Constraint Management in Loosely Coupled Distributed Databases *

Sudarshan S. Chawathe
Hector Garcia-Molina
Jennifer Widom

Department of Computer Science
Stanford University
Stanford, CA 94305-2140

{chaw,hector,widom}@cs.stanford.edu

Abstract

We provide a framework for managing integrity constraints that span multiple databases in loosely coupled, heterogeneous environments. Our framework enables the formal description of (1) interfaces provided by a database for the data items involved in multi-database constraints; (2) strategies for monitoring and maintaining multi-database constraints; (3) guarantees regarding the consistency of multi-database constraints. With our approach one can define "relaxed" constraints that only hold at certain times or under certain conditions. Such constraints appear often in practice and cannot be handled effectively by conventional, transaction-based approaches. We also describe a toolkit, based on our framework, for enforcing constraints over heterogeneous systems. The toolkit includes a general-purpose, distributed constraint manager that can be easily configured to a given environment and constraints. A first version of the toolkit has been implemented and is under evaluation.

1 Introduction

Integrity constraints arise in distributed databases even when the data is stored in loosely coupled, heterogeneous systems. For example, a construction company keeps data about a building under construction. This data must be consistent with the architect's design (e.g., walls must be in the same places), which is stored in an entirely different system. Similarly, a company building a jet engine for a new airplane must ensure that the engine thrust is the same as the airplane builder is assuming. Constraint management in distributed, heterogeneous environments is a much more difficult and complex problem than in centralized systems, because of several factors:

- Loosely coupled heterogeneous environments typically do not support multi-database transactions. Consequently, there is no mechanism for performing a consistent query or update across multiple databases, and there is no notion of an atomic unit of execution.

*Research sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Material Command, USAF, under Grant Number F33615-93-1-1339. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of Wright Laboratory or the US Government. This work was also supported by the Center for Integrated Systems at Stanford University, and by equipment grants from Digital Equipment Corporation and IBM Corporation.

- Each data repository (which may not even be a database system) may support different capabilities for accessing and monitoring the data items involved in a constraint. For example, one "uncooperative" repository might provide read-only access to a data item, while another "cooperative" repository might provide write access to a data item as well as automatic notification whenever the value of the data item changes.

- Communication delays may be incurred when checking and/or restoring constraints involving multiple databases.

Despite these problems, there is a clear need for integrity constraint management in loosely coupled distributed systems. Currently, such constraints are monitored or enforced by humans, in an ad-hoc fashion. For example, an architect may freeze the building design and send the latest specifications to the construction company so that consistency is "guaranteed." Of course, it is clear that these ad-hoc mechanisms do not work well in general; for example, the building may eventually not meet the specifications.

Our goal is to understand how constraints can be properly monitored or enforced in such environments. The key point is that it is *not* possible to make the level of guarantees that a centralized database system makes, i.e., it is usually not possible to say that every transaction sees consistent data any time it runs. But we will be able to make "relaxed" guarantees, e.g., a constraint is true from 8am to 5pm every day, or a constraint is true if some "Flag" is set. Understanding and formalizing these relaxed guarantees is the primary problem we address in this paper. This is especially challenging because we now have to deal with the timing of actions and of guarantees. (In centralized database systems, typically only order is important, not timing.) However, the advantages of being able to handle relaxed guarantees in heterogeneous systems are significant; there are many loosely coupled applications, and knowing precisely what holds and what does not hold, and when, will clearly lead to more trustworthy systems.

Our work takes a first step in addressing these problems by providing a framework and toolkit for managing constraints that span heterogeneous sites. The key components are illustrated in Figure 1. The Constraint Manager (CM) is shown as being centralized only to simplify the presentation of our framework. As discussed in Section 9, our toolkit uses a distributed CM. Our framework includes mechanisms for formally describing the following:

- **Interfaces**. For each data item involved in a constraint, the interface for that data item describes, using a rule-based notation, how the item may be read, written, and/or monitored by the CM. For example, the interface for a data item X might specify that a request from the CM to read X will be serviced within t_1 seconds[1], and that any user update to X will result in a notification to the CM within t_2 seconds. The interface for each data item is dependent on the facilities provided by the database system containing that item.[2]

[1]We consider seconds as our time unit in this paper, but our approach applies equally well with other, presumably shorter, time units.

[2]Note that we do not fix a specific granularity for "data items" here. For example, a data item might be a single object or it might be the set of all tuples in a database relation.

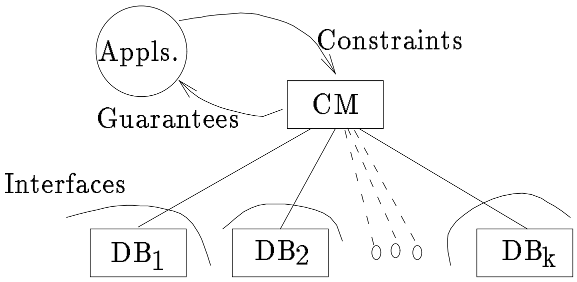

Figure 1: Constraint Management Architecture[4]

- **Strategies**. For a given constraint, a strategy is the specification of an algorithm used by the CM for monitoring or maintaining the constraint. Strategies also are described in a rule-based notation, and they incorporate the operations available for the data items involved in the constraint. For example (informally), a strategy for maintaining the constraint $X = Y$ might specify that all updates to X are propagated to Y, while all updates to Y are undone.

- **Guarantees**. For a given constraint, a guarantee is a logical description of the consistency guaranteed for that constraint. For example, a guarantee might state that a certain constraint holds provided that there have been no updates to any data items involved in the constraint within the last t seconds.

Figure 1 depicts the relationship between the constraint manager, the databases, and the applications or users.[3] Each database offers interfaces to the CM for its data items. The application or user informs the CM of each constraint that needs to be maintained. The CM provides a guarantee to the application or user, based on the interfaces and the strategy it decides to follow in order to maintain the constraint. (The strategy is used at run-time and is not shown in the figure.)

Using our mechanisms, it is possible to capture a wide variety of constraint management techniques over a wide variety of heterogeneous environments, and to provide formal guarantees for "relaxed" constraints. We also show how our strategies and interfaces can be used in practice. In particular, we describe a toolkit of general-purpose constraint management and translation services that can be easily configured to a given environment. Using the toolkit, one can describe the interfaces available for each database, and one can select strategies from a menu of proven strategies (examples of which are given in this paper). The constraint management services will then enforce the desired constraints. A first version of the toolkit has been implemented and is under evaluation.

We begin by discussing related work in Section 2. After presenting the framework in Section 3, we use it in Section 4 to specify several interfaces which we believe occur frequently in practice.

[3]Here and in the remainder of the paper we refer to the distributed sites as "databases". However, our framework applies equally well to environments with more general data repositories or information sources.

[4]This is a simplified version of Figure 2.

In Section 5 we describe how strategies are specified, and Section 6 illustrates the specification of guarantees. In Section 7, we present four extended examples dealing with the family of simple copy and inequality constraints, i.e., constraints of the form $X = Y$ and $X \leq Y$. Section 8 discusses a number of relevant issues and extensions. In Section 9, we present our constraint management toolkit and show how it can be used. For the sake of readability, a number of technical details have been omitted from the body of the paper; these are provided in Appendices A–E.

2 Related Work

Most previous work in database constraint management addresses centralized or tightly coupled distributed environments. Constraint management techniques for centralized systems are clearly inappropriate in loosely coupled environments such as those considered here. Methods for tightly coupled distributed databases also are inapplicable; both [SV86] and [Gre93] provide useful techniques for monitoring constraints in distributed databases, but these techniques rely on a traditional notion of data fragmentation and the presence of global transactions.

Reference [CW93] describes a constraint maintenance method for a multidatabase environment in which each database is relational, supports basic SQL operations, and has a production rules facility; in addition, there must be a persistent queue facility between sites. Similarly, [RSK91] describes a framework for interdatabase constraints based on a homogeneous relational interface to each database. Neither approach is applicable in a truly heterogeneous environment where each database offers a different interface to the constraint manager, and where some (or all) of the databases may not have the required features.

There has been some work on specific constraint management strategies in a loosely coupled environment. For example, the *Demarcation Protocol* [BGM92] is a method to maintain simple arithmetic constraints. Reference [GW93] describes a method for checking distributed constraints at a single site whenever possible. These are are special cases of the more general framework we present here.[5]

Another approach to constraint management in multidatabase environments is to extend the transaction concept to multidatabases by suitably weakening the traditional notion of correctness of schedules [Elm91]. This approach typically restricts the data items that may be involved in a constraint (e.g., constraints may be over local data only for *local-serializability* [BGMS92]). These approaches differ from ours in that with extended transactions there still is no mechanism to allow different interfaces at the participating sites, and no way to monitor constraints that only hold at particular times.

Work in temporal databases [Sno86] concentrates on extending database systems to allow the efficient storage and retrieval of temporal information, and shares with our work the issue of modeling time. However, here we are interested in modeling time to specify interfaces and strategies, and to express satisfaction of database constraints during certain intervals, not to represent historical

[5] In fact, in Section 7.4 we present the Demarcation Protocol in our framework and prove the associated guarantee.

data.

The problem of describing temporal aspects of systems has been studied before. The temporal logic programming [AM89] approach is a state-based approach that extends Prolog [CM81]. This approach is well-suited to describing reactive systems. However, for modeling distributed systems that communicate by message-passing, an event-based approach is more natural and convenient. This observation has also been made in [L+94]. Indeed, our event-based notation is very similar to that in RAPIDE [L+94].[6] However, RAPIDE is designed for the analysis and specification of distributed systems that are typically tightly coupled and that interact in much more complex ways than the systems we describe. As a result, RAPIDE is a richer, and consequently more complex, full-fledged programming language, which is far in excess of our needs. A similar observation applies to other event-based languages, including hardware description languages such as VHDL [IEE87] and Verilog [TM91], and software modeling languages such as LOTOS [BB87] and Esterel [BG92]. We believe that our model can more effectively and understandably describe our interfaces and strategies, since it is specifically targeted at these.

3 Model and Notation

In this section we introduce the notational and formal building blocks that are used throughout the paper. This includes a model of execution for loosely coupled distributed databases that is relatively simple but still expressive enough to capture those aspects of execution that are relevant to the monitoring and maintaining of multi-database constraints.

3.1 System State

Let $\{D_1, D_2, \ldots, D_n\}$ be the set of all data items in the system, including all the databases and any data stored by the constraint manager. An *interpretation* I is a function that maps each D_i to a value, yielding a state of the system. For example, if we have three data items $\{D_1, D_2, D_3\}$, then a possible interpretation is $\{D_1 = 7, D_2 = 14, D_3 = 49\}$. We permit an interpretation to "under-specify" the state by allowing some data items to map to null, meaning these data items can assume any value. The system passes through a sequence of states, each represented by an interpretation of its data items.

Note that although we are modeling the global state of a distributed system, this does not cause any of the problems usually associated with distributed state, since we use it for reasoning only. The interfaces and strategies are designed in such a way that no "access" to the global state is assumed or required.

[6] In fact, our rule-based language may be considered a very small subset of the RAPIDE specification language where we use only one of the many available "pattern operators," the "dependent" operator '$\rightarrow$'.

3.2 Events

The behavior of the databases and the constraint manager is described by *events*. For the purposes of constraint management, we divide events into two types:

- *Spontaneous events*, which occur as a result of users or application programs operating on the databases.
- *Generated events*, which occur as a (direct or indirect) result of a strategy being executed by the CM or an interface being maintained by a database.

Each event is represented using a six-tuple: $E =$ (time, desc, old, new, rule, trigger), where the components of the tuple are described below:

time: The time at which the event E occurs. For simplicity, we assume that all references are to global "physical" time. We use time mainly for reasoning about correctness, and as we will see, in practice we do not require synchronized clocks.

desc: The descriptor of the event, drawn from a fixed set of descriptors (see Section 3.2.1).

old: The interpretation representing the state of the system just before the event occurs.

new: The interpretation representing the state of the system just after the event occurs.

rule: If E is a generated event, this is a rule whose "firing" resulted in the occurrence of this event. If E is a spontaneous event, this component is null. Rules are described in Section 3.4.

trigger: If E is a generated event, this is the event which caused the rule above to fire. If E is a spontaneous event, this is null. Again, we elaborate on this in Section 3.4.

For an event E, we denote a component of the event using dot notation. For example, $E.old$ denotes the *old* component of the event E.

3.2.1 Event Descriptors

The following simple event descriptors suffice to model the interfaces and strategies we consider in this paper. (It is straightforward to add new event descriptors to our framework for more complex interfaces or strategies.) Throughout this paper we use lower-case letters to represent variables, Greek letters to represent constants, and capitalized words and upper-case letters to represent data items.

- *Spontaneous write:* A spontaneous write event of descriptor $W_s(X, \alpha, \beta)$ represents the update of a data item X from value α to value β by users or application programs operating on the database. For example, $W_s(D_1, 5, 10)$ is an event descriptor that changes the value of the data item D_1 from 5 to 10.

- *Generated write:* This is a write operation caused (directly or indirectly) by the CM. For example, $W_g(X, \beta)$ is the descriptor of an event that assigns the value β to X. (Here X may be a data item in a database or a data item maintained by the CM.)
- *Notify:* An event of descriptor $N(X, \beta)$ represents a database system notifying the CM that data item X has value β. (This is a generated event.)
- *Write request:* An event of descriptor $WR(X, \beta)$ represents a request from the constraint manager to a database system to update data item X to have value β. (This is a generated event.)

3.2.2 Example

Event E_1 = (

Time:	9:01 p.m.
Desc:	$W_g(X, 5)$
Old:	$\{(X = 10), (Y = 20), (Z = 0), \ldots\}$
New:	$\{(X = 5), (Y = 20), (Z = 0), \ldots\}$
Rule:	$WR(X, \beta) \rightarrow W_g(X, \beta);\ B \leq 5$
Trigger:	Event E_2)

This event occurs at 9:01 p.m. with descriptor $W_g(X, 5)$, which intuitively represents a write operation $X := 5$ generated by the constraint manager. The *old* interpretation maps X to 10, while the *new* interpretation maps X to 5. All other state variables are mapped to the same value by both *old* and *new*. Since this is a generated write, it has a rule that caused it to occur. While rules are described in detail later, the intuitive meaning of the rule in this event is that a write request event causes a write event to occur within 5 seconds. The trigger field specifies that event E_2 fired the rule. For brevity, to refer to an event we use the notation $E@t$, where E is the event's descriptor and t is its time. Thus the event above is referred to as $W_g(X, 5)$@9:01pm.

3.3 Event Templates

We define an *event template* to be an event descriptor in which some of the components are parameterized or "wild-carded." An event template represents the set of all event descriptors that can be obtained from the template by substituting particular values for the parameters and wild-cards. For example, $W_s(X, b)$ represents the infinite set of spontaneous write event descriptors that have X as the first component and any value as the second component. We use "_" to denote a wild-card—a parameter whose name is not important. Thus $W_g(\_, \_)$ represents the set of all generated write event descriptors (of any value to any data item in the system). We use $\mathcal{E}$ to denote event templates. In the sequel we use $W_s(X, b)$ as shorthand for $W_s(X, \_, b)$.

3.4 Rules

The simplest form of a rule is $\mathcal{E}_1 \rightarrow \mathcal{E}_2;\ B \leq \delta$, which states that the occurrence of any event matching the event template on the left hand side (LHS) causes the occurrence of an event satisfying the event template on the right hand side (RHS) within δ seconds. (We use the reserved word B to denote the bound on the delay between the occurrences of the LHS and the RHS events.) For example, the rule $W_s(X, b) \rightarrow N(X, b);\ B \leq 7$ states that if a spontaneous write, say $X := 5$, occurs at time t, then a notify event with descriptor $N(X, 5)$ will occur at some time in the time interval $[t, t+7]$. Note that the b in the RHS event template is a parameter whose value is obtained by "matching" the event template on the LHS with a particular event when it occurs. Matching is defined formally in Appendix A.

The LHS of a rule can also have a condition, evaluated when the LHS event occurs. To illustrate, consider the following rule. It intuitively states that the CM is notified every time the value of X changes by more than 10 percent, and that the notification occurs within 5 seconds of the change in X's value.

$$W_s(X, a, b) \wedge (|b - a| > a * 0.1) \rightarrow N(X, b);\ B \leq 5$$

Suppose a spontaneous write $W_s(X, 1, 2)$ occurs at time 10. The event $W_s(X, 1, 2)$ matches the event template on the LHS of the rule above, and when we substitute $\{(a = 1), (b = 2)\}$ the LHS condition is satisfied. Therefore, the rule fires, meaning that at some time $t' \in [t, t+5]$, the RHS event occurs. The RHS event is obtained from the RHS event template $N(X, b)$ by substituting $\{(a = 1), (b = 2)\}$, which yields $N(X, 2)$. We therefore conclude that an event with descriptor $N(X, 2)$ occurs at some time $t' \in [t, t+5]$.

Similarly, the RHS of a rule may also have a condition. As an example, consider the following:

$$N(X, b) \rightarrow (b \neq \mathrm{Cy})?W(\mathrm{Flag}, \mathrm{false});\ B \leq 7$$

Suppose an event with descriptor $N(X, 2)$ occurs at time t. This event matches the LHS event template of the above rule, and the rule therefore fires (since there is no LHS condition). This means that at some time $t' \in [t, t+7]$, the RHS condition $(b \neq \mathrm{Cy})$ is evaluated and if it is true, an event with descriptor $W(\mathrm{Flag}, \mathrm{false})$ occurs. If the RHS condition is false, no event is caused by the firing of this rule.

Finally, the RHS of a rule may contain a sequence of condition-event pairs. For example, consider the following rule:

$$\begin{aligned} N(X, b) \rightarrow\ & (b \neq Cy)?W_g(\mathrm{Flag}, \mathrm{false}), \\ & (b = Cy)?W_g(\mathrm{Flag}, \mathrm{true}), \\ & W_g(Tb, 5);\ B \leq 10 \end{aligned}$$

The above rule states that if a notify event $N(X, b)$ occurs, then the condition-event pairs on the RHS are evaluated in the order written. That is, first the condition $(b \neq Cy)$ is evaluated, and if true, the corresponding event $W_g(\mathrm{Flag}, \mathrm{false})$ occurs. Next, the second condition, $(b = Cy)$ is evaluated, and if true, the corresponding event $W_g(\text{Flag}, \text{true})$ occurs. Finally, the third condition-event pair (which has a null condition) is evaluated, so that the event $W_g(Tb, 5)$ occurs. Further, the $D : 10$ specifies that this entire process takes no more than 10 seconds.

A formal definition of rules and the semantics of rule firing is given in Appendix A.

3.5 Executions

Given a set $\mathcal{R}$ of rules (which are used to describe the interfaces and strategies of the system), an *execution* models the sequence of events that occur in the distributed system that are of interest for constraint management, along with the intervening states of each database. Formally, an *execution* is a sequence $(E_1, E_2, \ldots, E_n)$, where each E_i is an event as described in Section 3.2. (An execution in our model is analogous to the concept of *histories* in concurrency control theory.) Intuitively, we say an execution is *valid* if it "follows the rules," both the rules of normal system execution and the rules in $\mathcal{R}$. In valid executions, we assume in-order processing of events occurring at each site and in-order message delivery between sites.

As an example, consider a simple system with only one data item X and one rule:

r_1: $W_s(X, a, b) \wedge (b > a + 10) \rightarrow N(X, b)$; $B \leq 5$.

An example of a valid execution is the sequence of events: $(E_1, E_2, E_3, E_4, E_5)$, where

$E_1 = (9{:}01, W_s(X, 1, 3), \{X = 1\}, \{X = 3\}, null, null)$,
$E_2 = (9{:}03, W_s(X, 3, 19), \{X = 3\}, \{X = 19\}, null, null)$,
$E_3 = (9{:}04, W_s(X, 19, 23), \{X = 19\}, \{X = 23\}, null, null)$,
$E_4 = (9{:}05, N(X, 19), \{X = 23\}, \{X = 23\}, r_1, E_2)$,
$E_5 = (9{:}07, W_s(X, 23, 29), \{X = 23\}, \{X = 29\}, null, null)$.

Note that $E_2.old = E_1.new$, $E_3.old = E_2.new$, and so on. Furthermore, event E_2 satisfies the LHS of rule r_1 and indeed, we see that E_4 is triggered. Note that since the LHS condition of the rule is false for the other W_s events, the rule does not fire for those events. The complete definition of a valid execution is given in Appendix B.

A *valid execution prefix* is any prefix of a valid execution. Above, (E_1, E_2) is a valid execution prefix, but (E_1, E_2, E_5) is not.

4 Specifying Interfaces

The interface for a data item involved in a constraint describes how that data item may be read, written, and/or monitored by the constraint manager. Interfaces are specified using a restricted form of the *rule* notation described in Section 3.4. Note that it is important for interface specifications to correctly reflect the actual behavior provided by the database containing that item. For example, suppose an interface specification is too "conservative," e.g. it specifies that actions will take longer to execute than they actually do, or it does not include operations that actually are supported. Then, although the correctness of our approach is not compromised, the most efficient strategies and the strongest guarantees may not be provable. If an interface specification is too "liberal," e.g. if it specifies time intervals that are too short for the underlying database to honor, or if it specifies unsupported operations, then our framework may permit strategies that are not executable on the systems being modeled. Currently, we rely on the users of our framework to verify that the interfaces specified do faithfully represent the actual systems.

4.1 Interface Specification Language

For each data item, its interface is defined by a set of *interface statements* of the form:

$$\mathcal{E}_1 \wedge C \rightarrow \mathcal{E}_2;\ B \leq \delta$$

The meaning of this statement is: If event E_1 matching template $\mathcal{E}_1$ occurs at time t and condition C (involving local data items) is true at t, then the database guarantees that an event E_2 matching template $\mathcal{E}_2$ will occur at some time in $[t, t+\delta]$. The condition C is implicitly understood as being evaluated at the time the LHS event occurs. If C is the constant condition "true" it may be omitted. Formally, interface statements are just a specialization of *rules* in which the condition C_2 on the RHS of the '$\rightarrow$' is always true, and hence omitted.

Our interface notation can also specify events that are guaranteed *not* to occur. To denote that an event matching a certain event template $\mathcal{E}_1$ cannot occur, we write:

$$\mathcal{E}_1 \rightarrow \mathcal{F}$$

where $\mathcal{F}$ is a special event descriptor called the "false" event descriptor.[7] By definition, there can be no event matching $\mathcal{F}$, so there can be no valid execution in which an event matching $\mathcal{E}_1$ occurs.

4.2 Examples

In the heterogeneous system we model, each database offers a potentially different interface for each of its data items. These interfaces can be quite varied and complex. We believe that our language can describe most of the interfaces that actually occur in practice. Here are some examples:

Write: If a write request is received, the requested write operation will be executed within δ seconds.
$WR(X,b) \rightarrow W_g(X,b);\ B \leq \delta$

No Spontaneous Write: There are no spontaneous writes to the data item.
$W_s(X,a,b) \rightarrow \mathcal{F}$

Simple Notify: A notification will occur within time δ of each spontaneous write.
$W_s(X,a,b) \rightarrow N(X,b);\ B \leq \delta$

[7] The meaning of omitting "$B \leq \delta$" is that the RHS event occurs at some time in the future, with no specific bound.

Percent Change Notify: A notification will be received within time δ of a spontaneous write to X if the change in the value of X is more than P percent. This requires access to both the old and the new value of X.
$W_s(X, a, b) \wedge (|b - a| > a * P/100) \rightarrow N(X, b);\ B \leq \delta$

Periodic Notify: By defining periodic spontaneous events and a special data item T_{now} that represents the local time, one can define more complex interfaces. For example,

$$P(0, 1000) \wedge (8:00am < T_{now} < 5:00pm) \wedge (X = b) \rightarrow N(X, b);\ B \leq \delta$$

says that a notify message with the value of X will be sent between 8:00am and 5:00pm every 1,000 time units (starting at time 0). We do not further discuss these types of interfaces here; we merely want to point out they are possible.

Since the CM and database are likely to reside at separate sites, note that these interfaces incorporate the delay incurred by communication. Also note that interfaces can be defined over sets of items, e.g., the same definition may hold for all salary fields in a relation. This is further discussed in Section 8.

4.3 Failure Handling

We assume that the interface specifications are faithful descriptions of the actual interfaces. However, it is possible that due to unforeseen circumstances, a database may be unable to honor its interface specification. For example, there may be an uncharacteristically high load on the database, some deadlock may cause system standstill, or there may be some operating system or hardware failures. In such a situation, it is inevitable that the interface specification will be violated. In this section, we outline mechanisms to deal with such failures.

We can broadly classify failures in our system into two categories based on whether they can be detected by the constraint management system.

4.3.1 Detectable Failures

Consider a read interface described by the following interface specification, which says that if a CM issues a read request (RR) for a data item X, the database responds with the current value of X within δ seconds:

$$RR(X) \wedge (X = b) \rightarrow R(X, b);\ B \leq \delta$$

If the CM sends a $RR(X)$ to this database and does not receive a response in δ seconds, then it can detect this failure. In this situation the CM could pursue different options. The simplest is to flag a global failure state throughout the system, indicating that all guarantees are off until further notice. The system can then be reset by manual intervention after the problem with the database that failed is fixed. A slightly more sophisticated option is to flag as invalid only those guarantees that are dependent on the interface that has failed.

4.3.2 Undetectable Failures

Consider a notify interface of the kind described in Section 7.1:

$$W_s(X, b) \rightarrow N(X, b);\ B \leq \delta$$

If the database implementing this interface fails to honor it, there is no way the CM can detect this failure.[8] Such a failure leads to a failure of the CM system in the sense that some of the guarantees will no longer hold. However, such a phenomenon is part of every system's design. For example, undetected hardware failures can silently corrupt data on a disk, or the main memory, which in turn can potentially cause undetectable failures in applications. Timing assumptions such as the ones we make here always have to be made in reliable, asynchronous distributed systems [FLP85]. For example, one frequently assumes that an active site responds to a message within w seconds, else it is considered failed. The main difference is that we are making these timing assumptions *explicit* as opposed to "hiding" them in an assumed commit protocol or a reliable network layer. We need only ensure that the probability of such failures is low enough to be acceptable for a particular class of applications.

The probability of such failures of interfaces depends on the choice of the time constants (such as the constant δ above) in the interval specifications of interfaces. Typically, the constants would be determined based on the processing power of the information source, the expected load, the maximum estimated communication delay (including retries), etc. In practice, these constants could be chosen to ensure that the interface is honored with, say, 99.99% probability, while flagging a failure on the few occasions when the database fails to honor the interface. These probabilities can be reflected in the guarantees the CM offers by associating a probability with each guarantee, discussed later in Section 6.2.

5 Specifying Strategies

The strategy for a constraint describes the algorithm used by the constraint manager to monitor or maintain the constraint. Like interfaces, strategies are specified using a restricted form of the *rule* notation described in Section 3. In addition to performing operations on data items involved in a constraint, strategies may evaluate predicates over the values of data items (obtained through database read operations) and over private data maintained by the Constraint Manager. Once our framework has been used to specify a strategy—and, presumably, to verify the correctness of a guarantee—then the rule-based strategy specification must be implemented using the host language of the Constraint Manager. In most cases this translation should be relatively straightforward.

[8] Using this interface *only*. Detection using other interfaces may be possible. For example, if the database also supports a read interface, the CM could poll X and detect a change in value that it was not notified of.

5.1 Strategy Specification Language

The strategy for a given constraint is defined by a set of *strategy statements* of the form:

$$\mathcal{E}_1 \to C?\mathcal{E}_2;\ B \leq \delta$$

where $\mathcal{E}_1$ and $\mathcal{E}_2$ are event templates and C is a condition involving data items local to the CM and variables. This statement states that if an event matching template $\mathcal{E}_1$ occurs at time t, then there exists a unique time t' in $[t, t+\delta]$, such that if C is true at t' then an event matching template $\mathcal{E}_2$ occurs at time t'. If C is the constant condition "true" it may be omitted. More formally, strategy statements are a specialization of *rules* in which the condition C_1 on the LHS of the '$\to$' is always true, and hence omitted.

5.2 Example

Consider the copy constraint $X = Y$, where X and Y are at different sites. Further, let the interfaces for both X and Y be "notify" interfaces, meaning that the constraint manager is notified whenever there is a write to X or Y (the Simple Notify interface in Section 4.2). The Constraint Manager could follow many different strategies for monitoring or enforcing $X = Y$, each leading to a different guarantee. We illustrate a very simple strategy that consists of the CM maintaining a Boolean flag such that if the flag is true and if there have been no "recent" writes to X or Y, then $X = Y$. (This is the guarantee offered by the CM; we elaborate on it in Section 6.1.) The constant δ represents the bound on execution delay, which we assume is the same for each strategy statement. Data items Cx and Cy are local to the CM, and are used to record the most recent values of X and Y known to the CM.

$N(X,b) \to W(\mathrm{Cx}, b);\ B \leq \delta$
$N(X,b) \to (b \neq \mathrm{Cy})?W(\mathrm{Flag}, \mathrm{false});\ B \leq \delta$
$N(X,b) \to (b = \mathrm{Cy})?W(\mathrm{Flag}, \mathrm{true});\ B \leq \delta$
$N(Y,b) \to W(\mathrm{Cy}, b);\ B \leq \delta$
$N(Y,b) \to (b \neq \mathrm{Cx})?W(\mathrm{Flag}, \mathrm{false});\ B \leq \delta$
$N(Y,b) \to (b = \mathrm{Cx})?W(\mathrm{Flag}, \mathrm{true});\ B \leq \delta$

We assume there are no spontaneous writes to data items Cx, Cy, and Flag, which can be formalized with rules, as shown in Section 4.2. Intuitively, the above strategy specification can also be expressed as pseudo-code. Here we show the code for X; the one for Y is similar:

When a $N(X,b)$ occurs at time t do the following (in any order) {
  Cx $\leftarrow b$;
  if ($b = \mathrm{Cy}$) then Flag $\leftarrow$ true;
  if ($b \neq \mathrm{Cy}$) then Flag $\leftarrow$ false;
} within time $t + \delta$

In general, strategies could be quite complex, and our framework is capable of expressing these complex strategies. However, in practice we expect most of the constraints we deal with to be relatively simple (see Section 8), so strategies should be relatively simple as well.

6 Specifying Guarantees

A guarantee for a constraint specifies the level of global consistency that can be guaranteed by the Constraint Manager when a certain strategy for that constraint is implemented. Typically a guarantee is *conditional*, e.g., a guarantee might state that if no updates have recently been performed then the constraint holds, or if the value of a CM data item is true then the constraint holds.

6.1 Guarantee Specification Language

Before we describe our guarantee specification language, we define *time-expressions* and *occurrences*. A time-expression (texp) is one of the following:

1. a constant representing a time instant (e.g., *11:00 a.m.*)
2. a variable representing a time instant (e.g., t).
3. an arithmetic expression involving (1) and (2).
4. an interval with (1)...(3) as its end-points (e.g., the half-open interval $(11{:}00, 11{:}00 + t - 5]$).

We use $texp(t)$ to denote a time-expression that uses a variable t. We use time expressions to refer to events that occur in a particular time period. For example, $W_g(X,5)@[t, t+5]$ represents the *occurrence* of an event with descriptor $W_g(X,5)$ at *some* time in the interval $[t, t+5]$.[9] The general form of an occurrence is $E@texp$ where E is an event descriptor or template, and $texp$ is a time-expression.

Guarantees are expressed using *guarantee statements*, which are logical expressions involving occurrences and conditions. For guarantees, we look upon an occurrence as an atomic formula with a truth value. Thus $E@texp$ is true if event E occurs at a time specified by $texp$, and false otherwise. We then use $\neg(E@texp)$ to represent the negation of this occurrence. We use $\neg E@@texp$[10] as syntactic sugar for $\neg(E@texp)$; both state that event E does not occur at any time specified by $texp$. Hence, the operator @@ may be read as "at all." In guarantees, we also use conditions with time expressions: $C@texp$ states that condition C holds at some time in the interval denoted by $texp$; $C@@texp$ states that condition C holds in the entire time interval denoted by $texp$. In its most general form, a guarantee is an arbitrary Boolean expression involving occurrences, conditions, and time-expressions.

[9] Thus the operator @ is existential; it represents the event occurring at some instant of time in the given interval.

[10] The connective $\neg$ has a higher precedence than @ and @@, both of which have the same precedence as $\wedge$ and $\vee$.

A guarantee that the CM may offer for the constraint, interfaces and strategy described in Section 5.2 illustrates this:[11]

$$(\text{Flag} = \text{true})@t \wedge \neg W(X|Y)@@[t-\eta-\delta, t] \Rightarrow (X = Y)@@[t-\eta-\delta, t]$$

This guarantee states that if the value of the data item "Flag" is true at any time t, and if there has been no write to either X or Y in the last $\eta + \delta$ seconds, then the constraint $X = Y$ was true during the last $\eta + \delta$ seconds.

6.2 Probabilities and Guarantees

As discussed in Section 4.3, we can associate a probability of failure with each interface statement. Similarly, we can associate a probability of failure with each strategy statement. Using these, we can calculate the probability of failure of each guarantee. For example, a guarantee may read "$\neg W(X)@@[t-\eta-\delta-\omega, t] \Rightarrow (X = Y)@t$ with probability 99.99%." Note that the probability of failure for an interface will usually depend on the delay associated with the interface statements. Thus the choice of the delay and the failure probability are related. To arrive at particular values for these, we could model the implementations of the interface and strategy statements, then derive a graph of failure probability versus delay. Such a graph typically indicates how the failure probability decreases as the delay associated with the interface is increased. Depending on the application requirements, we can select a suitable point on this graph as a compromise between short delays and low failure probabilities.

This completes the presentation of our model. In the sections that remain we will show some complete examples (Section 7), discuss how guarantees can be used by application programs and users (Section 8), discuss various issues regarding the complexity of proving guarantees (Section 8), and finally, present an implemented toolkit that can be used in practice for enforcing constraints across heterogeneous databases (Section 9).

7 Extended Examples

We now present a set of examples that illustrate the application of our constraint management framework. In each example, we use our formalism to define the interfaces, describe the strategy and specify the guarantee offered by the constraint manager. The proofs of the guarantees are relegated to Appendix C. In Sections 7.1–7.3, we consider the copy constraint $X = Y$. In Section 7.4, we consider the Demarcation Protocol [BGM92] for the inequality constraint $X \leq Y$.

7.1 Notify–Write

Interfaces: Consider the constraint $X = Y$, where X and Y are at different sites. Let X have a "simple notify" interface, meaning that the CM is notified every time there is a spontaneous write

[11]We use the notation $E(D_1|D_2)$ as shorthand for $E(D_1) \vee E(D_2)$. Similarly, $W(X)$ is used as shorthand for $W_s(X) \vee W_g(X)$, where we omit the value written since it is unimportant.

to X. Let Y have a "write interface," meaning that on receiving a request from the CM to write a certain value to Y, that write operation will be performed within a certain time. Further, assume Y also has a "no spontaneous writes" interface, meaning that there can be no spontaneous writes to Y. Using our interface specification language, these interfaces are stated as follows:

$W_s(X, b) \to N(X, b);\ B \le \eta$ (simple notify) ....(1)
$WR(Y, b) \to W_g(Y, b);\ B \le \omega$ (write) ....(2)
$W_s(Y, \_) \to \mathcal{F}$ (no spontaneous writes) ....(3)

Constants η and ω represent the delay associated with the notification and the write request, respectively.

Strategy: Given the above interfaces for X and Y, a very simple way to maintain the constraint $X = Y$ is the propagation of updates from X to Y. In more detail, this means that on receiving a notification of a write to X, the CM makes a request that the same value be written to Y. We express this using our strategy specification language as follows:

$N(X, b) \to WR(Y, b);\ B \le \delta$ (update propagation) ....(4)

Constant δ represents the time required to emit the write request after receiving the notify.

Guarantee: Given that the CM propagates updates from X to Y in the above manner, and given that there can be no spontaneous updates to Y, it is intuitively clear that the value of Y should follow that of X. There is, however, a delay in the update propagation, during which more recent updates to X may not be reflected in Y. Since the propagation of an update from X to Y involves the "firing" of the three rules (1), (4), and (2) one after the other, it is easy to see that the maximum delay is the sum of the delays in each of these three rules, $\eta + \delta + \omega$. Therefore, at any given time, if there have been no updates to X in the last $\eta + \delta + \omega$ seconds, then $X = Y$. Using our guarantee specification language, we state this as follows:

$\neg W(X)@@[t - \eta - \delta - \omega, t] \Rightarrow (X = Y)@t$

The proof of this guarantee is presented in Section Appendix C.1.

7.2 Notify–Notify version 1

Interfaces: Consider again the constraint $X = Y$, and now let X and Y both have notify interfaces. Using our interface specification language, we state this as follows, where η is the maximum delay between a write to X or Y and the corresponding notify event:

$W_s(X, b) \to N(X, b);\ B \le \eta$ ....(1)
$W_s(Y, b) \to N(Y, b);\ B \le \eta$ ....(2)

Strategy: There are many different strategies that may be used in this scenario. Consider a strategy in which the CM maintains four data items of its own: Cy, which is a "cached copy" of Y, Cx, which is a cached copy of X, Tb, which is a data item storing a timestamp, and Flag, which is a boolean data item. The CM uses Cx and Cy to maintain Flag in such a manner that if Flag is true, then $X = Y$ during a "recent" time interval. The particulars are as follows. On receiving a notification of a write to X, the CM updates Cx to the new value of X and writes the current time to the timestamp data item Tb. (We assume the availability of the current time in a special data item T_{now}.[12]) If the new value of X is equal to the cached value of Y, the CM sets Flag to true; otherwise, Flag is set to false. Using our strategy specification language, we describe this strategy as follows:

$N(X,b) \rightarrow W_g(\mathrm{Cx},b);\ B \leq \delta$ .... (3)
$N(X,b)@t \rightarrow (b \neq \mathrm{Cy})?W_g(\mathrm{Flag},\mathrm{false}), (b = \mathrm{Cy})?W_g(\mathrm{Flag},\mathrm{true}), W_g(\mathrm{Tb},T_{now});\ B \leq \delta$ ...... (4)

Item Y is handled in a completely analogous manner; we would have four symmetric rules for it. We also need to specify that there are no spontaneous writes to the data items Cx, Cy, Flag and Tb. This adds four more rules, with $\mathcal{F}$ on the RHS, to the above strategy. The details are in Appendix C.2.

Guarantee: Intuitively, we see that the above strategy monitors the constraint $X = Y$ and indicates whether it holds by using the Flag data item. Further, Tb indicates the time after which the CM has not acted upon any notification. If rules fired instantaneously, this would mean that Flag = true implies $X = Y$ in the interval $[\mathrm{Tb}, t]$, where t is the current time. However, since there are delays involved in the firing of each rule, this interval needs to be adjusted to $[\mathrm{Tb}, t - \eta - \delta]$. We therefore state the guarantee as follows:

$((\mathrm{Flag} = \mathrm{true}) \wedge (\mathrm{Tb} = s))@t \Rightarrow (X = Y)@@[s, t - \eta - \delta]$
We prove this guarantee in Section Appendix C.2.

7.3 Notify–Notify version 2

Given constraint $X = Y$ and interfaces identical to those in the previous section, we consider a slightly different strategy in this section, which leads to a different guarantee.

Interfaces: (Same as in the previous section.)
$W_s(X,b) \rightarrow N(X,b);\ B \leq \eta$ .... (1)
$W_s(Y,b) \rightarrow N(Y,b);\ B \leq \eta$ .... (2)

Strategy: (Same as in the previous section, except the Tb data item is not maintained by the CM.)

$N(X,b) \rightarrow W_g(\mathrm{Cx},b);\ B \leq \delta$ .... (3)
$N(X,b) \rightarrow (b \neq \mathrm{Cy})?W_g(\mathrm{Flag},\mathrm{false});\ B \leq \delta$ .... (4)
$N(X,b) \rightarrow (b = \mathrm{Cy})?W_g(\mathrm{Flag},\mathrm{true});\ B \leq \delta$ .... (5)

As in the strategy in the previous section, we need to add analogous rules for Y and we need to

[12]Note that references to the "clock" data item T_{now} are the only use of actual (wall clock) time in our framework. These references are restricted to the Constraint Manager and therefore do not pose any of the problems associated with time in a distributed system. If the Constraint Manager is distributed, we may need to assume access to a global clock within some accuracy threshold, which we believe is not unreasonable.

specify that there are no spontaneous writes to the data items Cx, Cy and Flag. This adds six more rules as shown in Appendix C.3.

Guarantee: Using the above strategy, if there have been no writes to X or Y during a time interval, then Flag = true implies that $X = Y$ during that time interval, adjusted for timing delays. Formally, we state this as follows:

$(\text{Flag} = \text{true})@t \wedge \neg W(X|Y)@@[t-\eta-\delta, t] \Rightarrow (X = Y)@@[t-\eta-\delta, t]$
The proof of this guarantee is presented in Section Appendix C.3.

7.4 Demarcation Protocol

The Demarcation Protocol [BGM92] is a mechanism for maintaining simple arithmetic constraints over data items at different sites. We consider a simple case of the Demarcation Protocol for the constraint $X \leq Y$, where X and Y are at different sites. The protocol splits this constraint into three separate constraints: $X \leq X_l$, $Y \geq Y_l$, and $X_l \leq Y_l$. The *limit data items* X_l and Y_l are local to X's site and Y's site, respectively. Note that these three constraints imply the original constraint, $X \leq Y$, so it is sufficient if they are maintained instead of the original constraint. Further, since the constraints $X \leq X_l$ and $Y \geq Y_l$ are local to the databases containing X and Y, respectively, only $X_l \leq Y_l$ requires coordination between sites.

Interfaces: We assume that the local constraints $X \leq X_l$ and $Y \geq Y_l$ are maintained by the respective local database systems. That is, no value greater than X_l is written to X and no value less than Y_l is written to Y. Further, we assume the databases offer a write interface (with no spontaneous writes) for data items X_l and Y_l. For simplicity, we define an incremental write request event descriptor: $WR_i(D, a)$ is a request to increment data item D by an amount a. The databases also offer an *accept-change* interface, which is similar to the write request interface except that the requested write operation is not guaranteed to be performed within any given time; all that is guaranteed is that the write will eventually occur. (The accept-change interface is thus weaker than the write-request interface.) We define a new event descriptor, $AC(D, a)$ to represent the CM requesting a database to increment D by a, with no timing requirements. AC is a generated event. Formally, we define the interfaces as follows:

$W_s(X, b) \wedge (b > X_l) \rightarrow \mathcal{F}$ (local constraint $X \leq X_l$) ....(1)
$W_s(X_l, b) \rightarrow \mathcal{F}$ (no spontaneous writes) ....(2)
$WR_i(X_l, a) \rightarrow W_g(X_l, X_l + a)$; $B \leq \omega$ (incremental write) ....(3)
$AC(X, a) \rightarrow W_g(X_l, X_l + a)$ (accept change)[13] ....(4)
$W_s(Y, b) \wedge (b < Y_l) \rightarrow \mathcal{F}$ (local constraint $Y \geq Y_l$) ....(5)
$W_s(Y_l, b) \rightarrow \mathcal{F}$ (no spontaneous writes) ....(6)
$WR_i(Y_l, a) \rightarrow W_g(Y_l, Y_l + a)$; $B \leq \omega$ (incremental write) ....(7)

[13]Recall that when we omit the delay specification "$D : \delta$" from a rule, we mean that the RHS event occurs at some time in the future, with no specific bound.

$AC(Y,a) \rightarrow W_g(Y_l, Y_l + a)$ (accept change) .... (8)

Strategy: We describe a strategy that implements the Demarcation Protocol, as described in [BGM92]. (For simplicity, we do not describe the associated *policies*.)

The local constraint management system at each site is responsible for maintaining the local constraints ($X \leq X_l$ and $Y \leq Y_l$). When an application requests an update to the local data item (X or Y) that would violate the local constraint, the local constraint management system requests the global CM system to raise the associated limit. We use the event descriptor $CLR(X,a)$ ("*change limit request*") to denote the local CM system at X's site making such a request to raise the limit X_l to $X_l + a$.

The event descriptor $CL(X,a)$ represents a generated *change-limit* event produced in response to either a change-limit-request event at the local site or a change-limit request at another site. Similar to $CLR(X,a)$, $CL(X,a)$ is a request to change X_l to $X_l + a$.

A write to X_l or Y_l is called *safe* if it is guaranteed not to violate $X_l \leq Y_l$. Thus a decrement to X_l and an increment to Y_l are safe, while an increment to X_l and a decrement to Y_l are unsafe.

On receiving a change-limit request, the CM first checks if the requested increment or decrement to the limit data item is safe. If so, the requested change is performed using an incremental write request operation. After this safe write operation is known to have completed, an unsafe write operation that increments or decrements the other limit data item by the same amount is requested at the other site using the accept-change interface. Note that this request need not be made within any fixed time interval for the protocol to work. If the requested change-limit operation is unsafe, the CM first performs the corresponding safe operation at the other site. When that operation completes, the requested unsafe operation will be performed using an accept-change operation. Formally, we specify this strategy as follows:

$CLR(X,a) \rightarrow CL(X,a)$ .... (9)
$CL(X,a) \wedge (a > 0) \rightarrow CL(Y,a);\ B \leq \delta$ .... (10)
$CL(X,a) \wedge (a \leq 0) \rightarrow WR_i(X_l,a), AC(Y,a);\ B \leq \delta$ .... (11)
$CLR(Y,a) \rightarrow CL(Y,a)$ .... (12)
$CL(Y,a) \wedge (a < 0) \rightarrow CL(X,a);\ B \leq \delta$ .... (13)
$CL(Y,a) \wedge (a \geq 0) \rightarrow WR_i(Y_l,a), AC(X,a);\ B \leq \delta$ .... (14)

Guarantee: The Demarcation Protocol strategy ensures that if $X_l \leq Y_l$ at the initial time T_{init} then $X_l \leq Y_l$ at all times after T_{init}. This corresponds to Theorem 4.1 in [BGM92]. We state this formally as follows:

$(X_l \leq Y_l)@T_{init} \Rightarrow (X_l \leq Y_l)@t$
We prove this guarantee in Section Appendix C.4.

8 Discussion

Although we only illustrated four simple scenarios, we believe that our framework (with a few straightforward extensions) is flexible enough to handle most common interfaces and constraints that arise in loosely coupled environments. For example, an interface may specify that a certain data item is updated only during certain hours at night. We can also extend our our framework for set-oriented constraints to allow constraints over relations and to allow quantification.[14] We could then specify, for instance, that for every tuple in relation R, the salary attribute should be less than some maximum salary S stored at another site. This is simply a set of constraints, each of the forms we have dealt with in this paper. With appropriate extensions, we can also describe more complex constraints, such as that each salary in R must be less than that of the employee's manager, or the sum of the salaries should be less than some maximum.

As we deal with more complex constraints and interfaces, there is no question that the strategies and proofs of correctness will grow in complexity. Thus, one could argue that the framework we have presented is only of interest in a few simple scenarios. To counter this argument, we note, as we did in the Introduction, that one does not have to prove each scenario each time it is used. Our goal is to provide families of commonly used interfaces, strategies, and proven guarantees. Thus, the difficult proofs only have to be done once. Furthermore, we believe that much of the specification and proof process can be automated.

A second key argument is that simple constraints (like the $X = Y$ and $X \leq Y$ we considered here) are the most common in a loosely coupled environment where it is unlikely that autonomous data repositories will have very complex interdependencies. Furthermore, if there are complex constraints in a heterogeneous system, they are often split into distributed copy constraints plus local constraints. For example, consider the constraint $X = Y + Z$, where X, Y, and Z are at three sites. A common way to manage this constraint is to have cached copies of two of the items, say at the site where X is. Hence, we would have the constraints $X = Y_c + Z_c$, $Y_c = Y$ and $Z_c = Z$. Only the simple copy constraints are distributed and they can be handled by the strategies of Section 7, for instance. In summary, even if our framework can only be used for simple constraints, we believe it can cover the vast majority of the scenarios of interest for loosely coupled systems.

Our relaxed integrity constraint guarantees provide a formal guarantee in many real-life situations where conventional techniques can provide no guarantee at all, and are thus useful in their own right. However, if an application or end user wishes to use these guarantees directly, special care has to be taken. In general, the weaker the guarantee, the harder it is to use. For example, consider the guarantee in Section 7.3: $(\text{Flag} = \text{true})@t \wedge \neg W(X|Y)@@[t-\eta-\delta, t] \Rightarrow (X = Y)@@[t-\eta-\delta, t]$. If an application program reads the value of Y at a time t when Flag is true, how can the application tell if Y equals the current value of X? The only way is to know that there have been no updates to X in the past $\eta + \delta + \omega$ units of time, but this involves knowing what is happening at a remote site. One can avoid this problem by changing the strategy so we end up with a set of guarantees, one for each site, where the LHS of each only refers to local objects. This would allow applications to infer the RHS of guarantees using only local information. It is always possible to distribute guarantees in this manner, at the cost of complicating the strategy a bit. We have not done so in our examples in order to keep the presentation simple.

[14] See Section 9.3 for further discussion.

A guarantee like the one of Section 7.2, $((\text{Flag} = \text{true}) \wedge (\text{Tb} = s))@t \Rightarrow (X = Y)@@[s, t-\eta-\delta]$, is stronger because variables Flag and Tb tell us if and when the constraint holds. An application still has to read Flag and Tb at CM and Y at a data site before it can infer something about the state of X; however, the state of the CM may be more easily accessible than the state of X. An alternative is to distribute the guarantee by keeping cached copies of Tb and Flag at each site. (Whenever the CM modified Flag or Tb, it would propagate the update to each of the sites.) In this case, the local guarantee at each site would refer to the local copies of Flag and Tb on the LHS, and so an application can infer the RHS by querying data. Again, we have omitted this part in our presentation in the interest of simplicity.

Guarantees such as the one made by the Demarcation Protocol (Section 7.4) are not time dependent, but are weaker in the sense that we only know value ranges (e.g., $X \leq Y$) as opposed to knowing if X is identical to Y. In some applications, this type of "approximate" knowledge is all we need, so this type of guarantee is sufficient. Another type of weak guarantee that could be of use to applications is a probabilistic one, as discussed in Section 6.2.

9 A Toolkit for Constraint Management

In previous sections we have presented a framework and language for specifying interfaces, strategies, and guarantees for constraint management in heterogeneous environments. In this section we describe how we are using this framework to implement a toolkit for constraint management. The goal is to provide a set of easily configurable services that enforce or check constraints spanning heterogeneous systems. We first briefly describe the architecture and then use an extended example to illustrate some of the features of the system.

9.1 Architecture

Figure 2 depicts the architecture of our constraint management toolkit. At the lowest level we have the Raw Information Sources (RIS), which could be relational or object-oriented database systems, file systems, bibliographic information systems, electronic mail systems, network news systems, and so on. Each RIS has its own particular interface, which we call RISI. For example, for a Sybase RIS, the RISI is SQL-based and includes the protocols to send a query to the Sybase server and receive the results. The CM-Shell processes at the top of the figure implement the selected strategy, which is described in the Strategy Specification. Thus, each CM-Shell is a general-purpose process that is configured by reading the Strategy Specification file.

If the CM-Shell were to interface directly with the RIS, it would have to understand the peculiarities of each RISI. For example, to read a data item X stored in a relational database, a CM-Shell would have to issue a request in the particular dialect of SQL that the RIS understands. If X is stored in an OODB or a file system, the procedure to read X will be completely different.

[15] This is a detailed version of Figure 1.

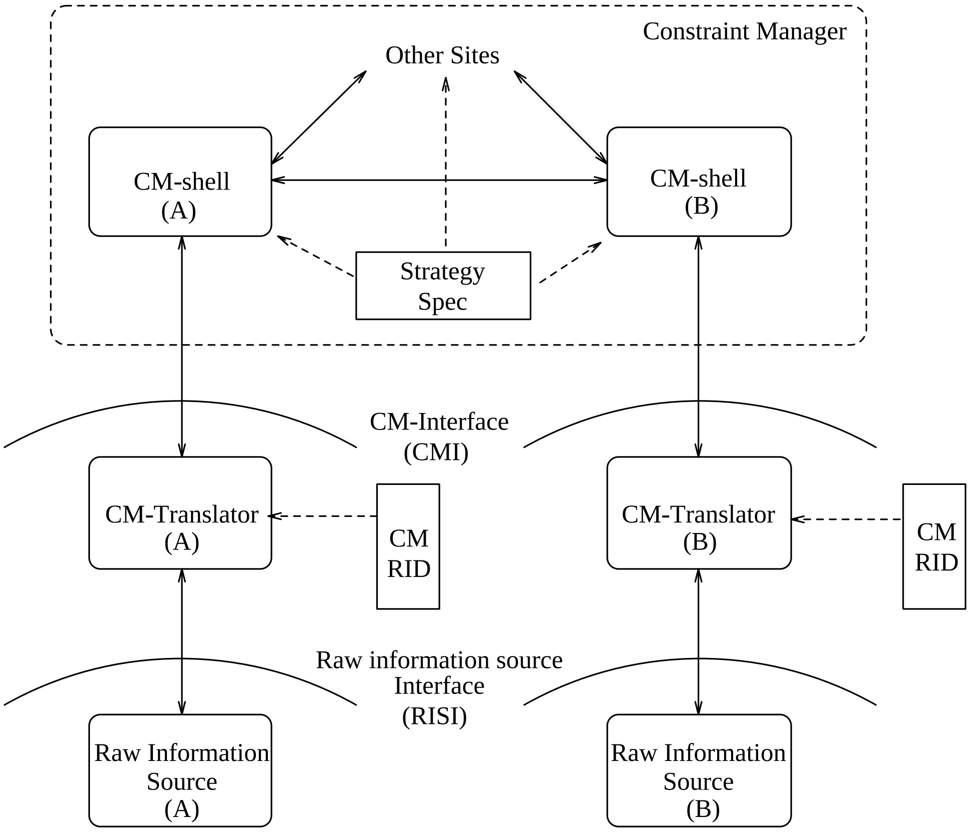

Figure 2: Constraint Management Toolkit Architecture[15]

To manage this complexity, we provide a CM-Translator (for each RIS) that presents to the CM-Shells the local capabilities in a standard fashion. This interface provided by the CM-Translator is the CM-Interface (CMI). The design and implementation of the CM-Translator is helped by the CM-Raw Interface Description (CM-RID) file, which configures standard CM-Translators to the particular underlying data source by presenting the specifics of the RISI in a standard format. For example, a CM-Translator for relational databases can be configured to interface with any DBMS (e.g., Sybase, Oracle) and any database (e.g., a payroll database, an inventory database) just by specifying the appropriate CM-RID.

A final component of our architecture (not shown in Figure 2) is a library of common interfaces and strategies. Thus, the contents of the Strategy Specification and the CM-RID files will usually be selected from available menus of proven strategies and interfaces.

Notice that our toolkit does *not* have a central Constraint Manager. Instead, the collections of CM-Shells cooperate to execute the specified strategy. However, some strategies are centralized in nature, and lead to one of the CM-Shells taking on a central role. For example, in the strategy of Section 7.2, we assumed that variables Tb, Flag, Cx, and Cy were available for rule execution. Thus, for this strategy to work, we would have to specify in the Strategy Specification that all these variables are at the same site. The CM-Shell at that selected site would then perform all the rule processing. The important thing to note is that this centralization is due to the strategy selected, not to the toolkit or our framework. One can develop a similar strategy that works when the constraint management variables are distributed.

9.2 Extended Example

To illustrate how CM-Translators and CM-Shells work, let us consider the Notify-Write example of Section 7.1 in more detail. Suppose that a RIS at a site B provides a write interface for data item Y, defined at a high level by $WR(Y, b) \rightarrow W_g(Y, b);\ B \leq \delta$. In practice, this means that the RIS at site B can be instructed to write object Y. The CM-RID is a configuration file that tailors the CM-Translator to handle such write requests. In addition to the interface statement $WR(Y, b) \rightarrow W_g(Y, b);\ B \leq \delta$, the CM-RID at B specifies the following:

- The object in the underlying RIS to which Y maps. In our example, suppose that the RIS B is located in New York and is a relational database containing payroll information of some company. Further, suppose that Y stands for "Smith's salary."

- The command that has to be issued to the RIS to perform the write. In our example, the CM-RID specifies that to write a value b to Y (which is "Smith's salary"), the SQL query UPDATE EMPLOYEES SET SALARY = b WHERE NAME = "SMITH" must be sent to the SQL server.

- Low-level details of the protocol for querying the SQL server. In our example, the CM-RID indicates that the underlying RID is a Sybase database, and also specifies the network name of the Sybase server, the port number to connect to, the name of the machine on which it is running, etc. Using these details, the CM-Translator can send the SQL query to the RIS and receive the response (acknowledgment).

The CMI exported by the B CM-Translator lets the CM-Shells inquire about the local capabilities and request services. For example, when the B CM-Shell is starting up, it may ask the B CM-Translator what interface it offers for Y. The response will be "$WR(Y, b) \rightarrow W_g(Y, b);\ B \leq \delta$." This tells the CM-Shell it can issue a $WR(Y, b)$ command and what the timing guarantees are. If a WR command is issued, the CM-Translator converts the request into one that the RIS can execute (SQL, in our example).

Next, consider the second RIS in our example of Section 7.1, which offers a notify interface for data item X at the RIS at site A. At a high level, this interface is defined by the rule $W_s(X, b) \rightarrow N(X, b);\ B \leq \eta$. This definition of the notify interface does not specify how it is implemented. For the purpose of this example, let us assume that the notify interface is implemented by setting a database "trigger" on the data item X. The CM-RID specifies what the CM-Translator at A needs to do to set the trigger, and what it should expect to receive from the RIS when X changes.

When the A CM-Shell queries the CM-Translator for the interface offered, the response is "$W_s(X, b) \rightarrow N(X, b);\ B \leq \eta$." The CM-Shell can then request notification, specifying where the notify messages are to be delivered. At this point, the CM-Translator sets up the appropriate trigger in the A RIS, as specified in the CM-RID. When a change notice is received, the CM-Translator forwards it to the A CM-Shell. Note that it is possible to implement the notify interface without using triggers, and that this does not cause a problem in our toolkit, as long as the semantics of the interface are preserved. Such an implementation may be necessary, for example, when the underlying RIS does not support triggers.

Now consider the strategy described in Section 7.1, which intuitively propagates updates from X to Y, and is described by the following strategy statement: $N(X, b) \rightarrow WR(Y, b);\ B \leq \delta$. This strategy specification is available to both the CM-Shells, as indicated in Figure 2. The Strategy Specification also indicates where objects are located, i.e., X is at site A and Y is at B. (In this case, there are no auxiliary objects such as Flag and Tb; if there were, their location would be given too.)

From the location of objects, the CM-Shells can determine who is responsible for each side of each rule. In our example, the CM-Shell at A is responsible for the LHS of the rule because X is at that site. This means that it sets up the local notification. When the A CM-Shell receives a $N(X, b)$ message from its CM-Translator, it forwards it to the B CM-Shell, since it is responsible for the RHS of the rule. The B CM-Shell then forwards the $WR(Y, b)$ command to its local CM-Translator. Based on the expected maximum execution time of each CM-Shell and the maximum transmission time between CM-Shells, one can compute an estimate for δ, the time guarantee in the rule. (See discussion on timing guarantees at the end of Section 4.)

9.3 Parametric constraints

So far we have assumed that constraints and interfaces deal with single objects such as X and Y. However, the toolkit can also handle sets of similar objects. For example, we can handle a constraint such as $\text{SALARY}(E) = \text{PAY}(E)$, where E is the name of any employee. In this case, each CM-RID is parameterized by the employee name. For instance, at the site where SALARY is stored, the CM-RID can specify that to write a value b into $\text{SALARY}(E)$, the following SQL statement needs to be executed: UPDATE EMPLOYEES SET SALARY $= b$ WHERE NAME $= E$. The format of the CM-RID thus includes simple pattern matching and parameter-passing and is general enough to enable a uniform description of diverse Raw Information Source Interfaces. The strategies can be parameterized in a similar fashion.

9.4 Implementation Status

We have implemented translators for Unix files and relational databases. The translators are designed using an object-oriented methodology with the goal of requiring only minor amounts of rewriting when moving to different kinds of raw sources (RIS). Currently, some low-level details for communicating with, say, Sybase (e.g., the names of the Sybase client-library functions used to connect to the server, send queries, parse results), have to be embedded in the CM-Translator code, and so this code has to be rewritten to port the CM-Translator to, say, an Oracle database. However, the amount of code that needs to be rewritten is typically less than a page. Similarly, porting the CM-Translator to a WAIS-like RIS involves incorporating the WAIS protocol for the submission of queries and the retrieval of results. We can avoid having to rewrite the CM-Translator by enhancing the CM-RID format to include a scripting language such as Tcl [Ous90]; there is a tradeoff here between the complexity in the CM-Translator and the complexity in the CM-RID. The design and implementation of translators is in itself a difficult and interesting issue. While we currently are building translators by hand, we hope to soon exploit related work we are doing in the context of a query mediation project [PGMW95].

The CM is implemented as a distributed, timed rule engine. The CM-Shell at each site forms a local component of this distributed rule engine. Whenever a CM-Shell produces an event, or learns of an event produced by its CM-Translator, it notifies all other CM-shells. In our next version of the toolkit, we plan to optimize the way the distributed rule engine works so as to minimize the communication between CM-shells. The current implementation does not perform any failure handling of the type described in Section 4.3, but such a facility can easily be added, and we plan to do that next.

10 Conclusion

Distributed integrity constraints arise naturally when information systems interoperate, due to interdependencies between data. Traditional constraint management techniques assume facilities like atomic transactions, locking, and consistent queries. While these are reasonable assumptions in centralized or tightly coupled distributed environments, they typically do not hold in loosely coupled heterogeneous environments, and traditional constraint management techniques are therefore inapplicable in such cases. Another characteristic of heterogeneous environments is that different databases offer different facilities for constraint management, which makes constraint management more difficult. Currently, constraints in heterogeneous environments are either not monitored at all, or monitored using ad-hoc techniques. Such techniques are error-prone and can lead to irreparable inconsistencies in the databases.

We have presented a framework and a toolkit for constraint management in loosely coupled, heterogeneous environments. Our framework allows the formal specification of the interfaces each database offers, and of the constraint management strategies. Given these, we can prove formal guarantees regarding the consistency of constraints. Our framework can express guarantees that are more "relaxed" than in conventional database systems, in the sense that constraints may hold only at given times or under certain conditions. This added flexibility is essential in real-world distributed scenarios where it is not possible to guarantee that integrity constraints are always satisfied. The toolkit we are building will provide configurable constraint management and translation processes, and a library of proven strategies, making it relatively easy to enforce relaxed constraints.

References

[AM89] Martin Abadi and Zohar Manna. Temporal logic programming. *Journal of Symbolic Computation*, 8(3):277–295, 1989.

[BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[BG92] Gerard Berry and Georges Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BGM92] Daniel Barbara and Hector Garcia-Molina. The demarcation protocol: A technique for maintaining linear arithmetic constraints in distributed database systems. In *Proceedings of the International Conference on Extending Database Technology*, pages 373–388, Vienna, Austria, March 1992.

[BGMS92] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *The VLDB Journal*, 1(2):181, October 1992.

[CGMW93] Sudarshan S. Chawathe, Hector Garcia-Molina, and Jennifer Widom. Constraint management in loosely coupled distributed databases. Technical report, Computer Science Department, Stanford University, 1993. Available through anonymous ftp from host `db.stanford.edu` as `pub/chawathe/1993/cm-loosely-coupled-dbs.ps`.

[CM81] W. Clocksin and C. Mellish. *Programming in PROLOG*. Springer-Verlag, Berlin, 1981.

[CW93] Stefano Ceri and Jennifer Widom. Managing semantic heterogeneity with production rules and persistent queues. In *Proceedings of the International Conference on Very Large Data Bases*, pages 108–119, Dublin, Ireland, August 1993.

[Elm91] Ahmed Elmagarmid, editor. *Special Issue on Unconventional Transaction Management,* Data Engineering Bulletin 14(1), March 1991.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Imposibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[Gre93] Paul Grefen. Combining theory and practice in integrity control: A declarative approach to the specification of a transaction modification subsystem. In *Proceedings of the International Conference on Very Large Data Bases*, pages 581–591, Dublin, Ireland, August 1993.

[GW93] Ashish Gupta and Jennifer Widom. Local verification of global integrity constraints in distributed databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 49–58, Washington, D.C., May 1993.

[IEE87] Inc. IEEE. *IEEE Standard VHDL Language Reference Manual.* 345 East 47th St., New York, NY 10017., March 1987.

[L+94] David C. Luckham et al. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 1994.

[Ous90] J. K. Ousterhout. Tcl: an embeddable command language. In *Proceedings of the Winter 1990 USENIX Conference*, Washington, D.C., January 1990.

[PGMW95] Yannis Papakonstantinou, Hector Garcia-Molina, and Jennifer Widom. Object exchange across heterogeneous information sources. In *International Conference on Data Engineering*, Taipei, Taiwan, March 1995.

[RSK91] Marek Rusinkiewicz, Amit Sheth, and George Karabatis. Specifying interdatabase dependencies in a multidatabase environment. *IEEE Computer*, 24(12):46–51, December 1991.

[Sno86] Richard Snodgrass. Temporal databases. *IEEE Computer*, 19(9):35–42, September 1986.

[SV86] Eric Simon and Patrick Valduriez. Integrity control in distributed database systems. In *Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences*, pages 621–632, 1986.

[TM91] D. E. Thomas and P. R. Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1991.

Template	Site
$W_s(x, \_)$	x's site
$W_g(x, \_)$	x's site
$N(\_, \_)$	CM
$WR(x, \_)$	x's site
$CL(\_, \_)$	CM
$AC(x, \_)$	x's site

Table 1: Site of an event is obtained by matching it against the templates.

Appendix A Sites, Matchings, and Rule Firings

In this section, we first define the *site* of an event and the meaning of an event *matching* an event template. We then use these concepts to formally describe the syntax and semantics of rule firing.

Intuitively, the site of an event is the site of its occurrence. For example, write and write-request events are located at the site of the data item involved, while notify events are located at the site of the CM. Formally, the site of an event is obtained by matching its descriptor with the entries in Table 1.

We now define what it means for an event to *match* an event template. We say an event E matches an event template $\mathcal{E}$ if there is an *interpretation* I of the variables in $\mathcal{E}$ such that substituting using I in $\mathcal{E}$ yields E. Such a matching interpretation I, if it exists, is denoted by $mi(E, \mathcal{E})$. The special *false* event template, $\mathcal{F}$, does not match any event (by definition).

The general form of a rule is

$$\mathcal{E}_0 \wedge C_0 \rightarrow C_1?\mathcal{E}_1, C_2?\mathcal{E}_2, \ldots, C_k?\mathcal{E}_k;\ B \leq \delta$$

where $\mathcal{E}_i$ are event templates, C_0 is a boolean expression involving data items local to the site of $\mathcal{E}_1$ and variables, and C_i are boolean expressions involving data items local to the site of $\mathcal{E}_1$ and variables.[16] The meaning of this rule is as follows: If an event matching the event template on the LHS occurs at a time t at which the condition C_0 is true, then there exist $t_i \in [t, t+\delta]$, $\quad i = 1 \ldots k$ where $t_i < t_{i+1}$, $\quad i = 1 \ldots k-1$ such that at time t_i, the condition C_i is evaluated, and if it evaluates to true, the event matching event template $\mathcal{E}_i$ occurs. The event corresponding to the event template $\mathcal{E}_i$ is obtained by substituting in $\mathcal{E}_i$ using the matching interpretation for the LHS, $mi(E_0, \mathcal{E}_0)$. Note that variables on the LHS are implicitly universally quantified, while variables on the RHS that do not occur on the LHS are implicitly existentially quantified. The bindings of variables from the LHS are passed on to the RHS (through the matching interpretation), so that a variable occurring on both sides takes on the same value.

[16]All the events on the RHS of a rule must have the same site.

Appendix B Valid Executions

A *valid execution* is an execution $(E_1, \ldots, E_n)$ that satisfies the following properties. Note that these properties reflect the semantics of rules described in Section 3.4 and Appendix A.

1. The events in the sequence are sorted in order of nondecreasing time.

   $\forall i, j \in [1, n], \quad E_i.time < E_j.time \Rightarrow i < j$

2. For each event in the execution: If the event descriptor is a (spontaneous or generated) write, then the *new* interpretation maps the corresponding data item to the value written, with all other data items being mapped to the same value in both *old* and *new*. If the event descriptor is not a write, then the *old* and *new* interpretations are identical. Formally,

   $\forall i = 1 \ldots n,$
   if $E_i.desc = W_s(X, \alpha, \beta)$ then $E_i.new = E_i.old - \{X = \alpha\} \bigcup \{X = \beta\}$
   else if $E_i.desc = W_g(X, \beta)$ then $E_i.new = E_i.old - \{X = \_\} \bigcup \{X = \beta\}$
   else $E_i.new = E_i.old$.

3. The *old* interpretation in each event is identical to the *new* interpretation in the immediately preceding event. That is, the only changes to the interpretations are those caused by events. Note that interpretations model only data items related to constraints, hence this restriction applies to only such constraint data; other data items may change their values.

   $E_i.old = E_{i-1}.new \quad i = 2 \ldots n$

4. For all $i = 1 \ldots n$, if E_i is a spontaneous event then both $E_i.rule$ and $E_i.trigger$ are null.

5. For all $i = 1 \ldots n$, if E_i is a generated event, then (informally) its rule component specifies the rule whose firing caused E_i to occur, and its trigger component specifies the event whose occurrence caused the rule to fire. Further, the LHS and RHS conditions of the rule must be satisfied by the appropriate interpretations.

   Formally, if E_i is a generated event then both $E_i.rule$ and $E_i.trigger$ are non-null. Further, the following properties are true:

   (a) $E_i.trigger$ is an event that matches the LHS event template of $E_i.rule$. Let the matching interpretation be I;

   (b) I can be extended[17] to an interpretation I' such that substituting using I' in a RHS event template $\mathcal{E}_j$ of $E_i.rule$ gives E_i;

   (c) The LHS condition of $E_i.rule$ is satisfied by $E_i.trigger.new$;

   (d) The RHS condition C_j (corresponding to $\mathcal{E}_j$) of $E_i.rule$ is satisfied by $E_i.old$.

[17] We say an interpretation I is *extended* to an interpretation I' if the set of non-null mappings in I is a subset of the set of non-null mappings in I'.

6. Informally, the converse of the previous property. That is, if an event matching the LHS event template of some rule occurs and if the LHS and (some) RHS conditions of that rule are satisfied at the appropriate times, then events matching the corresponding RHS event templates occur within the time specified by the rule.

   Formally, if E_i matches the LHS of a rule r: $\mathcal{E}_0 \wedge C_0 \rightarrow C_1?\mathcal{E}_1 \ldots C_k?\mathcal{E}_k;\ B \leq \delta$, and $E_i.new$ satisfies C_0, then there exist $t_j \in [E_i.time, E_i.time + \delta], j = 1 \ldots k$ where $t_j < t_{j+1}, j = 1 \ldots k-1$ and, for all $j = 1 \ldots k$, exactly one of the following holds true:

   - C_j is false at t_j;
   - there exists an event E_j, such that $E_j.time = t_j$, $E_j.old$ satisfies C_j, and substituting using matching interpretation $mi(E_i, \mathcal{E}_0)$ in $\mathcal{E}_j$ gives $E_j.desc$. Further, $E_j.rule = r$ and $E_j.trigger = E_i$.

7. This property formalizes our assumptions of in-order message delivery between sites and in-order processing at each site. To do this, we first introduce some additional notation.

   If E_i and E_j are events in an execution such that $E_j.trigger = E_i$ and $E_j.rule = R$ we write $E_i \leadsto^R E_j$.

   We say rules R_1: $\mathcal{E}_1^1 \wedge C_1 \rightarrow C_2?\mathcal{E}_2^1$ and R_2: $\mathcal{E}_1^2 \wedge C_1 \rightarrow C_2?\mathcal{E}_2^2$ are *related* if $site(\mathcal{E}_1^1) = site(\mathcal{E}_1^2)$ and $site(\mathcal{E}_2^1) = site(\mathcal{E}_2^2)$.

   The formal statement of this property is that if $E_1@t_1 \leadsto^{R_1} E_2@t_2$ and $E_3@t_3 \leadsto^{R_2} E_4@t_4$, where R_1 are R_2 are related rules, then $t_1 < t_3$ iff $t_2 < t_4$.

Appendix C Proofs of Guarantees

In this section, we use our formalism to prove the guarantees made in the examples of Section 7. Due to space limitations, we present only the outlines of the proofs here; the complete details are in [CGMW93].

Appendix C.1 Notify–Write

Interfaces:

$W_s(X, b) \rightarrow N(X, b);\ B \leq \eta$ (notify interface) .... (1)

$WR(Y, b) \rightarrow W(Y, b);\ B \leq \omega$ (write interface) .... (2)

$W_s(Y, \_) \rightarrow \mathcal{F}$ (no spontaneous writes) .... (3)

Strategy:

$N(X, b) \rightarrow WR(Y, b);\ B \leq \delta$ (update propagation) .... (4)

Guarantee:

$\neg W(X)@@[t - \eta - \delta - \omega, t] \Rightarrow (X = Y)@t$

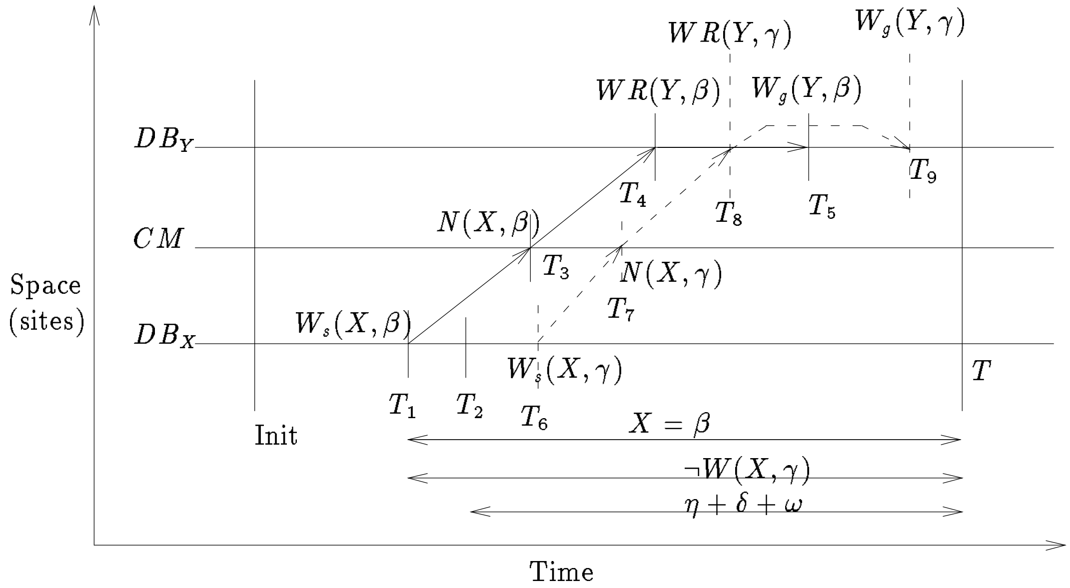

Figure 3: Timing diagram for proof in 7.1.

Proof outline: Refer to Figure 3. Let T be the current time. Consider the most recent write, $W_s(X,\beta)$[18] to X (at time T_1 in Figure 3), so $X = \beta$ at T. $W(X,\beta)$ triggers a notify event, $N(X,\beta)$, due to rule (1) (at time T_3 in Figure 3); this in turn triggers a write request event, $WR(Y,\beta)$, due to (4) (at time T_4 in Figure 3); this in turn triggers a write event, $W_g(Y,\beta)$, due to (2) (at time T_5 in Figure 3). The LHS of the guarantee tells us that $W_s(X,\beta)$ occurred at a time sufficiently long ago that $W_g(Y,\beta)$ occurs before T.

However, before we conclude that the write to Y implies $Y = \beta$ at T, we need to show that there are no other writes to Y between T_5 and T. By (3) we know there are no spontaneous writes, so we must show there are no generated writes. This is guaranteed by our "in-order firing" assumption, described in Appendix B. This assumption tells us that if $W_g(Y,\beta)$ occurs before (a hypothetical) $W_g(Y,\gamma)$ (time T_9 in Figure 3), then the events that triggered the writes must occur in that order, i.e, $W_s(X,\beta)$ must occur before $W_s(X,\gamma)$ (time T_6 in Figure 3). But this is a contradiction to the choice of the write operation $W_s(X,\beta)$ as the last write to X before T. We therefore conclude that there can be no write to Y in the period between the write $W_g(Y,\beta)$ and T, and hence, $Y = \beta$ at T. Since we showed earlier that $X = \beta$ at T, we conclude that $X = Y$ at T, as required. A complete formal proof is presented in [CGMW93].

Appendix C.2 Notify–Notify version 1

Interfaces: (notify interfaces)

[18]This write is a spontaneous write because there is no rule that triggers a generated write to X.

$W_s(X,b) \rightarrow N(X,b);\ B \leq \eta$ .... (1)
$W_s(Y,b) \rightarrow N(Y,b);\ B \leq \eta$ .... (2)

Strategy:

$N(X,b) \rightarrow W_g(\mathrm{Cx},b);\ B \leq \delta$ .... (3)
$N(X,b)@t \rightarrow (b \neq \mathrm{Cy})?W_g(\mathrm{Flag},\mathrm{false}), (b = \mathrm{Cy})?W_g(\mathrm{Flag},\mathrm{true}), W_g(\mathrm{Tb},T_{now});\ B \leq \delta$ ...... (4)
$N(Y,b) \rightarrow W_g(\mathrm{Cy},b);\ B \leq \delta$ .... (5)
$N(Y,b) \rightarrow (b \neq \mathrm{Cx})?W_g(\mathrm{Flag},\mathrm{false}), (b = \mathrm{Cx})?W_g(\mathrm{Flag},\mathrm{true}), W_g(\mathrm{Tb},T_{now});\ B \leq \delta$ .... (6)
$W_s(\mathrm{Tb},\_) \rightarrow \mathcal{F}$ .... (7)
$W_s(\mathrm{Cx},\_) \rightarrow \mathcal{F}$ .... (8)
$W_s(\mathrm{Cy},\_) \rightarrow \mathcal{F}$ .... (9)
$W_s(\mathrm{Flag},\_) \rightarrow \mathcal{F}$ .... (10)

Guarantee:

$((\mathrm{Flag} = \mathrm{true}) \wedge (\mathrm{Tb} = s))@t \Rightarrow (X = Y)@@[s, t - \eta - \delta]$

Proof outline: If Flag is true at time T_0, we can identify the write operation $W_g(\mathrm{Flag},\mathrm{true})$ that writes true to Flag, which is triggered by a notify event. Assume that it is $N(Y,\beta)@T_2$. This notify is, in turn, triggered by a write to Y, $W_g(Y,\beta)@T_3$. We can show that there can be no write to Y between T_3 and $T_0 - \eta - \delta$, so that $Y = \beta$ during $[T_3, T_0 - \eta - \delta]$.

Now consider the last write operation before $T_0 - \eta - \delta$ on X, $W_g(X,\gamma)@T_4$, so that $X = \gamma$ during $[T_4, T_0 - \eta - \delta]$. This write triggers a notify $N(X,\gamma)@T_5$, which in turn triggers a $W_g(\mathrm{Cx},\gamma)@T_6$. We can show that this is the last write to Cx before T_1, so that this is the value used to check the RHS condition of rule (6) at T_1. Since Flag is set true at T_1, we know that $(\mathrm{Cx} = \beta)$ at T_1, so we conclude $\beta = \gamma$. We have thus shown that $(X = Y)@@[max\{T_3, T_4\}, T_0 - \eta - \delta]$.

To complete the proof, we must show that the value of Tb at T_0 is greater than $max\{T_3, T_4\}$. The notify $N(Y,b)@T_2$ triggers a write $W_g(\mathrm{Tb},T_{now})@T_7$ to Tb according to rule (4). We can show that there can be no write to Tb between T_7 and T_0, so that the value of Tb at T_0 is T_7 (using the fact that $T_{now} = T_7$ at T_7, by definition). By performing arithmetic on the time intervals involved in all the above rule firings, we can show that T_7 (and hence Tb at T_0) is greater than $max\{T_3, T_4\}$, as required. A complete formal proof is presented in [CGMW93].

Appendix C.3 Notify–Notify version 2

Interfaces: (notify interfaces)

$W_s(X,b) \rightarrow N(X,b);\ B \leq \eta$ .... (1)
$W_s(Y,b) \rightarrow N(Y,b);\ B \leq \eta$ .... (2)

Strategy: (same as the preceding proof, except that the 'Tb' data item is not maintained)

$N(X,b) \rightarrow \quad W_g(\mathrm{Cx},b);\ B \leq \delta$ .... (3)
$N(X,b) \rightarrow \quad (b \neq \mathrm{Cy})?W_g(\mathrm{Flag},\mathrm{false});\ B \leq \delta$ .... (4)
$N(X,b) \rightarrow \quad (b = \mathrm{Cy})?W_g(\mathrm{Flag},\mathrm{true});\ B \leq \delta$ .... (5)
$N(Y,b) \rightarrow \quad W_g(\mathrm{Cy},b)];\ B \leq \delta$ .... (6)

$N(Y,b) \rightarrow \quad (b \neq \mathrm{Cx})?W_g(\mathrm{Flag}, \mathrm{false});\ B \leq \delta$ .... (7)
$N(Y,b) \rightarrow \quad (b = \mathrm{Cx})?W_g(\mathrm{Flag}, \mathrm{true});\ B \leq \delta$ .... (8)
$W_s(\mathrm{Cx}, \_) \rightarrow \mathcal{F}$ .... (9)
$W_s(\mathrm{Cy}, \_) \rightarrow \mathcal{F}$ .... (10)
$W_s(\mathrm{Flag}, \_) \rightarrow \mathcal{F}$ .... (11)

Guarantee:

$(\mathrm{Flag} = \mathrm{true})@t \wedge \neg W(X|Y)@@[t-\eta-\delta, t] \Rightarrow (X = Y)@@[t-\eta-\delta, t]$

Proof outline: Assume Flag is true at some time T. The last write operation on Flag must be $W_g(\mathrm{Flag}, \mathrm{true})$. Using a backward rule-firing argument similar to that in the previous section, we conclude that this write operation must be caused by the "most recent" notification $N(X,x)$ or $N(Y,y)$ (meaning that there are no notifications for either X or Y after this notification). Without loss of generality, we assume that the last notification received is $N(Y,y)$. Using the rule-firing argument in the backward direction once again, we conclude that this notification is caused by a corresponding "most recent" write operation $W_g(Y,y)$ at some time T_3, and therefore, $Y = y$ during the interval $[T_3, T]$.

Note that a notification $N(Y,y)$ at time T_4 will write true to Flag if and only if $y =$Cx at T_4. Since we know Flag was set true, we conclude that $(\mathrm{Cx} = y)@T_4$. We then identify the most recent write event $W_g(\mathrm{Cx}, y)$.[19] Reasoning backwards again, we identify the write event $W_g(X,y)$ at some time T_6 that triggered the write to Cx (through a notify event). Using the "in-order firing" property of valid executions, we show that there can be no write to X after this write and before T, and therefore, $X = y$ during the time interval $[T_6, T]$. Since we showed earlier that $Y = y$ during $[T_3, T]$, we conclude that $(X = Y)@@[max\{T_3, T_6\}, T]$. Finally, we show that $max\{T_3, T_6\} < T - \eta - \delta$, from which the RHS of the guarantee follows. A complete formal proof is presented in [CGMW93].
□

Appendix C.4 Demarcation Protocol

$W_s(X,b) \wedge (b > X_l) \rightarrow \mathcal{F} \quad$ (local constraint $X \leq X_l$) .... (1)
$W_s(X_l, b) \rightarrow \mathcal{F} \quad$ (no spontaneous writes) .... (2)
$WR_i(X_l, a) \rightarrow W_g(X_l, X_l + a);\ B \leq \omega \quad$ (incremental write) .... (3)
$AC(X,a) \rightarrow W_g(X_l, X_l + a) \quad$ (accept change) .... (4)
$W_s(Y,b) \wedge (b < Y_l) \rightarrow \mathcal{F} \quad$ (local constraint $Y \geq Y_l$) .... (5)
$W_s(Y_l, b) \rightarrow \mathcal{F} \quad$ (no spontaneous writes) .... (6)
$WR_i(Y_l, a) \rightarrow W_g(Y_l, Y_l + a);\ B \leq \omega \quad$ (incremental write) .... (7)
$AC(Y,a) \rightarrow W_g(Y_l, Y_l + a) \quad$ (accept change) .... (8)

Strategy: We describe a strategy that implements the Demarcation *Protocol*, as described in [BGM92]. (For simplicity, we do not describe the associated *policies*.) The event descriptor

[19] To guarantee that such an event exists, we make an assumption about the "initial state" of the system. Details are in [CGMW93].

$CL(X,a)$ denotes an application or user requesting a *change-limit* operation, that is, a request to change X_l to $X_l + a$. CL is a spontaneous event. A write to X_l or Y_l is called *safe* if it is guaranteed not to violate $X_l \leq Y_l$. Thus a decrement to X_l and an increment to Y_l are safe, while an increment to X_l and a decrement to Y_l are unsafe.

On receiving a change-limit request, the CM first checks if the requested increment or decrement to the limit data item is safe. If so, the requested change is performed using an incremental write request operation. After this safe write operation is known to have completed, an unsafe write operation that increments or decrements the other limit data item by the same amount is requested at the other site using the accept-change interface. Note that this request need not be made within any fixed time interval for the protocol to work. If the requested change-limit operation is unsafe, the CM first performs the corresponding safe operation at the other site. When that operation completes, the requested unsafe operation will be performed using an accept-change operation. Formally, we specify this strategy as follows:

$CLR(X,a) \rightarrow CL(X,a)$ ....(9)
$CL(X,a) \wedge (a > 0) \rightarrow CL(Y,a);\ B \leq \delta$ ....(10)
$CL(X,a) \wedge (a \leq 0) \rightarrow WR_i(X_l,a), AC(Y,a);\ B \leq \delta$ ....(11) ....(12)
$CLR(Y,a) \rightarrow CL(Y,a)$ ....(13)
$CL(Y,a) \wedge (a < 0) \rightarrow CL(X,a);\ B \leq \delta$ ....(14)
$CL(Y,a) \wedge (a \geq 0) \rightarrow WR_i(Y_l,a), AC(X,a);\ B \leq \delta$ ....(15) ....(16)

Guarantee: The Demarcation Protocol strategy ensures that if $X_l \leq Y_l$ at some time T_0 then $X_l \leq Y_l$ at all times after T_0. This corresponds to Theorem 4.1 in [BGM92]. We state this formally as follows:

$(X_l \leq Y_l)@T_0 \Rightarrow (X_l \leq Y_l)@t$

Proof outline: Consider an unsafe write to Y_l at time T_1. This write must be of the form $W_g(Y_l, Y_l + a)@T_1$, for some increment a where $a < 0$. Such a write could be the result of either rule (7) or rule (8) firing, triggered by events $WR_i(Y_l,a)$ or $AC(Y,a)$, respectively. $WR_i(Y_l,a)$ can occur only if rule (15) fires, but that requires $a \geq 0$, which contradicts $a < 0$. Therefore, the unsafe write $W_g(Y_l, Y_l + a)$ is caused by the firing of rule (8), which requires the $AC(Y,a)$ operation on the LHS of that rule to occur. Since $AC(Y,a)$ is generated by rule (12) only, we conclude that (12) must fire, which implies that the LHS event $CL(X,a)$ must have occurred earlier. Noting that $a < 0$, and using rule (11), we conclude that a $WR_i(X_l,a)$ event must occur, which in turn implies the occurrence of a write $W_g(X_l, X_l + a)$. Note that this write to X_l is a safe write. Simple arithmetic on the time intervals that restrict when each of the events described above occurs shows that the above safe write to X_l must occur before the original unsafe write to Y_l. We have thus shown that every unsafe write to Y_l is preceded by a safe write to X_l (with the same increment or decrement). Similarly, we can show that every unsafe write to X_l is preceded by a corresponding safe write to Y_l.

The value of X_l at any time t after T_0 is given by: $X_l@t = X_l@T_0 + \Sigma_{i=1}^{k_1} u_i^X + \Sigma_{i=1}^{k_2} v_i^X$, where $u_i^X > 0$ and $v_i^X \leq 0$. (The u_i^X represent the unsafe increments to X_l while the v_i^X represent the safe decrements.) Similarly, we have $Y_l@t = Y_l@T_0 + \Sigma_{i=1}^{k_3} u_i^Y + \Sigma_{i=1}^{k_4} v_i^Y$, where $u_i^Y < 0$ and $v_i^Y \geq 0$. (The u_i^Y represent the unsafe decrements to Y_l while the v_i^Y represent the safe increments.) Since an unsafe write to one limit data item is preceded by a corresponding safe write to the other limit data item, we conclude that $|\Sigma_{i=1}^{k_1} u_i^X| \leq |\Sigma_{i=1}^{k_4} v_i^Y|$ and $|\Sigma_{i=1}^{k_3} u_i^Y| \leq |\Sigma_{i=1}^{k_2} v_i^X|$. Therefore, given that $(X_l \leq Y_l)@T_0$, we conclude that $(X_l \leq Y_l)@t$.

We do not include a formal proof of the Demarcation Protocol since the proof is given in [BGM92].