (History-time Assertion)**

A history-time assertion is a formula of the form \( \text{Holds}(p, t) \), where \( p \) is a binary predicate among \( \text{prop}(), \text{instanceOf}(), \text{is}A \) and \( \text{att}() \) and \( t \) is a time interval. \( \text{Holds}(p, t) \) denotes the truth of \( p \) during time interval \( t \). Specifically, for a predicate \( p(x, y) \), \( \text{Holds}(p(x, y), t) \) means that there exists a time interval \( t_0 \) such that \( \text{prop}(\text{pid}, x, p, y, t_0) \land \text{contains}(t_0, t) \) is true, if \( p \) is a basic predicate, or that the truth of \( p(x, y) \) at time \( t \) is derivable form the knowledge base via the deductive rules, if \( p \) is a derived predicate.

**Definition 3.3.2 (Belief-time Assertion)**

A belief-time assertion is a formula of the form \( \text{Believed}(HT, t') \), where \( HT \) is a history-time assertion and \( t' \) is a time interval. \( \text{Believed}(HT, t') \) denotes that \( HT \) is believed by the knowledge base throughout the time interval \( t' \).

Having defined history-time and belief-time assertions we can now redefine the notion of restricted quantification. The restricted quantification forms \( \forall x/\mathcal{C} \ F \) and \( \exists x/\mathcal{C} \ F \) are short forms for the formulae:

\[
\forall x \ \forall t \ \text{Holds}(\text{instanceOf}(x, \mathcal{C}), t) \land \text{Holds}(\text{instanceOf}(t, \text{TimeInterval}), \text{Alltime}) \Rightarrow F
\]

and,

\[
\exists x \ \exists t \ \text{Holds}(\text{instanceOf}(x, \mathcal{C}), t) \land \text{Holds}(\text{instanceOf}(t, \text{TimeInterval}), \text{Alltime}) \land F
\]

The following example demonstrates the expression of assertions in RAL.

**Example 3.3.2** We restate some of the assertions of the previous example using the reified version of the language.

Constraints IC1 and IC3 and the deductive rule DR1 are now expressed as the formulae:
IC1: ∀ p/ConfPaper ∀ x/Author ∀ r/Referee ∀ t,t' / TimeInterval
\[ \text{Believed} (\text{Holds}(\text{author}(p, x), t), t') \land \text{Believed} (\text{Holds}(\text{referee}(p, r), t), t') \]
\[ \Rightarrow (r \neq x) \text{ (at 1988..x)} \]

IC3: ∀ p/Employee ∀ s, s' / Integer ∀ t_1, t_2, t / TimeInterval
\[ \text{Believed} (\text{Holds}(\text{salary}(p, s), t_1), t) \land \text{Believed} (\text{Holds}(\text{salary}(p, s'), t_2), t) \land \]
\[ \text{before}(t_1, t_2) \Rightarrow (s \leq s') \]

The above example shows that the expression of constraints in RAL is more succinct than the equivalent expression in AL. For the remainder of this chapter we will use RAL for the expression of assertions. We will switch back to AL in chapter 4, where the simplification method will be presented.

3.3.2 Syntactic Classification of Constraints

In this section, we elaborate on the syntactic classification of constraints. We claim that different types of temporal constraints lend themselves differently to the application of optimization and enforcement methods. Hence, the ability to distinguish syntactically temporal formulae representing constraints is beneficial for the application of simplification and enforcement methods, as well as for the automatic compilation of constraints into their simplified forms. For instance, a constraint of the form “sometime in the future ...” has to be enforced differently than, e.g., a constraint expressing a property that has to be enforced periodically.

Section 1.3 of the introduction, provided a classification of temporal formulae along five different dimensions. Here we provide some observations with respect to the syntactic form of temporal formulae expressing the various constraint types.

- **State** constraints express properties applicable to all states\(^6\) of the domain of discourse. State constraints are formulae universally quantified over history and belief time intervals. Moreover, since the constraint is applicable to all time intervals, there is no explicit temporal predicate restricting the history time variable(s).

- **Dynamic** constraints express properties referring to sequences of states of the domain of discourse. Hence, they are expressed as formulae universally quantified over history and belief time intervals. Since a dynamic constraint refers to two or more states of

\(^6\)The notion of a "state" of a knowledge base is discussed in section 3.4.
the domain, two or more temporal variables bound by a universal quantifier appear in their expression. The temporal ordering of the states of the domain is defined by explicit temporal predicates restricting the quantified temporal variables. There is no temporal predicate restricting belief time variables.

- **Periodic** constraints can be seen as a special case of dynamic constraints, where a property has to be verified at times that stand in a canonical relation. The expression of such properties requires the refinement of the domain of time intervals into subdomains, such as minutes, hours, days, weeks, etc. A periodic property is then expressed as a dynamic constraint, in which a history time variable is restricted to be the result of the transposition of a constant time interval or of another history time variable.

- **N-times aperiodic** constraints express properties that have to be verified at certain points in time only. This fact is denoted by the presence of existential quantification over history and belief time intervals possibly in the scope of a universal quantifier and, possibly, the presence of temporal predicates restricting the existentially quantified variables.

- **Relative** and **Real-time** constraints are distinguished by the presence of conventional dates, times and the special interval **Now**.

- **Object-level** and **Meta-level** constraints are distinguished by the type of domain (i.e., simple class, meta-class, etc.) the object variables range over.

- **Historical** constraints contain a single universally quantified belief time variable, whereas an **epistemic** contains at least two quantified belief time variables.

Similar observations can be made with respect to the rest of the classification dimensions. For instance, **dynamic epistemic** constraints, are characterized by the temporal quantification over both history and belief time intervals and by the presence of temporal predicates restricting history and belief time variables.

To the best of our knowledge, there has been no previous attempt to provide such a classification of temporal integrity constraints in the presence of a dual notion of time. The following definition summarizes the above discussion (as far as the presence of temporal
variables is concerned). For ease of exposition, we assume there exists a unique object sort $D$.

**Definition 3.3.3 (Constraint Types)**

1. A state constraint is a formula of the form $\forall \pi/\forall t_1, t_2/TimeInterval \; F(\pi, t_1, t_2)$, where $F$ is a formula of the assertion language containing no temporal predicates, $t_1$ is a history time variable and $t_2$ is a belief time variable.

2. A dynamic constraint is a formula of the form $\forall \pi/\forall t_1, t_2, t/TimeInterval \; F(\pi, t_1, t_2, t)$, where $F$ is a formula of the assertion language containing a temporal predicate restricting $t_1, t_2$ and $t$ is a belief time variable.

3. A periodic constraint is a dynamic constraint in which $t_1 = k \times t_2$ or $t_1 = k \times T$ for some integer $k$ and a constant interval $T$.

4. A N-times aperiodic constraint is a formula of the form $\forall \pi/\forall t_1, t/TimeInterval \exists t_2/TimeInterval \; F(\pi, t_1, t_2, t)$, where $F$ contains a temporal predicate restricting $t_1, t_2$ and $t$ is a belief time variable.

### 3.4 Constraint Satisfaction

In this section we elaborate on the notion of constraint satisfaction in the context of temporal knowledge bases. We adopt a model-theoretic view of a knowledge base [NG78], [Rei84]. Integrity constraints and deductive rules form a many-sorted first-order theory which we augment by an axiomatization of time as linear, discrete and unbounded, and of the structuring principles of the underlying knowledge model. A state of the knowledge base is an interpretation of the theory. Valid states are the ones that constitute finite models of the theory. The concept of constraint satisfaction, as defined in, e.g., [LS87] and [Cho92a], refers to a single only temporal dimension, namely that of history time. The notion of temporal validity must be modified so that both history and belief time are taken into account.

Intuitively, since a constraint itself has its own history and belief time intervals, it is meaningful to talk about the validity of a constraint during a certain time interval only if this time interval is contained in the belief time interval of the constraint. Hence, in the following, we will assume that the validity of a constraint is examined during an interval that satisfies the above requirement. We begin with some required formal machinery.
\textbf{Definition 3.4.1 (Knowledge Base)}

A \textit{Telos} knowledge base, \( KB \), comprises a set of propositions, \( KB_p \), defining the validity of predicates over (history) time intervals, as well as a set, \( KB_r \), of deductive rules and a set, \( KB_i \), of integrity constraints.

Since well-formed formulae of the assertion language refer both to objects in the domain of discourse and time intervals, we need to define the notions of \textit{object} and \textit{temporal} variable substitutions.

\textbf{Definition 3.4.2 (Object Substitution)}

An \textit{object variable substitution} \( \sigma \) is a function mapping a variable \( x_i \) of sort \( S_i \) to an instance of the corresponding \textit{Telos} class \( C_i \) so that \( \text{instanceOf}(\sigma(x_i), C_i; t) \in KB_p \) for some interval \( t \).

\textbf{Definition 3.4.3 (Temporal Substitution)}

Let \( I \) denote the set of time intervals with integer endpoints. A \textit{temporal variable substitution} \( \tau \) is a function mapping a temporal variable \( t \) of sort \textit{Time} to an interval in \( I \).

Then the satisfaction of temporal formulae is defined as follows\(^7\):

\textbf{Definition 3.4.4 (Satisfaction of Temporal Formulae)}

For a base predicate\(^8\) \( P \), object substitution \( \sigma \) and temporal substitution \( \tau \)

\begin{itemize}
  \item If \( P \) is ground, then \((KB, \sigma, \tau) \models P\) if and only if (iff) \( P \in KB_p \).
  \item \((KB, \sigma, \tau) \models P(x, t)\) iff there exists an interval \( i \in I \) such that \( \tau(t) \) is contained in \( i \) and \((KB, \sigma, \tau) \models P(\sigma(x), i)\).
  \item \((KB, \sigma, \tau) \models \neg P(x, t)\) iff there does not exist an interval \( i \) containing \( \tau(t) \) such that \((KB, \sigma, \tau) \models P(\sigma(x), i)\).
  \item If \( Q \) is also a base predicate, then \((KB, \sigma, \tau) \models P(x, t_1) \lor Q(x, t_2)\) iff \((KB, \sigma, \tau) \models P(x, t_1) \lor (KB, \sigma, \tau) \models Q(x, t_2)\).
\end{itemize}

\(^7\)We give the definitions of formulae satisfaction where formulae contain one variable of object and temporal sorts. The generalization to multiple variables of object and temporal sorts is straightforward.

\(^8\)The satisfaction of the basic temporal predicates is defined as in [Alb83].
\(\cdot (KB, \sigma, \tau) \models \forall x/C \ P(x,t) \iff (KB, \sigma[x/d], \tau) \models P(x,t) \) for all \(d\) such that \(d\) is an instance of \(C\) for some interval \(i\) containing \(\tau(t)\). \(\sigma[x/d]\) is the substitution that differs from \(\sigma\) only in the binding \(d\) for \(x\).

\(\cdot (KB, \sigma, \tau) \models \forall t/\text{Time} \ P(x,t) \iff \text{for all intervals } i \in I, (KB, \sigma, \tau[t/i]) \models P(x,t).\)

If \(P\) is a derivable predicate defined by a set of deductive rules with bodies \(R_1, \ldots, R_k\) and respective time intervals \(T_1, \ldots, T_k\), then

\(\cdot (KB, \sigma, \tau) \models P(x,t) \iff (KB, \sigma, \tau) \models \bigwedge_{i=1}^{k} R_i \land (t \ \text{during} \ T), \) where \(T\) is the intersection of the intervals \(T_i\).

Now we are in a position to define the satisfaction of temporal integrity constraints. We will use the notation \(C \ \text{[at} \ T\ \text{]}\) to denote an integrity constraint \(C\) associated with the history time interval \(T\).

**Definition 3.4.5 (Satisfaction of Temporal Integrity Constraints)**

If the temporal variables \(t_1, \ldots, t_k\) occur in the constraint \(C\), then \((KB, \sigma, \tau) \models C \ \text{[at} \ T\ \text{]}\)

\(\iff (KB, \sigma, \tau) \models C', \) where \(C' \equiv C \land \bigwedge_{i=1}^{k} \text{during}(t_i, T).\)

The above definition shows that the semantics of integrity constraints is taken into account in the definition of the notion of satisfaction. It would not make sense to evaluate an integrity constraint outside its validity interval. The satisfaction of temporal formulae over temporal elements follows similar lines: we require that the formula is satisfied for every interval in the element.

Having defined the satisfaction of temporal formulae, we now give the following definition for validity of temporal constraints:

**Definition 3.4.6 (Historical Validity)**

A temporal integrity constraint \(C\) is said to be historically valid, or simply valid, if \((KB, \sigma, \tau) \models C \ \text{[at} \ T\ \text{]}\) for every temporal substitution \(\tau\).

The above definition implies that, in order to decide the validity of some integrity constraint, one may have to examine its truth for intervals that extend to the future of current time. For instance, the history time interval of a constraint may be a right-infinite interval. Hence, this notion of validity is far too general.
Although time is infinite, the satisfaction of constraints has to be decided over a finite history of the knowledge base. For that, different notions of constraint validity have been defined [LS87],[HS90],[HS91], [Cho92a]. The definitions found in these papers refer to time points instead of intervals. These notions are recast here in an interval-based context.

**Definition 3.4.7 (Potential Validity)**

A constraint \( C \) is called *potentially valid* during a time interval \( T \), if the finite history of the knowledge base ending at the end of interval \( T \), can be extended to an infinite history in which \( C \) is valid.

Potential validity is a weaker notion than strict historical validity. In fact, this notion of validity is necessary for determining constraint (non-) violations for certain classes of constraints. Finally we introduce the notion of *historical admissibility* as an even weaker notion than potential validity.

**Definition 3.4.8 (Admissibility)**

A constraint is said to be admissible during a time interval \( T \), if it is true for every time interval during the history of the constraint up to \( T \).

The notion of admissibility is sufficient for formulae whose truth value can be determined using the current history only. However, there may be formulae, e.g. expressing a property of the form “sometime in the future ...”, that are potentially valid but not admissible. Hence, potential validity allows for certain types of constraints to be provisionally non-violated, if they are potentially valid and not violated in the history up to the present time.

### 3.4.1 Expressive Power

The choice of such a language is motivated by the requirement of being able to refer to both history and belief time and reason about the consistency of the knowledge base in past or present states. Moreover, it allows us to express different types of constraints not expressible in formalisms that employ a single only or no notion of time. We argue that we can express constraints and rules in a more succinct and intuitive way than formalisms such as FOTL [Eme90], [RU71]. As an example\(^9\), consider the constraints that express the requirements that “an order can be supplied only once" and that “orders should be

---

\(^9\)Example borrowed from [CN93].
fulfilled on a first-come first-served basis”. Let \( s(x) \) and \( f(x) \) denote that order \( x \) has been submitted and fulfilled respectively. Then, the expression of the constraints in the future fragment of FOTL are as follows:

1. \( \forall x \Diamond(s(x) \Rightarrow \Diamond \neg s(x)) \)
2. \( \forall x \forall y (x \neq y \land S(x) \land \Diamond((-f(x)) \text{ until } (s(y) \land \Diamond((-f(x)) \text{ until } (f(y) \land \neg f(x))))) \)

The same constraints can be expressed as follows\(^{10} \) in RAL.

1. \( \forall x/\text{Order} \forall t/\text{TimeInterval} \text{ Holds}(s(x), t) \Rightarrow \neg \exists t'(\text{Holds}(s(x), t') \land (t' \neq t)) \)
2. \( \forall x, y/\text{Order} \forall t_1, t_2/\text{TimeInterval} ((x \neq y) \land \text{ Holds}(s(x), t_1) \land \text{ Holds}(s(y), t_2) \land \text{ before}(t_1, t_2)) \Rightarrow \exists t_3, t_4/\text{TimeInterval}(\text{Holds}(s(x), t_3) \land \text{ Holds}(s(y), t_4) \land \text{ before}(t_3, t_4) \land \text{ meets}(t_1, t_3) \land \text{ meets}(t_2, t_4)) \)

The expression of the second constraint in particular is rather cumbersome when formulated in FOTL, and certainly less intuitive than its expression in the RAL.

As the following results state, the assertion language we propose can express all FOTL expressions. Hence, AL (RAL) is at least as expressive as FOTL. The above example showed how FOTL expressions of the form \( \square P, \diamond P \) and \( P \text{ until } Q \) can be expressed in our language. In a similar manner, we can express properties of the past, i.e., formulae of the form \( \blacksquare P, \blacklozenge P \) and \( P \text{ since } Q \).

Proposition 3.4.1 establishes an embedding of FOTL in RAL. The semantic equivalence of FOTL expressions with their RAL counterparts is expressed by proposition 3.4.2.

**Proposition 3.4.1** FOTL can be embedded in RAL.

**Proof:** Proving this proposition involves establishing a relationship between the notion of a temporal database as this is defined in formalisms using FOTL for expressing temporal constraints, and a knowledge base as it is used in the present work.

FOTL formulae are evaluated in finite sequences of first-order structures, where each structure \( S \) consists of a universe \( |S| \), an interpretation \( c^S \in |S| \) for each constant symbol \( c \) and an interpretation \( p^S \subseteq |S|^n \) for each relationship of arity\(^{11} \) \( n \). A binary predicate \( P \) is said to be true of a tuple \( (a_1, a_2) \) of elements of \( |S| \) if \( (a_1, a_2) \in p^S \). A temporal database \( D \) is a finite sequence \( (D_0, D_1, \ldots, D_k) \) of first-order structures over the same universe \( |D| \).

\(^{10}\)Belief time is omitted from the expression of the constraints here.

\(^{11}\)For the purposes of this proof we restrict attention to relationships of arity 2, since our knowledge bases can only directly represent binary relationships by means of propositions.
$D_i$ is called the database state at time $i$. If $v$ is a valuation and $t_1, t_2$ are terms (constants or variables), the truth of FOTL formulae in a temporal database is defined with respect to a state at a particular point in time:

- $D, v, i \models p(t_1, t_2)$ iff $(v(t_1), v(t_2)) \in p^{D_i}$
- $D, v, i \models \neg A$ iff it is not the case that $D, v, i \models A$
- $D, v, i \models A \land B$ iff $D, v, i \models A$ and $D, v, i \models B$
- $D, v, i \models \forall x A$ iff for all $d \in |D|$, $D, v[x/d], i \models A$
- $D, v, i \models \bullet A$ iff $i > 0$ and $D, v, i - 1 \models A$
- $D, v, i \models A$ since $B$ iff for some $j, 0 \leq j < i$, $D, v, j \models B$ and for all $k, j < k \leq i$, $D, v, k \models A$.
- $D, v, i \models \circ A$ iff $D, v, i + 1 \models A$
- $D, v, i \models A$ until $B$ iff for some $j, j > i$, $D, v, j \models B$ and for all $k, i \leq k < j$, $D, v, k \models A$.

In the knowledge bases considered in the present work, atomic formulae are associated with time intervals denoting the time span over which they are considered to hold. Hence, the analogue of testing the truth of a predicate in a database state at a particular point in time, is the testing of the existence of a proposition in the knowledge base that associates the truth of the predicate with an interval that includes the time point.

Given this observation, we proceed to define a mapping from the class of FOTL formulae interpreted over a finite temporal database $D$, to the class of RAL formulae interpreted over a knowledge base $KB$ encoding the same knowledge about the domain of discourse as $D$. We assume that the universe $|D|$ coincides with that of $KB$. The mapping produces formulae of RAL, that are in fact "weaker" than their FOTL counterparts, in the sense that the former will refer to entire intervals over which formulae will be considered to be true. Strict equivalence can be achieved by restricting the intervals to be unit intervals, whose endpoints are equal to the time instant at which the FOTL formula is evaluated. Without loss of generality, we will assume that the modeling of the domain history begins at time 0 and that the indices of the states of the temporal database are time instants, expressed
as integer numbers. Hence, for every binary predicate \( p \) true in state \( i \) of \( D \), there exists a proposition \( \text{prop}(\text{pid}, x, p, y, t) \) in the \( KB \) such that the time interval \( t \) contains the time instant \( i \).

For conciseness in notation, we introduce \( \iota \), a function mapping an integer to the unit time interval with the integer as its endpoints. Then, the mapping \( \mu \) is defined inductively as follows for the past fragment of FOTL. The mapping for the future fragment is defined quite similarly.

- If \( p \) and \( q \) are binary predicates, and \( i \) a time instant, then
  \(- \mu(p, i) = \text{Holds}(p, \iota(i)) \)
  \(- \mu(\bullet p, i) = \text{Holds}(p, \iota(i - 1)), \text{if } i > 0 \)
  \(- \mu(p \text{ since } q, i) = \text{Holds}(p, \iota(i)) \land \exists J/\text{TimeInterval} (\text{before}(J, \iota(i))) \land \text{Holds}(q, J) \land \forall K/\text{TimeInterval} (\text{meets}(J, K) \land \text{meets}(K, \iota(i))) \Rightarrow \text{Holds}(p, K) \)

- Assume that \( \phi_1, \phi_2 \) are FOTL formulae. Then,
  \(- \mu(\neg \phi_1, i) = \neg \mu(\phi_1, i) \)
  \(- \mu(\phi_1 \land \phi_2, i) = \mu(\phi_1, i) \land \mu(\phi_2, i) \)
  \(- \mu(\forall x \phi_1, i) = \forall x / D \mu(\phi_1, i) \)
  \(- \mu(\bullet \phi_1, i) = \mu(\phi_1, i - 1) \)
  \(- \mu(\phi_1 \text{ since } \phi_2, i) = \mu(\phi_1, i) \land \exists J/\text{TimeInterval} (\text{before}(J, \iota(i))) \land \mu(\phi_2, J) \land \forall K/\text{TimeInterval} (\text{meets}(J, K) \land \text{meets}(K, \iota(i))) \Rightarrow \mu(\phi_1, i) \)

\[\square\]

**Proposition 3.4.2** A FOTL formula \( \phi \) and its RAL counterpart are semantically equivalent, i.e., given a temporal database \( D \) over which \( \phi \) is evaluated, a valuation \( v \) and a knowledge base \( KB \) encoding \( D, D, v, i \models \phi \) iff \( (KB, v, \tau) \models \mu(\phi, i) \) for a temporal substitution \( \tau \).

**Proof:** Follows immediately from the embedding of FOTL in RAL (proposition 3.4.1) and the definition of temporal formulae satisfaction in a knowledge base. \[\square\]

Finally, we show that RAL is strictly more expressive than FOTL. It suffices to find an expression of RAL that is not expressible in FOTL.
Theorem 3.4.1 RAL is strictly more expressive than FOTL.

Proof: Propositions 3.4.1 and 3.4.2 established that RAL is at least as expressive as FOTL. To show that it is strictly more expressive, it suffices to find a property expressible in RAL that is not expressible in FOTL.

An example of such a RAL formula is one expressing a periodic property. There is no analog of the transposition operator in FOTL and periodicity properties cannot be expressed. As an example consider the constraint specifying the property that every university affiliate must author at least one conference paper every year. Then, assuming a time interval granularity of a year\(^2\), the constraint can be expressed as the formula
\[
\forall u/\text{UnivAffiliate} \exists p/\text{Confpaper} \exists t/\text{TimeInterval} \quad \text{Holds}(\text{author}(u,p), t) \Rightarrow \\
\forall k/\text{Integer} \exists p'/\text{Confpaper} \exists t'/\text{TimeInterval} \quad \text{Holds}(\text{author}(u,p'), t') \land \\
\text{during}(t', k \times t) 
\]
\[\square\]

3.4.2 Complexity Issues

Testing the validity of temporal formulae is a difficult problem. It has been shown [Cho92a] that testing for potential validity of FOTL formulae is \(\Pi_1^1\)-complete, that is, undecidable. Admissible validity is decidable for the Past fragment of FOTL. It has also been shown in [CN93] that, for the class of Future FOTL formulae for which all temporal operators apply to quantifier-free subformulae and in which all quantifiers over domain objects are universal and occur in the beginning of the formula, temporal integrity checking is decidable (in exponential time).

A precise complexity characterization of the problem of temporal integrity checking in our language is yet to be given. The validity problem in the unrestricted language is also undecidable, since the language can express all FOTL constraints. We are currently investigating the complexity of testing potential validity and admissibility and the restriction of the language into a maximal tractable subset.

\(^{12}\)This is done only for simplicity and conciseness in the expression of the formula.
Chapter 4

Temporal Integrity Monitoring

This chapter presents the first of the integrity maintenance methods we propose in this dissertation. It is termed an *integrity monitoring* method since it relies on the presence of a centralized subsystem that maintains knowledge about the consistency of the knowledge base. The proposed method is an incremental compile-time simplification method that accounts for implicit updates as well as temporal knowledge.

Our approach builds on the method proposed in [BDM88], which was adapted to an object-oriented setting in [JJ91]. It extends the latter by the treatment of temporal constraints and by the introduction of an efficient compilation scheme that allows us to optimize the computation of implicit updates and perform additional simplifications. The efficiency of the method stems from the separation of the task of constraint maintenance in two separate phases: a *compilation* phase, performed at schema definition time and an *evaluation* phase performed at knowledge base update time. During compilation, constraints and relevant rules are compiled into simplified forms whose evaluation can be triggered by the occurrence of affecting updates. Our proposal advances the method of [JJ91] by taking time into account and by optimizing the compile-time computation of implicit updates. Moreover, the simplified forms are organized into a *dependence graph*, a structure that reflects the logical and temporal interdependence of deductive rules and integrity constraints. The organization of the compiled forms in dependence graphs permits us to efficiently evaluate the implicit updates that may affect the integrity of the knowledge base and also apply additional optimization steps.

Compilation and simplification apply uniformly to integrity constraints and the bodies
of their relevant deductive rules. The assumptions that, first, the knowledge base is known to satisfy its constraints prior to an update and, second, that the types of updates can be anticipated, are exploited. Throughout the discussion, we will assume that the contents of the knowledge base are up to date and that they accurately reflect the history of the domain being modeled. Thus, the knowledge base can only be modified by an extension of the stored history.

An initial proposal was first given in [Ple93a] where simplified forms of constraints and rules with respect to a single affecting update were generated. The method was extended to handle arbitrary transactions in [Ple94] and [Ple95a].

4.1 Compilation and Simplification

We begin this section by presenting the theoretical framework upon which our method builds. The notions of affecting updates, transactions and literal dependence are defined first.

**Definition 4.1.1 (Updates and Transactions)**

An update is an instantiated literal whose sign determines whether it is an insertion or a deletion. A transaction is an arbitrary set of updates.

To avoid having to verify the entire set of constraints in the knowledge base we must first distinguish between those updates that may potentially violate an integrity constraint and those that cannot possibly affect its satisfaction and, thus, do not warrant the constraint’s evaluation. We will henceforth assume that constraints are written in disjunctive normal form (DNF). A constraint in DNF is affected by an update only when a “tuple” is inserted into the extension of a literal occurring negatively in the constraint, or when a “tuple” is deleted from the extension of a literal occurring positively in the constraint. The definition of relevance found in [JJ91] is not sufficient in the presence of time. Definition 4.1.2 provides sufficient conditions for “relevance” of a constraint to an update, by considering the relationship of the history time intervals participating in the literals of the constraint and the update in addition to that between literal occurrences (positive or negative) and types of updates (insertions or deletions).

**Definition 4.1.2 (Affecting Updates and Transactions)**
An update \( U(x, t) \) is an **affecting update** for a constraint \( C \) [at \( T \)] if and only if there exists a literal \( L(x, \ldots) \) in \( C \) such that \( L \) unifies with the complement of \( U \), and the intersection, \( t \cap T \), of intervals \( t \) and \( T \) is non-empty. A transaction \( X = \{ U_1, \ldots, U_m \} \) is called an **affecting transaction** for a constraint \( C \) [at \( T \)] if and only if at least one of \( U_1, \ldots, U_m \) is an affecting update for the constraint.

We define transactions as being sets rather than sequences of updates since, from the point of view of integrity maintenance, it is the net effect of a transaction rather than the order in which individual updates take place that has to be considered.

During compilation, along with each integrity constraint, deductive rules that may contribute to the constraint’s evaluation will be compiled. These are rules whose conclusion literal unifies with literals of the constraint. In this case, it is said that the constraint **directly depends** on the deductive rules. A constraint cannot directly depend on a rule whose conclusion literal does not match any of the constraint’s literals. It can however depend on a rule whose conclusion literal matches a condition literal of a rule on which the constraint depends. We define the notions of **dependence** and **direct dependence** along the lines of [JJ91].

**Definition 4.1.3 (Direct and Transitive Dependence)**

A literal \( L \) **directly depends** on a literal \( K \) if and only if there exists a rule of the form

\[
\forall x_1/C_1 \ldots \forall x_n/C_n (F \Rightarrow A)
\]

such that, there exists a literal in \( F \) unifying with \( K \) with most general unifier \( \theta \) and \( A\theta = L \). A literal \( L \) **transitively depends** (or, simply, depends) on literal \( K \) if and only if it directly depends on \( K \), or depends on a literal \( M \) that directly depends on \( K \).

Hence, a constraint \( C \) depends on a rule \( R \) if a literal in the constraint depends on the rule’s conclusion literal. Similarly, a rule \( R_1 \) depends on a rule \( R_2 \) if \( R_1 \)'s body contains a literal that depends on \( R_2 \)'s conclusion literal. These relationships define a **dependence graph** for a set of rules and constraints. A dependence graph represents how implicitly derived facts can affect the integrity of the knowledge base. Dependence graphs are discussed into more detail in section 4.2.

We now turn to the definition of compiled forms of constraints and rules. The key issue in compilation is to associate every constraint or rule body with the updates that may affect its evaluation. Moreover, for the purpose of restricting the search space for constraints or
rules affected by a transaction the concept of a concerned class [JJ91], [Ple93a] is used. The
definition given in [JJ91] is insufficient in the presence of time.

**Definition 4.1.4 (Concerned Class)**

The concerned class for a literal $L$ is the most specialized class $CC$ such that, inserting
or deleting an instance of $CC$ can affect the truth of $L$ and the time intervals of $L$ and $CC$
are unifiable.

The restriction on the time intervals of the literal and its concerned class in the above
definition is necessary since the specialization/generalization hierarchy may be modified
by the insertion of new classes and because metaclasses whose extension is expected to
be quite large, such as *Proposition*, qualify as concerned classes. To compensate for
schema changes that may result in a concerned class that is more specialized than the one
determined at compile time, a set of rules is introduced for computing the concerned class
for every literal in the constraint. These rules can be formulated in the form of meta-rules
that can be instantiated for each particular literal. The rules express the property that,
if $C$ is a concerned class for literal $L$ and, after an update, a subclass $C'$ of $C$ qualifies as
a concerned class of $L$, then $C'$ is the concerned class of $L$. Another rule expresses the

Concerned classes of literals are used for providing grouping for the simplified forms of
rules and constraints generated. This will become particularly important during evaluation,
since it will permit us to restrict attention only to a relevant set of constraints and rules.
Moreover, unnecessary - from a logical or temporal point of view- evaluations of rules or
constraints can be avoided. We will return to the use of concerned classes during the
evaluation phase in section 4.4.

The following definition specifies rules for the compile-time derivation of concerned
classes of literals that occur in integrity constraints or bodies of deductive rules.

**Definition 4.1.5 (Derivation of Concerned Classes)** Given a literal $L$ occurring in the
expression of an integrity constraint or the body of a deductive rule, the concerned class of
$L$ is derived as follows:

- **Instantiation literals**: for literals of the form $\text{instanceOf}(x,y,t_1,t_2)$, if $y$ is
  instantiated, then $y$ is the concerned class provided this class exists during $t_1$ and
its existence is believed during \( t_2 \); otherwise, the built-in class \texttt{InstanceOf} is the concerned class.

- **Generalization literals:** for literals of the form \texttt{isA}(x, y, t_1, t_2) where both \( x \) and \( y \) stand for classes, the concerned class is the built-in class \texttt{isa}, since the truth of an \texttt{isa}-literal does not depend on the insertion/deletion of instances to/from the extensions of \( x \) and \( y \).

- **Attribute literals:** for a literal of the form \texttt{att}(x, y, t_1, t_2), where \( att \) is an attribute of the class \( x \), if both \( x \) and \( y \) are uninstantiated, then the concerned class of the literal is the unique attribute class \( q \) with components \texttt{from}(q) = X, \texttt{label}(q) = att, \texttt{to}(q) = Y \) and \texttt{when}(q) = T, that is such that, \( x \) is an instance of \( X \) for \( t_1 \), \( y \) is an instance of \( Y \) for \( t_1 \) and both these are believed during \( t_2 \). In other words, the most specialized concerned class is the attribute class that includes all instantiated attributes that relate objects \( x \) and \( y \) of types \( X \) and \( Y \) respectively, under the assumption that to each attribute literal of the assertion language, corresponds a unique proposition with the above properties.

- For a literal of the form \texttt{prop}(p, x, y, z, t), if the components \( x \) and \( z \) are equal, then the concerned class is the built-in class \texttt{Individual}; if not, the concerned class is the class \texttt{Attribute}. In case none of \( x \) and \( z \) are instantiated, the concerned class is the class \texttt{Proposition}.

In what follows, \texttt{prop} literals will not be considered in the generation of simplified forms. We will assume that any proposition mentioned in a \texttt{prop} literal does exist in the knowledge base and is an instance of the class \texttt{Proposition}.

**Proposition 4.1.1** The rules of definition 4.1.5 yield the most restricted class that qualifies as a concerned class of a literal \( L \).

**Proof:** The property follows directly for the case of generalization and proposition literals. For instantiation literals, of the form \texttt{instanceOf}(x, y, t_1, t_2), the classes that qualify as concerned classes are the built-in class \texttt{InstanceOf}, and if \( y \) is instantiated in the literal \texttt{instanceOf}, the class denoted by \( y \) and all its superclasses. Hence, in either case, the class derived in definition 4.1.5 is the correct concerned class. Finally, in the case of
attribute literals, it suffices to note that there exists a unique class with the properties of a concerned class. This class relates object classes, instances of which occur in the attribute literal. Notice that, the concerned class of an attribute literal does not depend on whether its arguments are instantiated or not.

Compilation produces a *parameterized simplified structure* (PSS) for each such literal. This form contains a simplified form of the constraint or rule that suffices to be evaluated when an affecting update on the literal occurs at run time. Note that, for every occurrence of a literal in a constraint or rule, there is only one update that may affect the constraint or rule. Since the only updates possible are insertions and deletions in the extensions of literals, a literal occurrence can only be affected by one of the two operations.

### 4.1.1 Compile-time Simplification

We first list the rules for generating a parameterized simplified form for each update and then show how to obtain the simplified form for a transaction given the simplified forms for individual updates. Note that it is very costly to derive the simplified forms with respect to all possible transactions that may affect a given constraint, due to the exponential number of such transactions.

**Definition 4.1.6 (Parameterized Simplified Structure)**

Given a temporal constraint $C$ [at $T$] expressed in DNF and a literal $L$ occurring positively (negatively) in $C$, the *parameterized simplified structure* of $C$ with respect to $L$ is a 6-tuple $(L, \text{Params}, CC, T, T', SF)$, where, $\text{Params}$ is the list of instantiation variables\(^1\) of $L$, $CC$ is the concerned class of $L$, $T$ and $T'$ are the history and belief time intervals of the constraint respectively, and $SF$ is the simplified form of the constraint that suffices to be evaluated when a deletion from (insertion to) $L$ takes place. $SF$ is derived by the application of the following steps:

1. The quantifiers binding instantiation variables are dropped. Variables become parameters.

2. The temporal variables are constrained with respect to the history and belief time of the constraint: a temporal predicate of the form $\text{during}(t, T)$ is conjoined with $C$

\(^1\)Instantiation variables are $\forall$-quantified variables that are not in the scope of a $\exists$.  

63
for every history time variable \( t_i \) that occurs in \( C \); a temporal predicate of the form \( \text{during}(t_j, T') \) is conjoined with \( C \) for every belief time variable \( t_j \) that occurs in \( C \);

3. The literal into (from) whose extension a tuple is inserted (deleted) is substituted by the \textbf{True} (\textbf{False}) and absorption rules of first-order logic are applied. The standard absorption rules of first-order logic are shown in table 4.1. If a literal occurs more than once with different history time intervals, then select for replacement the literal with the greatest right endpoint.

4. Temporal simplification rules are applied, if applicable.

Before we proceed to present the temporal simplification rules a comment on the treatment of dynamic constraints in step 3 is in order. Dynamic constraints are distinguished from static constraints by the presence of explicit temporal constraints on history time variables. Since they express properties depending on two or more knowledge base states, some literals will occur more than once in their expression. Step 3 selects for replacement the one literal with a time interval extending to the future of the time intervals in the other literal occurrences. This is done because the constraint must be verifiable using the history of the knowledge base up to the present state. Determining which literal to replace can be done by comparison of the intervals if their endpoints are known, or by reasoning by cases based on the temporal relationship of the time intervals of the literals. Lemma 4.1.1 expresses the above property.

\textbf{Lemma 4.1.1} Let \( C \) be the constraint \( (\neg L(\overline{x}, t_1) \lor \neg L(\overline{y}, t_2) \lor \neg r(t_1, t_2) \lor R) \ [at \ T] \), where \( R \) is a temporal formula in DNF that does not mention any \( L \) literal and \( r \) is a temporal relation. Given that \( C \) is known not to be violated in the state prior to the occurrence of an affecting update on \( L \), \( C \)’s satisfaction can be determined in the state after the update if and only if the satisfaction of \( \overline{C} \) can be determined, where \( \overline{C} \) is the formula obtained from \( C \) as follows: if \( r \) is \textbf{before}, \textbf{during}, \textbf{overlaps}, \textbf{meets}, \textbf{starts} or \textbf{finishes}, then the literal

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
\( \phi \land \text{True} \equiv \phi \) & \( \phi \land \text{False} \equiv \text{False} \) & \( \phi \rightarrow \text{True} \equiv \text{True} \) & \( \phi \rightarrow \text{False} \equiv \neg \phi \) \\
\hline
\( \phi \lor \text{True} \equiv \text{True} \) & \( \phi \lor \text{False} \equiv \phi \) & \( \text{True} \rightarrow \phi \equiv \phi \) & \( \text{False} \rightarrow \phi \equiv \text{True} \) \\
\hline
\( \neg \text{True} \equiv \text{False} \) & \( \neg \text{False} \equiv \text{True} \) & \( \phi \leftrightarrow \text{True} \equiv \phi \) & \( \phi \leftrightarrow \text{False} \equiv \neg \phi \) \\
\hline
\end{tabular}
\caption{Absorption rules}
\end{table}
\( -L(y, t_2) \) is substituted with a Boolean constant and the variables \( y, t_2 \) become instantiated. If \( r \) is \textit{after}, \textit{contains}, \textit{overlapped-by}, \textit{met-by}, \textit{started-by} or \textit{finished-by}, then the literal \( -L(x, t_1) \) is substituted with a Boolean constant and the variables \( x, t_1 \) become instantiated. If \( r \) is \textit{equal}, any of the literals can be substituted.

\textbf{Proof:} \( \overline{C} \) is obtained in a way such that the variables occurring in the affected literals are instantiated with the appropriate update parameters. Under the assumption that the knowledge the system possesses is up to date, an update introduces knowledge that refers to the current state. Hence, by examining the type of the temporal relationship \( r \), we can identify the literal that refers to the current state. Constraint \( \overline{C} \) is derived by variable substitution and, thus, it is implied by the original constraint. The opposite direction follows from the assumption that it is known that the constraint is not violated prior to the occurrence of an affecting update. \( \square \).

\textbf{Example 4.1.1} Applying steps 1-4 to the dynamic constraint IC2 of example 3.3.1 will generate the following simplified form (capitalized variables denote parameters):

\[
\forall s/\text{Integer} \ \forall t_1/\text{TimeInterval} \ (\text{salary}(P, s, t_1) \ \land \text{during}(t_1, 02/01/1993..*) \\
\land \ \text{before}(t_1, T_2) \Rightarrow (s \leq S'))
\]

In this example, the literal \( \text{salary}(p, s', t_2) \) of the original constraint was replaced because of the relationship \( \text{before}(t_1, t_2) \) between \( t_1, t_2 \). \( \square \)

### 4.1.2 Temporal Simplification

The last step in the generation of parameterized simplified forms is temporal simplification. The objective of this step is to simplify a conjunction of temporal relationships into a single temporal relationship. Carrying out temporal simplification requires the employment of a temporal reasoner for deducing, in those cases this is possible, a temporal expression simplifying a conjunction of temporal relations. In its generality, the task may not be feasible. It has been shown [All83] that certain combinations of temporal constraints introduce incomplete knowledge (disjunction). In our case however, where at least one of the temporal variables in the temporal relations introduced is instantiated, temporal simplification can be performed efficiently.

Formally, the problem of temporal simplification is stated as follows: given a conjunction \( \text{during}(t, i_1) \land r_1(t, i_2) \), where \( r_1 \) is any of the 13 possible relationships between any two
time intervals or the negation of such a relationship, and \( i_1, i_2 \) are known time intervals, find a temporal relationship \( r \) and an interval \( i \) such that \( r(t, i) \) is satisfied if and only if the original conjunction is satisfied. The interval \( i \) is a function of the intervals \( i_1 \) and \( i_2 \). The fact that the intervals \( i_1 \) and \( i_2 \) are known permits us to derive a relationship \( r_2(i_1, i_2) \). This relationship serves to restrict the possibly multiple alternatives for \( r \). In fact, the expression that is simplified is the conjunction \( \text{during}(t, i_1) \land r_1(t, i_2) \land r_2(i_1, i_2) \).

It is not always possible to derive a single definite relation \( r \) that has the above property. For some combinations of temporal relationships \( r \) is a disjunction of temporal relationships. In those cases, and for the sake of completeness, we do not replace the original expression by the equivalent disjunction. Note also, that the presence of the additional relationship \( r_2(i_1, i_2) \) permits to perform simplifications in the case of negations of temporal relationships without introducing disjunction.

The table in figure 4.1 contains all simplifications that can be carried out at compile time. The rows are labeled with all 13 possible relationships between the time intervals \( i_1 \) and \( i_2 \), whereas the columns are labeled with those between intervals \( t \) and \( i_2 \). The column labels are abbreviations of the corresponding row labels. All relationships are treated as being mutually exclusive. The content of each table entry is the relationship \( r \) between \( t \) and \( i \) that fulfills the properties stated above, if such a relationship can be found without introducing indefiniteness. In those cases where this is not possible, the entries in the table indicate that no simplification is performed. Inconsistencies arising in some of the 169 possible combinations are also discovered. \( F \) indicates that a combination of relationships is unsatisfiable. A similar table can be defined for the cases where the negation of a temporal relationship appears in \( r_1 \). The operations of intersection and difference can be performed efficiently for intervals with known endpoints. We assume that the cost of performing these operations is negligible compared to that of evaluating an atomic formula of the assertion language. The task of temporal simplification does not introduce any prohibitive complexity. Its requirements include determining the relationship between the known time intervals, a constant time operation, a simple table lookup for finding the simplifying expression and simple operations between interval endpoints.

**Example 4.1.2** Consider the conjunction of temporal relationships:

\[
during(t, 01/88..09/88) \land before(t, 05/88..12/88) \land overlaps(01/88..09/88, 05/88..12/88)
\]

According to the simplification table of figure 4.1, the above expression can be simplified
# Temporal Simplification Table

<table>
<thead>
<tr>
<th>rl(t,i2)</th>
<th>b</th>
<th>d</th>
<th>o</th>
<th>m</th>
<th>s</th>
<th>f</th>
<th>e</th>
<th>a</th>
<th>c</th>
<th>ob</th>
<th>mb</th>
<th>sb</th>
<th>fb</th>
</tr>
</thead>
<tbody>
<tr>
<td>r2(t,i1,i2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>before</td>
<td>during i2</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>during</td>
<td>during i1</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>overlaps</td>
<td>during (i1,i2)</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>meets</td>
<td>during i1</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>starts</td>
<td>during i1</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>finishes</td>
<td>during i1</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>equal</td>
<td>during i1</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>after</td>
<td>during i1</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>contains</td>
<td>during (i1,i2)</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>overlapped by</td>
<td>during (i1,i2)</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>met by</td>
<td>during i1</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>started by</td>
<td>during i2</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>finished by</td>
<td>during (i1,i2)</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

**Legend**

- **F**: false
- **no simp.**: no simplification possible
- *****: intersection operator
- **-**: difference operator
4.1.3 Soundness and Completeness

The simplification method consists of a number of truth-preserving transformations that produce formulae which, if proven not to be satisfied in the resulting knowledge base, imply that the original formulae are not satisfied. Moreover, no inconsistency can be introduced by any of the simplification steps. The method is also complete, in the sense that all possible temporal transformations are performed. No transformation takes place in those cases where the derived temporal relationship is a disjunction of temporal predicates. Theorem 4.1.1 expresses the soundness and completeness properties of simplification.

Theorem 4.1.1 (Soundness)

A constraint $C$, known not to be violated in the state prior to the occurrence of an affecting update, is violated in the state resulting from the update if the formula produced after applying the simplification steps 1-4 to $C$ is.

Proof: We will first show that the simplification process for integrity constraints and deductive rules consists of a series of truth-preserving transformations so that, if the formula resulting from the application of the transformations is violated in the updated knowledge base, the original formula is violated. It is assumed that the knowledge base is in a consistent state before the update takes place.

Let $ic$ denote the constraint before the application of any one of the simplification steps and let $KB'$ denote the updated knowledge base. By $ic_{(i)}$ we denote the constraint resulting from the application of the $i$-th simplification step. Hence, $ic = ic_{(0)}$. We will show the following:

$$KB' \models (ic_{(i-1)} \Rightarrow ic_{(i)}), \ i = 1, 2, 3, 4$$
$$KB' \models \neg ic_{(4)} \Rightarrow KB' \models \neg ic$$

($i = 1$) We need to show that $KB' \models (ic \Rightarrow ic_{(1)})$, where $ic_{(1)}$ is the result of dropping all quantifiers binding instantiation variable from $ic$. Recall that instantiation variables are universally quantified variables not occurring in the scope of an existential quantifier. Hence we will only deal with the case where $ic$ is of the form...
\[ \forall x_1/C_1 \ldots x_m/C_m \; Q \; \phi(x_1, \ldots, x_m, \ldots) \] where \( Q \) is any alternation of \( \exists \) and \( \forall \), so that every universally quantified variable (if any) occurs in the scope of an existential quantifier (if any). It is known that \( \forall x \; \phi(x) \Rightarrow \phi(c) \) is valid for any first-order formula \( \phi \) and any constant \( c \). Hence, the formula \( \forall x \; (T(x) \Rightarrow \phi(x)) \Rightarrow (T(c) \Rightarrow \phi(c)) \) is also valid. Also, \( \forall x/T_1 \; \forall y/T_2 \; \phi(x,y) \equiv \forall x \forall y \; (T_1(x) \land T_2(y) \Rightarrow \phi(x,y)) \), and thus \( \forall x \forall y \; (T_1(x) \land T_2(y) \Rightarrow \phi(x,y)) \Rightarrow \forall y \; (T_1(c_1) \land T_2(y) \Rightarrow \phi(c_1,y)) \) is valid, as is the formula \( \forall y \; (T_1(c_1) \land T_2(y) \Rightarrow \phi(c_1,y)) \Rightarrow (T_1(c_1) \land T_2(c_2)) \Rightarrow \phi(c_1,c_2) \) for any constants \( c_1, c_2 \). From this, it follows that we can eliminate the quantifiers binding instantiation variables without affecting the truth of the formula. Hence, \( KB \models (\forall x_1/C_1 \ldots x_m/C_m \; Q \; \phi(x_1, \ldots, x_m, \ldots) \Rightarrow (C_1(x_1) \land \ldots \land C_m(x_m) \Rightarrow Q\phi(c_1, \ldots, c_m, \ldots)) \). Step 1 instantiates variables with parameters which at run time are guaranteed to be of the appropriate types. Hence, \( KB \models (ic \Rightarrow Q \; \phi(c_1, \ldots, c_m, \ldots)) \), i.e., \( KB \models (ic \Rightarrow ic_{(1)}) \).

(i = 2) We need to show that \( KB' \models (ic_{(1)} \Rightarrow ic_{(2)}) \), where \( ic_{(2)} \) is the result of conjoining with \( ic_{(1)} \) a conjunction of during predicates constraining the temporal variables to be intervals contained in the history and belief time intervals of the constraint. The semantics of temporal constraint satisfaction specify that an integrity constraint \( ic(x_1, \ldots, x_n, t_1, \ldots, t_l) \) with history and belief time intervals \( T \) and \( T' \) respectively is only applicable when the temporal variables occurring in the constraint are instantiated to intervals contained in \( T \) and \( T' \). Step 2 enforces this requirement. Hence if \( ic_{(2)} \) is violated, then either \( ic_{(1)} \) is violated or the temporal constraints are not satisfied. In the former case, we conclude that \( KB' \models (ic_{(1)} \Rightarrow ic_{(2)}) \). In the latter case, \( ic_{(1)} \) is not necessarily violated, but the constraint is not applicable from a temporal point of view. Hence, the implication trivially holds.

(i = 3) Step 3 replaces the atom \( \alpha \) in the literal unifying with the complement of the update by a Boolean constant, True or False, depending on whether the update is an insertion or a deletion. Without loss of generality, assume that \( ic_{(2)} \) is in conjunctive normal form.\(^2\) Thus, \( ic_{(3)} \equiv (l_{i,1} \lor \ldots \lor l_{i,j_1}) \land \ldots \land (l_{k,1} \lor \ldots \lor l_{k,j_k}) \land \ldots \land (l_{i,1} \lor \ldots \lor l_{i,j_i}) \), where \( i, j, k \geq 0 \) and each \( l_{u,v} \) is a literal. If \( \alpha \) occurs positively in some literal \( l_{i,j} \) of

\(^2\) It is easily seen that every constraint of the form defined in section 3.1, can be written as a conjunction of disjunctions.
a disjunct \((l_{i,1} \lor \ldots \lor l_{i,j})\) and the update is a deletion, then \(l_{i,j} \equiv \text{False}\) and the
truth value of the disjunct remains unchanged. If \(\alpha\) occurs negatively in some literal
\(l_{i,j}\) of a disjunct \((l_{i,1} \lor \ldots \lor l_{i,j})\) and the update is an insertion, then \(l_{i,j} \equiv \text{False}\)
and, again, the truth value of the disjunct remains unchanged. The absorption rules
applied are also known to be valid. Hence, \(KB' \models (ic_{(2)} \Rightarrow ic_{(3)})\).

\((i = 4)\) The temporal simplification step produces, in the cases where it is applicable,
expressions that are equivalent to the original disjunction. The truth of each transform-
ation can be easily verified. In cases, where an equivalent non-disjunctive formula
cannot be found, the temporal expression remains unchanged. In the cases where
inconsistency arises, either the original constraint contains an unsatisfiable temporal
expression, or temporal variables are instantiated to intervals that have an empty
intersection with the time intervals of the constraint. Hence, in all the above cases,
the implication \(KB' \models (ic_{(3)} \Rightarrow ic_{(4)})\) holds.

Finally, \(KB' \models \neg ic_{(4)} \Rightarrow KB' \models \neg ic\) follows easily from the proven implications. \(\square\)

As far as the completeness of the temporal simplification rule is concerned, exhaustive
testing of all entries in the table of figure 4.1, can verify that all possible simplifications by
a definite formula are contained in the table. Moreover, any valid temporal relationship \(r\)
between two time intervals \(i_1\) and \(i_2\) can be written in the form \(\text{during}(i_1, I) \land r_1(i_1, I') \land
r_2(I, I')\), with \(i_2\) a function of \(I, I'\). If both \(i_1\) and \(i_2\) are instantiated, then one choice for \(I\)
is the tighter interval that includes both \(i_1\) and \(i_2\). Then, the equivalence holds by setting
\(I' = (i_1+, I+), r_2 = \text{contains}\) and \(r_1 = r\). If one only of \(i_1, i_2\) is instantiated, then an
equivalent conjunction of temporal relationships can be determined by an exhaustive search
of the 169 choices for \(r_1, r_2\).

4.2 Dependence Graphs

In this section we examine the organization of the simplified forms of rules and constraints
into \emph{dependence graphs} and the optimization potentials that this scheme provides.

4.2.1 Graph Construction and Properties

The definitions of direct and transitive dependence define a directed graph representing how
literals, implicitly derived from deductive rules, can affect the integrity of the knowledge
base. The graph nodes are the PSSs of rules and constraints. Edges denote dependence of constraints/rules on rules.

**Definition 4.2.1** The dependence graph of a knowledge base KB is defined as $G(KB) = (V, E)$, where $V$ comprises one node for each PSS of an integrity constraint or deductive rule of $KB$. $V = V_I \cup V_R$, where $V_I$ and $V_R$ are the sets of nodes corresponding to integrity constraints ($KB_I$) and deductive rules ($KB_R$) respectively. $E = \{(v_i, v_j) | v_i \in V_R, v_j \in V_I \text{ and } v_j \text{ directly depends on } v_i\} \cup \{(v_i, v_j) | v_i, v_j \in V_R \text{ and } v_i \text{ directly depends on } v_j\}$.

**Example 4.2.1** Figure 4.2 shows the dependence graph for our working example. The edge from the PSS for literal **address** of rule **DR1** to the **works_in** literal of **IC2** denotes the direct dependence of **IC2** on constraint **DR1**, whereas the path from node **DR2.univ** to **IC2.works_in** denotes that an update on literal **univ** might cause a violation of constraint **IC2**.

From the graph definition it can be seen that the graph has a particular structure: there are no edges initiating at constraint nodes. A dependence graph has the form shown in figure 4.3. A dependence graph may contain cycles among deductive rule nodes. This happens in the case that the knowledge base contains mutually recursive rules. The graph is free of trivial cycles and enjoys the property expressed in the following theorem.
Theorem 4.2.1 For any Telos knowledge base, dependence graph construction yields a graph that may contain cycles of length at most equal to the number of deductive rules participating in the same recursive scheme.

Proof: The theorem can be proved by induction on the number δ of rules in a recursion scheme. The base case (δ = 1) and the treatment of mutually recursive rules are established by the graph construction. They are demonstrated by means of the following examples. It can be easily seen that the examples are directly generalizable to arbitrary rules.

Example 4.2.2 Assume the following rule and constraint have been defined in the knowledge base:

\[ r \equiv B(x, y) \land A(y, z) \Rightarrow A(x, z) \]
\[ c \equiv A(x, y) \land C(x, y) \]

Then, the compilation of \( r \) will create one node for literal \( B \) and one for the occurrence of literal \( A \) in the body of \( r \). Each of these compiled forms contains implicitly the information that the rule is recursive. The dependence graph generated is the one shown in figure 4.4.

The compilation scheme allows for the evaluation of recursive rule \( r \) by requiring that whenever an implicit update, such as \( A(x, z) \), is generated due to an explicit update, such as an insertion on \( B(x, y) \), then before following another edge for computing subsequent implicit updates, the derived literal has to be matched against the rule literals of the graph. In the example above, this procedure will return to node \( r_A(y, z) \) whenever the rule is evaluated. Note that the same result, as far as rule evaluation is concerned, can be obtained by introducing trivial cycles for the rule nodes. By implicitly encoding the information that the rule is recursive in the rule body nodes and by precomputing the implicit update operation in the compiled forms, the graph remains free of trivial cycles.

The following example shows the formation of cycles in the dependence graph when mutually recursive rules are contained in the knowledge base.
Example 4.2.3 Assume rules $r_1$ and $r_2$ shown below have been defined in the knowledge base.

\[
\begin{align*}
    r_1 &\equiv B(x, y) \land C(x, y) \Rightarrow A(x, y) \\
    r_2 &\equiv A(x, y) \land D(x, y) \Rightarrow C(x, y)
\end{align*}
\]

The corresponding dependence graph is shown in figure 4.5. The graph contains a cycle of length 2, equal to the number of rules involve in this recursive scheme.

In general, assume rules $r_1 \equiv L_{11} \land \ldots \land L_{1k} \Rightarrow A_1$ and $r_2 \equiv L_{21} \land \ldots \land L_{2l} \Rightarrow A_2$ are mutually recursive. This means that there exists a literal $L_{1i}$ in the body of $r_1$ unifying with $A_2$, as well as a literal $L_{2j}$ in the body of $r_2$ unifying with $A_1$. In the dependence graph generated during the compilation of these rules, there will be a node for each one of the literals of the bodies of $r_1$ and $r_2$. Moreover, there will be an edge from each one of the nodes of $r_1$ to the node of the literal $L_{2j}$ and an edge from each one of the nodes of $r_2$ to the node of $L_{1i}$. Hence, there will be an edge from $L_{1i}$ to $L_{2j}$ and one from $L_{2j}$ to $L_{1i}$, thus forming a cycle of length 2. No other edge may participate in the cycle, even if the rules are recursive in themselves, as the previous case demonstrated. Furthermore, there cannot be a cycle of length greater than 2 in the dependence graph resulting from the compilation of 2 mutually recursive rules, since then, either the rules would have to have more than one conclusion literals, or there would have to be edges connecting nodes corresponding to literals of the same rule.

Assuming that the property expressed by the theorem is true for the case of $\delta = n$, it remains to be shown that the property is true for $\delta = n + 1$. This means that there exist $n + 1$ deductive rules such that the $(n + 1)$-st rule body contains a predicate that occurs as the 1st rule’s head.\(^3\) We also assume that the $(n + 1)$-st rule’s head occurs in the body of the $n$-th rule. If we substitute the $(n + 1)$-st rule in the body of the $n$-th rule, the problem is

\(^3\)We assume that we can, without loss of generality, order the rules so that this property become true.
reduced to that of having a recursive scheme of \( n \) rules, which by the induction hypothesis preserves the property expressed by the lemma. Hence, given that a recursive scheme of \( n \) rules may contain a cycle of length at most \( n \), the graph for a recursive scheme of \( n+1 \) rules can be obtained by reversing the reduction of the induction step. Then the only edges that need to be added are those form the first rule’s body vertices to the vertex of the literal of the last rule that matches the first rule’s head. This addition creates a cycle of length \( n+1 \) since previously, there existed a path of length \( n \) from the \( n \)-th rule’s body literal vertices to one of the body literals of the 1st rule. \[ \]

To conclude the discussion on the dependence graph’s construction and properties, let us characterize the size of the graph in terms of the size of the knowledge base. The number of nodes in the dependence graph is in the order of the number of literals occurring in rule and constraint bodies, since one node is created for each compiled form. Let us also assume that the average number of attribute literals per rule or constraint can be estimated and let \( \alpha \) denote this number. The number of compiled forms generated, will then be equal to \( |V| = \alpha \times (|I| + |R|) \). The number of edges is \( |E| = |E_{RR}| + |E_{RC}| \), where \( E_{RR} \) and \( E_{RC} \) are the sets of edges between rule nodes and between rule and constraint nodes respectively. \( |E_{RC}| \) can be at most equal to \( |R| \) since, there exists an edge between compiled forms of a rule and a constraint only if the rule’s head unifies with the constraint’s literal. Hence \( |E_{RC}| \) is at most equal to the number of different literals occurring in rule heads which, in turn, is at most equal to the number of deductive rules in the knowledge base. Similarly \( |E_{RR}| \leq |R| \).

Thus, \( |E| = |E_{RC}| + |E_{RR}| \leq 2 \times |R| \). For \( \alpha > 2 \), \( |V| = \alpha \times (|I| + |R|) > 2 \times |R| \geq |E| \), which means that the graph is sparse. The graph’s sparsity will be exploited for the efficient maintenance of the graph’s transitive closure. The dependence graph is constructed once when the knowledge base is compiled, and is updated incrementally when new rules or constraints are inserted or deleted. Although sparse, the dependence graph for a large knowledge base will be quite large, even too large to fit in main memory. The problem of storage of the dependence graph in secondary storage remains as a future research problem.

The graph reflects both the logical and temporal interdependence of rules and constraints. Following paths from rules to constraints in the graph, permits us to derive implicit updates caused by explicit ones. The set of implicit updates can be precomputed at the time of graph construction using efficient algorithms for transitive closure computation, such as the \( \delta \)-wavefront algorithm of [QHK91] for solving the reachability problem. The
algorithm, applicable to directed cyclic graphs, has been modified to take advantage of the
dependence graph properties. The time complexity for computing implicit updates caused
by an explicit update matching some node in the graph is \( O(|E'|) \), and \( O(|V_R| \times |E|) \) for
computing the transitive closure of the entire graph by solving \( |V_R| \) single-source problems.
Experiments with randomly generated dependence graphs have shown that, on the average,
the complexity of computing transitive closure is sublinear in \( |E| \). At evaluation time reach-
ability information does not have to be recomputed. The implicit updates computed in this
manner are only potential updates. The actual updates can be obtained by instantiating
the potential updates and evaluating the rule bodies in which they occur, starting with the
ones matching the update’s literal and following the order in which the implicit updates
were computed. We elaborate on these properties in the following paragraphs.

Transitive Closure Computation

Devising efficient algorithms for the computation of the transitive closure of database re-
lations has been a subject of intense research in the recent years. An early proposal for
an algorithm for computing the transitive closure of a graph is due to Warshall [War62].
Several improvements of the algorithm have been proposed ever since. Transitive closure
algorithms are classified into iterative (e.g., the seminaive algorithm of [BR86]) and direct
(e.g., [Agr90]). Iterative algorithms repeatedly compute the transitive closure of a relation
until no new tuples are found. The number of iterations they perform depends on the
underlying database. Direct algorithms on the other hand, process each element (tuple, node or edge) a constant number of times and terminate after processing is complete. The
number of iterations they perform is independent of the underlying database. Transitive
closure algorithms are furthermore distinguished in matrix (e.g., [War75]), graph [Ita88] or
hybrid [AJ90] algorithms depending on the representation and the techniques used. Hy-
brid algorithms utilize the optimizing features of graph algorithms in a matrix framework.
Most algorithms on graphs solve a variety of problems involving path computations such as,
critical path, shortest path, bill of materials etc [Yan90]. We are interested in solving the
reachability problem on the dependence graph for the purpose of potential implicit update
precomputation. For the time being, we will restrict ourselves to studying the problem of
computation and incremental update of transitive closure in primary storage. That is we
will assume that enough virtual memory is available to hold the entire dependence graph
and the computed transitive closure. We leave the problem of Input/Output Complexity of transitive closure as a future research issue. The problem has been studied fairly extensively in the literature (e.g., [UY90], [IR88]). The general algorithms proposed need to be modified to take advantage of the particular properties of the dependence graph and the storage scheme used for Telos knowledge bases.

The reachability or single-source transitive closure problem is defined as the problem of determining, given a dependence graph \( G = (V, E) \) and a source vertex \( s \) in \( V \), which vertices of \( V \) are reachable from \( s \). Solving the reachability problem permits the computation of the dependence graph's transitive closure, as well as, the precomputation of the potential implicit updates. The former is possible by solving a number of single-source problems. Specifically, because of the particular structure of the dependence graph, the complete transitive closure can be computed by only solving \( |V_R| \) single-source transitive closure problems. Recall that the vertices corresponding to simplified forms of constraints have no outgoing edges and thus, one only has to consider rule nodes that form inference paths. Hence, in the worst-case, the complete transitive closure computation will solve \( |V_R| \) single-source computations. Implicit updates are computed as the following paragraph explains.

Every vertex \( v \) in a path form a source vertex \( s \) is a potential implicit update if \( v \in V_R \), or indicates a potential constraint violation if \( v \in V_I \). Specifically, given an assumed explicit update on a literal, implicit updates and affected constraints are found as follows. First, the set \( D \) of dependence graph vertices whose characteristic literal matches the update must be located. Set \( D \) can be partitioned in two sets \( D_I \) and \( D_R \) comprising vertices in \( V_I \) and \( V_R \) respectively. The integrity constraints corresponding to the vertices in \( D_I \) are potentially violated by the update if the update operation is of the type (insertion/deletion) that affects the constraint. Recall that each simplified form is affected by only one type of update operation and this information is stored along with the simplified form. The nodes of \( V_R \) reachable from any node in \( D_R \) constitute the implicit updates which may also affect the constraints if there is already a path connecting these vertices with constraint vertices. This information is available if the transitive closure is already computed.

At knowledge base compilation time, the dependence graph construction can also yield additional information that allows the efficient computation of transitive closure. Specifically, an adjacency list representation of the dependence graph can be obtained incrementally and while the graph is built in the compilation phase. Provided the dependence graph
is expected to be sparse, an adjacency list representation is preferable to an adjacency matrix representation. Having the adjacency list representation of the graph, one can apply a suitable algorithm for computing the transitive closure. Such an algorithm should take advantage of the graph’s sparsity and be applicable to cyclic graphs. Moreover, as will become apparent in the discussion to follow, additional information is needed for the efficient maintenance of the dependence graph. Specifically, direct predecessor vertices need also be kept for each graph vertex. Lists of predecessor vertices are needed for the maintenance of the graph, and its computed transitive closure when integrity constraints or deductive rules are deleted and for the purpose of determining actual updates at run-time.

Instantiated transitive closure queries, such as the ones we have to answer while computing transitive closure and implicit updates, can be evaluated efficiently using the $\delta$-wavefront algorithm of [QHK91]. The algorithm is applicable to general (cyclic) graphs. The time complexity of the algorithm for solving the reachability problem is $O(|E|)$. For computing the complete transitive closure, the algorithm has a worst-case complexity of $O(|V| + |E|)$.

The $\delta$-wavefront algorithm is shown in figure 4.6. It has been modified for the computation of the transitive closure of the dependence graph. The procedure accepts as parameter the source node $s$. It assumes that the graph $G$ is represented by the set of adjacent vertices and the associated labels. In the end, the set $R$ contains all vertices that are reachable from $s$. The idea behind the algorithm is to propagate a "wave" of vertices across the graph, computing at each stage an addition to the transitive closure. Specifically, the wave starts as the set consisting of the single source vertex, and the addition to the initially empty transitive closure consists of the vertices that are reachable in one step from the vertices in the wave. The wave is updated in each iteration by removing the vertices that already appear in the closure. The original proposal for the $\delta$-wavefront procedure in [QHK91] assumed a matrix representation of the graph $G$. Our modification incurs a constant time overhead but is expected to yield a lower average-case complexity since it takes advantage of the graph’s sparsity.

A close examination of the algorithm reveals that the algorithm can be quite efficient for solving the reachability problem for a given set of source nodes. The worst-case complexity of solving a single-source transitive closure problem is $O(|E|)$. The algorithm has sublinear time complexity on the average. The complexity measure we use is the number of edge
Procedure $\delta$ - wavefront($S$)

begin

$\text{wave} := S; \text{R} := S$;
while ($\text{wave} \neq \emptyset$) {

$\text{temp} := \text{wave}$;
for each $v$ in $\text{wave}$ {

for each $u$ in the adjacency list of $v$

$\text{temp} := \text{temp} \cup \{u\}$;

$\text{temp} := \text{temp} \setminus \{v\}$;

$\text{wave} := \text{temp} \setminus \text{R}$;

$\text{R} := \text{wave} \cup \text{R}$;
}

$\text{R} := \text{R} \setminus \text{S}$;
end

Figure 4.6: The $\delta$-wavefront Algorithm

traversals. If the full transitive closure of the graph has to be computed, then one can either solve $|V_R|$ single-source transitive closure problems, or one reachability problem with the set of source nodes initiated to the set of all nodes of the graph. In both cases, the worst-case complexity is $O(|V_R| \cdot |E|)$, whereas on the average the time complexity is expected to be substantially smaller. In the case of dependence graphs, which are sparse, the average case complexity is sublinear on the number of edges, since not all edges will have to be traversed. Moreover, the complexity remains $O(|E|)$ when the entire set of nodes is used as the set of source nodes, since the algorithm does not recompute anew the transitive closure of nodes that have been visited in previous iterations. This property is of particular interest since it provides for an efficient way to compute a set of implicit updates given a number of source nodes. In the case of constraint enforcement, the source nodes are the rule nodes that match an explicit update. It also allows for the incremental computation of transitive closure for additions/deletions of rules/constraints.

In this section we only dealt with the time complexity of computing the transitive closure of the dependence graph. The space and I/O complexity are also major issues that have to be taken into account. The $\delta$ - wavefront algorithm has been shown to have non-linear I/O complexity. These issues currently remain as future research considerations.
**Algorithm:** Minimize Intervals \((G,S)\)

\[
\text{begin}
\text{For every node } v \in S \{ \\
\quad Temp := \emptyset \\
\quad \text{For every node } u \in V_R \text{ such that } (u,v) \in E \{ \\
\quad \quad u.T := (v.T \ast u.T) \\
\quad \quad newSF := u.SF \lor \bigwedge_{t_i \in gtv(u,SF)} \text{during}(t_i,u.T) \\
\quad \quad u.SF := temp\_simp(newSF) \\
\quad \quad Temp := Temp \cup \{u\} \}
\}
\quad S := Temp \\
\quad \text{Minimize Intervals } (G,S)
\text{end}
\]

Figure 4.7: Interval Minimization

### 4.3 Optimizee

We now present additional optimizations that take into account the temporal properties of constraints and rules. They take place after the graph’s construction and aim at producing more efficiently evaluable temporal formulae.

The first optimization step replaces the temporal intervals of the PSSs that comprise the graph nodes by smaller intervals. Algorithm *Minimize Intervals* (see figure 4.7) is applied after the construction of the dependency graph is complete and whenever the dependence graph structure is modified by the insertion of a new constraint or rule. The algorithm starts with an initial set of nodes and, for every node \(v\) in this initial set, it replaces the history time intervals of each of its incident nodes with the time interval that results from computing the intersection of the time interval of \(v\) with that of its incident node. The algorithm proceeds in a breadth-first fashion until no more replacements can take place. In this manner, the validity time intervals of the formulae whose satisfaction has to be determined at run-time are minimized. Since the history time intervals of the constraints and rules are used to constrain the temporal variables of their respective expressions, the minimized intervals must be used in their place. As was shown in section 4.1.1, the time intervals originally associated with a rule or constraint are factored in the rule or constraint expression using a *during* relationship that captures the semantics of temporal validity. When the minimized
interval of a PSS is computed, the new interval must also be factored into the simplified form. The during relationship that introduced the original interval in the temporal formula expressing the constraint or rule may not appear explicitly due to simplification step 4 that replaces conjunctions of temporal relationships by a new relationship. Hence, a new during relationship introducing the minimized interval must be conjoined with the simplified form. Additional simplifications may be carried out if applicable. In the presentation of the algorithm we assume that each node is represented as a PSS and we use the “.” (dot) operator to refer to its components. The call to temp simp symbolizes the application of step 4 of definition 4.1.6. The function qtv returns the quantified temporal variables of a simplified form.

Initially, the algorithm is applied to a dependence graph $G$ with the set of nodes $V_I$ corresponding to simplified forms of integrity constraints as the set of source nodes $S$. When $G$ is modified by the insertion of a new integrity constraint, the set of source nodes is the set comprising the PSSs of the new constraint. If a new rule is inserted, then the set of source nodes is the set of successor nodes of all the PSSs of the newly inserted rule. Note that we do not need to apply the algorithm for the case of deletions of constraints or rules since deletion can only disconnect graph nodes. Since a dependence graph may be cyclic, special attention must be paid to the graph’s strongly connected components which can be identified at compile time. The algorithm can be easily modified to take the graph’s strongly connected components into account.

We argue that the optimization steps presented above can yield considerable savings in evaluating simplified forms at run-time by minimizing the intervals over which the evaluation of formulae must take place and by, possibly, carrying out additional temporal simplifications. The transformations carried out preserve the satisfaction of constraints since only the simplified forms of their relevant rules are modified. Moreover, the derivation of the actual implicit updates becomes simpler since the formulae that have to be verified have been simplified. Lemma 4.3.1 and theorem 4.3.1 establish the correctness of the algorithm.

**Lemma 4.3.1** Let $SF$ be the simplified form of a rule whose associated time interval is restricted by algorithm Minimize Intervals and let $SF'$ be the new simplified form. Then, the satisfaction of $SF$ in the knowledge base implied the satisfaction of $SF'$.

**Proof:** The new simplified forms are obtained by conjoining one or more temporal
predicates. The result follows from the definition of constraint satisfaction (definition 3.4.5) and the soundness of temporal simplification (theorem 4.1.1).

\[\square\]

**Theorem 4.3.1** Algorithm *Minimize Intervals* is correct: if the violation of a constraint \( C \) can be determined using the dependence graph \( G \), then it can also be determined using the graph resulting from applying *Minimize Intervals* to \( G \).

**Proof:** Follows immediately from lemma 4.3.1.

\[\square\]

### 4.4 Evaluation Phase

The evaluation of the simplified constraints takes place after the actual updates occur. The actual values specified in the updates are substituted for the parameters in the parameterized simplified forms of the constraints affected by the update. Recall that the definition of an affecting update takes into account the temporal validity of constraints. Then, the instantiated simplified constraints must be verified against the knowledge base.

In this section we describe the evaluation phase of our method. We first discuss how the dependence graph generated in the compilation phase is used to check the integrity constraints at the time of update. Then we describe how this graph is incrementally maintained as the set of integrity constraints and deductive rules is modified.

#### 4.4.1 Using Dependence Graphs for Constraint Checking

The implicit changes to the knowledge base caused by the (explicit) updates must also be derived as explained in the previous sections. Hence, the parameterized simplified rules whose literals match the update literal are instantiated, their conclusions are deduced and treated as normal updates, insertions or deletions depending on the sign of the matching literals. The deduced updates can cause constraint violations and the derivation of further implicit changes. The dependence graph constructed during the compilation phase allows us to avoid recomputing the (non-ground) implicit updates. Instead, implicit updates are found as literals on paths in the dependence graph starting from a literal whose extension is affected by an explicit update. This eliminates the cost of performing unification and of evaluating temporal expressions since the dependence relation, the transitive closure of direct dependence has been computed and stored. The literals on the paths of the
dependence graph are potential updates, since they are not instantiated\(^4\) and it is not known at compile time whether the bodies of the rules in which they participate is satisfied. At update time the actual values specified in the update are substituted for the parameters. A potential update is an actual update if the body of its rule is satisfied in the new state of the knowledge base.

The dependence graph reflects both the logical and temporal interdependence of rules and constraints. To check if an update \( U \) affects an integrity constraint, we first locate all literals \( L_i \) in the dependence graph that unify with the update. The set of integrity constraints that may be violated are those which have at least one node on a path initiating at a literal \( L_i \). As mentioned earlier, the dependence graph transitive closure can be precomputed at compile time. Hence, at update time reachability information does not need to be recomputed. It suffices to verify that the potential implicit updates are actual updates by instantiating the literals of the implicit updates and evaluating the bodies of rules in which they belong. Concerned classes provide a short-cut in locating the literals unifying with the update. It suffices to derive the concerned class \( C \) for the update literal and seek for the parameterized simplified forms that agree on their concerned class component with \( C \). The concerned class component of the compiled forms can be used to provide a grouping of nodes with respect to their concerned classes. The dependence graph can be indexed with respect to concerned classes of nodes. Searching for affected rules or constraints will involve accessing the nodes in the graph can be performed efficiently.

**Example 4.4.1** In figure 4.2 an update on literal \( \text{univ} \) might cause a violation of constraint IC2 since one of IC2's literal's lies on a path with source from \( \text{univ} \).

The evaluation process is summarized in the algorithm of figure 4.8. A word on the notation used should be said first. For each simplified form corresponding to a node \( v \) of the dependence graph \( l(v) \) returns the characteristic literal of the rule or constraint. Moreover, for each node \( v \in V_R \), \( S_v \) denotes its successor set, whereas for each node \( v \), \( P(v) \) denotes the set of \( v \)'s direct predecessors in the graph. Set \( A(v) \) denotes the set of direct predecessors of \( v \). Obviously, \( A(v) \subseteq S(v) \). The set \( Path \) obtained by intersecting a successor set \( S(u) \) and a predecessor set \( P(v) \) restricts search to the rules in a path from \( u \) to \( v \). It has to be noted that the set \( S(v) \) does not have to be maintained explicitly for

\(^4\)No values for the parameters of the literals are known at this point.
Algorithm: Evaluation

begin

for each update $U$ of a transaction $T$ {
    for each $v$ in $V_I$ s.t. $l(v)$ unifies with $U$ {
        instantiate the parameters with the actual values in $U$
        evaluate the constraint against the KB
        if false reject $T$
    }
    for each $u$ in $V_R$ s.t. $l(u)$ unifies with $U$
        if exists $v$ in $V_I$ reachable from $u$ {
            instantiate the parameters with the actual values in $U$
            evaluate rule $u$
            if true {
                $Path := P(v) \cap S(u)$
                repeat {
                    evaluate each rule $w$ in $Path$
                    if false
                        reject
                } $Path := P(w) \cap S(u)$
                until ($Path = \emptyset$)
            evaluate constraint $v$
            if false reject
        }
    }
}

end

Figure 4.8: Evaluation phase algorithm
each \( v \). It can be obtained from the transitive closure in \( O(|V|) \) worst-case time.

As far as the complexity of the algorithm is concerned, the iteration over the constraint nodes requires \( O(|V_R|) \) time. The iteration over the predecessor nodes requires, in the worst case, \( O(|E|) \) time and so the iteration over the rule nodes has a worst-case time complexity of \( O(|V_R| \cdot |E|) \). Hence, the total time required for each update is \( O(|V_R| \cdot |E|) \). It is expected that the average time complexity will be significantly smaller since not all edges of the graph will be relevant to a single chain of inference from an explicit update to one or more constraints, neither will all rule nodes match the explicit update.

Finally, a few words should be said about the verification method for the instantiated simplified constraints. It has to be shown that each of these constraints is a theorem of the knowledge base. Most integrity maintenance methods rely in the use of some resolution theorem proving procedure (e.g., SLDNF [Llo87]) for the task of verifying constraints. However, in the presence of explicit temporal predicates in the simplified constraints, standard SLDNF-resolution is not sufficient. The use of negation as failure is problematic in the presence of temporal information which may introduce incomplete knowledge. A special purpose temporal reasoner must be employed in conjunction with SLDNF. Hence a hybrid theorem prover needs to be defined as, e.g., in [MS90]. One possibility is to combine existing methods via Theory Resolution [Sti85] or via Universal Attachment [Mye91]. It remains as a current research objective to devise an efficient hybrid reasoner for the evaluation of constraints.

### 4.4.2 Generating Simplified Forms for Transactions

We now turn to the derivation of simplified forms of constraints with respect to arbitrary transactions. This derivation takes place at run-time when the actual transaction is specified, but uses the simplified forms that were generated with respect to the individual updates at compile-time.

As far as the integrity of a knowledge base is concerned, transactions are considered to be atomic. Hence, the effect of a transaction made up of any subset of the affecting updates for a particular integrity constraint is the same independently of the order in which the updates occur in the transaction.\(^5\) This is easy to see, if we consider that it is known that the constraint is not violated in the state prior to the transaction’s execution and we apply

\(^5\)Of course, the order of updates is important for producing the result intended by the transaction specifier.
the simplification steps for all updates occurring in the transaction simultaneously. The
above observation allows us to use the simplified forms generated for individual updates
in order to derive one simplified form that suffices to be evaluated in order to verify the
satisfaction of a constraint with respect to a transaction containing an arbitrary sequence
of the individual updates.

**Definition 4.4.1** Let \( X = \{ U_1, \ldots, U_m \} \) be a transaction affecting constraint \( C \), and let
\( U_i \) be an update in \( X \) that affects \( C \). Let \( X' \) be the set of updates in \( X \) that and affect \( C \)
and do not include \( U_i \). Then, the simplified form of \( C \) with respect to \( X \), is obtained by
applying all updates in \( X' \) simultaneously to the PSS \((L, Params, CC, T, T', SF)\) of \( C \) that
is such that \( L \) is unifiable with \( U_i \).

**Example 4.4.2** Consider the constraint
\[
\forall \overline{x}, \overline{y} \ (P(\overline{x}) \land \neg Q(\overline{x}, \overline{y})) \lor (R(\overline{x}, \overline{y}) \land \neg P(\overline{x})) \lor S(\overline{x}, \overline{y})
\]
and the transaction \( X = \{ \neg P(\overline{x}), \neg R(\overline{x}, \overline{y}) \} \). If the updates were treated independently,
two formulae would need to be verified, namely \( \forall \overline{y} \ (R(\overline{x}, \overline{y}) \lor S(\overline{x}, y)) \) and \( (P(\overline{x}) \land \neg Q(\overline{x}, \overline{y})) \lor S(\overline{x}, \overline{y}) \). This should be contrasted with the formula \( S(\overline{x}, \overline{y}) \) that is the
simplified form of \( C \) that suffices to be verified when \( C \) is simplified with respect to \( X \). \( \square \)

The following theorem establishes the soundness of the simplification method with re-
spect to transactions. Its proof uses the result of theorem 4.1.1.

**Theorem 4.4.1 (Soundness)** A temporal constraint \( C \), known not to be violated in the
state prior to the execution of an affecting transaction, is violated in the state resulting
from the transaction’s execution if the formula produced after applying simultaneously all
updates affecting the constraint is.

**Proof:** Let \( X = \{ U_1, \ldots, U_m \} \) be a transaction affecting constraint \( C \), and let \( U_i \) be
an update in \( X \) that affects \( C \). Let \( X' = \{ U_i, \ldots, U_k \}, k \leq m \) be the set of updates in \( X \)
that affect \( C \). Without loss of generality we impose an arbitrary order in this set obtaining
the sequence \( [U_i, \ldots, U_k] \). Theorem 4.1.1 established the soundness of the simplification
method with respect to the individual updates that may affect a constraint. We will prove
the theorem by induction on the length of \( X' \).

The case where the length of \( X' \) is 0, is trivial. The transaction does not affect the
constraint and the result holds vacuously. In the case where, \( X' \) contains a single update,
the result holds by virtue of theorem 4.1.1. Let $C_{U_j}$ denote the simplified form generated by applying the update indexed $j$ in the sequence to the form generated by the simplification of $C$ with respect to updates $U_i, \ldots, U_{j-1}$. Let us assume that the result is true for the transaction $X'' = \{U_i, \ldots, U_{k-1}\}$. Then, we will have $C \Rightarrow C_{U_{k-1}}$. The effect of transaction $X'$ on the constraint $C$ is obtained, if each update in the sequence is applied to the simplified form generated according to the one preceding it in $X'$. Because of the soundness of the simplification process, we will have: $C_{U_{k-1}} \Rightarrow C_{U_k}$ and, thus, $C \Rightarrow C_{U_k}$.

\[ \square \]

4.4.3 Updates of Integrity Constraints and Deductive Rules

The situation is more complicated in the case of transactions that insert or delete constraints and rules. The previously defined compilation method needs to be extended in order to provide for an incremental compilation of the newly inserted constraints and rules without having to reconsider integrity constraints and deductive rules that have already been compiled. This means that, in both the case of insertion and deletion of integrity constraints or deductive rules, the dependence graph must be modified minimally.

The organization of simplified forms in a dependence graph permits the incremental compilation of newly inserted constraints and rules without having to reconsider those that have already been compiled. The dependence graph definition and construction permits such a minimal incremental modification: only direct dependence between rule conclusion and constraint literals has to be determined. The number of new nodes to be added is of the order of the number of literals occurring in the newly added deductive rules or constraints. When a deductive rule is added, this number is equal to that of the existing parameterized constraint forms that directly depend on the conclusion of the newly added rule. When a new constraint is added, nodes corresponding to its parameterized forms are introduced and connected to the rule nodes on which they directly depend. A similar removal of nodes and edges takes place when rules or constraints are deleted. Note that only direct and temporal dependence needs to be determined when the dependence graph is modified. Dependence is then determined by the paths created from the previously existing paths in the dependence graph and the newly added edges. The transitive closure of the direct dependence relation has been computed in the knowledge base compilation phase. Updates of integrity constraints and deductive rules cause changes to the dependence which must be reflected in the computed transitive closure. These changes can be computed along with the
incremental changes to the dependence graph. In the algorithms to follow, the maintenance of the transitive closure is carried out differently, depending on the update operation. We present algorithms that modify the dependence graph minimally and incrementally and characterize the complexity of the modification operations. The incremental changes to the transitive closure and the complexity of the associated operations will be presented in a subsequent section. For the following discussion, we will assume that the transitive closure of the dependence graph is represented in a form that permits checking the reachability between two graph vertices in $O(1)$ time.

**Updates of Integrity Constraints**

The traditional semantics attributed to the insertion of an integrity constraint specifies that a new integrity constraint must be completely evaluated against the knowledge base. If found true it is accepted; otherwise it is rejected. An alternative semantics for the insertion of integrity constraints gives preference to constraints over base facts. Adopting this view would require to accept the constraint without evaluating it. In case its evaluation after the occurrence of an affecting update shows an inconsistency, a sequence of updates can be initiated so that no inconsistency occurs in the resulting state. Techniques employed for proving finite satisfiability of constraints [BM87] can be used for deriving a sufficient sequence of updates\(^6\) in a parallel version of the knowledge base extension. The same techniques can be employed for integrity recovery. This process can be automated by extending the approach followed in [CW90]. For the time being we assume the traditional semantics.

When an integrity constraint is inserted, the constraint is evaluated and only if found true, it is transformed into a set of parameterized simplified forms, one for each of its literals. These forms are added as nodes to the dependence graph and, in case there exist deductive rules already in the graph on which the constraint directly depends, the appropriate edges are added and labeled. Hence, compilation and evaluation have to be interleaved in the case of new constraint definitions.

As far as the deletion of integrity constraints is concerned, it cannot cause an inconsistency. Hence, in the case of deletion of an integrity constraint, only reorganization of the dependence graph is required in order to delete the edges specifying the dependence

---

\(^6\)More than one such sequences are in general possible.
**Algorithm:** Insert Constraint IC

**begin**

Evaluate IC over KB

**if** true

**for each** literal $L$ in IC {
    generate a parameterized simplified form for $L$
    create a node $n_L$ in $V$
    **for each** parameterized rule $n_r$ in $V$
        **if** $L$ directly depends on $n_r$ {
            create an edge $e$ from $n_r$ to $n_L$ in $E$
            **if** $n_L$ and $n_r$ are temporally dependent
                label $e$
        }
    update $TC$
}
**else**
    reject update
**end**

**Figure 4.9:** Integrity Constraint Insertion

relationships between deductive rules and the deleted constraint. In the method proposed in [JK90], if this process results in isolated nodes representing parameterized simplified rules on which the deleted constraint previously depended, then these nodes have to be removed as well. Alternatively, in our approach, these nodes are not removed in order to avoid their recompilation in case future updates introduce integrity constraints depending on these rules.

The treatment of knowledge base updates by insertion/deletion of integrity constraints is summarized in the algorithms of figures 4.9 and 4.10. It is assumed that the dependence graph $G$ is represented by a set of vertices $V$ and a set of edges $E$, created as discussed in previous sections. $TC$ denotes the stored transitive closure of the dependence graph.

The insertion of an integrity constraint in the dependence graph has the worst-case complexity $O(|V_d|)$, since the newly introduced vertices have to be connected with as many deductive rule nodes as the number of rules with conclusion literal matching the constraint
**Algorithm:** Delete Constraint IC

begin
    for each parameterized simplified form of IC
        delete corresponding node from V
        delete all adjacent edges from E
        update TC
end

Figure 4.10: Integrity Constraint Deletion

literal. On the average, it is expected that the complexity of dependence graph modification will be much smaller, since only a subset of the deductive rules will match the constraint literals. To precisely characterize the cost of insertion, let us define a function $F : L \rightarrow [0, 1]$, which returns, for each literal $l$, the frequency of its occurrences in rule heads.\footnote{This type of information can be available after knowledge base compilation and can be maintained incrementally after modifications.} $\alpha$ has been defined in section 4.2 as the average number of literals per rule body or constraint. Hence, on the average the number of edge additions required for the insertion of an integrity constraint will be equal to $\sum_{l \in IC} |V_R| \cdot F(l) \cdot \alpha$. We define this quantity as $Cost_{IC_{insert}}$.

From the above description of the algorithm for deletion of constraints, it can be easily seen that the worst case time complexity for removing a vertex of one simplified form is $O(|E|)$, since, in the worst case, all other vertices are connected to the vertex in question. On the average, the number of edges to be removed is expected to be much smaller. In total the deletion process of an integrity constraint is linear in the number of deductive rules whose conclusion literal matches some literal of the constraint. As was the case in insertion, the average number of edges that have to be added is equal to $\sum_{l \in IC} |V_R| \cdot F(l) \cdot \alpha$. We define this quantity as $Cost_{IC_{delete}}$.

**Updates of Deductive Rules**

The case of updates of deductive rules appears to be more complicated. The reason is that the insertion (deletion) of a deductive rule may cause implicit changes to the knowledge base which are also candidates for violating integrity.

When a new deductive rule is inserted, the dependence relationships of the newly in-
serted rule must be determined and represented in the dependence graph. Hence, it must be checked whether there exist parameterized simplified constraints or rules with literals that unify with the rule’s conclusion literal. Temporal dependence has to be determined as well. In that case, the conclusions of the rule must be derived and checked for possible constraint violations (i.e., they must be treated as normal insertions). These implicit updates may trigger subsequent implicit updates if there exist already compiled rules with body literals that unify with the inserted rule’s conclusion literal. This information is available since the transitive closure of the graph has been computed and updated after each insertion/deletion of rules and constraints.\(^8\) One only has to check if a constraint node appears as a successor (not necessarily direct) of the deductive rule nodes that match the inserted rule’s conclusion literal. If no violation of constraints arises, and if there does exist a literal of a rule or constraint that unifies with the rule’s conclusion, then the rule is transformed into a set of parameterized simplified forms, one for each literal in the rule’s antecedent, and inserted in the dependence graph.

The above description of the rule insertion procedure is summarized in the algorithm of figure 4.11. It assumes that the computed transitive closure is available in TC. Note the savings incurred by the presence of the computed transitive closure: only matching literals have to be located. The information about implicit updates needed to guarantee integrity is already available. Moreover the search for matching literals is done only once and is used for creating the new vertices and edges in case no violation of integrity occurs and the rule is compiled into its simplified forms. The problem of predicate indexing for efficient matching at run-time remains for the time being a research consideration. Concerned classes however provide a way for indexing the graph nodes, by grouping together nodes that have the same concerned class.

The complexity of deductive rule insertion is dominated by that of the first for loop of the algorithm. The worst case complexity of the loop is \(O(|V_R| \times |V|)\), since a vertex can have at most \(|V|\) successors in the computed transitive closure. However, as was noted in a previous section, the dependence graph is sparse, thus making \(|V|\) much larger than \(|E|\). In the above algorithm, the number of successors is at most equal to \(|E|\) which means that the algorithm’s worst-case time complexity is \(O(|V_R| \times |E|)\).

Let us analyze a bit further the costs of finding the possibly affected constraints and

\(^8\)The incremental update of the transitive closure is examined in the next section.
**Algorithm:** Insert Ded. Rule $DR(B \rightarrow H)$

begin

$C := \emptyset$;

for each $v$ in $V_R$ matching $H$

    for each successor $u$ of $v$ in $TC$

        if $u \in V_I$

            $C := C \cup \{u\}$;

    for each $v$ in $V_I$ matching $H$

        $C := C \cup \{v\}$;

    for each $e$ in $C$

        check corresponding constraint

if no violation occurs {

    compile $DR$

    update $TC$ }

else

    reject update

end

Figure 4.11: Deductive Rule Insertion
modifying the dependence graph by adding the nodes corresponding to the new rule and the appropriate dependence edges. The number of graph vertices corresponding to deductive rules matching the rule head $H$ will be equal to $|V_R| * F(H)$. Moreover each such node can have at most $|E_{RC}|$ successors in the transitive closure. Again, in the worst case, all these successors will be in $V_I$ which makes the cost of computing possibly affected constraints equal to $|V_R| * F(H) * |E_{RC}|$. Similarly, there will be at most $|V_I| * F(H)$ nodes matching $H$. Hence, the total maximum cost of identifying the possibly affected constraints is $|V_R| * F(H) * |E_{RC}| + |V_I| * F(H)$. Let $c = \frac{|V_R|}{|V_I|}$ denote the ratio of rule nodes over constraint nodes on the graph. Then $\frac{|E_{BB}|}{|E_{RC}|} \approx c$ and the above cost becomes $|V_I| * F(H) * (1 + |E| \times \frac{c}{1+c})$. Let also $r = \frac{|V_I|}{|E|}$ denote the ratio of nodes over edges in the dependence graph. Then the cost can be written as $\frac{r}{1+c} * F(H) * (1 + c + c * |E|) * |E|$. Finally, the modification of the dependence graph requires that edges are added from rule nodes in the dependence graph to the newly added nodes. This additional cost is derived as in the case of a constraint insertion and is on the average equal to $\sum_{l \in B} |V_R| * F(l) * \alpha$.

In total, $\text{Cost}_{DR_{\text{insert}}} = \frac{r}{1+c} * F(H) * (1 + c + c * |E|) * |E| + \frac{r}{1+c} * |E| * \sum_{l \in B} F(l) = \frac{r}{1+c} * |E| * (\alpha * \sum_{l \in B} F(l) + F(H) * (1 + \frac{c}{1+c} * |E|))$

In the case that an already compiled rule is to be deleted, if there exist deductive rules or integrity constraints with literals matching the rule’s negated conclusion, then the literals deducible with this rule must be treated as normal deletions. If these deletions do not cause any integrity violation, then the parameterized forms of the rule must be deleted along with all their incident edges. The algorithm for deductive rule deletion is very similar to that for the insertion of deductive rules and is depicted in figure 4.12. Again, the computed transitive closure provides the information of whether it is possible that an implicit deletion caused by the rule’s deletion violate an integrity constraint. If no violation occurs all simplified forms of the rule are deleted and the transitive closure is updated.

As was the case for deductive rule insertion, deductive rule deletion requires worst-case time of $O(|V_R| * |E|)$, under the same assumptions. A similar analysis of the average cost is applicable in the case of deletions: $F(H)$ must be substituted by $F(-H)$ in the cost formulas. The edge removal process has a worst case complexity of $O(|E|)$ whereas in the average case the number of edges to be removed is derived similarly to the case of insertion.

Hence, $\text{Cost}_{DR_{\text{delete}}} = \frac{r}{1+c} * F(-H) * (1 + c + c * |E|) * |E| + \frac{r}{1+c} * |E| * \sum_{l \in B} F(l) = \frac{r}{1+c} * |E| * (\alpha * \sum_{l \in B} F(l) + F(-H) * (1 + \frac{c}{1+c} * |E|))$
Algorithm: Delete Ded. Rule DR(B → H)

begin
  \( C := \emptyset \);

  for each \( v \) in \( V_R \) matching \( \neg H \)
  
  \( \text{for each successor } u \text{ of } v \text{ in } TC \)
  
  \( \text{if } u \in V_I \)
  
  \( C := C \cup \{ u \} \);

  for each \( v \) in \( V_I \) matching \( \neg H \)
  
  \( C := C \cup \{ v \} \);

  for each \( e \) in \( C \)
  
  check corresponding constraint

  if true {
    remove every compiled form of \( DR \) and all incident edges
    update \( TC \)
  }

  else
    reject update

end

Figure 4.12: Deductive Rule Deletion
From the above discussion, it can be seen that an adequate treatment of updates of
deductive rules and integrity constraints requires interleaving of compilation and evaluation.
This results in an increase of the complexity of the method and may also degrade
performance. This cost however is traded for the cost of rolling-back the knowledge base
in its previous state in case a violation is discovered. In our method, as well as in the one
proposed in [JK90], violations of constraints are discovered as soon as possible.

**Incremental Modification of Transitive Closure**

In this section, we examine the maintenance of the transitive closure of the dependence
graph. The transitive closure is modified every time an integrity constraint or deductive rule
is inserted or deleted. This corresponds to the addition or deletion of vertices and edges in
the dependence graph. Since only the insertion or removal of edges affects transitive closure
we will analyze the case of insertion or removal of a single edge in the dependence graph.
Recall that, when a new vertex is introduced, it is connected with all matching or affecting
vertices already in the dependence graph. Also, when a vertex is removed, all adjacent
edges are removed as well. Hence, it is the insertion or removal of edges that causes changes
to the transitive closure.

A number of algorithms have been proposed for solving the problem of on-line mainte-
nance of transitive closure for graphs. The method proposed in [Ita88], an improvement on
the one proposed in [IK83], is applicable to acyclic graphs only. It is readily applicable to
the condensed form of the dependence graph provided an appropriate indexing of the graph
vertices is performed. It is not applicable however for general, possibly cyclic, graphs. In
this section we propose a method for computing the incremental changes to the computed
transitive closure for the dependence graph structure.

Let us consider edge insertion first. Insertion of an edge between vertices $u$ and $v$ of
the dependence graph makes vertex $v$ and everyone of its successors reachable from $u$ and
every vertex from which $u$ is reachable. The reachability between any other pair of vertices
is not affected. For every pair of nodes $(x, y)$ such that $y$ is reachable from $x$ before the
insertion, $y$ remains reachable from $x$ after the insertion. No special care should be taken
for vertices in cycles. This procedure is described in the algorithm *Insert Edge* of figure
4.13. Its worst-case complexity is quadratic on the number of edges for the case of sparse
graphs.
Algorithm: Insert Edge \((u, v)\)

begin
\[ S'(u) := S(u) \cup \{v\} \cup S(v) \]
for each \(x\) s.t. \((x, u) \in TC\) and \((x, v) \notin TC\) {
insert \((x, v)\) in \(TC\)
for each \(y\) in \((S'(u) - S(u))\)
if \((x, y) \notin TC\)
insert \((x, y)\) in \(TC\) }
\[ S(u) := S'(u) \]
end

Figure 4.13: Edge Insertion

Let us introduce some notation first. For each node \(v\), \(S(v)\) denotes the set of successor nodes of \(v\) in the transitive closure. This set does not have to be explicitly maintained for each vertex, but can be retrieved from the transitive closure matrix in time \(O(|V|)\) for dense graphs, and in time \(O(|E|)\) for sparse graphs, since in a graph with \(|E| \leq |V|\) edges, one node can have at most \(|E|\) successors. Finally, \(P(v)\) denotes the set of direct predecessors of \(v\) in the dependence graph. The cost of constructing \(P(v)\) is in the worst-case \(O(|E|)\).

The algorithm maintains correctly the transitive closure of a directed graph in case of an edge insertion even in the case that the graph contains cycles and the inserted edge is a cycle forming one. The situation however is simpler in the case of the dependence graph because of its particular structure. We will distinguish two cases: when the inserted edge is between a rule and a constraint vertex and when the inserted edge is between two rule vertices. Note that we never insert edges that connect constraint vertices.

The cost of computing the set \(S'(u)\) is at most equal to \(2 \times |E|\) since a node can have at most \(|E|\) successors in the graph. Similarly, the cost of computing \(P(v)\), is equal to \(\max(|E_{RC}|, |E_{RR}|)\) depending on whether \(v\) is a rule or a constraint node. The main cost is incurred by the iteration over the nodes from which \(u\) is reachable. Since \(u\) cannot be a constraint node, there can be at most \(|E_{RR}|\) such nodes. The inner iteration however is at most equal to \(|E|\). Hence the total cost is \(|E_{RR}| \times |E|\) or \(O(|E_{RR}|^2)\) with a coefficient much smaller than 1. Below we examine the cost of computing the new reachability information for the cases where the inserted edge is between two rule or a rule and a constraint node.
When an edge \((r, e)\) between a rule and a constraint vertex is inserted, there is no possibility of cycle formation since constraint vertices have no outgoing edges. In that case, only the successor sets of \(r\) and all its predecessors in the transitive closure have to be updated, as well as, the set of direct predecessors of \(e\). Specifically, the cost of computing the new successor set for \(r\) is at most \(|E|\), whereas the cost of computing the direct predecessors of \(e\) is at most \(|E_{RC}|\) since a constraint node cannot have any predecessors among constraint nodes. The iteration over the nodes from which \(r\) is reachable before the edge addition, has a worst-case cost of \(|E_{RR}|\). If however, there exist cycles or multiple non-disjoint paths to \(r\), the number of such nodes will be less than \(|E_{RR}|\). The inner iteration, iteration only has to consider one node, since the only addition to the successor set of \(r\) is the node \(e\). In total, the worst-case cost is \(|E|\), whereas on the average, the cost is expected to be sublinear in \(|E_{RR}|\).

In the latter case, of insertion of an edge between two rule nodes \(r_1, r_2\) that is, the cost can be as much as \(|E_{RR}|*|E|\) due to the fact that the iteration over the nodes that can reach \(r_1\) can involve as many as \(E_{RR}\) nodes, whereas the inner iteration can involve as many as \(|E|\) nodes. Again, on the average we can expect a much smaller cost since the dependence graph is sparse, and certainly not complete, i.e., not all nodes have \(|E|\) successors or predecessors.

Let us now examine, how the transitive closure can be maintained when deletions of edges take place. When an edge \((u, v)\) is removed, if the only path from \(u\) to \(v\) is via the directed edge \((u, v)\), then \((u, v)\) must be removed from the transitive closure. All nodes that could previously reach \(v\) but not \(u\), remain unaffected since they can still reach \(v\). The same holds for all nodes that could previously reach \(v\) via a path that did not include the edge \((u, v)\). For any node \(x\) whose only path to \(v\) is via \(u\), \((x, v)\) must be removed from the transitive closure. Also, for every successor \(y\) of \(v\) and for every node \(x\) that can reach \(u\), if the only path from \(x\) to \(y\) is via \((u, v)\), then \((x, y)\) must be removed from the transitive closure. Algorithm Delete Edge of figure 4.14 describes the procedure for edge deletion sketched above.

The worst-case complexity of the edge deletion process is quadratic in the number of edges between rule nodes since constraint nodes do not have successors. Hence, as was the case in edge insertion, edge deletion requires a maximum of \(O(|E_{RR}|^2)\) changes in the transitive closure of the graph. Below, we analyze the special cases of removal of an edge between a rule and a constraint node or between two rule nodes.
Algorithm: Delete Edge $(u, v)$

begin
    compute $S(u), S(v), P(v)$
    $X := \emptyset$
    for each $x \neq v$ s.t. $(x, v) \in TC$ and $(x, u) \in TC$
        if $S(x) \cap P(v) = \{u\}$
            $X := X \cup \{x\}$
            $S(x) := S(x) - \{v\}$
            remove $(x, v)$ from $TC$
        $P(v) := P(v) - \{u\}$
    for each $y$ in $S(v)$
        for each $x$ in $X$
            if $(path(x, y) = 0)$
                remove $(x, y)$ from $TC$
end

Figure 4.14: Edge Deletion

If an edge $(r, c)$ between a rule and a constraint node is to be removed, then the cost of computing $P(c)$ is at most $|E_{RC}|$. There can be at most $|E|$ nodes that can reach both $r$ and $c$ and at most $|E|$ successors in each such node. The case is similar when an edge has to be removed. The difference is in the computation of the successors of the destination node, but the complexity remains quadratic in $|E|$ in the worst-case. Again, the average case performance is expected to be much better, in fact sublinear on $|E|$.

The description of the maintenance procedures reveals that the presence of cycles complicates the on-line transitive closure computation: the worst-cases in the performance of the algorithms occurs when the edge to be inserted/removed forms/belonged to a cycle in the graph. We expect to be able to perform better in the average case because of the structure of the dependence graph. Specifically, no inference path can involve a constraint node unless it is the final node in the path. Hence, if the edge to be removed is between a rule and a constraint node only paths formed by edges connecting rule nodes have to be searched. In this case the number of edges to be examined is $|E_{RC}|$ rather than $|E|$. The graphs in figures 4.15, 4.16 compare the cost of incremental transitive closure computation
Figure 4.15: Performance of Incremental T.C. Computation (Edge Insertion)

with that of recomputing $TC$ from scratch in the cases of edge insertion and removal and for random sparse dependence graphs. The graphs show that on-line maintenance of the implicit update information can be carried out efficiently, with a cost as low as $0.1 \times |E|$ on the average.
Figure 4.16: Performance of Incremental T.C. Computation (Edge Deletion)
Chapter 5

Transaction Modification

In the previous chapter, we presented a compile-time simplification method for temporal deductive knowledge bases, where integrity constraints, specified in a many-sorted temporal assertion language [MBJK90], are specialized and simplified with respect to the anticipated updates. In this chapter, we undertake a dual approach, namely compile-time transaction modification in the context of knowledge bases containing (temporal) integrity constraints and deductive rules. In particular, we examine how transactions specified by means of precondition-postcondition pairs, can be modified so that the constraints they - directly or indirectly - affect are guaranteed not to be violated in the state resulting from transaction execution. Specifying database transactions by means of pre- and post-conditions, permits the database designer to express what the effect of transactions should be and not how this effect should be accomplished. Database transaction specifications, should be given in a formal notation that possesses both notational suitability and the capacity of supporting formal treatment. The formal specification language should be accompanied by the appropriate machinery for proving properties of specifications. We choose first-order (temporal) logic as the language for expressing transaction pre/post-conditions, constraints and rules. We address the problem of proving integrity maintenance by relating it to that of reasoning about actions [GS87], and elaborate on the impact that the frame problem [McC69] and the ramification problem [Fin88] have in transaction specifications. We will bypass the equally infamous qualification problem [McC69] by making the assumption that, all the preconditions for transaction execution are listed explicitly in the transaction specification. This is a valid assumption from the point of view of integrity maintenance,
since integrity constraints are regarded as being postcondition rather than precondition oriented.

In the majority of the existing methods for integrity constraint maintenance by transaction modification, the frame and ramification problems have either been ignored or bypassed by means of implicit assumptions that state that "nothing but what is explicitly declared to change in the update procedure does". An exception to that is Transaction Logic [BK95], where the frame and ramification problems are alleviated by the use of both procedural and declarative knowledge in the specification of direct and indirect effects of actions respectively.

Given a set of transaction specifications, the problem of succinctly stating that "nothing else changes" except the aspects of the state explicitly specified, has been called the frame problem. The ramification problem amounts to devising a way to avoid having to specify indirect effects of actions explicitly. Several attempts to solve these problems have appeared in the Artificial Intelligence planning literature of the recent years. [LR94] presents a solution to the frame problem with application to database updates. [BMR93] and [BMR95] show why the frame problem becomes particularly acute in object-oriented specifications, where transactions are inherited and specialized from superclasses to subclasses. The arguments presented therein apply directly to the specification of database transactions in deductive, active and object-oriented databases. The effects of rule evaluation or firing have to be accounted for and checked against integrity constraints. Moreover, transaction specifications can be specialized from superclasses to subclasses and conjoined to form complex transactions. The effects of specialized or synthesized transactions must be precisely characterized. To the best of our knowledge, there has not been any previous attempt to link the ramification problem with that of proving safety of transactions. In fact, a solution of the ramification problem can suggest a strategy for integrity maintenance: transform the specifications of transactions to embody implications of constraints (these may be simpler formulae than the constraints themselves); if the new specifications are not met, then the constraints are violated. Embodying the implications of constraints in transaction specifications means that the constraints need not be checked at run-time.

The solution proposed in [LR94] consists of systematically deriving a set of successor-state axioms that completely describe how fluents\footnote{Fluents are predicates whose truth value may change from state to state [McC69].} can change truth value as a result of
some action taking place. These axioms formulate closed-world assumptions for actions. A syntactic generator of successor-state axioms in the presence of binary constraints and stratified definitions of non-primitive fluents is proposed in [Pin94]. We relate this solution to the problem of integrity maintenance in the context of temporal deductive knowledge bases. The following example shows the rationale behind the use of ramifications for simplifying the task of proving transaction safety.

**Example 5.0.3** Transaction *EnrollInCourse* records the enrollment of student *st* in course *crs*. *size* and *classlimit* are function symbols representing the class size and enrollment limit respectively, whereas *EnrolledIn* is a predicate symbol. We adopt the unprimed/primed notation\(^2\) to refer to the values of variables, functions or predicates immediately before and after the execution of the transaction.

<table>
<thead>
<tr>
<th><em>EnrollInCourse</em> (<em>st</em>, <em>crs</em>)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Precondition:</strong> (-EnrolledIn(st, crs))</td>
</tr>
<tr>
<td><strong>Postcondition:</strong> <em>size</em>'(crs) = <em>size</em>(<em>crs</em>) + 1 (\land) <em>EnrolledIn</em>'(st, <em>crs</em>)</td>
</tr>
<tr>
<td><strong>Invariant:</strong> (\forall c/Course\ \ <em>size</em>(<em>c</em>) \leq \ <em>classlimit</em>(<em>c</em>))</td>
</tr>
</tbody>
</table>

In this specification, the precondition requires that the student is not already enrolled in the course, whereas the postcondition specifies the effect of the transaction on the final state: the size of the class is incremented and the predicate *EnrolledIn* becomes true of *st* and *crs*. The invariant requires that the size of a course should not exceed its limit. We can easily verify that, if the condition \((\ *size*(*crs*) + 1 \leq \ *classlimit*(*crs*))\) is conjoined with the postcondition, then no implementation that meets the augmented postcondition can possibly violate the invariant. Indeed, the invariant follows as a logical consequence of the augmented postcondition, the invariant in the previous state and the assumption that in the state before transaction execution the invariants are known to be satisfied. As will be shown in the sequel, the derived addition to the postcondition is a ramification of (a form of) the invariant and the initial postcondition. This is different than simply conjoining constraints to the transaction postconditions [Sto75]. Let us assume for a moment that the invariant had the form \(\forall c/Course\ \ *size*(_c_) \leq \ *classlimit*(c) \land \psi\), where \(\psi\) does not mention predicate

\(^2\)We will abandon this notation when we present the formal solution to the frame and ramification problem. Instead, we will use a many-sorted logic and factor time into predicates and functions.
\textit{EnrolledIn} or function \textit{size}. Then, under the assumption that the invariant was satisfied before the transaction execution, the conjunct $\psi$ can be eliminated since its satisfaction persists. The augmentation to the postcondition that suffices to guarantee the constraint is the same as above, i.e., $(\text{size}(\text{crs}) + 1 \leq \text{classlimit}(\text{crs}))$. A non-optimizing transaction modification technique, such as, e.g., in [Sto75], would include $\psi$ as a condition that would need to be conjoined with the transaction postcondition. \hfill $\Box$

We review the concept of \textit{ramifications} in section 5.1. We describe briefly the solution to the frame and ramification problems proposed in [LR92], as well as an extension to the procedure proposed in [BMR93] and [Pin94] for systematically solving the problem in the case of determinate transaction specifications. Section 5.2 presents the ramification method for transaction modification along with several examples of the application of the method on static and dynamic constraints. The results are extended in section 5.3 to deal with multiple transactions, conjunction and inheritance of specifications. Section 5.4 discusses the integration of the proposed method in the database design process.

\section{5.1 Ramifications}

In this section, we provide the necessary background for the ramification method for transaction modification. In the first part, we review the concept and properties of ramifications and, in the second, we describe the solution to the ramification problem for the class of deterministic transaction specifications. The bulk of the background material is based on [Fin88] and [Pin94] but is recast here in database terminology. Moreover, the results presented therein are extended as propositions 5.1.1 - 5.1.2 describe.

\subsection{5.1.1 Background}

Let us assume that a knowledge base $KB$ comprises a sequence of knowledge base states $(KB_0, KB_1, \ldots, KB_n)$, a set $R$ of deductive rules and a set $I$ of integrity constraints. Constraints and rules are expressed as range-restricted formulae of a many-sorted first-order logic in which we distinguish one sort $\text{Time}$ for time points. We will refer to this logic as MSTL for short. Atomic formulae of MSTL are predicates with one argument of sort $\text{Time}$ and evaluable predicates using the standard comparison operators. Well-formed formulae of MSTL are built from atomic formulae by the use of the standard logical connectives
and by quantification over time points. Time is interpreted as being relative, linear and discrete. Each knowledge base state $KB_i$ is a first-order structure comprising a universe $|KB|$, an interpretation $e^{KB_i} \subseteq |KB|$ for each constant symbol $e$ and an interpretation $p^{KB_i} \subseteq |KB|^n$ for each predicate $p$ of arity $n$. One state, $KB_c$, is distinguished as being the current knowledge base state.

The entailment of an MSTL formula from a knowledge base is defined as follows:

- If $p$ is a predicate symbol of arity $k$ and sort $S_1 \times \ldots \times S_k$, $a_1, \ldots, a_{k-1}$ are elements of $|KB|$ of sorts $S_1, \ldots, S_{k-1}$ respectively, and $t$ is an element of sort $Time$, then $KB \models p(a_1, \ldots, a_{k-1}, t)$ iff $(a_1, \ldots, a_{k-1}, t) \in p^{KB_c}$.

- If $\phi$ and $\psi$ are MSTL formulae then the entailment of $\neg \phi, \phi \lor \psi, \forall x_1/S_1, \ldots, x_k/S_k \phi$ is defined in the standard manner [End72].

- If $\phi$ is an MSTL formula then $KB \models \forall t/Time \phi$ iff for every state $KB_i$ of $KB$, $KB_i \models \phi$.

Intuitively, a ramification of a formula $\phi$ is a formula $N$ such that, $N$ is inevitably true if $\phi$ is true [Fin88]. This definition is amenable to different interpretations in different world models. If the world model in question - the knowledge base - is expressed as a first-order theory, then the concept of ramifications can be captured by first-order entailment.

**Definition 5.1.1 (Ramification)** Given a knowledge base $KB$ and a formula $\phi$ a formula $N$ is a ramification of $\phi$ in $KB$, if $KB \models (\phi \Rightarrow N)$.

**Example 5.1.1** Consider the knowledge base$^3$ $KB = \{\forall x \forall y \forall z (A(x) \land C(y) \Rightarrow D(x, y, z))\}$ and the formula $\phi \equiv \exists x \exists y (A(x) \land B(x, y) \land C(y))$. Then, the formula $N \equiv \exists x \exists y \forall z D(x, y, z)$ is a ramification of $\phi$ in $KB$ according to the above definition. \(\square\)

It is easy to verify the following properties of ramifications:

1. If a ramification of a formula is known to be unsatisfiable, then the formula itself is unsatisfiable, under the assumption that the knowledge base is satisfiable.

2. Ramifications of formulae can reduce the search space for the formulae satisfaction

---

$^3$For the sake of simplicity we assume that this knowledge base is atemporal and that all variables range over the same sort which is omitted in the expression of formulae.
3. Transformations may be applicable to formulae ramifications but not to the formulae themselves

Hence, if there exists a way to systematically generate ramifications from a set of formulae, then the derived ramifications can be used for optimizing the evaluation of the formulae themselves.

Generating ramifications of a formula \( \phi \) essentially involves augmenting a partial description of a state where \( \phi \) holds with additional descriptions that stem from the knowledge of the consistency of the state. A procedure for generating ramifications may require an arbitrary amount of inferencing. From the semi-decidability of first-order entailment, it follows that the problem of finding ramifications is, in its generality, intractable. Tractability can be achieved by restricting the class of formulae for which ramifications are sought. For instance, the task is tractable for ordered conjunctions of literals [Fin88]. Furthermore, not all derivable ramifications may be useful for simplifying the task of proving a formula. For that, the generator may be guided to derive only “useful” ramifications by providing appropriate input clauses. In fact, the solution to the ramification problem that we present in section 5.1.2 generates exactly those ramifications that are needed for proving the safety of transaction specifications.

As an example, consider the following use of ramifications for formula simplification. During the process of generating formula ramifications, the generator can be guided to generate “useful” only ramifications by, for instance, using an appropriate set of input clauses with set-of-support resolution [GN87].

**Example 5.1.2** Consider the knowledge base \( KB = \{ c_1 \equiv \forall x \forall y \forall z (A(x) \land C(y) \Rightarrow D(x, y, z)) \} \) and the formula \( G \equiv \exists x \exists y (A(x) \land B(x, y) \land C(y)) \). Figure 5.1 shows the derivation tree for a ramification of \( G \). Star-ed variables denote Skolem constants.

Incorporating the derived ramification into the formula yields \( G' \equiv \exists x \exists y [A(x) \land B(x, y) \land C(y) \land \forall z D(x, y, z)] \). Then, literals \( A(x) \) and \( C(y) \) can be eliminated because the presence of literal \( D \) in \( G' \) and in the head of the clause \( c_1 \) renders \( A \) and \( C \) redundant. \( \Box \)

The process of modifying a formula by conjoining it with additional constraints so that the reformulated formula is less expensive to check than the original one, has been termed *supersumption* in [Fin88]. The methods proposed therein apply only to conjunctive formulae. It is easy to establish the soundness of supersumption for formulae in disjunctive normal
form (DNF). Proposition 5.1.1 is a direct consequence of the definition of ramifications and proposition 5.1.2 follows from the properties of first-order entailment.

**Proposition 5.1.1** If \( N \) is a ramification of \( \phi \) in the knowledge base \( KB \), and \( N \) is falsified in \( KB \), then so is \( \phi \).

**Proof:** Follows directly from definition 1 and the contraposition law of predicate logic. \( \square \)

**Proposition 5.1.2** Let \( \phi \) be a formula in DNF, i.e. \( \phi \equiv \bigvee_{i=1}^{n} \phi_{i} \), where each \( \phi_{i} \) is a conjunction of literals. Let also \( N_{i} \) be a ramification of \( \phi_{i} \) for each \( i = 1, \ldots, n \) and \( KB \models (\phi_{i} \Rightarrow N_{i}) \). Then, if for each \( i = 1, \ldots, n \) \( KB \models \neg N_{i} \), then \( KB \models \neg \phi \).

**Proof:** Assume that \( W \models \neg N_{i} \) holds for \( i = 1, \ldots, n \). Then, for each \( i \), \( W \models \neg \phi_{i} \) holds by theorem 1, and hence, \( W \models \bigwedge_{i=1}^{n} \neg \phi_{i} \). This is equivalent to \( W \models \neg \bigvee_{i=1}^{n} \phi_{i} \) which in turn is equivalent to \( W \models \neg \phi \). \( \square \)

According to proposition 5.1.2, given a formula in DNF, \(^4\) one can derive ramifications of the conjuncts of the formula and then test whether all ramifications are falsified. If this is the case, then the initial formula is falsified. This strategy is useful if one is interested in monitoring when a formula becomes violated rather than proving that it is always satisfied. As will be seen in the next section, this strategy is sufficient to monitor integrity constraint violations in knowledge bases. The next corollary establishes the soundness of supersumption.

\(^4\)We can additionally require that the rules in the KB and the formulas are range-restricted so that efficient evaluation can be achieved.
**Corollary 5.1.1** *(Soundness of Supersumption)*

Let $\phi$ be the formula $Q_1 x_1, \ldots, Q_m x_m \hat{\phi}, Q_i \in \{\forall, \exists\}, i = 1, \ldots, m$, where $\hat{\phi}$ is quantifier-free in DNF with each of the variables $x_1, \ldots, x_m$, appearing in at least one of the disjuncts, and let $N$ be a ramification of $\phi$ in the knowledge base $KB$. Then $KB \cup \{\phi\} \models \hat{\phi} \land N$.

### 5.1.2 The Frame and Ramification Problems

In this section, we present the solution to the frame and ramification problems for a class of constraints that encompasses static and transition constraints. The solution to the frame problem was initially proposed in [Rei91] in the framework of situation calculus [McC69]. It was extended in [LR94] and [Pin94] for dealing with the ramification problem as well. The solution relies on the automatic generation of complete characterizations of the conditions under which, predicates or functions may change (truth) value as a result of transaction execution. Here, we present an extension to the method of [Pin94] for dealing with transition constraints.

The method generates *successor-state axioms* from a given set of *effect axioms*, in the presence of a limited class of constraints and definitions of non-primitive predicates. Effect axioms specify the direct effects of transactions on predicates. For instance, a direct effect of transaction *EnrollInCourse* (see example 5.0.3), is that the size of the course in which a student enrolls is incremented by one. Successor-state axioms characterize all conditions under which predicates and functions may change value as a result of transaction execution. Such axioms serve as a formalization of a closed-world assumption about the transactions themselves rather than the knowledge base.

Integrity constraints are of the form:

$$\forall x_1/S_1, \ldots, x_k/S_k, \forall t_1, t_2/Time \ \phi(x_1, \ldots, x_k, t_1, t_2) \lor$$

$$(-)p_1(x_1, \ldots, x_k, t_1) \lor (-)p_2(x_1, \ldots, x_k, t_2)$$

where, $p_1, p_2$ are $(k + 1)$-ary predicates, intensional or extensional, $S_1, \ldots, S_k$ are object sorts and $\phi(x_1, \ldots, x_k, t_1, t_2)$ is a formula in which variables $x_1, \ldots, x_k, t_1, t_2$ occur free, if at all, and does not mention any predicate other than evaluable predicates. This class of constraints is an extension of the class of *binary constraints* of [Pin94], since it allows the temporal variables to occur in evaluable predicates. It includes static constraints and transition constraints, i.e., constraints referring to two consecutive states of the knowledge base, but not general dynamic constraints. For example, the transition constraint specifying
the property that an employee's salary can never decrease, can be specified by the formula
\[ \forall e/Employee \ \forall s_1,s_2/Salary \ \forall t_1,t_2/Time \ (s_1 < s_2) \lor -(t_1 < t_2) \lor \\
-salary(e,s_1,t_1) \lor -salary(e,s_2,t_2). \]

A transaction \( T \) with parameters \( \overline{x} \) is specified by a pair \( (pre_T(\overline{x}), post_T(\overline{x})) \), where \( pre_T(\overline{x}) \), and \( post_T(\overline{x}) \) denote the transaction pre- and post-condition respectively, both specified as well-formed formulae of MSTL.

The solution assumes that deductive rules are not recursive. Moreover, they are assumed to be stratified. It also presupposes the existence of causal rules describing the direct effects of transactions. These causal rules are expressed by direct effect axioms which, for a transaction \( T(\overline{x}) = (pre_T(\overline{x}), post_T(\overline{x})) \) have the form:
\[ \forall \overline{x}/S \ \forall t/Time \ (Occur(T(\overline{x}),t) \Rightarrow pre_T(\overline{x},t) \land post_T(\overline{x},next(t))) \]
where the term \( next(t) \) denotes the state resulting from the execution of the transaction at time \( t \). Given any transaction specification, the effect axioms are derived independently of any other transaction specification. Hence, we can avoid having to specify the axioms in a refined logic that interprets the predicate \( Occur \) outside a standard first-order interpretation. The above axiom can now be written as:
\[ \forall \overline{x}/S \ \forall t/Time \ T(\overline{x},t) \Rightarrow pre_T(\overline{x},t) \land post_T(\overline{x},next(t)). \]
From the direct effect axioms we can systematically generate positive and negative effect axioms \([BMR95]\) for every predicate \( P \) that occurs in \( post_T \), as described in the following steps.\(^5\) The rationale is to describe concisely all conditions that are necessarily true when a predicate changes value from \( False \) to \( True \) (\( True \) to \( False \) respectively).

1. Construct the following positive and negative axioms:
\[ \forall \overline{x}/S \ \forall t/Time \ (\neg P(\overline{x},t) \land P(\overline{x},next(t))) \land T(\overline{x},t) \Rightarrow False \]
\[ \forall \overline{x}/S \ \forall t/Time \ (P(\overline{x},t) \land \neg P(\overline{x},next(t))) \land T(\overline{x},t) \Rightarrow False \]

2. If \( post_T \) is \( P(\overline{x},next(t)) \) (\( \neg P(\overline{x},next(t)) \)), add \( True \) as a disjunct to the positive (negative) effect axiom for \( P \).

3. If \( post_T \) is of the form \( \gamma(\overline{x},t) \Rightarrow (\neg)P(\overline{x},t) \), where \( \gamma \) does not contain terms referring to any time point except \( t \), add a disjunct \( \gamma(\overline{x},t) \) to the positive (negative) effect axiom for \( P \).

\(^5\)We only show the derivation of effect axioms for predicates. Effect axioms for functions can be derived quite similarly.
4. If $post_T(f)$ is of the form $\exists \gamma (\gamma(f, \overline{f}, t) \Rightarrow (\neg P(\overline{w}, next(t)))$, where $\overline{w}$ consists of constants and variables from $\overline{f}, \overline{f}$, then augment the positive (negative) axiom for $P$ with a disjunct $\exists \gamma (\gamma(f, \overline{f}, t) \land (f = \overline{w}))$.

This process results in a set $T_{ef}$ of effect axioms of the form:

$\forall \gamma \in \mathcal{S} \forall t/Time -P(f, t) \land -\Phi_1P(f, t) \Rightarrow -P(f, next(t)) \quad (1)$

$\forall \gamma \in \mathcal{S} \forall t/Time \quad P(f, t) \land -\Phi_2P(f, t) \Rightarrow P(f, next(t))) \quad (2)$

These axioms concisely describe how transactions directly affect the truth values of predicates. It remains to describe the indirect effects that are due to the presence of integrity constraints and deductive rules.

In addition to the effect axioms $T_{ef}$, the knowledge base is augmented with an axiomatization of time ($T_{time}$) formalizing the properties of discreteness and unboundedness, as well as unique name axioms ($T_{num}$) for predicates and functions. A new set $T_{ef}'$ is obtained from $T_{ef}$ by replacing each derived predicate occurring in an effect axiom by the disjunction of the bodies of the rules that define it. Since the set of deductive rules is assumed to be stratified, the process of replacing derived predicates by their definitions will terminate with all effect axioms mentioning only primitive predicates.

Let $C \equiv \forall \gamma \in \mathcal{S} \forall t_1, t_2/Time \phi(f, t_1, t_2) \lor P(f, t_1) \lor Q(f, t_2)$ be a constraint that has to be satisfied at all times. Then, for each effect axiom of type (1) for $P$, the following axiom for $Q$ is generated:

$\forall \gamma \in \mathcal{S} \forall t/Time -P(f, t) \land -\Phi_1P(f, t) \land -\phi(f, t, next(t)) \Rightarrow Q(f, next(t))$

This axiom expresses the property that, if predicate $P$ is known not to be true in the state prior to the execution of a transaction and constraint $C$ is known to be satisfied in the same state, then, if the conditions that cause $P$ to change truth value from $False$ to $True$ are not satisfied, $Q(f, next(t)) \lor \phi(f, t, next(t))$ has to be true in order for the constraint to remain satisfied in the state after the transaction execution.

Symmetrically, for each effect axiom of type (1) for $Q$, generate the following axiom for $P$:

$\forall \gamma \in \mathcal{S} \forall t/Time -Q(f, t) \land -\Phi_1Q(f, t) \land -\phi(f, t, next(t)) \Rightarrow P(f, next(t))$

The respective process takes place if the constraint contains negated predicates. Then, under the assumption that the given specifications characterize all transactions, we can generate the set $T_{so}$ of successor-state axioms as follows: Let $\Psi_P(f, t) = -\Phi_1Q(f, t) \land -\phi(f, t, next(t))$ and $\Psi_Q(f, t) = -\Phi_2Q(f, t) \land -\phi(f, t, next(t))$, $\Psi_Q(f)$ and $\Psi_Q(f)$ are defined analogously.
Then, the successor-state axiom for \( P \) is:
\[
\forall \bar{x}/S \forall t/Time \; \Psi_P(\bar{x}, t) \lor (\neg \Psi_{\neg P}(\bar{x}, t) \land P(\bar{x}, t))
\]

**Example 5.1.3** Example 5.0.3 showed the definition of transaction \( EnrollInCourse \) which affects the predicate \( EnrolledIn \) and the function \( size \). Let us assume that another transaction, \( DropCourse \) is defined as follows:

<table>
<thead>
<tr>
<th>DropCourse(st,crs)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Precondition:</strong> EnrolledIn(st,crs)</td>
</tr>
<tr>
<td><strong>Postcondition:</strong> size'(crs) = size(crs) - 1 &amp; \neg EnrolledIn'(st,crs)</td>
</tr>
</tbody>
</table>
| **Invariant:** \( \forall c/\text{Course} \; \text{size}(c) \leq \text{classlimit}(c) \)

The generation process described in this section will generate the following successor-state axioms for \( EnrolledIn \) and \( size \):
\[
\forall st/\text{Student} \; \forall crs/\text{Course} \; \forall t/Time
\]

\[
\neg \text{EnrolledIn}(st, crs, t) \lor \text{EnrolledIn}(st, crs, next(t)) \lor \text{DropCourse}(st, crs, t)
\]

\[
\forall crs/\text{Course} \; \forall t/Time \; \text{size}(crs, t) = \text{size}(crs, next(t)) \lor \exists st \; \text{EnrollInCourse}(st, crs, t) \lor \exists st \; \text{DropCourse}(st, crs, t)
\]

Under the assumption that the transactions \( EnrollInCourse \) and \( DropCourse \) are the only ones affecting \( EnrolledIn \) and \( size \), the above axioms characterize all conditions under which the predicate or function can change (truth) value as a result of transaction execution. Specifically, the first axiom expresses the property that if a student is enrolled in a course in the state prior to a transaction’s execution but is not enrolled in the course after the transaction’s execution, then it has to be the case that transaction \( DropCourse \) occurred and it cannot be the case that any other transaction may have occurred. The second axiom says that, if function \( size \) changes value because of transaction execution, then it is the case that either transaction \( EnrollInCourse \) occurred or transaction \( DropCourse \) occurred.

It has been shown in [LR94] that a set, \( T_{ss} \), of successor-state axioms constitutes a solution to the frame and ramification problem if, for every predicate \( p \), the condition \( T_{una} \models \neg(\Psi_p \land \Psi_{\neg p}) \) is satisfied. Moreover, the correctness of the syntactic generation has been proven in [LR94] and [Pin94]. We are now in a position to state the relationship of the syntactic generation of successor-state axioms with ramifications of constraints. A similar result holds for the case in which predicates occur negated in the expression of a constraint.
Proposition 5.1.3 For a knowledge base $KB$ and a constraint

$$I \equiv \forall \bar{x}/S, \forall t_1, t_2/Time \phi(\bar{x}, t_1, t_2) \lor P(\bar{x}, t_1) \lor Q(\bar{x}, t_1, t_2)$$

the following entailment relation holds:

$$KB \cup \{I\} \models \left(\neg \Phi_1 P(\bar{x}, t) \land \neg \phi(\bar{x}, t, next(t)) \lor \neg \Phi_1 Q(\bar{x}, t) \land \neg \phi(\bar{x}, t, next(t))\right)$$

Proof: Follows from the syntactic generation process.

The above result is significant to the problem of proving transaction safety, since it provides a way to produce systematically necessary conditions (ramifications) for the satisfaction of constraints in the state resulting from the transaction execution.

5.2 Integrity Maintenance by Transaction Modification

In this section, we establish the relationship between the ramification problem and the maintenance of integrity constraints in the context of knowledge bases containing deductive rules and temporal knowledge.

In this context a formula $\phi$ is called a ramification of an integrity constraint $I$ of the knowledge base $KB$ if $KB \models (I \Rightarrow \phi)$. This means that a ramification of an integrity constraint is an entailment of the knowledge base and its integrity constraints according to the definition of entailment in section 5.1.1.

5.2.1 Ramifications and Integrity Maintenance

The problem of integrity maintenance is defined as follows: Given a knowledge base $KB$ with constraint set $I$ and a set of transaction specifications $T = \{T_1, \ldots, T_k\}$ with $T_i = (pre_i, post_i)$, can the set $I$ be systematically partitioned into sets $I_t, I_f, I_c$ so that: (a) constraints in $I_t$ are provably maintained by $T$, (b) constraints in $I_f$ are provably violated by some $T' \subseteq T$, and (c) constraints in $I_c$ have to be checked after execution of some transactions in $T$ but possibly in some simplified form? Since we are following a transaction modification approach, the problem is equivalent to transforming the set, $T$, of transactions into a set, $T'$, of transactions with the property that, for each transaction $T'_i$ in $T'$, either $T'_i = T_i$ and $T_i$ has been shown not to violate any of the integrity constraints, or $T_i$ has been modified to $T'_i$ and every implementation meeting its new specification cannot possibly violate any of the constraints in $I$. 

111
Integrity constraints defined in the knowledge base specify properties that each state of the knowledge base must respect in order to be considered a valid state. Assuming that transactions that access and/or modify the knowledge base are specified by means of precondition/postcondition pairs, integrity constraints serve the role of invariants of transactions. Each transaction must maintain its invariants, i.e., not violate the relevant integrity constraints in order to be accepted. The problem of proving that a transaction maintains its invariants is formalized in the following definition.

**Definition 5.2.1 (Invariant Maintenance)**

Let $T$ be a transaction with precondition $P$, postcondition $Q$ and invariant $I$. $T$ is said to maintain invariant $I$ when executed over a knowledge base $KB$, if $KB \cup \{I, P\} \models (Q \Rightarrow I')$, where $I'$ denotes the invariant in the state resulting after the transaction takes place.

Proving that transactions maintain invariants is a difficult task since it requires theorem proving. Furthermore, the cost of undoing the transaction in case it is discovered to violate the invariants is high. A way to avoid checking whether transactions maintain their invariants is to augment their postconditions in a way such that the invariant is maintained as a result of meeting the postcondition. In other words, if for any invariant/postcondition pair $(I, Q)$, an augmented postcondition $R \equiv Q \wedge N$ can be found, such that meeting the augmented postcondition has the effect of maintaining the invariant,

We now show that such an augmented postcondition can be found by computing ramifications. The following discussion is based on a number of completeness assumptions. These assumptions specify that, firstly, all transaction specifications are known at the time that integrity constraints are specified and, secondly, that transaction invariants are known to be maintained in the state before the transaction execution. We discuss in the sequel how we can incrementally accommodate newly defined transactions.

We consider the single-transaction case first. Assume a transaction $T$ specified by a pair $(P, Q)$ of a precondition and a postcondition expressed in MSTL. Let $I$ be an integrity constraint relevant to the transaction\(^6\). We need to find a formula $N$ such that $KB \cup \{P, I\} \models (Q \wedge N \Rightarrow I')$, or equivalently that, $KB \cup \{P, I\} \models (Q \wedge \neg I' \Rightarrow \neg N)$. If $\neg N$ is a ramification of $Q \wedge \neg I'$ as computed by the syntactic generator, then the desired entailment relationship holds. This leads to the following theorem, whose proof follows from definitions

\(^6\)The notion of “relevance” is defined formally in the sequel.
5.1.1, 5.2.1 and corollary 5.1.1:

**Theorem 5.2.1** For any transaction $T$ specified in terms of precondition $P$ and postcondition $Q$ expressed in MSTL, and any invariant $I$ of the transaction, if a ramification $N$ of $Q \land \neg I'$ can be found by the syntactic generator, then $\neg N$ can be conjoined with $Q$ and the augmented postcondition has the property that, the invariant is maintained in the state resulting after the execution of the transaction as a result of meeting the augmented postcondition.

**Proof:** Follows from the above discussion and the soundness property of supersetumption. $\Box$

This result has significant impact to both the areas of procedure specification and constraint enforcement: if the process of suggesting additions to the postconditions of transactions can be automated, the transaction specifier actually realizes the implications of transaction invariants and the procedure implementor is saved the burden of finding ways to meet the postcondition in a way such that no invariant is violated. In fact, the implementor may not be familiar with all the invariants that a certain transaction may affect. The theorem also suggests a way of enforcing integrity constraints by requiring that updating transactions meet postconditions that embody implications of constraints: first, ramifications of the conjunction of the postcondition and the negation of the invariant instantiated in the state after transaction execution are computed; the negation of the computed ramifications is conjoined with the transaction postcondition to form a postcondition which should be met by the implementation in order not to violate the invariants. The fact that postconditions describe all the direct effects of transactions can be exploited to simplify the formula of which ramifications are sought. Specifically, (truth) values of predicates or functions changed by the transaction can be assumed to be known in the state resulting from transaction execution. Hence, the (truth) values can be substituted for the predicates or functions and logical simplifications may be applicable. The invariants themselves need not be verified in the state resulting from the transaction execution since their satisfaction is guaranteed by the transformation process. Moreover, the ramifications generated may be simpler formulae than the invariants and hence, the transformation of postconditions can incur considerable savings in testing for the satisfaction of invariants. An exact assessment of the run-time complexity of testing the satisfaction of augmented postcondition for the
The class of transaction specifications considered here is a topic of current research.

Although the transformation of postconditions is unidirectional, it is sufficient for the problem of constraint maintenance. Satisfaction of the constraint’s ramification does not, in general, imply the constraint’s satisfaction; however, if a ramification of a constraint is violated, the constraint itself is violated. The following examples show the merits and problems associated with the technique of incorporating ramifications into transaction specifications.

**Example 5.2.1** Transaction *EnrollInCourse* was defined in example 5.0.3. In this example, we show the derivation of ramifications for postcondition augmentation, by rewriting the postcondition and invariant in MSTL.

We first construct the conjunction of the postcondition $Q$ and the negation of the invariant instantiated as follows:

$$
\neg I' \equiv \neg (size(\text{crs}, t + 1) \leq \text{classlimit}(\text{crs}))
$$

$$
\neg I' \land Q \equiv \neg (size(\text{crs}, t + 1) \leq \text{classlimit}(\text{crs})) \land (size(\text{crs}, t + 1) = size(\text{crs}, t) + 1) \land EnrolledIn(st, \text{crs}, t + 1)
$$

By substituting True for *EnrolledIn(st, crs, t + 1)* and $size(\text{crs}, t) + 1$ for $size(\text{crs}, t + 1)$, the conjunction becomes $\neg (size(\text{crs}, t) + 1 \leq \text{classlimit}(\text{crs}))$. This formula is in fact a ramification of $\neg I' \land Q$ and can be computed as shown in section 5.1.2.

According to theorem 5.2.1, it suffices to conjoin the negation of the ramification to the postcondition. The invariant is no longer needed for verifying the safety of transaction *EnrollInCourse*, since the invariant is embodied in the new postcondition. The augmented transaction specification now becomes:

<table>
<thead>
<tr>
<th>EnrollInCourse (st, crs)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Precondition:</strong> $\neg EnrolledIn(st, crs)$</td>
</tr>
<tr>
<td><strong>Postcondition:</strong> $size'(\text{crs}) = size(\text{crs}) + 1 \land EnrolledIn'(st, \text{crs}) \land (size(\text{crs}, t) + 1 \leq \text{classlimit}(\text{crs}))$</td>
</tr>
</tbody>
</table>

The next example shows that certain ramifications can suggest that invariants do not have to be checked and postconditions need not be modified in order to guarantee the invariants.
Example 5.2.2 (Special cases) The specification of transaction DropCourse was given in example 5.1.3. Intuitively, the invariant cannot be violated as a result of executing DropCourse, and hence the specification of the transaction need not be modified. In fact, the ramification generation process produces the Boolean constant False as a ramification of the negation of the conjunction formed by instantiating and negating the invariant in the state after the transaction execution, and by conjoining it with the transaction postcondition. According to theorem 5.2.1, it suffices to augment the postcondition with the negation of the derived ramification, i.e., the Boolean constant True. This means that it suffices for the implementation meet the transaction postcondition as was initially specified, in order to maintain the invariant.

The case in which the propositional constant False is derived as a ramification is of particular interest since, as the following corollary specifies, no change in the postcondition is needed in order to meet the invariant.

Corollary 5.2.1 If False is derived as a ramification of \( Q \land \lnot I' \) by the process described in section 5.1, then \( I' \) is maintained by a transaction meeting \( Q \).

Proof: By theorem 5.2.1, if False is derived as a ramification of \( Q \land \lnot I' \), then the invariant is maintained if the postcondition \( Q \land \lnot \text{False} \) is met. In this case, the strengthened postcondition is equivalent to the original postcondition \( Q \). If \( Q \) is met the invariant is maintained.

In the case of an inconsistent transaction specification, the inconsistency will be introduced in the generation of the effect axioms. Hence, the process of generating ramifications can also discover inconsistent specifications of transactions that may have escaped the specifier’s attention.

A valid question that arises is whether a similar approach where preconditions rather than postconditions of transaction specifications can be augmented so that the maintenance of the invariants is a result of the satisfaction of the transaction precondition. In general, the two approaches are not equivalent. They are equivalent only in the case where derived ramifications refer only to the state before the transaction execution. The following section shows an example of a transaction specification where the derived ramification refers to the state after the transaction’s execution. It is, thus, unnatural to augment the postcondition
with a condition that refers to the state resulting from the transaction execution.

5.2.2 Dynamic Integrity Constraints

We would like to be able to propose similar augmentations to postconditions when the invariant refers to any number of knowledge base states, both before and after the state in which a transaction is executed. In other words, we need to extend the method for the enforcement of dynamic integrity constraints. The solution to the ramification problem presented in section 5.1.2 does not deal with constraints more general than transition constraints. It is applicable in the cases where the checking of conditions over multiple consecutive states can be reduced to checking conditions over pairs of consecutive states. The extension of the method to general dynamic constraints is a topic of current research. Some initial results are given through examples of the use of ramifications for transactions that involve transition and general dynamic constraints. These examples also motivate the use of a temporal calculus for expressing transaction specifications. As will be seen from the different examples presented, we do not commit ourselves to a particular language for specifying transactions or invariants, but limit ourselves to demonstrate instances of the applicability of the approach to a number of more expressive formalisms.

Example 5.2.3 (Transition constraints as transaction invariants) Transaction RaiseSalary assigns an employee an increase to her salary.

<table>
<thead>
<tr>
<th>RaiseSalary (emp, new_sal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precondition: ( \exists \text{old} ) salary(emp, old, ( T_b ))</td>
</tr>
<tr>
<td>Postcondition: salary(emp, new_sal, ( T_a ))</td>
</tr>
</tbody>
</table>

Invaraint: \( \forall e, s, s'/D \forall t, t'/T \ [\text{salary}(e, s, t) \land \text{salary}(e, s', t') \land (t \leq t') \Rightarrow (s \leq s') \) |

\( T_b \) and \( T_a \) are used to denote time points before and after the transaction respectively. They are parameters whose exact values are not known at transaction specification time\(^7\). For the purpose of deriving the successor state axioms, one needs to instantiate the time component of predicates referring to the state prior to the transaction with \( T_b \). The time component of predicates referring to the state after the transaction are instantiated with \( T_a \).

---

\(^7\)These time points are not unique. It suffices that \( T_b \) is a time point at which the constraints have to be verified (before the transaction commits) and that \( T_a \) a time point before the transaction begins execution and at which it is known that the knowledge base is in a consistent state.
The ramification derived is: \((\text{old}^* > \text{new}_{\text{sal}})\). Its addition to the postcondition suffices to ensure the maintenance of the invariant.

For the sake of demonstrating the applicability of using ramifications with constraints strictly more general than the ones dealt with so far, we now switch to using first-order temporal logic (FOTL) [MP91] as the specification language.\(^8\) FOTL allows one to express constraints referring to an arbitrary number of states. We need to assume, without loss of generality, that exactly one action can occur between two successive states of the knowledge base. The following example also demonstrates why it is unnatural to consider augmentations of the precondition of a transaction specification in order to achieve the maintenance of invariants. We argue, contrary to [LR92], that dynamic integrity constraints can be effect-yielding constraints. Specifically, the following example shows that it is unnatural to consider constraints from this particular class as precondition-oriented.

**Example 5.2.4 (General dynamic integrity constraints)** The formula expressing the property “If \(P(x)\), then sometime in the future \(Q(x)\)” is an invariant for transaction \(\text{InsertP}\) that inserts a tuple \((x,t)\) in the extension of base predicate \(P\).

<table>
<thead>
<tr>
<th>(\text{InsertP} (x,t))</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Precondition</strong>: True</td>
</tr>
<tr>
<td><strong>Postcondition</strong>: (P(x,t))</td>
</tr>
<tr>
<td><strong>Invariant</strong>: (\forall x/D \forall t/\text{Time} \ [P(x,t) \Rightarrow \exists t'/\text{Time} (t' &gt; t \land Q(x,t'))])</td>
</tr>
</tbody>
</table>

In addition, assume that the knowledge base includes the following deductive rules:

\(R_1: \forall x/D \forall t/\text{Time} \ P(x,t) \rightarrow R(x,t + 1)\)

\(R_2: \forall x/D \forall t/\text{Time} \ R(x,t) \rightarrow Q(x,t)\)

Intuitively, after the transaction \(\text{InsertP}\) finishes execution, the knowledge base is in a consistent state, since the constraint is satisfied due to the implicit updates. Because of the presence of the rules, no precondition exists that will guarantee the invariant. The invariant however contributes to effects (ramifications) that can be used to eliminate the need for proving the invariant. We generate the ramifications by, first, instantiating and negating the invariant and then computing its logical consequences given the knowledge base. The negated invariant is \(\neg I' \equiv \forall t'[P(x,t) \land (Q(x,t') \Rightarrow (t' \leq t))]\). From the postcondition and

\(^8\)Similar results can be obtained when transactions are specified in a FOTL.
rule $R_1$ we derive $R(x,t+1)$. Using rule $R_2$, we can now derive $Q(x,t+1)$. Using the negated invariant and the postcondition we derive the ramification $N \equiv Q(x,t+1) \Rightarrow \text{False}$ The ramification is equivalent to $\neg Q(x,t+1)$ which in turn is equivalent to the False. Hence, the invariant is maintained if the postcondition is met. Notice that it is unreasonable to include $Q(x,t+1)$ as a precondition to a transaction of which it is an implicit consequence. The constant $\text{False}$ is generated as a ramification by the syntactic generator by replacing the derived predicates that occur in $\neg I' \land Q$ by their definitions and then applying the steps described in section 5.1.2.

Albeit artificial, example 5.2.4 shows that dynamic constraints can have useful effects for transactions. Even in cases where the satisfaction of a constraint cannot be determined because the constraint refers to the yet undefined future, the method yields a ramification that can be used in place of the original constraint. This is the case in example 5.2.4 if the rules are omitted. We cannot determine whether the constraint is satisfied, or violated, since it refers to the possibly infinite set of subsequent states. In this case, the method can propose a simpler condition, that is actually a ramification of the original constraint and the postcondition. The formula $N \equiv \neg \forall t' (t' \leq t \lor \neg Q(x,t'))$ is derived and the new constraint that suffices to be verified is $\neg N \equiv \exists t' (t' > t \land Q(x,t'))$. It contains the property that has to be verified by the future states and can be treated as the original constraint would.  

Similar results can be obtained when transactions are specified in FOTL [MP91]. Assume also that transactions are the units of state change, i.e., a “next” state results from the current state after the execution of a transaction. Let us look again at the previous example now formulated in FOTL.

**Example 7: First-Order Temporal Logic**

In this example, the transaction specification of example 6 is reformulated in first-order temporal logic.

<table>
<thead>
<tr>
<th>$InsertP (x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Precondition:</strong> True</td>
</tr>
<tr>
<td><strong>Postcondition:</strong> $\diamond P(x)$</td>
</tr>
<tr>
<td><strong>Invariant:</strong> $\square (P(x) \Rightarrow \diamond Q(x))$</td>
</tr>
</tbody>
</table>

---

9One possibility for checking satisfaction of constraints of the above form, is to create an agenda including the conditions that have to be verified at appropriate times in the future.
The deductive rules $R_1$ and $R_2$ are rewritten as:

$$R_1: \Box(P(x) \rightarrow \diamond R(x))$$

$$R_2: \Box(R(x) \rightarrow Q(x))$$

The generation of ramifications from the invariant and the postcondition is slightly different because of the different syntax. $I'$ is obtained by instantiating the invariant for the state after the transaction execution. Instantiation means replacing the $\Box$ operator in the invariant expression by $\diamond$. The use of instantiation is legal since the invariant will be violated if its instantiated form is violated at any point in time. The formula thus obtained for $I'$ is:

$$I' \equiv \diamond(P(x) \Rightarrow \diamond Q(x)) \equiv (\diamond P(x) \Rightarrow \diamond Q(x))$$

Then $I'$ is negated and conjoined with the postcondition of the transaction yielding:

$$\diamond P(x) \land \neg \diamond Q(x) \land \diamond P(x) \equiv \diamond P(x) \land \neg \diamond Q(x)$$

Now we can use the knowledge that in the updated state the postcondition has to be met and simplify the above formula into

$$\neg \diamond Q(x) \equiv \neg \Box Q(x) \equiv \neg \Box \neg Q(x)$$

Using the deductive rules $R_1$ and $R_2$ we can derive $\diamond Q(x)$. From that and $\diamond \neg Q(x)$, False is derived as a ramification. According to corollary 2, the invariant of the transaction will be maintained if the postcondition $\diamond P(x)$ is maintained.

To summarize this section we outline a method that can be used to suggest modifications to a transaction specifier with respect to the satisfaction or not of the defined constraints and, possibly, the inclusion of a simpler test for verifying that constraints remain satisfied after transaction execution.

**Method Outline:** From $I$ derive $I'$, the instance of the constraint in the state resulting from the transaction execution. Transform $\neg I'$ in disjunctive normal form (DNF). Transform each of the disjuncts in clausal form and use set-of-support resolution [GN87] with initial set of support the literals of the disjunct and input set the knowledge base, for deriving ramifications of the conjunction of each disjunct with the transaction postcondition. The disjunction of ramifications generated for each of the disjuncts is a ramification of $Q \land \neg I'$.

This method is inherently incomplete since resolution is used as a derivation rather than a refutation procedure. The class of constraints for which the method is also complete has to be precisely characterized.

---

10 Any first-order formula has an equivalent expression in DNF.
5.3 Extensions

In this section, we present extensions of the ramification method for dealing with multiple constraints, multiple transactions, conjoining transaction specifications, and inheritance of specifications.

5.3.1 Multiple Transactions

The approach followed in the previous section is certainly applicable to the problem of reasoning about the set of constraints of a knowledge base in the presence of multiple transaction specifications. In this case, all transaction specifications have to be taken into account since a constraint may be relevant to, or affected by, more than one transaction. The notion of relevance is formally defined here. It is based on the notion of dependence, as this is defined in definition 4.1.3. We will assume that the specifications are given in MSTL and that no interleaving of transactions is allowed. The difficulty lies in the characterization of the implicit changes taking place in the knowledge base because of the presence of deductive rules. In the absence of deductive rules, interleaved transaction execution can be dealt with in conjunction with a concurrency control protocol ensuring serializability. Transactions are regarded being atomic and as the only means of knowledge base state change.

Definition 5.3.1 (Relevance)

A constraint $I$ is relevant to a transaction $T = (pre, post)$ if $post$ contains a literal on which some literal of $I$ depends.

The definition of dependence is repeated here for ease of reference.

Definition 5.3.2 (Dependence)

A literal $L$ directly depends on a literal $K$ if and only if there exists a rule of the form $\forall x_1/C_1 \ldots \forall x_n/C_n (F \Rightarrow A)$ such that, there exists a literal in the body $F$ of the rule unifying with $K$ with mgu $\theta$ and $A\theta = L$. (Dependence) A literal $L$ depends on literal $K$ if and only if it directly depends on $K$, or depends on a literal $M$ that directly depends on $K$. A constraint/rule depends on a rule if its literal depends on the rule’s conclusion literal.

For instance, a constraint $I$ whose expression contains predicate $R$ will be relevant to a transaction that inserts a tuple in the extension of $R$, if the the transaction postcondition
contains predicate $R$ in the state resulting from the transaction execution. The definition has to be refined appropriately when a different syntax is used for transactions. Obviously, this definition of relevance is not sufficient in the presence of implicit updates, where an integrity constraint may be violated as a result of the derivation of an implicit consequence of the explicit update. To deal with implicit updates we must extend the definition of relevance to take deductive rules into account. Such a notion of relevance has been used in chapter 4, where integrity constraints are compiled along with their relevant deductive rules so that, at run time, the implicit consequences of explicit updates can be easily derived. Similarly, in the present, the deductive rules that are relevant to the transaction specification have to be found as well. A constraint is relevant to a transaction specification if it is relevant to the head of a rule that is relevant to the specification. Hence, from now on, when we refer to a constraint as being relevant to a transaction, we will mean that it is either directly or transitively relevant to the transaction’s specification.

An integrity constraint relevant to a set of transactions $\{T_i = (\text{pre}_i, \text{post}_i) | i = 1, \ldots, k\}$ has to be considered for the modification of each $\text{post}_i$, so that the execution of any $T_i$ provably maintains the constraint. Hence, it suffices to repeat the process presented in section 5.2 for every $T_i$. The process may be optimized by reusing the derivation of ramifications for transactions that involve common predicates.

**Example 5.3.1 (Multiple Transactions)** Consider the transactions $\text{EnrollInCourse}$ and $\text{DropCourse}$, both with parameters $(\text{st, crs, trm})$. In the former a student $\text{st}$ is enrolled in a course $\text{crs}$ for term $\text{trm}$, whereas in the latter, student $\text{st}$ drops course $\text{crs}$ for for term $\text{trm}$. Both transactions change the value of a global function $\text{enrolNo(crs, trm)}$. The global function $\text{classLimit(crs, trm)}$ returns the maximum number of students that can be enrolled in a course for a particular term. The invariant expresses the property that there must exist a course whose enrollment number is greater than zero at all terms.

```
\text{EnrollInCourse(st, crs, trm)}

\text{Precondition1:}
\text{enrolNo(crs, trm) < classLimit(crs, trm) \land \exists t \text{ enrolledin}(st, crs, t)}

\text{Postcondition1:}
\text{enrolNo'(crs, trm) = enrolNo(crs, trm) + 1 \land enrolledin'(st, crs, trm)}
```
\textbf{Invariant:} \( \exists y \forall z (\text{enrolNo}(y, z) > 0) \)

\begin{tabular}{|l|}
\hline
\textbf{DropCourse}(st, crs, trm) \\
\textbf{Precondition2:} enrolledin(st, crs, trm) \\
\textbf{Postcondition2:} \\
enrolNo'(crs, trm) = enrolNo(crs, trm) - 1 \land \neg enrolledin'(st, crs, trm) \\
\hline
\end{tabular}

\textbf{Invariant:} \( \exists y \forall z (\text{enrolNo}(y, z) > 0) \)

\textbf{Invariant} is relevant to both transactions. Applying the ramification method in turn, yields the conditions \( \neg (\text{enrolNo}(crs, trm) \leq 0) \) and \( \neg (\text{enrolNo}(crs, trm) \leq 1) \), which are sufficient for the maintenance of the invariant by \textit{EnrollInCourse} and \textit{DropCourse} respectively.

The constraint is negated and conjoined with each transaction specification for deriving ramifications that will be used for augmenting postconditions.

\[-I' \equiv (\text{enrolNo}'(crs, trm) \leq 0). \text{ Then, } -I' \land \text{Postcondition1} \equiv \]

\{(\text{enrolNo}'(crs, trm) \leq 0) \land (\text{enrolNo}'(crs, trm) = \text{enrolNo}(crs, trm) + 1) \land \\
\text{enrolledin}'(st, crs, trm) \} \Rightarrow (\text{enrolNo}(crs, trm) < 0) \equiv N.

Hence, \(-N \equiv (\text{enrolNo}(crs, trm) \geq 0).

For transaction \textit{DropCourse}, \(-I' \land \text{Postcondition2} \equiv \{(\text{enrolNo}'(crs, trm) \leq 0) \land \\
(\text{enrolNo}'(crs, trm) = \text{enrolNo}(crs, trm) - 1) \land \neg \text{enrolledin}'(st, crs, trm) \} \Rightarrow \\
(\text{enrolNo}(crs, trm) \leq 1) \equiv N. \text{ Hence, } -N \equiv (\text{enrolNo}(crs, trm) > 1).

We can see that the derived ramifications is what one intuitively expects are sufficient conditions for the invariants to be maintained.

The above example shows that the method can be used in the case where more than one transactions are relevant to a single invariant. The applicability of the method allows to provide a characterization of the constraint set of a knowledge base with respect to the need for verifying the constraints after each transaction takes place. As we have shown in the previous section, the derivation of ramifications simplifies the task of constraint enforcement by either revealing cases where constraints do not need to be checked or by guaranteeing the constraints’ satisfaction by modifying the transactions’ postconditions.
5.3.2 Multiple Constraints

The symmetric case, where a transaction specification is associated with more than one invariants, is dealt with by simply taking the conjunction of the invariants as the new invariant. Then the derived ramification depends on all invariants, provided that the union of the invariants is a satisfiable set.

5.3.3 Conjoining Transaction Specifications

In this paragraph, we discuss the applicability of the simplification method with respect to conjoining specifications of transactions to form new transaction specifications. Leaving aside the problem that the frame axioms specifying the "nothing else changes" property, we assume that the the conjunction of specifications - denoted by the operator $\mid$ - is formed by conjoining the respective pre/post-conditions. Then, as theorem 5.3.1 suggests, it suffices to conjoin the ramifications of the two invariant-postcondition pairs, to guarantee that the invariant will be maintained if the new postcondition is met.

**Theorem 5.3.1** Let $T_1 = (pre_1, post_1)$ and $T_2 = (pre_2, post_2)$ be two transaction specifications sharing invariant $I$. If there exist ramifications $N_1$ and $N_2$ which, if conjoined with the postconditions $post_1$ and $post_2$ guarantee the maintenance of $I$ in $T_1$ and $T_2$ respectively, then $N_1 \land N_2$ is a ramification which, if conjoined with $post_1 \land post_2$ guarantees the maintenance of $I$ in $T = T_1 || T_2$.

**Proof:** We can easily prove that for any first-order formulae $\phi_1, \phi_2, \psi_1, \psi_2$ the formula

$$[(\phi_1 \rightarrow \psi_1) \land (\phi_2 \rightarrow \psi_2)] \rightarrow [(\phi_1 \land \phi_2) \rightarrow (\psi_1 \lor \psi_2)]$$

is valid. From that and the fact that ramifications that suffice to guarantee the invariants have been derived, we can derive the following:

$$KB \cup \{P, I\} \models (post_1 \land N_1) \Rightarrow I \quad \text{and} \quad KB \cup \{P, I\} \models (post_2 \land N_2) \Rightarrow I$$

$$KB \cup \{P, I\} \models (post_1 \land post_2) \land (N_1 \land N_2) \Rightarrow I$$

**Example 5.3.2** (Conjoining Specifications)

We apply the theorem the case of conjoining the transaction specifications of example 8. We define a new transaction $ChangeCourse(st, crs1, crs2, trm)$ as $DropCourse(st, crs1, trm) || EnrollInCourse(sr, crs2, trm)$. Theorem 4 specifies that it suffices to augment the new postcondition
enrolNo'(crs1, trm) = enrolNo(crs1, trm) - 1 \land -enrolledin'(st, crs1, trm) \land
enrolNo'(crs2, trm) = enrolNo(crs2, trm) + 1 \land enrolledin'(st, crs2, trm)

with the conjunction of the previously derived ramifications:

(enrolNo(crs1, trm) > 1) \land (enrolNo(crs2, trm) \geq 0)

Intuitively, if the condition specified by the combined ramifications is satisfied then the
invariant is maintained if the new postcondition is met.

An important consequence of theorem 5.3.1 is the ability to accommodate new invariants
without having to redo the entire process. Specifically, if a new invariant is to be added and
is relevant to a transaction specification whose postcondition has already been augmented
by computed ramifications, it suffices to verify that the new invariant does not introduce
any contradiction, and, if this is the case, to generate ramifications of the new invariant
and the postcondition. The new ramification can be conjoined with the previously derived
ones, so that the new postcondition guarantees the invariants.

5.3.4 Inheritance of Transaction Specifications

In object-oriented specification languages, inheritance of transaction (method) specifications
is traditionally accomplished by conjoining the superclass’ method specification to that of its
subclasses [SP87]. We will ignore for the moment the possibility of inconsistencies between
the frame axioms required for both the subclass and superclass method specifications. We
will assume that the generation of explanation closure axioms [BMR93] suffices to provide
the required knowledge about the predicates and functions that change or remain unaffected
by the transaction. The axioms derived by the method proposed in [BMR93] form part of
the knowledge base and are used in the process of deriving ramifications. We examine
whether ramifications derived for the superclass can be inherited by the subclasses.

Assume a transaction $T_2 = (pre_2, post_2)$ is a specialization of $T_1 = (pre_1, post_1)$ and
that there exists a formula $N_1$ with the property $KB \cup \{P, I\} \models (post_1 \land N_1) \Rightarrow I$. The
specification of $T_1$ is inherited by $T_2$. It is the responsibility of the specifier to ensure that
neither of the conjunctions $(pre_1 \land pre_2)$ and $(post_1 \land post_2)$ is a contradiction. Then,
according to theorem 5.3.1, if a ramification $N_2$ can be found, such that $KB \cup \{P, I\} \models
(post_2 \land N_2) \Rightarrow I$, then augmenting $post_2$ with $N_1 \land N_2$ suffices to guarantee that the
invariant will be maintained if the augmented refined postcondition is met.

124
Example 5.3.3 (Inheritance of Specifications)

Transaction DropSpecialCourse is a specialization of the transaction DropCourse of example 5.3.1 for courses that have corequisites. We assume that there exists a predicate coreq($c_1$, $c_2$) which is true if $c_2$ is among the corequisites of $c_1$.

\[
\text{DropSpecialCourse}(st, crs, trm) \\
\text{Precondition2: True} \\
\text{Postcondition2:}
\begin{align*}
\forall c & \quad \text{coreq}(crs, c) \quad \Rightarrow (\text{enrolNd}(c, trm) = \text{enrolNo}(c, trm) - 1) \land \\
\neg & \text{enrolledIn}'(st, c, trm)
\end{align*}
\]

\[
\begin{array}{c}
\text{Invariant: } \exists y \forall z \ (\text{enrolNo}(y, z) > 0)
\end{array}
\]

\text{DropSpecialCourse} inherits the specification of \text{DropCourse} for which ramification \( N_1 \equiv (\text{enrolNo}(crs, trm) > 1) \) has been derived. The application of the method for transaction \text{DropSpecialCourse} will yield \( N_2 \equiv \{ \forall c \ \text{coreq}(crs, c) \ \Rightarrow (\text{enrolNo}(c, trm) > 1) \} \). Then, \( KB \cup \{ P, I \} \models ((post_1 \land post_2) \land (N_1 \land N_2) \Rightarrow I) \). \hfill \Box

5.4 Discussion

The previous sections showed how the task of proving integrity maintenance can be assisted by the adaptation of a systematic solution to the frame and ramification problems. The method presented is applicable to a fairly large class of transaction specifications, namely that of determinate specifications, and a class of constraints that encompasses the types of constraints that are usually supported by commercial database systems and most research prototypes. Furthermore, the results extend to the case of object-oriented specifications and for the synthesis of specifications by conjoining existing specifications.

The process of designing a knowledge base can be assisted by a tool that, when given a set of transaction specifications, suggests modifications to the postconditions so that the integrity constraints are provably maintained by any implementation meeting the modified specification. Moreover, inconsistencies in the specifications can be discovered. We argue that this form of feedback is crucial for the knowledge base design process, since it provides a systematic way of testing whether certain desirable aspects of a “good” design can be achieved. It has to be noted that the results presented here are not tied to a particular specification language. A similar generation process can be devised for specifications given
in a language like SQL or its extensions
Chapter 6

An Application: Analysis of Business Processes

In this chapter, we present an application of the transaction modification approach to the problem of analyzing business processes [Ham94]. Preliminary results of this work first appeared in [Ple95b]. Most of the machinery required for performing certain types of analysis, has already been introduced in the previous chapter. Some of the concepts are recast in a different context. The proofs of the results presented in this chapter follow closely their counterparts in the previous chapter.

6.1 Business Process Analysis

The role of Artificial Intelligence (AI) in Business Process Reengineering (BPR) is two-fold: AI can provide both the enabling technology for automatically reengineering processes and tools to support process redesign by the user [Ham94]. The majority of attempts to put AI to work in BPR have insisted on the first of these roles. It remains a challenging issue to develop tools for evaluating and for assisting the production of designs. There’s a growing requirement to represent processes through which work is achieved. Process representation thus becomes a vital issue in redesigning work and allocating responsibilities. Many languages aiming to assist in the requirements engineering activity of information systems development have been proposed (see for instance RML [Gre84], Albert [DDBDP94], KAOS [vLDM94] and Telos [MBJK90]). These languages address the problem of representing processes to a greater or lesser extent, however few of them provide a solid theoretical basis
for reasoning about processes. Several other languages intending to support process enactment (e.g., [SR92]) or process redesign (e.g., [Yu95]) have also been proposed. There is a notable shortage of languages that intend to support the analysis of processes. Despite the importance of the representation issue, we, in this chapter, will focus on the problems of simulating and analyzing business processes. We propose the use of a novel logic programming language as a tool that will support the analysis phase in process reengineering.

Unlike other areas of process modeling, e.g., software process modeling [HL88] where the initial emphasis has been put on the parts of processes that can be automated so that they can be machine executable, business process modeling attempts to capture phenomena enacted by humans rather than machines. This creates additional requirements, specifically for modeling actions that only involve acquiring or communicating knowledge or information. Reasoning about properties of business processes, such as, e.g., invariance or consistency with respect to a set of constraints, deadlock or communication bottleneck detection [Ell80], requires that actions of the above sort be formalized in addition to actions whose effect it is to change the state of affairs in the domain modeled. GOLOG [LLL+94] is a novel language suitable for representing business processes that also provides a theoretical platform for reasoning about both physical and perceptual actions, i.e., actions whose effect it is to change an agent’s knowledge of the state of affairs.

GOLOG was initially conceived as a language for high-level robot programming. Hence, in itself, GOLOG lacks the conceptual richness of languages such as Telos [MBJK90] or KAOS [vLDM94] that are intended to be used in requirements modeling and conceptual modeling in general. However, GOLOG can be coupled with a requirements modeling language and function within the simulation and analysis component of a business process management system (BPMS) [Kar94]. GOLOG can also support the process enactment phase, where activity plans can be scheduled. Then the viability of the synthesized plans can be tested.

For the purposes of this chapter, we will adopt a functional/behavioral representation of processes [CKO92]. We will focus on the sequencing of process elements or their iterations, as well as on necessary and sufficient conditions for process execution. These will be exploited for proving properties of processes within a formal framework. The majority of process modeling languages lack mathematical formality or employ a very low level thereof. The higher the degree of formal precision a language possesses, the easier the en-
actment of processes on a machine is. Business processes involve a relatively high degree of non-determinism, hence a formal language aiming to model business processes should incorporate machinery that goes beyond traditional programming paradigms.

A process modeling language should permit the evaluation of the adequacy of a proposed process. The model should be analyzable for properties such as syntactic correctness and consistency with respect to constraints. In this chapter, we argue that a formal method for analyzing the consistency of process specifications with respect to constraints can be devised by using a solution to the frame and ramification problems. These problems were introduced in the previous chapter and in the context of proving properties of database transactions. The existing approaches were reviewed there as well. Perceptual actions are dealt with in [LLL+94].

In the area of information systems development and in process modeling in particular, the frame and ramification problems have traditionally been either ignored or bypassed by means of explicit assumptions (see, e.g., [DDBDP94], [HL88]). In [BMR93], a systematic solution to the frame and ramification problems for determinate transaction specifications is presented. We will employ this solution for strengthening process specifications - at design time - to guarantee the maintenance of constraints by any implementation that meets the specifications. Moreover, as will be shown in the sequel, a solution to the ramification problem can be used for proving whether constraints are preserved or violated by action execution.

We now introduce an example that is used throughout the chapter for demonstrating the representation and simulation of actions and the process analysis method. We limit ourselves to presenting action specifications in an abstract language. We assume that actions are specified in terms of precondition/postcondition pairs, where both of these conditions are specified in a variant of first-order predicate calculus. The idea for the example below is borrowed from [DvLF93] although the specifications presented here are different. In this section, we only give an initial specification of representative actions and objects. Primed predicates or function terms will denote the respective (truth) values in the state resulting from the action execution. The example will be refined as needed in the sections to follow.

**Example 6.1.1** Consider a library management domain that includes as entities libraries, books and borrowers. Primitive actions include checking-out books, issuing reminders to borrowers, charging fines for late returns and so forth. A goal to be achieved in the library
**Action Check-Out** \((br, bk, dt)\)

**Precondition:** \(\text{available}(bk, dt) \land \text{requested}(bk, dt)\)

**Postcondition:**

\(-\text{available}'(bk, dt) \land \text{checkedOut}'(bk, dt) \land \text{borrows}'(br, bk, dt) \land (\#\text{borrows}'(br) = \#\text{borrows}(br) + 1)\)

end

**Action Issue-Reminder** \((br, bk, dt)\)

**Precondition:**

\(\exists d \ (\text{borrows}(br, bk, d) \land (\text{today}() - d > 2) \land \forall d' (d < d' < \text{today}()) \Rightarrow (\text{reminderIssued}(br, bk, d') \land \neg \text{returns}(br, bk, d')))\)

**Postcondition:** \(\text{reminderIssued}'(br, bk, dt)\)

end

Figure 6.1: Primitive action specifications in the library management system

management system will be to satisfy book requests. Figure 6.1 depicts the specifications of the actions **Check-Out** and **Issue-Reminder**. The following predicates and functions occur in action specifications: \(\text{borrows}(br, bk, dt)\), meaning that borrower \(br\) borrows book \(bk\) on date \(dt\), \(\text{requests}(br, bk, dt)\), meaning that prospective borrower \(br\) requests book \(bk\) on date \(dt\), \(\text{returns}(br, bk, dt)\), meaning that borrower \(br\) returns book \(bk\) on date \(dt\), \(\text{reminderIssued}(br, bk, dt)\), meaning that a reminder has been issued to borrower \(br\) on date \(dt\) concerning the loan of book \(bk\), \(\text{available}(bk, dt)\), meaning that a book is available on a certain date, \(\text{requested}(bk, dt)\), meaning that a book has been requested on a certain date, and \(\text{checkedOut}(bk, dt)\), meaning that a book is checked-out on a certain date. Functions \(\#\text{borrows}(br)\) and \(\text{limit}(br)\) return the number of books borrowed (and not yet returned) by a borrower and the limit on the number of books one can borrow respectively. For simplicity, we assume a very coarse time line where the basic unit is that of a day. The global function \(\text{today}()\) returns the current date.

The precondition of the action **Check-Out** states that the action is possible only if the book is available on that particular day and the prospective borrower has issued a request for it. The action postcondition expresses the immediate effects of the action on the predicates and functions occurring in the domain specification. Specifically, in the state resulting after
the action’s execution, the book is not available, it is checked-out by the borrower. The number of books the borrower has borrowed is updated and the predicate borrows becomes true of the borrower \( br \), the book \( bk \) and the date \( dt \). The second action, \textit{Issue-Reminder}, is possible only if a book has been borrowed and has not been returned for more than two weeks and no reminder has been issued in the meantime. Its postcondition makes the predicate \textit{reminderIssued} true of the action’s parameters. Given such specifications, we would like to be able to infer whether action execution, either in isolation or as part of processes involving other actions, may violate constraints expressing library policies. □

The rest of this chapter is organized as follows. Section 6.2 gives a short introduction to GOLOG and the situation calculus. The solution to the frame and ramification problems that is employed by the process analysis methods we propose is also sketched. Section 6.3 deals with the representation of business processes in GOLOG and the formulation of system goals by means of procedures in the language. Finally, section 6.4 presents our process analysis method which relies on the solution to the frame and ramification problems.

### 6.2 Situation Calculus and GOLOG

GOLOG [LLL+94] is a novel language that is suitable for defining and executing complex actions. The language was initially conceived for high-level robot control and is the result of extending the \textit{situation calculus} with perceptual and complex actions, including non-determinate ones. The development of GOLOG is part of an ongoing project in cognitive robotics that aims at the integration of reasoning, perception and action within a uniform framework that possesses a strong theoretical basis. The language has a situation calculus semantics and an interpreter that executes actions in a real or simulated environment.

Before presenting the complex action expressions that make up GOLOG, a few words should be said about the situation calculus. The situation calculus is a first-order language for representing dynamically evolving domains. Changes are brought to being in states of the world, \textit{situations}, as results of actions performed by an agent. A situation calculus structure thus contains a set \( A \) of actions and a set \( S \) of situations. For an action \( \alpha \in A \) and a situation \( s \in S \), the term \( do(\alpha, s) \) denotes the situation that results from the execution of action \( \alpha \) in situation \( s \). Relations whose truth values may differ from one situation to another are called \textit{fluents}. They are denoted by predicate symbols having a situation term
as their last argument. Similarly, the term functional fluent is used to denote functions whose denotation varies from one situation to another.

All actions in the situation calculus are assumed to be primitive and determinate. GOLOG permits the definition of complex actions through sequencing, iteration, non-deterministic choice of actions and non-deterministic choice of action parameters. Complex actions, or simply actions henceforth, are defined as follows: Simple actions are actions. If \( \alpha_1, \alpha_2 \) are actions, then \( [\alpha_1; \alpha_2] \) is the action that consists of \( \alpha_1 \) followed by \( \alpha_2 \), \( [\alpha_1|\alpha_2] \) is the action consisting of non-deterministically choosing between \( \alpha_1 \) and \( \alpha_2 \), \( \Pi_x(\alpha_1) \) denotes the non-deterministic choice of parameter \( x \) for \( \alpha_1 \). Other actions include, for a situation calculus formula \( \phi \), tests (?\( \phi \)), conditionals (if \( \phi \) then \( \alpha_1 \) else \( \alpha_2 \)) and iteration (while \( \phi \) do \( \alpha_1 \)). Complex actions are treated as macros for situation calculus expressions. A predicate \( Do(\alpha, s, s') \), where \( \alpha \) is an action and \( s, s' \in S \), is taken to mean that the agent's executing action \( \alpha \) in situation \( s \), leads to (a not necessarily unique) situation \( s' \).

GOLOG possesses the property that complex actions defined therein decompose into primitive actions. Moreover, the language interpreter is, in essence, a theorem prover that performs arbitrary first-order reasoning. This is required for executing actions that include tests, conditionals or iterations.

In GOLOG, one can express goals to be achieved by schematic plans. The details about how actions are to be performed are inferred by the theorem prover during the course of evaluating action conditions. In using GOLOG, the user provides a specification of primitive actions in terms of their preconditions and their effects on the world modeled. As will be shown in the sequel, such a specification along with a completeness assumption, suffice for performing process analysis for determining consistency of specifications with respect to constraints and for strengthening specifications, so that the process execution will not violate any of the constraints. Procedures comprising complex actions can then be written and executed by the system interpreter.

Because of the presence of arbitrary situation calculus formulae as parts of tests or while-loops in procedures, GOLOG maintains the world model by keeping track of the effects of actions and by modifying the model to reflect the knowledge acquired through perceptual actions. To reason about actions and their effects, a form of regression [Rei91] can be used to reduce conditions on arbitrary states to ones involving only the initial state. Alternatively, the progression of the knowledge base describing the world model is proposed.
in [LR92] as a less costly - in certain cases - method of bringing the model up to date.

### 6.2.1 The Frame and Ramification Problems

Before we proceed into the simulation and analysis of business processes with GOLOG, we briefly present the core of the theoretical framework behind GOLOG. This consists of a solution to the frame and ramification problems for complex and perceptual actions. We only sketch the solution and apply it to our working example. More details can be found in [Rei91], [LR92], [Pin94] and in chapter 5. Another detailed example is given in the appendix.

The solution relies on the generation of successor-state axioms for processes. As before, the intent of the generation method is to “compile” the constraints into the successor-state axioms. We use a many-sorted first-order language in which we distinguish one sort for situations; the rest of the sorts are object sorts \(^1\). Constraints are of the form:

\[
\forall x_1/S_1, \ldots, x_k/S_k, \forall t_1, t_2/S \phi(x_1, \ldots, x_k, t_1, t_2) \lor (-p_1(x_1, \ldots, x_k, t_1) \lor (-p_2(x_1, \ldots, x_k, t_2)
\]

where, \(p_1, p_2\) are \((k+1)\)-ary predicates, \(S_1, \ldots, S_k\) are object sorts and \(\phi(x_1, \ldots, x_k, t_1, t_2)\) is a formula in which variables \(x_1, \ldots, x_k, t_1, t_2\) occur free, if at all, and does not mention any predicate other than evaluable predicates. This class includes static constraints and transition constraints, but not general dynamic constraints. For example, the transition constraint specifying the property that an employee’s salary can never decrease, can be specified by the formula

\[
\forall e/Employee \ \forall s_1, s_2/\text{Salary} \ \forall t_1, t_2/S \ (s_1 < s_2) \lor -(t_1 < t_2) \lor \\
\neg \text{salary}(e, s_1, t_1) \lor \neg \text{salary}(e, s_2, t_2)
\]

An action \(\alpha\) with parameters \(\pi\) is specified by a pair \((\text{pre}_\alpha(\pi), \text{post}_\alpha(\pi))\), where \(\text{pre}_\alpha(\pi)\) and \(\text{post}_\alpha(\pi)\) denote the action pre- and post-condition respectively.

The generator assumes the existence of causal rules describing the direct effects of transactions. These rules, called the direct effect axioms, have the form \(\forall \pi (\text{Occur}(\alpha(\pi), s) \Rightarrow \text{pre}_\alpha(\pi, s) \land \text{post}_\alpha(\pi, \text{do}(\alpha, s)))\). Given any action specification, the effect axioms are derived independently of any other action specification. Hence, we can avoid having to specify the axioms in a reified logic that interprets the predicate \(\text{Occur}\) outside a stan-

\(^1\)This language is a variant of MTL that was used in the previous chapter for the specification of database constraints and transactions.
standard first-order interpretation. The above axiom can now be written as: \( \forall x (\varphi(x, s) \Rightarrow \text{pre}_\alpha(x, s) \land \text{post}_\alpha(x, \text{do}(\alpha, s))) \). For instance, from the specification of the action Check-out in figure 6.1, we can derive the following direct effect axiom\(^2\):

\[
\forall br/Borrower \forall bk/Book \forall dt/Date \ (\text{Check-Out}(br, bk, dt) \Rightarrow \text{available}(bk, dt) \land \text{requested}(bk, dt) \land \neg \text{available}'(bk, dt) \land \text{checkedOut}'(bk, dt) \land \text{borrows}'(br, bk, dt) \land \#\text{borrows}'(br) = \#\text{borrows}(br) + 1)
\]

From the direct effect axioms we can systematically generate positive and negative effect axioms [BMR93] for every predicate \( P \) that occurs in \( \text{post}_\alpha \). A positive (negative) effect axiom for a predicate \( P \) expresses necessary conditions for the change of the truth value of \( P \) from True (False) to False (True respectively) during the transition from a situation \( s \) to the situation \( \text{do}(\alpha, s) \) resulting from the execution of action \( \alpha \). The systematic generation of positive and negative effect axioms follows the lines of section 5.1.2. Here, we only show the effect axioms derived for the predicate \( \text{borrows} \), under the assumption that the only actions available are the actions Check-Out and Issue-Reminder:

\[
\forall br/Borrower \forall bk/Book \forall dt/Date \neg \text{borrows}(br, bk, dt) \land \#\text{borrows}'(br, bk, dt) \land \\
\text{Check-Out}(br, bk, dt) \Rightarrow \text{available}(bk, dt) \land \text{requested}(bk, dt) \land \\
\neg \text{available}'(bk, dt) \land \text{checkedOut}'(bk, dt) \land \text{borrows}'(br, bk, dt) \land \\
(\#\text{borrows}'(br) = \#\text{borrows}(br) + 1) \tag{2}
\]

\[
\forall br/Borrower \forall bk/Book \forall dt/Date \text{borrows}(br, bk, dt) \land \neg \text{borrows}'(br, bk, dt) \land \\
\text{Issue-Reminder}(br, bk, dt) \Rightarrow \text{False} \tag{3}
\]

The consequent of the negative effect axiom is \( \text{False} \) since there is no action specification with the effect of falsifying the fluent \( \text{borrows} \).

In addition to the effect axioms, the knowledge base is augmented with unique name axioms for actions, predicates and functions.

The next step is the generation of successor-state axioms (see section 5.1.2 for a formal presentation). As mentioned earlier, successor-state axioms characterize all conditions under which fluents may change truth value. Thus, the generation of a set of successor-state axioms for the fluents of a particular domain constitutes a solution to the frame problem.

\(^2\)We use the primed/unprimed notation here to avoid the quantification over situations.
To account for constraint ramifications, the fluents occurring in the expression of constraints are considered as well.

For example, assume that the following constraint is defined:

\[
\forall br/Borrower \exists bk/Book \exists dt/Date \rightarrow \text{borrows}(br, bk, dt) \lor \neg \text{returns}(br, bk, dt)
\]

It expresses the property that, on a certain date, a person may be only borrowing or returning a book. For brevity in presentation, let us rewrite the positive effect axiom (2) for \text{borrows} as

\[
\forall br/Borrower \exists bk/Book \exists dt/Date \rightarrow \text{borrows}(br, bk, dt) \land \text{returns}(br, bk, dt) \rightarrow \phi \quad (2')
\]

where \( \phi \) is the formula

\[
\text{Check-Out}(br, bk, dt) \rightarrow \text{available}(bk, dt) \land \text{requested}(bk, dt) \land \neg \text{available}'(bk, dt) \land \text{checkedOut}'(bk, dt) \land \text{borrows}'(br, bk, dt) \land (\#\text{borrows}'(br) = \#\text{borrows}(br) + 1)
\]

The syntactic generator will generate the following successor-state axiom for the predicate \text{borrows}:

\[
\forall br/Borrower \exists bk/Book \exists dt/Date \rightarrow \neg \text{borrows}(br, bk, dt) \land \neg \phi \land \text{returns}(br, bk, dt) \rightarrow
\neg \text{borrows}'(br, bk, dt)
\]

or, equivalently,

\[
\forall br/Borrower \exists bk/Book \exists dt/Date \rightarrow \neg \text{borrows}(br, bk, dt) \land \text{borrows}'(br, bk, dt) \rightarrow
\phi \lor \neg \text{returns}(br, bk, dt)
\]  

(3)

Axiom (3) captures the ramification \( \neg \phi \land \text{returns}(br, bk, dt) \) of (a form of) the constraint and the postcondition of the action \text{Check-Out} as a necessary condition for the change in the truth value of the predicate \text{borrows}.

Finally, we show how knowledge producing actions are taken into account [LLL+94]. The successor-state axioms in this case only state that a particular predicate or function term becomes known as a result of a knowledge producing action taking place.

An accessibility relation \( K \) between situations is introduced. For \( s, s' \) in \( S \), \( K(s', s) \) means that situation \( s' \) is accessible from \( s \). Then, if actions \( \alpha_1, \ldots, \alpha_n \) are all knowledge producing actions and each one is associated with a formula \( \phi_i \) expressing the predicate or function term that becomes known, then the successor state axiom for \( K \) is as follows:
**Action** Request \((br, bk, dt)\)

**Precondition:** True

**Postcondition:** \(\text{requests}(br, bk, dt)\)

end

Figure 6.2: Specification of the library management system (cont’d)

\[
\forall s, s''(K(s''), \text{do}(\alpha, s)) \equiv
\exists s'(K(s', s) \land (s'' = \text{do}(\alpha, s')) \land ((\alpha = \alpha_1) \rightarrow \phi_1) \land \ldots \land ((\alpha = \alpha_n) \rightarrow \phi_n))]
\]

The correctness of the syntactic generation has been proven in [LR92], [BMR93], [Pin94], [PM96] and in chapter 5 of this dissertation.

### 6.3 Process Simulation

In this section, we present the simulation of complex business processes in GOLOG. The language permits the representation of goals to be achieved as programs consisting of actions that achieve subgoals of the original goal. The use of GOLOG language constructs is illustrated through process specification in the library management example.

First, a specification of the application domain is needed. This consists of the definition of primitive actions and the derivation of the action precondition and effect axioms. Both types of axioms can be derived from action specifications of the type used in the examples of section 6.1.

We consider as primitive the actions *Check-Out* and *Issue-Reminder* of example 6.1.1, as well as the action *Request* which is specified in figure 6.2. We assume that the action *Request* is always possible. Figure 6.2 depicts the specification of the new actions. From this specification of actions, the effect axioms can be easily derived as described in section 6.2.1. For example, since the fluent *requests* only occurs positively in the postcondition of action *Request*, the (positive) effect axiom for *Request* will have the form:

\[
\forall br, bk, dt \neg \text{requests}(br, bk, dt) \land \text{Request}(br, bk, dt) \Rightarrow \text{requests}'(br, bk, dt)
\]

Now we are in a position to write procedures in GOLOG that implement goals in the library management system. For instance, the goal *SatisfyRequests* expresses the requirement that the system should serve all requests for book loans while such requests exist. The GOLOG procedure *SatisfyRequests* first “senses” all requests (if such exist) for book
loans on a particular day. While there exists a request to be served, the system non-deterministically chooses some request and checks if it can be satisfied. The procedure is shown in figure 6.3. \textit{Check – Status} is the name of another GOLOG procedure that is called by \textit{SatisfyRequests}. The specification of the procedure is shown in figure 6.4. The procedure checks if an unavailable book is overdue. If this is the case, then it issues a reminder to its borrower.

In a similar fashion, we can write procedures in GOLOG to achieve all goals that are set forth in the system’s requirements specification. As shown in procedure \textit{SatisfyRequests}, non-deterministic and knowledge producing actions can be part of procedure specifications. These specifications can be also executed by a GOLOG interpreter. What is needed apart from the domain axiomatization and the procedure specification, is the specification of the initial state. The specification of the initial state need not be complete. This gives the ability to model incompletely known domains as well. The domain modeled is maintained by the interpreter by regression, so that testing of conditions on arbitrary domain state, amount to evaluating conditions on the initial state only.

Albeit short and simple, the above example aims at depicting the advantages of the specification of complex business processes in a high-level programming language. Firstly,
:- op(950,xfy,[:]). /* Event sequence. */
:- op(950,xfy,[#]). /* Nondeterministic event choice. */

/* Primitive events */
primitive_event(checkOut(Br,Bk)).
primitive_event(issueReminder(Br,Bk)).
primitive_event(request(Br,Bk)).

/* Process SatisfyRequests */
proc(satisfyRequests,
    while(some(Br,(some Bk, requests(Br,Bk)))).
    pi([[Br,Bk], if (and(requests(Br,Bk), available(Bk))],
      checkOut(Br,Bk), checkStatus(Bk)))).

/* Preconditions for Primitive Events */
poss(checkOut(Br,Bk),S):- available(Bk,S), requested(Bk,S).
poss(issueReminder(Br,Bk),S):- borrows(Br,Bk,S1), today>S1, 
    not(reminderIssued(Br,Bk,S2), S1<S2, S2<today).
poss(request(Br,Bk),S).

/* Successor state axioms for primitive events. */
requests(Br,Bk,do(A,S)):- A=request(Br,Bk); not A=request(Br,Bk), 
    requests(Br,Bk,S).

reminderIssued(Br,Bk,do(A,S)):- A=issueReminder(Br,Bk);
    not A=issueReminder(Br,Bk), reminderIssued(Br,Bk,S).
...
...

Figure 6.5: Event and axiom specifications for the library management example
perceptual actions, that are highly likely to be part of a business process, can be included in the procedures. In addition, a formal account of such actions exists. Secondly, non-deterministic actions can be specified. Thirdly, processes can be written in a modular manner and be executed via an interpreter. These procedures specify how goals are to be achieved without necessarily specifying all the details about how the goal is to be achieved. It's the responsibility of the theorem-prover to fill all the details about action execution.

The interpreter of GOLOG accepts definitions of primitive and complex actions in Prolog. The specifier must also supply the precondition and effect for primitive events. The successor-state axioms can then be compiled automatically for the effect axioms. Figure 6.5 contains part of an initial specification of the library management example in GOLOG, and under the assumption that situations are identified with the time parameters of the specifications. The simulation is rather simplistic at this stage, as compared to other simulation tools (e.g., [DDBD94]), which do not perform theorem proving or plan generation. The interpreter returns a complete trace of the execution of processes, showing the actions that took place since the initial situation. As a side-effect of the evaluation of conditions in tests or while-loops, a complete plan that achieves the goals that the procedures implement is returned. The declarative approach to process specification permits the simulation of processes even when only partial specifications are given.

6.4 Process Analysis

In this section, we present a formal technique intended to assist in the analysis phase of business process reengineering. The method is used to determine whether consistency with respect to defined constraints is preserved or violated as a result of process execution. In cases where such proof or disproof is not possible at process specification time, strengthenings to specifications of actions that are relevant to the constraints are proposed so that, any implementation meeting the strengthened specifications provably guarantees that constraints will not be violated in the state resulting from action execution. The technique is based on the assumptions that primitive actions are determinate and that all action specifications are known. Primitive actions are considered to be the basic means of effecting change. Processes are formed by combining primitive actions. Our results show that, if primitive actions can be proved to be correct with respect to the maintenance of invariants,
then the synthesized processes are correct. In effect, invariants cannot be violated at the beginning or end of each atomic action, and, thus, at the beginning or end of processes.

We argue that providing such a functionality to the process analysis component of a BPMS, is essential for the process reengineering effort. The process specifier realizes the implications of actions and the implementor is saved the burden of finding ways to meet the postcondition and maintain the invariant. Consistency is guaranteed by the correctness of an analysis tool such as the one presented here. Moreover, optimized forms of conditions to be verified can be incorporated into process specifications.

Constraints serve the role of invariants of actions. No action may violate the relevant constraints in order to be accepted. The problem of proving that action execution does not violate its invariants is formalized in the following definition. The following definition and results are based on the premise that a knowledge base over which actions take place is expressed as a first-order theory.

**Definition 6.4.1 (Invariant Maintenance)**

Let $A$ be an action specification with precondition $P$, postcondition $Q$ and invariant $I$. $A$ is said to maintain invariant $I$ when executed over a knowledge base $KB$, if $KB \cup \{I, P\} \models (Q \Rightarrow I')$, where $I'$ denotes the precondition in the state resulting after the transaction takes place.

Proving that actions maintain invariants is a difficult task since it requires theorem proving. A way to avoid checking whether actions maintain their invariants is to augment their specifications in a way such that the invariant is maintained as a result of meeting the new specification. We consider the single-action case first. Assume an action specification $A$ includes a pair $(P, Q)$ of a precondition and a postcondition expressed in first-order predicate calculus. Let $I$ be an integrity constraint relevant to the action$^3$. We need to find a formula $N$ such that $(Q \land N \Rightarrow I')$, or equivalently $(Q \land \neg I' \Rightarrow \neg N)$ is logically implied by the knowledge base and the axiomatization of the domain. If $\neg N$ is a ramification of $Q \land \neg I'$, then the desired entailment relationship holds. This leads to the following theorem whose proof is an easy consequence of the definition of ramifications and invariant maintenance$^4$:

---

$^3$The notion of "relevance" is defined formally in the sequel.

$^4$Chapter 5 and [PM96] contain similar results for the case of database transactions. The proof of theorem 6.4.1 and its corollaries follow the lines of theorem 5.2.1 and its corollaries respectively.
**Theorem 6.4.1** Let $A$ be an action specification including a precondition/postcondition pair $(P, Q)$ and let $I$ be an invariant of $A$. If $R$ is a ramification of $Q \land \neg I'$, then the invariant is maintained in the state resulting after the action’s execution if the postcondition $Q \land \neg R$ is met.

Although the transformation of postconditions is unidirectional, it is sufficient for proving constraint maintenance. It is assumed that actions are always executed in states that respect the constraints. Satisfaction of a constraint’s ramification does not, in general, imply the constraint’s satisfaction; if a ramification of a constraint is violated, the constraint itself is violated. Moreover, the ramifications of constraints can be simpler formulae than the constraints themselves.

Certain ramifications can suggest that invariants do not have to be checked and postconditions need not be modified in order to guarantee the invariants. The case in which the propositional constant $False$ is derived as a ramification, is of particular interest since, as the following corollary specifies, no change in the postcondition is needed in order to meet the invariant.

**Corollary 6.4.1** If $False$ is derived as a ramification of $Q \land \neg I'$, then $I'$ is maintained by any action execution meeting $Q$. If $True$ is derived as a ramification of $Q \land \neg I'$, then the set \{Q, I'\} is inconsistent.

Hence, the process of generating ramifications can also discover inconsistent specifications of actions that may have escaped the specifier’s attention. The following example will help illustrate the application of the ramification method.

**Example 6.4.1** Assume that the following constraint has been defined in the library management system for the purpose of maximizing the number of book requests the library can serve:

$$I \equiv (\forall br/Borrower \#borrows(br) \leq \text{limit}(br))$$

The constraint expresses the property that there exists an upper limit on the number of books a borrower can borrow from the library. Clearly, the constraint can be violated as a result of executing the action *Check-Out*. Violation of the constraint can be prevented if the action postcondition is strengthened by conjoining it with a ramification of the constraint. The constraint in the state resulting from the action execution is expressed as
\[ I' \equiv \#borrowed'(br) \leq limit(br) \]. Then, we form the conjunction of the negated invariant and the action postcondition:

\[
- I' \land Post \equiv - (\#borrowed'(br) \leq limit(br)) \land - \text{available}'(bk, \text{dt}) \land \text{checkedOut}'(bk, \text{dt}) \land \text{borrows}'(br, bk, \text{dt}) \land (\#\text{borrows}'(br) = \#\text{borrows}(br) + 1)
\]

Exploiting the fact that after an action's execution it's direct effects, as these are specified by the postcondition, become true, we can derive \#\text{borrowed}'(br) \geq limit(br) as a consequence of the above conjunction. Given that \#\text{borrowed}'(br) = \#\text{borrowed}(br) + 1, we derive \#\text{borrowed}(br) \leq limit(br) - 1 as the condition that suffices to be added to the postcondition of action Check-Out in order not to violate the constraint. 

The above example intends to convey the idea behind using the ramification method for strengthening action specifications. The process of deriving ramifications can be automated as described in section 6.2.1. The ramifications required to augment the postconditions are part of the successor-state axioms derived. Appendix A shows the application of the method to another example.

We would like to be able to propose similar augmentations to postconditions when the invariant refers to any number of states, both before and after an action takes place. In other words, we need to extend the method to take into account dynamic constraints. The solution to the ramification problem, as formulated in [Pin94], does not deal with constraints more general than transitional ones. We have extended the syntactic generator of ramifications to a more general class of constraints, as shown in chapter 5 and in [PM96], where some initial results were given through examples. The extension of the method to general dynamic constraints is a topic of current research.

The above results concerned the single-action single-constraint case. In the remainder of this section, we show how the method applies to the case of multiple actions, multiple constraints, conjunction and inheritance of specifications.

### 6.4.1 Multiple Actions and Multiple Constraints

In this case, all action specifications have to be taken into account since a constraint may be relevant to, or affected by, more than one actions. The notion of relevance is formally defined here. For that we will assume that the specifications are given in first-order predicate
calculus (ordinary or temporal). Moreover, we will assume that no interleaving of actions is allowed. An action is regarded as the only means of state change.

**Definition 6.4.2** Relevance A constraint $I$ is relevant to an action specification $A$ that includes a precondition/postcondition pair $(pre, post)$, if $post$ contains a literal that occurs in $I$.

A constraint relevant to a set of action specifications $A_i$ with respective pre/post-conditions $(pre_i, post_i)$ has to be considered for the modification of each $post_i$, so that the execution of any action provably maintains the constraint. Hence, it suffices to repeat the process presented before for every $A_i$. The process may be optimized by reusing the derivation of ramifications for actions whose postconditions involve common predicates.

The symmetric case, where an action specification is associated with more than one invariants, is dealt with by simply taking the conjunction of the invariants as the new invariant. Then, the derived ramification depends on all invariants, provided that the union of the invariants is a satisfiable set.

### 6.4.2 Conjoining Action Specifications

The conjunction of specifications $\alpha_1, \alpha_2$ - denoted by $\alpha_1 \parallel \alpha_2$ - is formed by conjoining the respective pre/post-conditions. Conjunction of action specifications corresponds to the sequencing $\alpha_1 \cdot \alpha_2$ of $\alpha_1$ and $\alpha_2$. Then, as theorem 6.4.2 suggests, it suffices to conjoin the ramifications of the two invariant-postcondition pairs, to guarantee that the invariant will be maintained if the new postcondition is met. The theorem provides a sufficient condition for the maintenance of constraints by sequences of actions.

**Theorem 6.4.2** Let $A_1, A_2$ be action specifications with pre/post-conditions $(pre_1, post_1)$ and $A_2 = (pre_2, post_2)$ respectively. Let $I$ be a constraint relevant to both $A_1$ and $A_2$. If there exist ramifications $N_1$ and $N_2$ which, if conjoined with the postconditions $post_1$ and $post_2$ guarantee the maintenance of $I$ in $A_1$ and $A_2$ respectively, then $N_1 \land N_2$ is a ramification which, if conjoined with $post_1 \land post_2$ guarantees the maintenance of $I$ in $A = A_1 \parallel A_2$.

An important consequence of theorem 6.4.2 is the ability to accommodate new invariants without having to redo the entire process. Specifically, if a new invariant is to be added and
is relevant to an action specification whose postcondition has already been augmented by computed ramifications, it suffices to verify that the new invariant does not introduce any contradiction, and, if this is the case, to generate ramifications of the new invariant and the postcondition. The new ramification can be conjoined with the previously derived ones, so that the new postcondition guarantees the invariants.

6.4.3 Inheritance of Action Specifications

Refinement of business processes can be accomplished via specialization in object-oriented specification languages. The inheritance of action specifications implies that the pre/post-conditions of the specialized actions are conditions stronger than those of the more general action, and, hence, include the pre/post-conditions of the more general action as conjuncts in their expression.

Assume action specification $A_2$ with pre/post-conditions $(pre_2, post_2)$ is a specialization of action specification $A_1$ with pre/post-conditions $(pre_1, post_1)$ and that there exists a ramification $N_1$ with the desired property of maintaining the action's invariants $I$ if used to augment the postcondition $post_1$. The specification $A_1$ is inherited by $A_2$. If neither of the conjunctions $(pre_1 \land pre_2)$ and $(post_1 \land post_2)$ is a contradiction, then, according to theorem 6.4.2, if a ramification $N_2$ can be found, then augmenting $post_2$ with $N_1 \land N_2$ suffices to guarantee that the invariant will be maintained if the strengthened postcondition is met.

6.5 Summary

This chapter have presented an adaptation of ideas from AI for the development of tools intended to assist in the reengineering of business processes. Specifically, we elaborated on the use of a high-level logic programming language, GOLOG, for the specification and simulation of business processes. The language's features for specifying sequences or iterations of actions, non-deterministic and perceptual actions make it suitable for specifying business processes. Moreover, GOLOG can be used for the simulation of processes and for proving schematic plans of actions correct. Most importantly, the theoretical basis of GOLOG which includes a solution to the frame and ramification problems, permits us to develop formal techniques for process analysis. We proposed techniques for verifying the consistency of specifications with respect to the defined constraints and for strengthening
action specifications so that constraints are provably guaranteed by any implementation that meets the new action specifications. We also demonstrated the ability to incrementally accommodate new invariants or action specifications and the conjunction and inheritance of process specifications. We argue that providing tools of this kind is essential for the development of business process management systems.
Chapter 7

Concluding Remarks

7.1 Summary

In this dissertation, new methods for the maintenance of temporal integrity in knowledge bases were proposed. We were motivated by the pragmatic demands of new applications and the inability of existing methods to provide satisfactory solutions to the problem in question. Generality and efficiency have been the main objectives of this work. We argue that such techniques can be adopted in the development of knowledge base management systems with the potential of providing integrity maintenance with acceptable performance. To the best of our knowledge, the problem has not been examined before in as general a context.

We first introduced a language for defining temporal assertions. The language is an extension of the assertion language of Telos [MBJK90], which is based on Allen’s interval-based model [All83]. It incorporates history and belief time for modeling both the history of the evolution of the domain of discourse, as well as the system’s knowledge of this history. The language permits the specification of integrity constraints that are not specifiable in other formalisms with a single or no notion of time. Constraints definable in our assertion language are classifiable along several dimension, each one giving rise to different requirements as far as integrity maintenance is concerned. We also defined appropriate notions of constraint satisfaction in a bitemporal context.

The first of the proposed methods is a two-phase simplification method for temporal integrity monitoring in large knowledge bases and in the presence of deductive rules. Compile-time simplifications are applied uniformly to static and dynamic constraints. The
proposed technique focuses on simplifying formulae as much as possible at schema definition time, and aim at yielding acceptable performance at update time. During compilation integrity constraints and deductive rules are simplified with respect to the anticipated types of updates and organized in a graph structure that permits the efficient derivation of implicit updates. This structure is incrementally modifiable for accommodating dynamic changes to the rule and constraint set. Preliminary performance results showed that the maintenance of this scheme can be carried out efficiently. Moreover, additional optimizations on the formulae expressing the simplified forms generated at compile-time were proposed. These optimizations minimize the temporal intervals over which the validity of formulae has to be checked in order for their satisfaction to be verified at update-time. They exploit the semantics of constraint satisfaction and the organization of simplified forms in the dependence graph structure.

Another contribution is the simplification of temporal formulae with respect to arbitrary transactions. The simplified formulae produced by the compile-time simplification method are used in order to derive a single formula that suffices to be verified at update-time when a transaction takes place. Hence, a single formula incorporating the affecting updates of an arbitrary transaction needs only to be verified, as opposed to a number of more complex formulae incorporating only individual updates.

Subsequently, we followed an orthogonal direction into devising an integrity maintenance method using transaction modification. We started by showing that the use of a solution to the frame and ramification problem can provide valuable feedback during the transaction design phase and suggested a new technique for integrity maintenance in knowledge bases. In particular, we presented an adaptation of ideas from artificial intelligence into the problem of maintaining the integrity of a knowledge base. We extended a systematic solution to the frame and ramification problem for simplifying - at compile time - the task of proving that transaction execution does not violate the integrity constraints. This became possible by the syntactic generation of successor state axioms which, for the case of determinate transaction specifications, completely characterize under which circumstances predicates or functions change value after a state transition. We extended the method to apply to a class of constraints which includes static and transition constraints, and showed that the introduction of new transaction specifications or new constraints can be accommodated incrementally. This technique can lead to the development of a tool which, as part of
a knowledge base management system [MCP+96], will suggest additions to transaction postconditions, whose effect will be to maintain the invariants. This tool will assist the knowledge base design process by providing feedback to the designer of transactions and by automating the task of verifying the safety of transactions. Furthermore, a new technique for integrity constraint checking is suggested. Checking for the satisfaction of constraints after each transaction execution is saved for the constraints for which, it can be decided at compile-time that they remain provably true. Similarly, for those that are provably violated, the cost of undoing transactions is saved. Otherwise, a simpler condition sufficient for ensuring that constraints will not be violated is derivable. The proposed method is a promising avenue for achieving acceptable run-time performance for integrity maintenance in large knowledge bases.

Finally, we presented the application of the transaction modification approach to the development of tools intended to assist in the reengineering of business processes. Specifically, we elaborated on the use of a high-level logic programming language, GOLOG, for the specification and simulation of business processes. The language’s features for specifying sequences or iterations of actions, non-deterministic and perceptual actions make it suitable for specifying business processes. Moreover, GOLOG can be used for the simulation of processes and for proving schematic plans of actions correct. Most importantly, the theoretical basis of GOLOG which includes a solution to the frame and ramification problems, permits us to develop formal techniques for process analysis. We proposed techniques for verifying the consistency of specifications with respect to the defined constraints and for strengthening action specifications so that constraints are provably maintained by any implementation that meets the new action specifications. We also demonstrated the ability to incrementally accommodate new invariants or action specifications and the conjunction and inheritance of process specifications. We argue that providing tools of this kind is essential for the development of business process management systems.

7.2 Outlook

Many research issues arising from this work warrant further investigation. We have yet to define a maximal tractable class of constraints specifyable in our assertion language and provide an exact characterization of its expressive power, as well as of the complexity of
formula evaluation.

Current research focuses on devising historical knowledge minimization techniques for integrity constraints, in the lines of [Cho92a] and [SL88a], but in the richer context of knowledge bases introduced in this paper. We feel that additional performance gains can be obtained by the reuse and adaptation of proofs of temporal formulae satisfaction over dependence graphs. The graphs have the property of representing derivation paths for implicitly derived literals. Furthermore, the derivation of incrementally evaluable formulae as simplified forms of constraints expressed in the interval based assertion language is examined. Incrementally evaluable formulae lend themselves to an implementation using an active rule model, as, e.g., in [GL93], [CT94] and [SW95b]. The combination of compile-time optimizations with the efficiency of active rules is a promising avenue towards providing the functionality of integrity maintenance.

Temporal formulae evaluation will benefit from the existence of an efficient hybrid theorem prover for the evaluation of temporal constraints, in the flavor of [MS90]. Such a theorem prover may be enhanced with techniques for reduction of temporal formulae referring to long histories into formulae evaluable a pair of states only [SL88b]. The performance assessment of the method must be completed. The method needs to be compared against one-phase methods that interleave simplification and evaluation [Kuc91] and run-time methods. Moreover, the I/O complexity of graph computations and the secondary storage of rules and constraints are issues that we intend to pursue further.

As far as the orthogonal direction of transaction modification is concerned, we are currently investigating the possibility of devising similar syntactic generators for general dynamic constraints, as well as the application of additional optimization steps for generating simpler conditions that suffice to guarantee invariant maintenance. For the case of temporal constraints in particular, the minimization of the temporal information required to verify the constraints appears to be possible by using knowledge about the satisfaction of the constraints in the history of states up to the current state.

The last of the chapters presented the application of transaction modification in the area of business process analysis. A number of issues need to be looked further into. In particular, we intend to extend the current results for proving properties of processes synthesized by any of the constructs provided by GOLOG. A concurrent version of GOLOG is also under development. A solution to the frame and ramification problems for the case of concurrent
actions will provide the basis for extending the results presented in this paper for the case of concurrent processes. The time-sensitive nature of requirements that arise in business processes require that optimizations of the sort proposed here, should be extended to deal with arbitrary dynamic and real-time constraints. Last, but not least, we are looking into interfacing GOLOG with a more abstract language for process definition so that the precondition and successor-state axioms can be derived automatically from the specification of processes.
Bibliography


Also available as Technical Report KRR-90-7 on Knowledge Representation and Reasoning.


Appendix A

Example: A Simple Elevator Controller

In this section, we depict the application of specification strengthening to a simple example borrowed from [DvlF93]. Constraints here are expressed in a variant of first-order temporal logic [MP91]. The example aims at showing the applicability of the proposed analysis techniques to the case where specifications are given in a temporal logic. Hence, we will focus on postcondition strengthening rather than giving a fairly complete specification of the domain in question.

Let us assume that the constraint expressing the requirement that doors be closed while the elevator is moving ensures the abstract goal of safe transportation in the elevator system. The constraint is formally expressed as the following many-sorted temporal formula, where the operators $\circ$ and • mean “in then next state” and “in the previous state” respectively:

$I \equiv \forall l/Lift \, d/Door \, f, f’/Floor \, PartOf(d, l) \Rightarrow (LiftAt(l, f) \land \circ LiftAt(l, f’) \land (f \neq f')) \Rightarrow ((d.state = \text{"closed"}) \land \circ (d.state = \text{"closed"})))$

For simplicity, let us assume that the only actions relevant to the constraints are the actions $\textit{GotoFloor}$ and $\textit{OpenDoors}$, whose specification is given in figure A.1. We need to add the axiom

$$\forall d/Door \, (d.state = \text{"open"}) \Rightarrow \neg (d.state = \text{"closed"})$$

in order to be able to reason about the state of elevator doors.

We first consider the action $\textit{GotoFloor}$ for the derivation of ramifications. The first step is to instantiate the constraint $I$ in the state after the action’s execution. This amounts to
Action \textit{GotoFloor} \((l, f, f')\)

\begin{description}
\item[Precondition:] \textit{LiftAt}(l, f)
\item[Postcondition:] \textit{LiftAt}(l, f') \land (f \neq f')
\end{description}

end

Action \textit{OpenDoors}(l, d)

\begin{description}
\item[Precondition:] \textit{PartOf}(d, l) \land (d.\textit{state} = \textit{closed'})
\item[Postcondition:] (d.\textit{state} = \textit{open'})
\end{description}

end

Figure A.1: Specification of an Elevator Controller

parameterizing the universally quantified variables corresponding to the parameters of the action and to replacing every occurrence of \(oP\) by \(P\) and every occurrence of \(P\) with \(\bullet P\) for every predicate \(P\). Negating the instantiated constraint and conjoining it with the action’s postcondition yields

\[-I' \land \text{Post} \equiv \textit{PartOf}(d, l) \land \textit{LiftAt}(l, f) \land \textit{LiftAt}(l, f') \land (f \neq f') \land (\neg \bullet (d.\textit{state} = \textit{closed'}) \lor \neg (d.\textit{state} = \textit{closed'}))\]

By exploiting the facts that the predicates occurring positively in the action’s postcondition must evaluate to \textit{True} in the state after the action’s execution and that the precondition must have been satisfied, \(-I' \land \text{Post}\) implies the formula

\[-\bullet \textit{PartOf}(d, l) \land (\neg \bullet (d.\textit{state} = \textit{closed'}) \lor \neg (d.\textit{state} = \textit{closed'}))\]

We now exploit the assumption that the constraint was known to be satisfied in the state prior to the action’s execution. This means that either the antecedent \(\bullet \textit{PartOf}(d, l)\) of \(I\) is \textit{False} or, both the antecedent of \(I\) and its consequent are \textit{True}. In the former case, we derive \textit{False} as a ramification and we can conclude that the constraint remain unaffected by the transaction. In the latter case, \(\bullet \textit{PartOf}(d, l)\) can be substituted by \textit{True} and the derived formula is \(-R \equiv \neg \bullet (d.\textit{state} = \textit{closed'}) \lor \neg (d.\textit{state} = \textit{closed'}),\) which yield the ramification \(R \equiv \bullet (d.\textit{state} = \textit{closed'}) \land (d.\textit{state} = \textit{closed'}).\) Intuitively, this means that both in the state before and in the state after the action execution, the state of the elevator door must be “closed”.

Similarly, we can derive ramifications with respect to the action \textit{OpenDoors}. In this case, the ramification derived is expressed by the formula
Action $GotoFloor( l, f, f' )$

**Precondition:** $LiftAt(l, f) \land (d.\text{state} = \text{“closed”})$

**Postcondition:** $LiftAt(l, f') \land (f \neq f') \land (d.\text{state} = \text{“closed”})$

end

Action $OpenDoors(l, d)$

**Precondition:** $PartOf(d, l) \land (d.\text{state} = \text{“closed”})$

**Postcondition:** $PartOf(d, l) \land (d.\text{state} = \text{“open”}) \land \exists f, f'/Floor\ (\bullet LiftAt(l, f) \land LiftAt(l, f')) \Rightarrow \neg (f \neq f')$

end

Figure A.2: Strengthened Specification of an Elevator Controller

\[ R \equiv \exists f, f'/Floor\ (\bullet LiftAt(l, f) \land LiftAt(l, f')) \Rightarrow \neg (f \neq f') \]

which means that both before and after the action execution the elevator must be at the same floor. It is easy to verify that if we strengthen the action specifications by conjoining their postconditions with the derived ramifications, then the constraint is maintained by any implementation of the actions that meet the new specifications. Figure A.2 shows the strengthened specification of the actions $GotoFloor$ and $OpenDoors$. 