Integrity Constraint and Rule Maintenance in Temporal Deductive Knowledge Bases

Dimitris Plexousakis
Department of Computer Science
University of Toronto
Toronto, Ont. M5S 1A4, Canada
E-mail: dp@cs.toronto.edu

Abstract

The enforcement of semantic integrity constraints in data and knowledge bases constitutes a major performance bottleneck. Integrity constraint simplification methods aim at reducing the complexity of formula evaluation at run-time. This paper proposes such a simplification method for large and semantically rich knowledge bases. Structural, temporal and assertional knowledge in the form of deductive rules and integrity constraints, is represented in Telos, a hybrid language for knowledge representation. A compilation method performs a number of syntactic, semantic and temporal transformations to integrity constraints and deductive rules, and organizes simplified forms in a dependence graph that allows for efficient computation of implicit updates. Precomputation of potential implicit updates at compile time is possible by computing the dependence graph transitive closure. To account for dynamic changes to the dependence graph by updates of constraints and rules, we propose efficient algorithms for the incremental maintenance of the computed transitive closure.

1 Introduction

The problem of efficient management and enforcement of semantic integrity constraints constitutes a cornerstone issue in the development of knowledge base management systems (hereafter KBMSs) [12]. Undoubtedly knowledge bases (KBs) will form an integral part of information systems of the future. Aimed at modeling a multitude of application domains, these systems will be required to represent consistently and reason efficiently with large amounts and a wide range of knowledge. Generic tools, providing robust and efficient implementations for *storage management, query optimization, concurrency control* and *integrity enforcement* are needed for managing such large KBs [12].

Proceedings of the 19th VLDB Conference
Dublin, Ireland 1993

Integrity constraints specify the valid states of a KB (*static*) as well as the allowable KB state transitions (*dynamic*). The ability to express general constraints in KBs is crucial, since constraints express additional application dependent semantics that cannot be built-into or expressed by the data structures used for representing knowledge. Additionally, they constitute a means for controlling the quality of information stored in knowledge repositories.

A number of issues arise within the problem of integrity constraint management. A first such issue is the *declarative* – as opposed to *procedural* – specification of constraints. Constraints specify **which** properties must hold in valid KB states and transitions, and not **how** these properties are to be enforced. Procedural constraint specification inadvertently 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 KB and allows automating their optimization process. Telos [11] regards constraints as objects of their own right, and as such, they can be inserted or removed, and are subject to consistency violation. Constraints in Telos are specified declaratively via a first-order assertion language.

A second issue is that of efficient *constraint checking*. This consists of determining, after each update, whether all constraints are satisfied in the resulting state. The expressive power of the assertion language, the number of constraints in the KB and the inherent complexity of first- or higher-order deduction constitute major impediments to constraint checking. This paper focuses on the problem of simplifying the process of constraint checking in large, semantically rich KBs. Lastly, it has to be ensured that the KB is in a consistent state after a constraint violation. *Integrity recovery* is the undertaking of appropriate action for restoring consistency once it has been violated. At present, we adopt a rather coarse-grained approach to integrity recovery, namely the rejection of any integrity violating transaction. A transaction is not committed until all constraints are found to be satisfied.[1]

The naive approach of checking all integrity constraints after each update is highly impractical given the anticipated large number of integrity constraints in a KB. *Incremental* integrity checking methods are based on the premise that constraints are known to be satisfied prior to an update [13]. Only a subset of the constraints need to be verified after the update, namely those that are affected by it. Incremental integrity checking is made possible by specializing integrity constraints with respect to the anticipated types of updates and by performing simplifications on the specialized forms. A number of incremental constraint checking techniques for relational (e.g. [13]), deductive (e.g. [4], [2], [8]) and, most recently, object-oriented databases [7] have appeared in the recent literature. Most methods concentrate on the enforcement of static constraints. Dynamic constraint checking methods are mainly based on temporal logics and are run-time methods (e.g. [3]). To the best of our knowledge, there has been no proposal for compile-time simplification of temporal (static and dynamic) integrity constraints. We propose such a method in this paper. The presence of deductive rules also complicates constraint enforcement: explicit updates may cause implicit updates which may violate integrity constraints. Most techniques compute implicit updates at run time, after the actual update is specified. In our proposal, potential implicit updates can be computed at KB compilation time. This information is stored as the transitive closure of a logical dependence relation that defines a graph structure. The transitive closure of the *dependence graph* is affected by updates of rules or constraints and has to be incrementally maintained on-line.

The rest of this paper is organized as follows. Section 2 presents a brief overview of Telos and introduces a working example. The compilation of constraints and rules into simplified structures and the precomputation of implicit updates is introduced in section 3. Section 4 discusses the treatment of updates of deductive rules and integrity constraints and describes incremental algorithms for maintaining the computed implicit update information. Preliminary performance results are also presented and the paper is concluded in section 5 with an outlook for further research.

2 Overview of Telos

The representational framework of Telos [11] is a generalization of graph-theoretic data structures used in se-

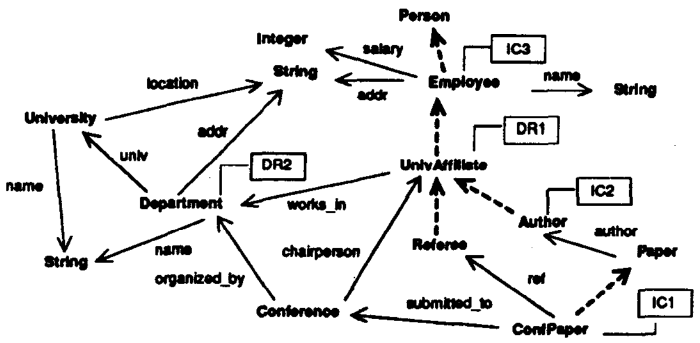


Figure 1: An example Telos KB

mantic networks, semantic data models and structurally object-oriented representations. Among the distinguishing aspects of Telos are the novel treatment of attributes, 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 [16]. Telos KBs 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 KB. Telos propositions are divided into two disjoint categories: *individuals*, representing concrete or abstract entities, and *attributes*, representing relationships between entities or relationships. Individuals and attributes are treated uniformly and are the building blocks of the structured objects that comprise a Telos KB. Propositions are organized along the structuring dimensions of *instantiation*, *generalization* and *aggregation*. Telos offers a number of built-in classes at all levels of an 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. Instantiation and specialization relationships are grouped by the built-in classes `InstanceOf` and `IsA`. Figure 1 depicts an example KB modeling 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 through attributes to classes.

2.1 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

[1] A finer grained approach could initiate a sequence of updates so that constraints are satisfied in the resulting state.

dimensions for *historical* and *belief* time. Telos adopts Allen's[1] interval-based time model for representing historical information about an application domain. Belief time records the history of the KB itself. The primitive notion of the time model is that of an *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**. The model also includes temporal constants (dates and times), semi-infinite time intervals and the special time interval **Alltime**.

2.2 The Assertional Component

Telos provides an assertional language (AL) for the expression of deductive rules and integrity constraints. AL is a first-order language with equality whose terms are variables, constants and the functions **from(t), label(t), to(t)** and **when(t)** when applied to terms, and 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**.[2] For any terms **x** and **y** in AL and every temporal or evaluable predicate θ, $x\ \theta\ y$ is an atomic formula with the obvious meaning. The well formed formulae (wffs) of AL are defined recursively from atomic formulae in the usual manner. Assertional knowledge is also organized along the structuring dimensions of Telos.

Integrity constraints and deductive rules are expressed as rectified[3] closed wffs of AL. An integrity constraint can be in one of the forms $\mathtt{I} \equiv \forall x_1/\mathtt{C}_1 \ldots \forall x_k/\mathtt{C}_k\ F$ or $\mathtt{I} \equiv \exists x_1/\mathtt{C}_1 \ldots \exists x_k/\mathtt{C}_k\ F$, 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 $\mathtt{C}_i$ 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* [4]. This class of constraints is equivalent to both the *restricted quantification form* of [2] and the *range form* of [7]. 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 [9] the following equivalences hold: $\forall x/T\ F \equiv \forall x\ T(x) \Rightarrow F$ and $\exists x/T\ F \equiv \exists x\ T(x) \wedge F$. The typed quantifications $\forall x/\mathtt{C}\ F$ and $\exists x/\mathtt{C}\ F$ are short forms for the formulae:

$\forall x\ \forall t\ \mathtt{instanceOf}(t, \mathtt{TimeInterval}, \mathtt{Alltime}) \wedge$
$\mathtt{instanceOf}(x, \mathtt{C}, t) \Rightarrow F$ and
$\exists x\ \exists t\ \mathtt{instanceOf}(t, \mathtt{TimeInterval}, \mathtt{Alltime}) \wedge$
$\mathtt{instanceOf}(x, \mathtt{C}, t) \wedge F$.

The transformation preserves the range-restriction property despite the introduction of new universally quantified temporal variables: they are restricted by the **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. Deductive rules are considered to be special cases of integrity constraints. Their general form is $\mathtt{DR} \equiv \forall x_1/\mathtt{C}_1 \ldots \forall x_n/\mathtt{C}_n\ (F \Rightarrow A)$, 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 *stratified* [9]. Constraints and rules are associated with history and belief time intervals. If no such association appears explicitly with their definition, both intervals are assumed to be equal to **(systime..*)**, where **systime** denotes the current system time.

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. Apart from static and transition constraints expressible in first-order predicate calculus and temporal logics respectively, AL provides the ability to express constraints referring to the epistemic state of the KB. To provide a better characterization of the types of constraints expressible in AL, we will refer to *state*, *dynamic* and *dynamic epistemic* 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 AL. *Dynamic* constraints specify properties dependent on two or more domain states. Finally, *dynamic epistemic* constraints refer to two or more epistemic states of the KB in addition to multiple domain states. From the above definitions, it is expected that in the expression

[2] The belief time component of these predicates is omitted when they appear in the head of deductive rules.

[3] A formula is rectified if no two quantifiers introduce the same variable [2].

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.
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 KB's 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.

2.3 Working Example

This section introduces a working example that will be used throughout the paper. A number of integrity constraints and deductive rules have been defined, for the sake of depicting the application of the method, and attached to classes (see figure 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: $\forall c/\texttt{ConfPaper}\ \forall r/\texttt{Referee}\ \forall a/\texttt{Author}$
$\forall t_1,t_2/\texttt{TimeInterval}\ (ref(c,r,t_1,t_2)\ \wedge$
$author(c,a,t_1,t_2) \Rightarrow (r \neq a\ [\texttt{at}\ t_1, \texttt{believed at}\ t_2]))$
IC2: $\forall c/\texttt{Conference}\ \forall p/\texttt{ConfPaper}\ \forall a/\texttt{Author}$
$\forall d/\texttt{Department}\ \forall t_1,t_2/\texttt{TimeInterval}$
$(submitted\_to(p,c,t_1,t_2) \wedge organized\_by(c,d,t_1,t_2) \wedge$
$author(p,a,t_1,t_2) \wedge works\_in(a,d,t_1,t_2)\ \Rightarrow \texttt{False})$
IC3: $\forall p/\texttt{Employee})(\forall s,s'/\texttt{Integer}$
$\forall t_1,t_2,t_3/\texttt{TimeInterval}\ (salary(p,s,t_1,t_2) \wedge$
$salary(p,s',t_3,t_2) \wedge \texttt{before}(t_1,t_3) \Rightarrow (s \leq s'))$
DR1: $\forall u/\texttt{UnivAffiliate}\ \forall d/\texttt{Department}\ \forall s,s'/\texttt{String}$
$\forall t_1,t_2/\texttt{TimeInterval}\ (address(u,s,t_1,t_2)\ \wedge$
$D\_addr(d,s',t_1,t_2)\ \wedge\ (s = s'[\texttt{at}\ t_1, \texttt{believed at}\ t_2])$
$\Rightarrow works\_in(u,d,t_1))$
DR2: $\forall d/\texttt{Department}\ \forall u/\texttt{University}\ \forall s/\texttt{String}$
$\forall t_1,t_2/\texttt{TimeInterval}\ (univ(d,u,t_1,t_2) \wedge$
$location(u,s,t_1,t_2) \Rightarrow D\_addr(d,s,t_1))$
IC4: $\forall p,c,l/\texttt{Proposition}\ \forall t,t'/\texttt{TimeInterval}$
$(\texttt{prop}(p,c,l,c,t) \wedge \texttt{instanceOf}(p,\texttt{Class},t,t') \Rightarrow$
$(\forall T,T'/\texttt{TimeInterval}\ (\texttt{overlaps}(t,T) \wedge$
$\texttt{overlaps}(t',T')\ \Rightarrow\ \texttt{instanceOf}(p,\texttt{Class},T,T')))$ ◇

3 Constraint Simplification

In this section the compilation of constraints and rules into simplified forms and their organization in a *dependence graph* is presented. Our method builds on the *compilation* method that was initially proposed in [2] and was later adapted to an object-oriented setting in [7]. The efficiency of the method stems from the separation of the task of constraint enforcement in two separate phases: a *compilation* phase, performed at schema definition time and an *evaluation* phase performed at KB 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 [7] by taking time into account and by optimizing the compile-time computation of implicit updates.

3.1 Compilation of Constraints and Rules

Let us first explain the rationale behind simplification by means of an example. Assume that constraint IC1 of section 2.3 has the right-infinite interval $t_1 =$ (01/01/1988..*) as its associated history time interval. Assume also that the constraint defining transaction was processed on January 2, 1988. This means that the constraint is believed by the system from 02/01/88 and on. Let $t_2 =$ (02/01/1988..*). Constraint IC1 is *relevant* to any update processed on or after 02/01/88 and which refers to a history time interval contained in t_1. IC1 is satisfied if it is entailed by the KB for any history time interval t_3 such that $\texttt{during}(t_3,t_1)$, with belief time t_4 such that $\texttt{during}(t_4,t_2)$. Updates with time intervals overlapping t_1 and t_2 are considered relevant only for the subintervals that occur during t_1, t_2. Assume now that an update introduces an instance of the class ConfPaper with title Krypton, author Brachman and time 05/01/88, and that the transaction is processed on 08/01/88. Then the constraint is relevant to this update since, for t_3 =(05/01/88..*) and t_4 =(08/01/88..*), $\texttt{during}(t_3,t_1)$ and $\texttt{during}(t_4,t_2)$ are both satisfied. After instantiating variables and dropping quantifiers the constraint that needs to be checked becomes:
$\forall r/\texttt{Referee}\ (ref(\texttt{Krypton},r,\texttt{05/01/88..*},\texttt{08/01/88..*})$
$\wedge\ author(\texttt{Krypton},\texttt{Brachman},\texttt{05/01/88..*},\texttt{08/01/88..*})$
$\Rightarrow (r \neq \texttt{Brachman}\ [\texttt{at 08/01/88..*}, \texttt{believed at 08/01/88..*}]))$.
Moreover, it is known that the literal
$author(\texttt{Krypton},\texttt{Brachman},\texttt{05/01/88..*},\texttt{08/01/88..*})$ is

true in the new state and thus, it can be replaced by the Boolean constant **True**. What remains to be checked is the simplified form:

$\forall r/$**Referee**$(ref($**Krypton**$, r,$ **05/01/88..*, 08/01/88..***$)$
$\Rightarrow (r \neq$ **Brachman [at 08/01/88..*, believed at 08/01/88..*]**$))$

Such a simplified form can be generated for each update that may affect the constraint. The actual values of the variables in the constraint are not known. They are replaced by parameters which are instantiated at evaluation time. Note however that only $\forall$-quantified variables not governed by $\exists$ (called *instantiation variables*) can be replaced by parameters. As shown in [13], replacing $\exists$-quantified variables or $\forall$-quantified variables governed by $\exists$ with constants or parameters, may lead to counter-intuitive or incorrect instances. Let us introduce some terminology for formalizing the simplification method. Compilation and simplification steps apply uniformly to integrity constraints and the bodies of deductive rules defined for both simple classes and metaclasses.

Definition: An *update* is an instantiated literal whose sign determines whether it is an insertion or a deletion. ◇

Given the general form of constraints defined in section 2, it can be seen that a constraint 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 [7] is not sufficient in the presence of time. The following definition provides sufficient conditions for "relevance" of a constraint to an update, by considering the relationships of the time intervals participating in the literals of the constraint and the update.

Definition:(*Affecting Update*) An update $U(\_,\_,t_1,t_2)$ is an affecting update for a constraint I with history and belief time intervals T and T' respectively, if and only if there exists a literal $L(\_,\_,\_,\_)$ in I such that L unifies with the complement of U and the intersections $t_1 * T$ and $t_2 * T'$ are non-empty. ◇

For each literal occurring in a constraint, the compilation process generates a *Parameterized Simplified Structure* (PSS). A PSS allows for efficient selection of constraints affected by an update and for indexing with respect to time or its characteristic literal or class. A PSS has the form shown is figure 2. PSSs are readily representable in Telos via its metaclassing mechanism. The *Literal* component is the constraint literal with respect to which the PSS is generated. *Parameters* is the list of instantiation variables occurring as components to *Literal* in the initial form of the constraint and which are treated as parameters in the simplified form. Parameters will be replaced by the constants appearing in an update at run-time. The *History* and *Belief Time* components contain the history and belief time intervals associated with the constraint. The *Concerned Class* for a literal L is a class C such that, inserting or deleting an instance of C can affect the truth of L [7]. The role of a concerned class is to limit the search space for constraints affected by an update. This is possible because of the fine granularity - not found in relational databases - provided by ***aggregation***. In the presence of time and specialization, this definition must be refined appropriately. Intuitively, the concerned class for a literal $L(\_,\_,t_1,t_2)$ should be the most specialized class that fulfills the above requirements and is such that the time intervals of the literal L overlap with those of the class. This restriction 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. The concerned class for each literal is determined at compile-time, when the constraint is transformed into its simplified form. 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 transitivity of the **Is-A** relationship. Concerned classes are determined as follows:

- **Instantiation literals**: for a literal of the form **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 **InstanceOf** is the concerned class.
- **Generalization literals**: for a literal of the form **isA**$(x,y,t1,t_2)$ where both x and y stand for classes, the concerned class is the built-in class **IsA**, the truth of an **isA**-literal does not depend on the insertion/deletion of instances to/from the extensions of classes x and y.
- **Attribute literals**: for a literal of the form $att(x,y,t_1,t_2)$, 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 **from**$(q) = X,$ **label**$(q) = att,$ **to**$(q) = Y$ and **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 AL, corresponds a unique proposition with the above properties.

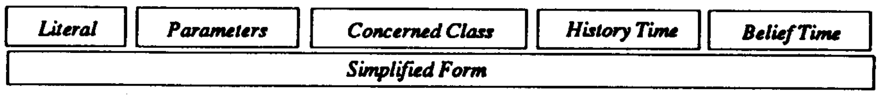


Figure 2: A Parameterized Simplified Structure

$\phi \wedge \mathtt{T} \equiv \phi$	$\phi \wedge \mathtt{F} \equiv \mathtt{F}$	$\phi \Rightarrow \mathtt{T} \equiv \mathtt{T}$	$\phi \Rightarrow \mathtt{F} \equiv \neg\phi$
$\phi \vee \mathtt{T} \equiv \mathtt{T}$	$\phi \vee \mathtt{F} \equiv \phi$	$\mathtt{T} \Rightarrow \phi \equiv \phi$	$\mathtt{F} \Rightarrow \phi \equiv \mathtt{T}$
$\neg\mathtt{T} \equiv \mathtt{F}$	$\neg\mathtt{F} \equiv \mathtt{T}$	$\phi \Leftrightarrow \mathtt{T} \equiv \phi$	$\phi \Leftrightarrow \mathtt{F} \equiv \neg\phi$

Table 1: Absorption rules

• For a literal of the form $\mathtt{prop}(p, x, y, z, t)$, if the components x and z are equal, then the concerned class is the built-in class **Individual**; if not, the concerned class is the class **Attribute**. In case none of x and z are instantiated, the concerned class is the class **Proposition**. However, because of the *referential integrity* constraint imposed in Telos,[4] the prop literals will not be considered in the generation of simplified forms.

Finally, the *Simplified Form* is derived by applying the following steps to the constraint.

Step 1: the quantifiers binding instantiation variables are dropped. Instantiation variables become parameters.

Step 2: the temporal variables are constrained with respect to the history and belief times of the constraint, and the resulting temporal relationships are conjoined with the constraint

Step 3: the atom into (from) whose extension a tuple is inserted (deleted) can be substituted by the Boolean constant **True** (**False**) since after the update it is known that the fact expressed by the literal is true (false respectively).

Step 4: absorption rules are applied to simplify the resulting formula as shown in table 3.1. [5]

Step 5: temporal simplification rules are applied if applicable. [6]

Example: The concerned class for literal *author* of constraint **IC1** is the attribute class defined by the proposition (**Paper**, *author*, **Author**, t), with t satisfying the properties of the above rules. Applying the above steps to constraint **IC1** for literal *author* yields:

$\forall r/\mathbf{Referee}(ref(\mathtt{c}, r, \mathtt{t}_1, \mathtt{t}_2) \wedge \mathbf{during}(\mathtt{t}_1, 01/01/88..*)$
$\wedge\ \mathbf{during}(\mathtt{t}_2, 02/01/88..*) \Rightarrow (r \neq \mathtt{a}\ [\mathbf{at}\ \mathtt{t}_1,$
$\mathbf{believed\ at}\ \mathtt{t}_2])).$

At update time, if the constraint is affected by an update, the form that will have to be verified will be

$\forall r/\mathbf{Referee}\ (ref(\mathtt{c}, r, \mathtt{t}_1, \mathtt{t}_2) \Rightarrow (r \neq \mathtt{a}\ [\mathbf{at}\ \mathtt{t}_1,$
$\mathbf{believed\ at}\ \mathtt{t}_2])) \diamond$

Temporal Simplification

The last step in the generation of parameterized simplified forms is the application of temporal simplification rules. The objective of temporal simplification is to simplify a conjunction of temporal relationships into a single temporal relationship. Hence, the number of subformulae to be evaluated at run-time is reduced. 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 [1] 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, it is feasible to define a number of rules that allow to derive new temporal relations from already existing ones.

Formally, the problem of temporal simplification is stated as follows: given a conjunction $\mathtt{during}(t, i_1) \wedge \mathtt{r}_1(t, i_2)$, where i_1 and i_2 are known time intervals, find a temporal relationship $\mathtt{r}$ and an interval i such that $\mathtt{r}(t, i)$ 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 $\mathtt{r}_2(i_1, i_2)$. This relationship is exploited for restricting the possibly multiple alternatives for $\mathtt{r}$. In fact, the expression that is simplified is the conjunction $\mathtt{during}(t, i_1) \wedge \mathtt{r}_1(t, i_2) \wedge \mathtt{r}_2(i_1, i_2)$. It is not always possible to derive a single definite relation $\mathtt{r}$ that has the above property. For some combinations of temporal relationships $\mathtt{r}$ 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.

The table in figure 3 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 $\mathtt{r}$ 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

[4] It is possible to express such a property as a meta-constraint in AL.

[5] ϕ stands for any wff of AL.

[6] In several cases the combination of temporal predicates introduced in step 2 with ones in the constraint may result in simpler forms. Step 5 is applied repeatedly until no further simplification is possible.

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. In the case that the negation of a temporal relationship appears in r_1 one can only suggest a weaker condition r, which if satisfied guarantees that the original conjunction is satisfied. A table similar to that of figure 3 can be defined for the cases where negation 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 table lookup for finding the simplifying expression and simple operations between interval endpoints.

Example: Consider the conjunction $\mathtt{during}(t, 01/88..09/88) \wedge \mathtt{before}(t, 05/88..12/88) \wedge \mathtt{overlaps}(01/88..09/88, 05/88..12/88)$. According to the table in figure 3, the above expression can be simplified into $\mathtt{during}(t, 01/88..05/88)$. It is also easy to verify that, if $\mathtt{during}(t, 01/88..05/88)$ is satisfied, then the conjunction $\mathtt{during}(t, 01/88..09/88) \wedge \neg\mathtt{overlaps}(t, 05/88..12/88) \wedge \mathtt{overlaps}(01/88..09/88, 05/88..12/88)$ is satisfied. ◇

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 KB, imply that the original formulae are not satisfied. Moreover, no inconsistency can be introduced by any of the simplification steps. Hence, the simplification method is sound. 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. Detailed proofs are found in [15].

Dynamic Constraints

Simplification is applicable in the case of dynamic (epistemic) constraints as well. The validity of the constraints in the KB history up to the state prior to the update and the system's knowledge about it is exploited for producing, in those cases possible, forms with a reduced number of literals. Dynamic (epistemic) constraints are distinguished from state constraints by the presence of explicit temporal constraints on the history (belief) time variables. In addition, since dynamic constraints express properties depending on two or more KB states some literals will occur more than once in the expression of constraints. In such cases, the compilation process will generate one PSS for each literal occurrence. The forms will differ in their lists of parameters, as well as in their simplified forms. The original constraint will be violated if any of the simplified forms is. However, in such a case, not all occurrences of the literal can be replaced by their truth values on the basis that both the update and the fact that constraints were satisfied before the update are known. This would be possible only if it were known that the constraints were non-trivially satisfied in the previous state. This kind of knowledge requires the maintenance of meta-level information about the satisfaction of constraints. For the moment we will assume that no such knowledge is available and that a PSS is generated for each literal occurrence in an integrity constraint. The following example shows the application of the compilation process in the case of a dynamic constraint.

Example: Assume that the history and belief time intervals of constraint IC3 of section 2.4 are T and T' respectively. The literal *salary* occurs twice in the expression of the constraint. Hence, two simplified forms are generated from the compilation process. Only one of the history time variables t_1 and t_3 will be instantiated in each of the two forms. It is known that the constraint is satisfied before an update to a *salary* literal occurs. This means that, according to the current beliefs of the system, either all employees have not had a change in salary, or for those that have had a salary change, this change was an increase. These two cases correspond to trivial and strict constraint satisfaction respectively. If no information exists about whether the satisfaction of the constraint prior to the update is strict or trivial, the following two forms can be generated by the compilation process:

$\forall s/\mathtt{Integer}\ \forall t_1/\mathtt{TimeInterval}\ (salary(p, s, t_1, t_2) \wedge \mathtt{during}(t_1, T) \wedge \mathtt{during}(t_3, T) \wedge \mathtt{before}(t_1, t_3) \wedge \mathtt{during}(t_2, T') \Rightarrow (s \leq s'))$

$\forall s'/\mathtt{Integer}\ \forall t_3/\mathtt{TimeInterval}\ (salary(p, s', t_3, t_2) \wedge \mathtt{during}(t_1, T) \wedge \mathtt{during}(t_3, T) \wedge \mathtt{before}(t_1, t_3) \wedge \mathtt{during}(t_2, T') \Rightarrow (s \leq s'))$

Were it known that IC3 was non-trivially satisfied, only one simplified form would be generated, namely the form resulting from dropping all quantifiers from the above forms and replacing the *salary* literals by True. If however it was trivially satisfied before the update, i.e., at least one of the *salary* literals was false or the temporal constraint was violated, then the *salary* literals cannot be eliminated. The rest of the simplification steps are applied as before. ◇

3.2 Dependence Graph Organization

In the compilation phase, along with each integrity constraint, deductive rules that may contribute to the constraint's evaluation are compiled. These are the

r1(t,i2) / r2(i1,i2)	b	d	o	m	s	f	e	a	c	ob	mb	sb	fb
before	during i1	F	F	F	F	F	F	F	F	F	F	F	F
during	F	during i1	F	F	F	F	F	F	F	F	F	F	F
overlaps	during i1-i1*i2	during i1*i2	overlaps i1*i2	finishes i1-i1*i2	starts i1*i2	F	F	F	F	F	F	F	F
meets	during i1	F	F	F	F	F	F	F	F	F	F	F	F
starts	F	during i1	F	F	F	F	F	F	F	F	F	F	F
finishes	F	during i1	F	F	F	finishes i1	F	F	F	F	F	F	F
equal	F	during i1	F	F	F	F	F	F	F	F	F	F	F
after	F	F	F	F	F	F	F	during i1	F	F	F	F	F
contains	during (i1-,i2-)	during i2	no simp.	finishes (i1-,i2-)	starts i2	finishes i2	equal i2	during (i2+,i1+)	no simp.	no simp.	starts (i2+,i1+)	starts (i2-,i1+)	finishes (i1-,i2+)
overlapped by	F	during (i1-,i2+)	F	F	F	finishes (i1-,i2+)	F	during (i2+,i1+)	F	no simp.	starts (i2+,i1+)	F	finishes (i1-,i2+)
met by	F	F	F	F	F	F	F	during i1	F	F	F	F	F
started by	F	during i2	F	F	F	finishes i2	F	during i1-i2	F	no simp.	starts i1-i2	F	F
finished by	during (i1-,i2-)	during i2	no simp.	finishes (i1-,i2-)	starts i2	F	F	F	F	F	F	F	finishes i1

Legend

F:	false	no simp. :	no simplification possible
* :	intersection operator	t-	: left endpoint
- :	difference operator	t+	: right endpoint

Figure 3: Temporal Simplification Table

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 transitively on a rule whose conclusion literal matches a condition literal of a rule on which the constraint depends either directly or transitively. Formally, we can define the notions of *dependence* and *direct dependence* along the lines of [7].

Definition:*(Direct Dependence)* A literal L directly depends on literal K if and only if there exists a rule of the form $\forall x_1/\mathtt{C}_1 \ldots \forall x_n/\mathtt{C}_n\ (F \Rightarrow A)$ such that, there exists a literal in the body F of the rule unifying with L with most general unifier θ and $A\theta = K$. *(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.◇

The above relationships define a *dependence graph* for a set of rules and constraints. The dependence graph is a directed graph representing how implicitly derived facts from deductive rules can affect the integrity of the KB. The graph nodes are the PSSs of rules and constraints. Edges denote dependence of constraints on rules. There exists an edge form the node of a rule R to that of a constraint C, if C directly depends on R. An edge from a node of a rule R to the node of a constraint C is labeled "T", if the history and belief time intervals of R overlap those of the constraint C. Formally, the dependence graph of a 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. The set V of nodes is equal to the union of the set of nodes corresponding to integrity constraints (I) with the set of nodes corresponding to deductive rules (R). Hence, $V = V_I \cup V_R$, where V_I and V_R are the aforementioned sets. The set E of edges is defined as: $E = \{(v_i, v_j) | v_i \in V_R, v_j \in V_I \textit{ and } v_j \textit{ directly depends on } v_i\} \cup \{(v_i, v_j) | v_i, v_j \in V_R \textit{ and } v_i \textit{ directly depends on } v_j\}$. The set E of edges is made up of edges between rule nodes (E_{RR}) and edges from rule to constraint nodes (E_{RC}). 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 may contain cycles among deductive rule nodes. This happens in the case the KB contains mutually recursive rules. As shown in [15] the graph is free of trivial cycles and enjoys the property expressed in the following lemma.

Lemma: For any Telos KB, 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. ◇

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 α denote this number. The number of compiled forms generated, will then be equal to $|V| = \alpha * (|I| + |R|)$. The number of edges is $|E| = |E_{RR}| + |E_{RC}|$. $|E_{RC}|$ 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, $|E| = |E_{RC}| + |E_{RR}| \leq 2 * |R|$. For $\alpha > 2, |V| = \alpha * (|I| + |R|) > 2 * |R| \geq |E|$, which means that the graph is sparse. The graph's sparsity will be exploited for deriving efficient algorithms for transitive closure computation. The dependence graph is constructed once when the KB is compiled and is updated incrementally when new rules or constraints are inserted or deleted. Although sparse, the dependence graph for a large KB 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 [17] for solving the *reachability problem*. The algorithm, applicable to directed acyclic 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 $O(|V_R| * |E|)$ 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, reachability information does not have to be recomputed. Space restrictions do not permit a detailed analysis of the algorithm in this paper. The algorithm and its analysis are found in [15]. 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. This process

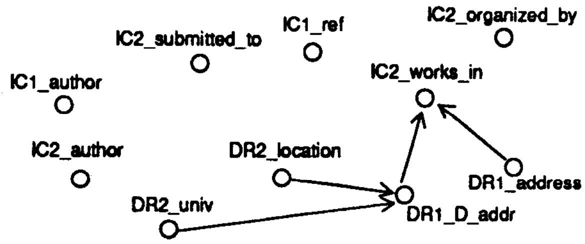


Figure 4: A Dependence Graph

can take place only after the explicit update is specified and interleaves the generation of actual implicit updates with formula evaluation. Its complexity is linear on the number of potential implicit updates multiplied by the cost of formula evaluation.

Figure 4 shows the dependence graph organization for our working example. The edge from the compiled structure 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**. ◇

4 Updates of Integrity Constraints and Deductive Rules

The compilation scheme presented in section 3.1 permits the efficient treatment of transactions of literal updates: for every update in a transaction, the simplified constraints affected by the update are selected and instantiated along with the simplified rules whose literal matches the update. The set of implicit updates corresponding to the rule's evaluation is instantiated and each implicit update is treated as a normal update. The situation is more complicated in the case of transactions that insert or delete constraints and rules. The organization of simplified forms in a dependence graph allows for incremental compilation of newly inserted constraints and rules without having to reconsider those that have already been compiled. In both the cases of insertion and deletion the dependence graph must be modified minimally. Determining however how dependence is affected and updating the stored transitive closure is a difficult and costly task for cyclic graphs. In this section we describe incremental graph modification procedures, characterize their complexity and show how the computed transitive closure can be updated without having to recompute it from scratch. We also present results from experiments with random graphs. It will be assumed that the computed transitive closure (TC) is represented in a form that permits checking reachability between any two graph nodes in $O(1)$ time. A detailed presentation and analysis of the algorithms is found in [15].

4.1 Updates of Integrity Constraints

Insertion: For the time being we assume the traditional semantics attributed to the insertion of an integrity constraint, namely that a new constraint must be evaluated against the KB and be accepted only if found true; otherwise it has to be rejected. Only when a new constraint is found true, it is transformed into a set of parameterized forms, one for each of its literals. These forms are added as nodes to the dependence graph and in case there exist rules already in the graph on which the constraint directly depends, edges from the rule nodes to the constraint nodes are added and labeled. The worst-case complexity of the dependence graph modification is $O(|V_R|)$, since the newly introduced nodes have to be connected with as many rule nodes as the number of rules whose conclusion literal matches the constraint 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 characterize the cost of insertion more precisely, let us define a function $F : L \rightarrow [0,1]$, which returns, for each literal l, the frequency of its occurrences in rule heads.[7] On the average the number of edge additions required for the insertion of an integrity constraint will be equal to $Cost_{IC\_insert} = \sum_{l \in IC} |V_R| * F(l) * \alpha$.

Deletion: Deletion of constraints cannot cause an inconsistency. All nodes corresponding to some simplified form of the constraint are removed along with their incident edges.[8] The worst-case complexity of the deletion process is $O(|E|)$ and corresponds to a situation in which all edges are adjacent to the nodes to be deleted. The average cost of edge deletion is $Cost_{IC\_delete} = \sum_{l \in IC} |V_R| * F(l) * \alpha$.

4.2 Updates of Deductive Rules

The case of updates of deductive rules appears to be more complicated since insertion or deletion may cause implicit changes which are also candidates for violating integrity.

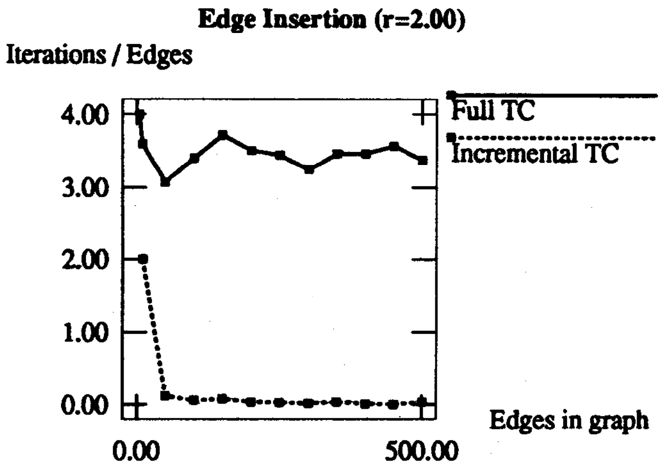
Insertion: When a new rule $B \rightarrow H$ is inserted, its direct dependence relationships to existing rules or constraints must be determined and represented in the graph. It must be checked whether there exist PSSs of constraints or rules with literals unifying with the rule's conclusion literal. In that case, the conclusions of the rule must be derived and checked for possible constraint violations. 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 graph's transitive closure has been computed and updated after each insertion/deletion of rules and constraints. One only has to check if a constraint node appears as a successor 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/constraint that unifies with the rule's conclusion, then the rule is transformed into a set of PSSs and inserted in the dependence graph. The worst-case complexity of this process is $O(|V_R| * |E|)$. More specifically, the number of graph nodes corresponding to deductive rules matching the rule head H is equal to $|V_R| * F(H)$. Moreover each such node can have at most $|E_{RC}|$ successors in the transitive closure. 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 $c = \frac{|V_R|}{|V_I|}$.Then $\frac{|E_{RR}|}{|E_{RC}|} \approx c$ and the above cost becomes $|V_I| * F(H) * (1 + |E| * \frac{c}{1+c})$. Let also $r = \frac{|V|}{|E|}$. Then the cost can be written as $\frac{r}{(1+c)^2} * F(H) * (1+c+c*|E|) * |E|$. Finally, the modification of the dependence graph requires that edges are added from rule nodes to the newly added ones. This cost is derived as in the case of constraint insertion and is, on the average, equal to $\sum_{l \in B} |V_R| * F(l) * \alpha$. In total, $Cost_{DR\_insert} = \frac{r}{1+c} * |E| * (\alpha * \sum_{l \in B} F(l) + F(H) * (1 + \frac{c}{1+c} * |E|))$.

Deletion: If an already compiled rule is to be deleted, then, if there exist rules or constraints with literals matching the rule's negated conclusion, the literals deducible with this rule must be treated as normal deletions. If they do not cause integrity violation, the parameterized forms of the rule must be deleted along with all their incident edges. The computed transitive closure provides the information of whether an implicit deletion caused by the rule's deletion can violate an integrity constraint. As was the case for rule insertion, rule deletion requires worst-case time of $O(|V_R| * |E|)$. As before, the edge removal cost is estimated as $Cost_{DR\_delete} = \frac{r}{1+c} * |E| * (\alpha * \sum_{l \in B} F(l) + F(\neg H) * (1 + \frac{c}{1+c} * |E|))$.

From the above description, it can be seen that an adequate treatment of updates of rules and constraints requires interleaving compilation and evaluation. This increase in complexity however is a tradeoff for the cost of KB roll-back in case a violation is discovered.

[7] This type of information can be available after KB compilation and can be maintained incrementally after modifications.

[8] In [7], if this process results in isolated nodes representing simplified rules on which the deleted constraint previously depended, then these nodes are removed as well. In our approach, these nodes are not removed in order to avoid their recompilation in case future updates introduce constraints depending on these rules.



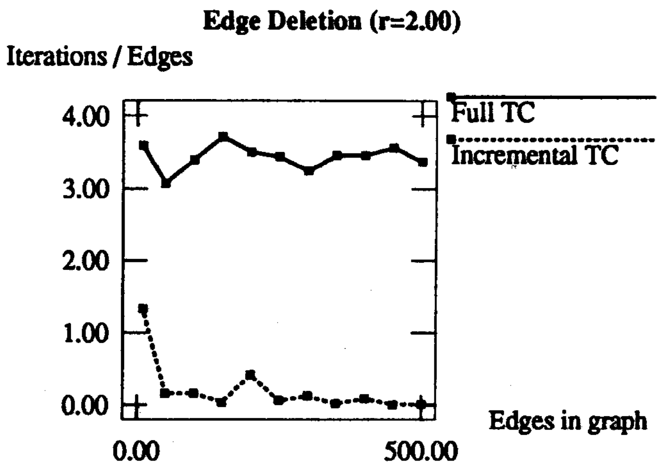


Figure 5: Performance of Incremental T.C. Computation

4.3 Incremental Modification of Transitive Closure

In the previous procedures for dependence graph maintenance during rule or constraint updates, the precomputed transitive closure has to be updated, since changes in the direct dependence relation induce changes to the dependence relation between literals. A number of algorithms have been proposed for on-line maintenance of transitive closure (e.g. [6]) but are not applicable to cyclic graphs. In [15] we propose algorithms that maintain reachability information for edge insertions and deletions in the dependence graph. These algorithms are shown to be correct for graphs with the properties of a dependence graph. We briefly describe each of them here. Insertion of an edge from node u to node v makes v and all its successors reachable from u and every node from which u is reachable. The reachability between any other pair of nodes 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. The worst-case complexity of edge insertion is $O(|E_{RR}| * |E|)$. In particular, when an edge from a rule node to a constraint node is inserted the worst-case cost is $|E|$, whereas the cost is $|E_{RR}| * |E|$ when an edge between two rule nodes is inserted. The cause for the high cost is the possible presence of cycles in the graph. When an edge (u, v) is removed, for all nodes x whose only path to v is via u, (x, v) must be removed from TC. Moreover, if the only path from each such node x to a successor y of v is via (u, v), then (x, y) must be removed from TC. The maintenance of TC in the case of an edge removal assumes path information is available. This information can be computed along with TC and be maintained incrementally with a total added cost of at most $|E|$. The complexity of the deletion process is quadratic on the number of edges in the graph.

It becomes apparent from the description of the maintenance procedures, that the presence of cycles complicates the on-line transitive closure computation. 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 figure 5 compare the cost of incremental computation of TC with that of recomputing it 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 $0.1 * |E|$ on the average. The performance of the incremental algorithms improves even more when the degree of sparsity of the dependence graph increases ($r = 3, r = 4$).

5 Conclusions and Outlook

This paper presented an integrity constraint simplification method for large Telos KBs. The main contribution of this work lies in the definition of a sound and complete simplification method that treats uniformly temporal and non-temporal (static and dynamic) constraints. Temporal simplification is performed efficiently by a table lookup. Other attempts to treat uniformly static and dynamic constraints restrict attention to specific types of constraints (e.g. transition constraints) and do not contain explicit temporal information [14]. Chomicki's techniques [3] are beneficial to the enforcement of dynamic constraints since they permit their evaluation without having to consider the entire history of the KB. The

method assumes a rather cumbersome formulation of constraints in Past Temporal Logic with no explicit presence of time and that the set of constraints does not change. We consider this to be a major restriction for temporal KBs modeling an evolving domain. Techniques proposed for temporal integrity monitoring (e.g. [5]) are run-time methods, whereas we have focused on simplifying formulae as much as possible at schema definition time. Performance is less critical at compile-time.
Current research focuses on the definition of an efficient hybrid theorem prover for the evaluation of temporal constraints, in the flavor of [10]. Such a theorem prover may be enhanced with techniques for reduction of temporal formulae referring to long histories into formulae evaluable in a pair of states only [18]. A performance assessment of the method is in progress. The method needs to be compared against one-phase methods that interleave simplification and evaluation [8] and run-time methods. Moreover, the I/O complexity of graph computations and the secondary-memory storage of rules and constraints need to be studied.

6 Acknowledgements

The author wishes to thank Professors John Mylopoulos and Yannis Ioannidis for their valuable advice, as well as Thodoros Topaloglou, Manolis Koubarakis and the anonymous referees for comments that helped improve the paper. Financial support was provided by the Department of Computer Science and the Information Technology Research Centre.

References

[1] J. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[2] F. Bry, H. Decker, and R. Manthey. A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases. In *1st Int. Conference on Extending Data Base Technology*, pages 488–505, Venice, Italy, 1988.

[3] J. Chomicki. History-less Checking of Dynamic Integrity Constraints. In *8th Int. Conference on Data Engineering*, pages 557–564, Phoenix,AZ, 1992.

[4] H. Decker. Integrity Enforcement in Deductive Databases. In *Expert Database Systems, 1st Int. Conference* , pages 271–285, 1986.

[5] K. Hulsmann and G. Saake. Representation of the Historical Information Necessary for Temporal Integrity Monitoring. In *2nd Int. Conference on Extending Data Base Technology*, pages 378–392, Venice, Italy, 1990.

[6] G. Italiano. Finding Paths and Deleting Edges in Directed Acyclic Graphs. *Information Processing Letters*, 28(1):5–11, 1988.

[7] M. Jeusfeld and M. Jarke. From Relational to Object-Oriented Integrity Simplification. In *2nd Int. Conference on Deductive and Object-Oriented Databases*, pages 460–477, Munich, Germany, 1991.

[8] V. Kuchenhoff. On the Efficient Computation of the Difference Between Consecutive Database States. In *2nd International Conference on Deductive and Object-Oriented Databases*, pages 478-502, Munich, Germany, 1991.

[9] J. Lloyd. *Foundations of Logic Programming*. Springer Verlag, 1987. 2nd edition.

[10] S. Miller and L. Schubert. Time Revisited. *Computational Intelligence*, 6:108–118, 1990.

[11] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: A Language for Representing Knowledge in Information Systems. *ACM Transactions On Information Systems*, 8(4):325–362, 1990.

[12] J. Mylopoulos, V. Chaudhri, D. Plexousakis, and T. Topaloglou. A Performance Oriented Approach to Knowledge Base Management. In *1st Int. Conference on Information and Knowledge Management*, pages 68–75, Baltimore, MD, 1992.

[13] J.-M. Nicolas. Logic for Improving Integrity Checking in Relational Databases. *Acta Informatica*, 18:227–253, 1982.

[14] Olivé, A. Integrity Constraints Checking in Deductive Databases. In *17th VLDB Conference*, pages 513–523, Barcelona, Spain, 1991.

[15] D. Plexousakis. Integrity Maintenance in a Telos based KBMS. Technical report, Department of Computer Science, University of Toronto, 1993. Forthcoming.

[16] D. Plexousakis. Semantical and Ontological Considerations in Telos: a Lanugage for Knowledge Representation. *Computational Intelligence*, 9(1):41–72, 1993.

[17] G. Qaddah, L. Henschen, and J. Kim. Efficient Algorithms for the Instantiated Transitive Closure Queries. *IEEE Transactions on Software Engineering*, 17(3):296–309, 1991.

[18] G. Saake and U. Lipeck. Foundations of Temporal Integrity Monitoring. In C. Roland, editor, *Temporal Aspects in Information Systems*, pages 235–249. North Holland, 1988.