A Toolkit for Constraint Management in Heterogeneous Information Systems*

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 present a framework and a toolkit to monitor and enforce distributed integrity constraints in loosely coupled heterogeneous information systems. Our framework enables and formalizes weakened notions of consistency, which are essential in such environments. Our framework is used to describe (1) interfaces provided by a database for the data items involved in inter-site constraints; (2) strategies for monitoring and enforcing such constraints; (3) guarantees regarding the level of consistency the system can provide. Our toolkit uses this framework to provide a set of configurable modules that are used to monitor and enforce constraints spanning loosely coupled heterogeneous information systems.

1 Introduction

We address the management of distributed integrity constraints over data that is stored in a collection of loosely coupled heterogeneous information systems. Distributed integrity constraints arise naturally whenever data that is semantically related is stored in different systems. For example, a construction company keeps data about a building under construction in its private database. 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 database.

Throughout this paper, we use the term "database" to mean any information system. In addition to traditional database systems, we include bibliographic information systems, "whois" servers, legacy systems, file systems, etc. We use the term "loosely coupled" to refer to information systems that do not offer standard control facilities such as those found in traditional distributed databases. In particular, such databases do not support multi-database transactions or multi-database query and update mechanisms that guarantee data consistency. Often, one or more of the component databases does not even support (local) transactions. Another characteristic of such

*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 Anderson Faculty Scholar Fund, the Center for Integrated Systems at Stanford University, and by equipment grants from Digital Equipment Corporation and IBM Corporation.

environments is that different databases support different modes of access to the database. For example, one database might provide only read access to its data, while another might provide both read and write access. Yet another may provide notification of updates.

This heterogeneity in the method of database access and control is one of the prime reasons why traditional integrity constraint management techniques cannot be applied to loosely coupled heterogeneous environments. In particular, traditional approaches to constraint management assume various facilities such as distributed transactions, remote locking, and prepare-to-commit interfaces, which are usually not supported by the databases involved in a loosely coupled system. Further, most previous work assumes that all databases in the system offer a homogeneous access and control method, as discussed in Section 2.

In spite of the difficulties outlined above, integrity constraint management is very important in many loosely coupled scenarios. Currently, systems involving data stored in several loosely coupled databases have no systematic method for monitoring or enforcing integrity constraints over the data. In most such systems, integrity constraints are simply not monitored, are monitored manually, or are monitored in an ad-hoc manner. Monitoring integrity constraints manually or using ad-hoc techniques is tedious and error-prone, while neglecting integrity constraint management altogether often leads to irreparable inconsistencies and costly correction measures.

We argue that in the loosely coupled environment that we study, it is not, in general, possible to make the kind of strict consistency guarantees that traditional constraint management systems make. For example, it is usually not possible to ensure that every application sees strictly consistent data any time it executes. Given this situation, our work focuses on how weakened notions of consistency may be defined, implemented, and used. We propose a uniform, rule-based framework in which we can formally define *guarantees* of weak consistency. Our framework is also used to define the *interfaces* (modes of access) provided by each database to the constraint manager. Further, we use our rule-based framework to specify constraint management *strategies*. We consider two kinds of constraint management. Constraint *enforcement* involves doing the work necessary to make (usually a weakened form of) the constraint valid. In some situations, however, the best we can do is constraint *monitoring*. This involves indicating when the constraint is valid and when it is not. We have also developed a set of proof rules that enable us to derive the validity of guarantees based on interface and strategy specifications [CGMW94]. However, due to space constraints, we do not discuss that work in this paper.

Using our framework, it is possible to capture a wide variety of constraint management techniques over a wide variety of loosely coupled heterogeneous environments, and to provide formal guarantees for weakened constraints. We also show how our interfaces and strategies can be used in practice. We describe a toolkit of general-purpose constraint management and translation services that can easily be configured to a given heterogeneous environment (for e.g., relational databases, object-oriented databases, file systems, bibliographic information systems etc.) 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) that conform to the interfaces. Based on the interfaces and the strategy, the toolkit offers guarantees of weak consistency to applications. At run-time, the toolkit will monitor or enforce the constraints so that the guarantees are valid.

This paper is organized as follows. In Section 2, we discuss how our work relates to traditional database constraint management literature as well as other related areas. We present a short overview of the framework underlying our approach to constraint management in Section 3. We expand on the framework by discussing interfaces, strategies and guarantees in Sections 3.1, 3.2, and 3.3, respectively. In Section 4, we present our constraint management toolkit and illustrate its use with an extended example. We discuss how failures are handled in Section 5. Some additional scenarios that illustrate the use of our framework and toolkit are presented in Section 6, while Section 7 discusses the use of guarantees and some issues in implementing distributed strategies. We summarize our conclusions in Section 8. The appendix presents the formal definition of the rule language used in our framework, which is described informally in Section 3.

2 Related Work

Most previous work in database constraint management addresses centralized or tightly coupled distributed environments. The techniques presented in such work are not applicable in the loosely coupled, heterogeneous environment we study because they assume many facilities such as distributed transactions, remote locking, prepare-to-commit interfaces, etc. For example, 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 multi-database 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 and [GW94] presents a set of protocols for inter-database 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. (In fact, we can express the Demarcation Protocol in our framework and prove the associated guarantee. This is discussed in Section 6.)

Another approach to constraint management in multi-database environments is to extend the transaction concept to multi-databases 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 hold only at particular times.

Finally, the formal aspects of our framework are related to work in Metric Temporal Logic (MTL) [Koy92]. Our formalism can be considered an extension of MTL in which events are modeled explicitly, and distributed rules are used as the primary constructs for specification. A formalism of events and rules is more convenient than a purely state-based formalism for studying many systems like the ones we model; a similar observation is made in [L^+93]. While our formal framework shares its interest in specification with software modeling languages such as LOTOS [BB87] and Esterel [BG92], our formalism is much simpler than those languages since it is targeted at modeling constraint management systems.

3 Framework

In this section, we describe our logical constraint management architecture and define the three main components of our framework, namely *interfaces*, *strategies*, and *guarantees*. (The toolkit is covered in Section 4.) In the interest of saving space and keeping the discussion intuitive, we introduce the concepts in our framework by example. In the appendix, we present the formal definition of the rule language we use, and its execution semantics. appendix. As mentioned in Section 2, the theoretical framework is similar to Metric Temporal Logic [Koy92], with some additions that make it easy and natural to specify constraint management in loosely coupled systems.

The key components of the logical architecture are illustrated in Figure 1. Our distributed constraint manager (CM) consists of a collection of Constraint Manager Shells (CM-Shells). The CM-Shell interacts with the local database and cooperates with other CM-Shells to monitor or enforce the inter-site constraints. If it is not possible to have a CM-Shell at the site of some database, its duties can be performed by one or more of the other CM-Shells, as for Site 3 in the figure. In the sequel, we use the term CM to refer to one or more of the CM-Shells acting on the behalf of the constraint manager. The three major components of our framework are described below:

- **Interfaces**. For each data item involved in a constraint, the interface for that data item describes, 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. Note that we do not fix a specific granularity

[1] We consider seconds as our time unit in this paper, but our approach applies equally well with other time units. Note also that the use of time does not, in general, require synchronized clocks; this issue is discussed further in Section 7.

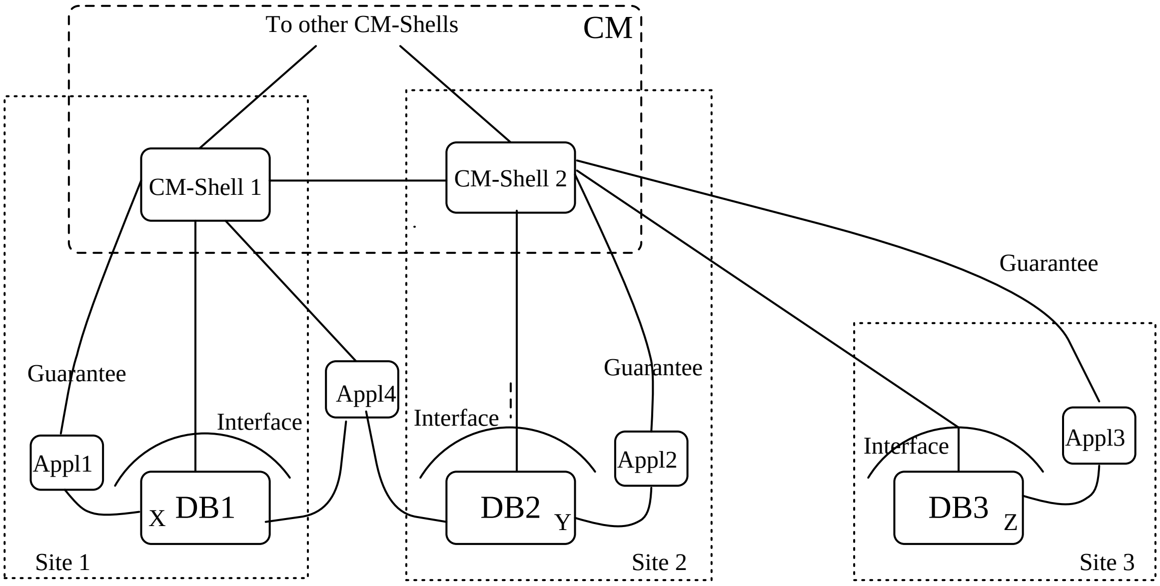

Figure 1: Constraint Management Architecture

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. Our framework also lets us define a single interface for a set of related data items (e.g., the set of salaries of all employees in the Sales department).

- **Strategies**. For a given constraint, a strategy is the specification of an algorithm used by the CM for monitoring or enforcing the constraint. Strategies incorporate the operations available for the data items involved in the constraint. For example (informally), a naive strategy for maintaining the copy 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 level of consistency guaranteed for that constraint. For example (informally), given a *copy constraint* $X = Y$, where X is the primary copy, a guarantee may state that Y takes every value that X takes; that is, that no values of X are "lost."

Figure 1 depicts the relationship between the Constraint Manager, the databases, and the applications (or users). Each database offers interfaces to the CM for its data items. Applications inform the CM of each constraint that needs to be maintained. The CM provides guarantees to the applications, based on the interfaces and the strategy it decides to follow in order to maintain the constraint. Our approach applies to both single-site and multi-site applications. In the case of multi-site applications (for e.g., application 3 in the figure), the application chooses the CM-Shells at one of its sites to be its "local" CM-Shell. This choice is arbitrary and does not affect the validity of our approach.

Our formal framework includes detailed semantics and proof rules that allow us to prove guarantees from interface and strategy specifications. While arbitrary interfaces, strategies and guarantees may be expressed using our framework, in practice we expect most often to use interfaces and strategies from menus provided by the toolkit, with previously proven guarantees. We also plan to extend the toolkit so that it can help the system designer derive new guarantees for different interfaces and strategies.

3.1 Specifying Interfaces

The interface for a data item involved in a constraint describes how that data item may be read, written, or monitored by the constraint manager. Interfaces are specified using a rule-based notation. Note that, as in any specification system, it is important for interface specifications to correctly reflect the actual behavior provided by the database containing that item. As stated above, the database administrators at each site can choose the appropriate interfaces from a menu or they may write their own custom interfaces.

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

$$\mathcal{E}_1 \wedge C \rightarrow_\delta \mathcal{E}_2$$

The meaning of this statement is the following: If an *event* E_1, of the form indicated by *event template* $\mathcal{E}_1$, occurs at time t, and condition C (involving the event and 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 the interval $[t, t+\delta]$. The condition C is evaluated at the time the left-hand side event occurs, and it may be omitted when not needed.

3.1.1 Examples

In the heterogeneous systems we model, interfaces for data items may vary within and across database systems. These interfaces can be quite varied and complex. We believe that our language is useful to describe many interfaces that occur in practice. Below, we present some examples of interfaces. We use the term CM to denote the CM-Shell responsible for the database offering the interface (usually the local CM-Shell).

Write Interface: When a database offers a write interface for a data item X, it promises to perform write operations to X requested by the CM within some time bound. We use the event template $WR(X, b)$ to represent the database receiving a request for the write operation $X \leftarrow b$ from the CM. Similarly, we use the event template $W(X, b)$ to represent the database performing the operation $X \leftarrow b$. Let δ be the time bound within which the database promises to perform the requested write operation. This write interface is expressed as follows:

$$WR(X, b) \rightarrow_\delta W(X, b)$$

Note that *parameters* such as b in the above rule are to be distinguished from local data items. Parameters are simply artifacts of the rule language, whereas local data items refer to actual data in the local database. We represent parameters by lower-case letters and local data items by upper-case letters.

No Spontaneous Write Interface: When a database offers this interface for a data item X, it promises not to update X spontaneously. An event is called *spontaneous* if it can occur at any time, independent of constraint management. Spontaneous events model actions performed by local applications that may be unaware of the CM, and that operate on the local database independently. We use the event template $W_s(X, b)$ to represent an application performing the spontaneous write operation $X \leftarrow b$. The "no spontaneous writes" interface guarantees that there will be no $W_s(X, b)$ events. We express this in our interface specification language by using a special event $\mathcal{F}$ (for *false*), which, by definition, can never occur. Using $\mathcal{F}$, we write the following specification for this interface:

$$W_s(X, b) \rightarrow \mathcal{F}$$

Note that this interface does not mean that X can never be updated, only that it cannot be updated without involving the CM. (Data items may have more than one interface.)

Notify Interface: When a database offers a notify interface for a data item X, it promises to notify the CM within some time bound every time X is updated spontaneously. By using $N(X, b)$ to represent the CM receiving a notification of the update operation $X \leftarrow b$, and using δ to represent the time bound on notification, we express this interface as follows:

$$W_s(X, b) \rightarrow_\delta N(X, b)$$

Conditional Notify Interface: This is a refinement of the Notify Interface. In this interface, the database notifies the CM only when, in addition to X being updated spontaneously, some condition is satisfied. In addition to reducing communication costs, such an interface is useful when the local database can evaluate conditions that cannot be evaluated from the outside. A simple example is an interface that sends a notification to the CM only when the update changes the value of X by more than 10%. To express this interface, we use a spontaneous write event of the form $W_s(X, a, b)$, which represents X being updated from a to b. We then write the following:

$$W_s(X, a, b) \wedge (|b - a| > a * 0.1) \rightarrow_\delta N(X, b)$$

Periodic Notify Interface: Another kind of notify interface is one in which the current value of the data item is sent to the CM periodically. To describe such periodic interfaces, we use the notion of periodic events of the form $P(p)$, which occur every p seconds by definition. A 300-second periodic notify interface is expressed as follows:

$$P(300) \wedge (X = b) \rightarrow_\delta N(X, b)$$

This interface states that every time a $P(300)$ event occurs (every 300 seconds), a notification with the current value of X is sent to the CM within δ seconds.

Read Interface: When a database offers a read interface for a data item X, the CM can send it a read-request and the database will respond with the current value of X within some time bound δ. We use the event $RR(X)$ to represent the database receiving a read request from the CM, and the event $R(X, b)$ to represent the CM receiving the response from the database. We express this interface as follows:

$$RR(X) \wedge (X = b) \rightarrow_\delta R(X, b)$$

Parameterized Interfaces: In each of the above interfaces, the data item name X may be parameterized, yielding an interface for a set of related data items. For example, let $phone(n)$ denote "the phone number of n," where n is the name of an employee. Then, to specify that the CM is notified every time the phone number of any employee n is updated (spontaneously), we use a Parameterized Notify Interface, written as:

$$W_s(phone(n), b) \rightarrow_\delta N(phone(n), b)$$

3.2 Specifying Strategies

The strategy for a constraint describes the algorithm used by the constraint manager to monitor or enforce the constraint. Strategies are specified using a rule-based notation similar to that used for interfaces.

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

$$\mathcal{E}_1 \rightarrow_\delta C?\mathcal{E}_2$$

where $\mathcal{E}_1$ and $\mathcal{E}_2$ are event templates and C is a condition involving the events and data items local to the site of the event matching template $\mathcal{E}_2$. (Each event has a unique site.) This statement states that if an event matching template $\mathcal{E}_1$ occurs at time t, then there exists a time t' in the time interval $[t, t + \delta]$ such that if C is true at t' then an event matching template $\mathcal{E}_2$ occurs at time t'. The condition may be omitted when it is not needed. The events represented by templates $\mathcal{E}_1$ and $\mathcal{E}_2$ can be at different sites, however, the condition C can refer to data at the site of the right-hand side event only.

3.2.1 Example

Consider the copy constraint $X = Y$, where X and Y are at different sites. Let X have a Notify Interface (recall Section 3.1.1) and let Y have a Write Interface. A simple strategy in this case is the propagation of updates from X to Y by making a write request at Y whenever a notification is received from X. Assuming the write request can follow the notification within 5 seconds, we write:

$$N(X, v) \rightarrow_5 WR(Y, v)$$

Just like interfaces, strategies can be parameterized. For example, let $phone1(n)$ denote "the phone number of n," and let $phone2(n)$ denote the same phone number stored in another database.[2] We can specify that a write request is sent to $phone2(n)$ within 5 seconds every time a notification is received from $phone1(n)$ using the following rule:

$$N(phone1(n), v) \rightarrow_5 WR(phone2(n), v)$$

In general, our framework is capable of expressing more complex strategies than these examples. Each CM-Shell can have private data, stored in the CM-Shell itself, for use in strategies. This data may be read and written in the right-hand side of strategy rules. For example, the CM can use a local data item Cx to cache the value of X (obtained, for example, through a Periodic Notify Interface) using the following rule:

$$N(X, b) \rightarrow_5 W(Cx, b)$$

Note that Cx is a local data item maintained by the CM, while b is a parameter used only to express the rule. To forward a write request to a remote data item Y only when the new value of X differs from the cached value, we can write the rule:

$$N(X, b) \rightarrow_5 (Cx \neq b)?WR(Y, b)$$

The reader may observe that this and the previous rule are triggered by the same event, and that this rule must fire before the previous one. As explained in Appendix A.1, our rule language permits a sequence of conditions and events on the right-hand side of rules to achieve this. Note that the CM-Shell at each site can use only data that is local to that site, therefore strategies do not need global data access.

Once our framework has been used to specify a strategy (and to verify the correctness of a guarantee) then the rule-based strategy specification is implemented using the host language of the Constraint Manager. In our toolkit, the implementation uses a distributed rule engine, although other implementations could also be used.

3.3 Specifying Guarantees

A guarantee is essentially a modified (usually weakened) form of the constraint being managed. As we will see, guarantees vary in strength, from guarantees like "$X = Y$ always," which is very useful but very difficult to achieve, to guarantees like "$X = Y$ if there are no updates for a day," which is easy to achieve but not very useful. In between these two extremes is a spectrum of weakened guarantees that are both useful and relatively easy to achieve. One of the strengths of our approach is that it lets us specify guarantees anywhere in this range, unlike existing systems where one either

[2] Note that the two databases can be of different types. For example, the first may be a relational database while the second is a flat-file system. The complexity of translation to and from these different data models is handled by the *CM-Translators*, described in Section 4.

gets strong consistency (with distributed transactions, when applicable), or no consistency at all. However, we note that identifying the right weakened guarantees that are meaningful to applications and that can be enforced is challenging. We return to this issue in Section 7.1.

Guarantees are logical expressions involving occurrences of events and predicates over data items and time. The basic construct of the guarantee language is the following:

$$\{Event|Condition\}@Time\_variable$$

For example, $W(X,5)@t_1$ means that there is a write operation "$X \leftarrow 5$" performed at time t_1. Similarly $(X = 25)@t_2$ means that the data item X has value 25 at time t_2. In addition to this construct, we have predicates over data items, variables, and constants, and the usual logical connectives such as *and* ($\wedge$), *or* ($\vee$), *not* ($\neg$), *implies* ($\Rightarrow$), etc. We also permit a limited form of quantification for variables representing time, as described in the first example below. (Note that quantification over data involved in a constraint is achieved by means of parametrized data names, as explained in Section 3.1.1.)

3.3.1 Example: Some Guarantees for Copy Constraints

Consider the case of an inter-site copy constraint $X = Y$ between a data item X at site S_1 and a data item Y at site S_2. Suppose we wish to maintain Y as a copy of X. Below, we discuss some guarantees, one or more of which may be useful for a given application. For simplicity in presentation we consider a single copy constraint, but our guarantees also apply to a set of inter-site copy constraints over related data items, where X and Y are replaced by parameterized data names.

- A simple guarantee that is desirable in many situations is that at no time should Y have a value not previously taken by X. Informally, we call this the "Y follows X" guarantee. Formally, we express this by saying that if Y has a certain value at time t_1, then X must have had that value at some time t_2 before t_1. Note that, implicitly, variables on the left-hand side of the $\Rightarrow$ sign are universally quantified, while those on the right-hand side are existentially quantified:

  $(Y = y)@t_1 \Rightarrow (X = y)@t_2 \wedge (t_2 < t_1)$ .... (1)

- In some cases, one may also want that every value taken by X is eventually reflected in Y. That is, if $X = x$ at some time, we are guaranteed that $Y = x$ at some later time; there are no missing values. Informally, we call this the "X leads Y" guarantee. Formally, we express this as follows:

  $(X = x)@t_1 \Rightarrow (Y = x)@t_2 \wedge (t_2 > t_1)$ .... (2)

  Note that this guarantee may not be desirable in all situations. For example, if X represents the position of a "player" in an interactive distributed game, we usually are only interested in the latest position of the player (the most recent value of X), and we do not care about a missed update. On the other hand, there are situations in which it is important for each value to be propagated. For example, if X represents the phone number of an employee, and if Y is a copy of X on another system that is interested in recording all the phone numbers of this employee over time, then guarantee (2) is desirable.

- In many cases, the order in which the updates are propagated is important in addition to the assurance that they are eventually propagated. For example, if X represents the position of a robot and Y is its copy on a system that plots the robot's path, we would like to receive the updated positions of the robot in the order in which the updates are actually made. Informally, we call this the "Y strictly follows X" guarantee. Formally, we express this as follows:

  $(Y = y_1)@t_1 \wedge (Y = y_2)@t_2 \wedge (t_1 < t_2) \Rightarrow (X = y_1)@t_3 \wedge (X = y_2)@t_4 \wedge (t_3 < t_4)$ ....(3)

We call guarantees such as the ones above *non-metric* since they do not make explicit reference to time intervals. That is, they specify only the order in which events occur and predicates are satisfied, not an explicit delay between them. In contrast with non-metric guarantees, we also have *metric* guarantees, which state that some event must occur or some predicate must be true within some fixed time bound of another event or predicate. We can extend non-metric guarantees (1) and (2) above to metric guarantees by placing a bound on the delay between the time at which the two conditions mentioned in the guarantees are true. For example, the metric form of guarantee (1) is the following:

$(Y = y)@t_1 \Rightarrow (X = y)@t_2 \wedge (t_1 - \kappa < t_2 < t_1)$ ....(4)

where κ is a constant. This guarantee specifies that if $Y = y$ at time t_1, then $X = y$ at some time t_2 that is at most κ seconds before t_1. Informally, Y takes values held by X no more than κ seconds ago.

4 A Toolkit for Constraint Management

We have presented a framework and language for specifying interfaces, strategies, and guarantees for constraint management in heterogeneous systems. In this section we describe how we have used this framework to develop and implement a toolkit for constraint management. The toolkit provides a set of easily configurable services that monitor or enforce constraints spanning multiple loosely coupled databases. We first briefly describe the architecture of the toolkit, and then we use an example to illustrate some of its features.

4.1 Architecture

Figure 2 depicts the architecture of our constraint management toolkit, which is a realization of the logical architecture depicted in Figure 1. At the lowest level we have the *Raw Information Sources* (RIS), which could be relational or object-oriented database systems (OODBs), file systems, bibliographic information systems, electronic mail systems, network news systems, and so on. Each

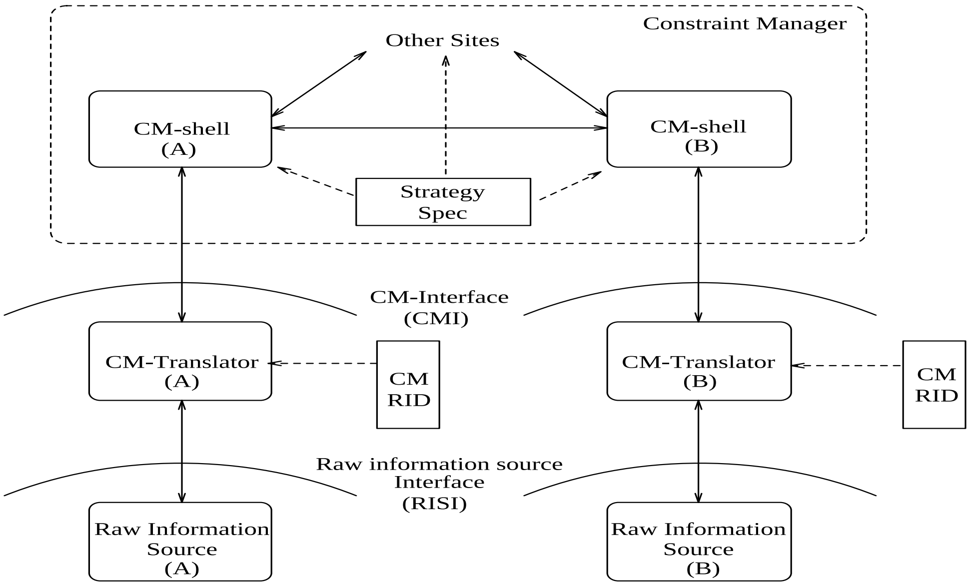

Figure 2: Constraint Management Toolkit Architecture

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 interact 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. To factor this complexity away from the CM-Shells, 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 can usually be selected from available menus of proven strategies and interfaces. However, the toolkit is extensible and can accommodate custom interface and strategy descriptions written using our rule language.

During initialization, the CM-Shells query the CM-Translators about the local capabilities and services. The CM-Translators respond with the interface specifications. The CM then suggests strategies that are applicable to these interfaces, along with the associated guarantees. The system administrator can either select one of the suggested strategies, or specify a different strategy using the strategy specification language. Once a strategy is specified, the CM distributes the rules of the strategy to CM-Shells based on the site of the event on the left-hand side of the rule. Each rule is executed in the CM-Shell handling the site at which the left-hand side event occurs. Based on this distribution of rules, the CM also determines, for each event template in each rule, the CM-Shells and/or the CM-Translators to which an event matching that template must be forwarded. During initialization, the CM-Translators also perform any set-up required for supporting the selected interface. For example, a CM-Translator supporting a Notify Interface for a Sybase RIS may need to declare triggers on the underlying database.

At run-time, the CM-Shells process events received from their respective CM-Translators and fire rules appropriately. The events that are produced as a result of rules firing are forwarded to the local CM-Translator and other CM-Shells as determined during initialization. CM-Translators implement the events using the native facilities of the RIS, thus executing the strategy. The CM-Shell supports a simple programmatic interface to allow applications to read auxiliary CM data for the guarantees that refer to it.

4.2 Example

Consider the following scenario. A company stores the personnel information for some of its employees in a local San Francisco branch database A. Personnel information also is stored in a database B at the headquarters in New York. These databases are loosely coupled in the sense described in Section 1. We wish to maintain the following constraint: For each employee in the San Francisco database, the salary stored in database A must equal the salary stored in database B. This is an example of a parameterized copy constraint. Let $salary1(n)$ denote the salary of n in database A, where, intuitively, n represents the employee ID. Similarly, let $salary2(n)$ denote the salary of n in database B. The constraint is then $salary1(n) = salary2(n)$ for all n in database A.

We first demonstrate how the toolkit is used to define interfaces for $salary1(n)$ and $salary2(n)$, and then we show how a simple strategy is specified and implemented. Finally, we discuss the validity of different guarantees, and we show how, with very little effort, we can continue to enforce the copy constraint even when the interface for $salary1(n)$ changes. The reader may wish to refer to Figure 2 as the description proceeds.

4.2.1 Interfaces

Suppose the RIS B is a Sybase relational database that provides a write interface for data item $salary2(n)$, defined in our language as $WR(salary2(n), b) \rightarrow_{\delta} W(salary2(n), b)$. In practice, this interface means that the RIS at site B can be instructed to write object $salary2(n)$. The CM-RID tailors the CM-Translator to handle such write requests. In addition to the interface statement, the CM-RID at B specifies the following:

- 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 *salary2*(n), the SQL query "UPDATE EMPLOYEES SET SALARY = b WHERE EMPID = n" must be sent to the SQL server. Note that we use the parameter n in the query. Our CM-Translator performs the necessary substitution given a particular instance of n.

- 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 acknowledgment.

Suppose the RIS at site A offers a notify interface for data item *salary1*(n). This interface is defined by the rule $W_s(\mathit{salary1}(n), b) \rightarrow_\delta N(\mathit{salary1}(n), b)$. For the purpose of this example, let us assume that the notify interface is implemented by declaring a database trigger on the data items *salary1*(n). The CM-RID specifies what the CM-Translator at A needs to do to declare the trigger, and what it should expect to receive from the RIS when *salary1*(n) changes.

4.2.2 Strategy

Consider the following simple strategy: Make a write request to *salary2*(n) within δ seconds whenever a notification of a write to *salary1*(n) is received. We express this using our strategy specification language as follows:

$N(\mathit{salary1}(n), b) \rightarrow_\delta WR(\mathit{salary2}(n), b)$

This strategy specification is processed by both of the CM-Shells. The strategy specification also indicates where objects are located, i.e., that *salary1*(n) is at site A and that *salary2*(n) is at B. As explained earlier, from the site of the event template on the left-hand side of the rule, the toolkit can determine which CM-Shell is responsible for executing each rule. In our example, the CM-Shell at A is responsible for the left-hand side of the rule because *salary1*(n) is at that site. When the A CM-Shell receives a $N(\mathit{salary1}(n), b)$ event from its CM-Translator, it forwards the event to the B CM-Shell, since the B CM-Shell is responsible for the right-hand side of the rule. The B CM-Shell then sends the $WR(\mathit{salary2}(n), b)$ event to its local CM-Translator. Based on the expected maximum execution time of each CM-Shell and the maximum transmission time between CM-Shells, the database administrators can compute an estimate for δ, the time guarantee in the rule. (See the discussion on timing guarantees at the end of Section 5.)

4.2.3 Guarantees

Given the interfaces and the strategy above, we can prove that guarantees (1), (2) and (3) of Section 3.3.1 are all valid. We can also prove that the associated metric guarantee (4) is valid for an appropriate κ. Intuitively, it is easy to see why these guarantees are all valid; yet, there are important details (such as a requirement for in-order message processing) that were discovered during the process of verification of the guarantee using our proof rules.

Now consider what happens if the administrator at site A decides to change the interface for data item *salary1*(n) from the above notify interface to a read interface (described in Section 3.1.1). Now the CM is no longer notified of updates to *salary1*(n); instead, the database at A only offers to respond with the current value of *salary1*(n) whenever it is requested. Since the only way to find out about changes to *salary1*(n) in this scenario is to periodically read the salaries, we must use a polling strategy. The simplest strategy is to periodically read *salary1*(n) and propagate the value read to *salary2*(n).[3] We express this strategy as follows:

$P(60) \rightarrow_\delta RR(X)$
$R(X, b) \rightarrow_\delta WR(Y, b)$

Recall that the event $P(60)$ represents a periodic event that occurs every 60 seconds.

Guarantees (1), (3) and (4) from Section 3.3.1 are valid in this scenario, while guarantee (2) is not. Intuitively, it is easy to see why guarantees (1), (3) and (4) are valid. The reason guarantee (2) is not valid is that since we are polling *salary1*(n) periodically, it is possible for us to "miss" updates when two or more updates to *salary1*(n) occur in the same polling interval.

4.3 Implementation Status

We have implemented CM-Translators for Unix files and relational databases. The translators are implemented using an object-oriented approach that requires only minor amounts of rewriting when moving to different kinds of raw sources (RIS). Currently, some low-level details for communicating with, say, Sybase must be embedded in the CM-Translator code. 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. Porting the CM-Translator to, say, a WAIS-like RIS involves incorporating the WAIS protocol for the submission of queries and the retrieval of results. We could avoid the need 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 complexity in the CM-Translator and 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].

We have used our toolkit to implement several constraint management scenarios such as copy constraints for data with read, write and notify interfaces. Strategies include update propagation, polling, and the Demarcation Protocol (described in Section 6). We are currently implementing a large scenario for distributed constraints involving several databases at Stanford. The databases include the Stanford "whois" database, the Computer Science Department's custom personnel

[3]Note that we could certainly do better, for example by caching *salary1*(n) at the CM and propagating updates to *salary2*(n) only when the value of *salary1*(n) changes. In the interest of simplicity, we do not consider this strategy here.

database ("lookup"), the database group's Sybase database, and a bibliographic database. There are copy constraints for different personnel data such as phone numbers, addresses, etc., stored in the different databases. We also have referential integrity constraints, such as one that specifies that every paper authored by a Stanford database researcher as reported by the bibliographic database must also be mentioned in the Sybase database.

Using our toolkit, we coordinate the activities of the loosely coupled, heterogeneous databases without modifying the databases or the existing applications, thus maintaining database autonomy. Heterogeneity in the modes of database access and control is handled in a uniform way by describing interfaces using our rule language. Furthermore, incorporating new databases or changing the interface to an existing database requires very little work, since only the high-level interface and strategy specifications have to be modified (and can be chosen from a menu in most cases). Even though the databases in the system are not transactional, our toolkit provides a formal notion of data consistency that is useful in practice.

5 Failure Handling

In a distributed environment, especially a loosely coupled one, coping with failures is an important component of any coordinating software. We classify failures of the databases in our architecture into the following two types:[4] We say a database interface has had a *metric failure* when it is unable to honor the time bounds specified in the interface specifications. In such a scenario, the actions mandated by the interface statements are eventually performed, but not within the time bound specified. Such failures may be caused by the underlying database being overloaded or crashing. (In many cases, crashes can be mapped to metric failures if the database has some basic recovery facilities and can "remember" messages that need to be sent out upon recovery.) When a metric failure occurs on one or more of the sites involved in a constraint, the metric guarantees for that constraint are no longer valid. However, the non-metric guarantees continue to be valid, which may allow many applications to continue to function.

The second kind of failure is one in which the interface statements are no longer valid at all. We call this a *logical failure*. Such a failure may be caused by catastrophic failure of one or more of the databases, and we expect such failures to be very infrequent. When a logical failure occurs, both metric and non-metric guarantees involving the failed site are no longer valid until the system is reset.

In our current implementation, failures are detected and flagged. In the future, we plan to incorporate a more sophisticated failure handling scheme into our toolkit, permitting applications to deal with failures in a more sophisticated manner. Recall that in our toolkit, the CM-Translator translates the raw interface (RISI) of the underlying database to the CM-Interface presented to the CM-Shell. The CM-Translator also maps (when possible) failures of the RISI into metric or logical

[4]Throughout this paper, we assume a reliable network. Therefore, we consider only site (database) failures here. Of course, network failures can be viewed as the failure of the sites sending the affected message.

failures of the CM-Interface. On detecting a failure, the CM-Translator notifies the local CM-Shell, which then propagates the information to other CM-Shells so that the affected guarantees may be marked as invalid.

The method used by the CM-Translator to detect failures of the RISI depends on the nature of the RISI and the CM-Interface being supported. For example, consider a CM-Translator implementing a Read Interface with a Unix file system as the underlying database. In this case, the the CM-Translator will use the *read()* system call interface to the Unix file system. Failure of this system call can be detected based on the return value, and such a failure can be flagged as a failure of the Read Interface. Depending on the reason for the failure of the *read()* call, the Read Interface failure is flagged as either a metric or a logical failure. Similarly, a CM-Translator for a Sybase database can detect and flag interface failures based on the error codes returned by calls to the Sybase library routines that communicate with the SQL server.

Sometimes, however, the nature of the RISI may make detecting failures very difficult or impossible. For example, consider a CM-Translator supporting a Notify Interface for a legacy database, and suppose the database simply sends a message to the CM-Translator whenever there is an update to some data item. If the database fails silently and does not report some update, there is no way for the CM-Translator to detect the failure. If it is not possible to ensure that the probability of such undetectable failures is acceptably low, then a Notify Interface should not be used for this database. Often, one can use another available interface, such as a Read Interface, and use polling to simulate notification, as in the example in Section 4. Note that some probability of undetectable failures exists in most systems. For example, undetected hardware failures (e.g., double parity errors on disk or memory reads) can silently corrupt data on disk or in memory, and this is not detected until some application fails unexpectedly.

The probability of a metric failure of an interface depends on the choice of the time bound in the rules specifying that interface. Typically, these time bounds 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. It is well known that all fault tolerant systems have to select constants (timeouts, number of retries, etc.) in a similar manner [Cri89]; the difference is that our toolkit makes the effects of choosing these constants explicit in the form of metric guarantees, so that applications do not have to guess.

6 Additional Constraint Management Scenarios

Our framework and toolkit for constraint management can be applied in a variety of scenarios, with interfaces and strategies ranging from simple (such as those described so far) to complex. In this section, we briefly illustrate some additional scenarios to show the wide range of interfaces, strategies and guarantees that our framework and toolkit can cover.

6.1 Demarcation Protocol

Consider the inequality constraint $X \leq Y$ where X and Y are at different sites. The *Demarcation Protocol* [BGM92] is applicable in this scenario if the sites of X and Y both offer certain interfaces. The protocol guarantees that the constraint $X \leq Y$ is always valid. The protocol uses local *limit data items* X_l and Y_l (located at the sites of X and Y respectively) and uses the constraint managers of the underlying databases to enforce the local constraints $X \leq X_l$ and $Y \leq Y_l$. Using our framework, we can accurately specify the interfaces assumed by the Demarcation Protocol, which are fairly complex. We can also specify the Demarcation Protocol itself. Then, using our proof rules, we can prove the guarantee $X \leq Y$ based on the specification of the protocol strategy and the interfaces. Thus, our framework is capable of expressing a complex scenario, in which we use the facilities of the underlying database (such as local constraint managers) in order to implement a global constraint, in addition to the simple ones we discuss elsewhere in this paper.

There are many ways of implementing the Demarcation Protocol such that the above guarantee is valid, and some of these are less desirable than the others. For example, an implementation that simply does not change the limit data items X_l and Y_l will satisfy the above guarantee, but is not very desirable since it does not permit X (and Y) to ever exceed (respectively, fall below) their original limit values. We can formalize this intuitive guarantee by introducing an event to denote a request for a limit-change operation, and by specifying that if there is enough "slack" at one site, then a change-limit request at the other site must be granted within some time. Different implementations of this protocol (called *policies* in [BGM92]) can then be compared using this guarantee. In [CGMW94], we have presented a formal specification of interfaces, strategies and guarantees for the Demarcation Protocol.

6.2 Referential Integrity

Consider a referential integrity constraint that states that for every employee ID having a record describing a project assignment ("project record" for short) in one database, there must be a record in another database with the salary information for that employee ID ("salary record" for short). A weakened form of this constraint, more suitable to loosely coupled heterogeneous systems, is the following guarantee: The above referential integrity constraint may be violated for any one employee ID for a period of at most 24 hours. We express this guarantee using an *exists* predicate.[5] An exists predicate $E(X)$, where X is a data item name (usually parameterized), is true if and only if the data item X exists in the database. In our example, let $project(i)$ denote the project record of an employee with employee ID i, and let $salary(i)$ denote the salary record of that employee as stored in the second database. The guarantee states that if the project record exists at time t in the first database, the salary record must exist in the second database within 24 hours (86,400

[5] We use an explicit exists predicate because our language does not include general quantification for data item names (although we do have quantification of time variables).

seconds), and is expressed as follows:

$E(project(i))@t \Rightarrow E(salary(i))@(t+86400)$

A simple strategy to realize the above guarantee is the following. At the end of each working day, the CM deletes all project records from the projects database that do not have a corresponding salary record in the salary database, perhaps notifying the database owner of the deleted records. This strategy assumes the projects database permits the CM to delete records. If this is not the case, then there may be no way for the CM to enforce the referential integrity constraint. However, the CM could still monitor the constraint using a technique similar to that described in the next section.

6.3 Monitor

Consider a scenario where we have a copy constraint $X = Y$, both X and Y have notify interfaces (see Section 3.1.1), and the CM cannot update either data item. In this case, the best the CM can do is to monitor the constraint. One method of monitoring the constraint is to maintain some auxiliary data items at the site of the application[6] interested in this constraint. (We discuss the storage and access of auxiliary data in Section 7.1.) These auxiliary data items indicate the validity of the constraint over time. In particular, we may offer the following guarantee to an application:

$((Flag = true) \wedge (Tb = s))@t \Rightarrow (X = Y)@@[s, t-\kappa]$

Here *Flag* and *Tb* are auxiliary data items The guarantee states that if *Flag* is true, then $X = Y$ was true at all times in the time interval indicated on the right-hand side. The auxiliary data item Tb is used to keep track of the start of the time interval during which $X = Y$. (In the guarantee, we use Tb by accessing its value s on the right-hand side.) Details of the strategy used to ensure this guarantee, as well as the proof of the guarantee using the interface and strategy specifications is in [CGMW94].

6.4 Periodic Guarantees

A periodic guarantee is one that states that the constraint is valid periodically. For example, consider an old-fashioned banking environment in which all update transactions occur between 9 a.m. and 5 p.m. Suppose we have a set of copy constraints stating that balances for each account at the local branch and the head office must be equal. A simple strategy in this situation is to propagate the new values of account balances from the branch to the head office at the end of each working day. If the branch offers an interface that guarantees that there will be no updates to account balances between 5 p.m. and 8 a.m., and if the propagation of new values at the end of the day takes 15 minutes, we can offer a periodic guarantee that the copy constraints will be valid every day from 5:15 p.m. to 8 a.m. the next day. Such a guarantee permits, for example, a financial

[6]In the case of multi-site applications, this is the site of the CM-Shell that services the application. The choice of a CM-Shell can be arbitrary.

analysis application at the main office to proceed with the assurance of consistency, assuming it runs in the above time interval.

7 Discussion

In this section, we discuss some details of how applications can use the guarantees offered by our toolkit. We also explain why the implementation of strategies does not require global data access and clocks, and we discuss how auxiliary CM data is stored and accessed.

7.1 Using Guarantees

From the viewpoint of an application, weakened notions of consistency as expressed by our guarantees are not as easy to use as conventional, strict consistency is. Yet, given the restrictions on data access and control in a loosely coupled heterogeneous environment, weakened consistency is usually all that can be offered. Weakened consistency is certainly more useful than no consistency guarantees at all, which is what usually is offered to applications in current systems. In this section, we discuss how applications can use weakened consistency guarantees.

The ease of use of a guarantee depends on the "strength" of the guarantee. Guarantees that are relatively strong are easy to use. For example, a strong non-metric guarantee like $X \leq Y$ (as offered by the Demarcation Protocol) permits the application know at all times that the data it sees is consistent. Similarly, consider guarantees (1) and (2) of Section 3.3.1 from the viewpoint of an application that runs at Y's site and tabulates the different values taken by X. This application can read Y and be assured that Y is a value previously taken by X (due to guarantee (1)) and that Y does not miss any values that X takes (due to guarantee (2)). Hence, these guarantees permit the application to know that its tabulation is correct.

The guarantees that are harder to use are those that are conditional on the values of some auxiliary data items. For example, consider the following guarantee introduced in Section 6.3:

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

In order to use this guarantee, the application must read the values of the auxiliary data items *Flag* and *Tb*. Such auxiliary data may be stored in a private database of the CM-Shell at the site of the application. In this case, the application reads their values through the CM-Shell. Alternatively, the underlying database at the site may offer to store such auxiliary data for the CM-Shell. In this case, the application can read the auxiliary data directly from the database. In our toolkit, for example, auxiliary data for a CM-Shell that is at the site of a Sybase database is stored in the Sybase database, while the auxiliary data for a CM-Shell that is at the site of a bibliographic information system is stored in a private database.

Note that guarantees are always designed in such a way that that applications can use them without the need for global transactions. This is done by ensuring that reading local data only is sufficient to conclude some desirable properties of the global data. In the above guarantee, for example, by reading local (auxiliary) data items *Flag* and *Tb*, an application can determine the validity of the global constraint $X = Y$. It is a straightforward extension of the strategy and guarantee in that example to replicate *Flag* and *Tb* at the site of each application interested in the guarantee, so that each application accesses only local data.

The reader will note that it is important to read the data items on the left-hand side of the $\Rightarrow$ in the guarantee consistently. If the auxiliarly data items *Flag* and *Tb* are stored in the CM-Shell, the CM-Shell ensures that they are read consistently, since they are under its control. If, however, they are stored in the underlying database, then the database must have some facility to permit consistent read of these data items.

Once the application reads the auxiliary data items *Flag* and *Tb*, it can determine whether the constraint was valid during the interval specified on the right-hand side of the guarantee. To see how this guarantee is useful, suppose that the application received some query results based on the values of X and Y at some time in the past. Then this guarantee permits the application to check whether that the query was computed based on a consistent state of the data. If $X = Y$ was true at the time at which the earlier query was computed, the application can proceed with confidence in the query results. If the guarantee is inconclusive about whether $X = Y$ was true at the query-computation time (either because *Flag* is false or because the time interval on the right-hand side of the guarantee does not include the time of interest to the application), the application can either proceed with the understanding that the query results may not be accurate, or it can recompute the query in the hope that the new computation will be performed on a consistent state.

In this paper, we have illustrated guarantees only for some simple copy, inequality, and referential integrity constraints. As constraints get more complex, their guarantees will also increase in complexity, making them more difficult to use. However, note that simple constraints like the ones we consider here are the most common kind of constraints in a loosely coupled heterogeneous environment, where it is unlikely that autonomous data repositories will have very complex interdependencies. Furthermore, if there are complex constraints in a loosely coupled 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 different sites. A common way to manage this constraint is to have cached copies Y_c and Z_c of Y and Z, respectively, 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 3.3.1, for instance. Thus, even if our framework is used for simple constraints only, we believe it can cover the vast majority of the scenarios of interest for loosely coupled heterogeneous systems.

7.2 Executing Strategies

In Section 7.1 we have seen how the use of guarantees does not require global transactions. Similarly, execution of strategies (rules) does not require global transactions. To see this, note that while the left-hand side and right-hand side of a rule each (separately) execute "atomically," the entire rule does not. Further, each side of a rule is restricted to accessing data that is all at the same site.

Thus the atomic execution of each side of the rule can be implemented in the local CM-Shell.

Another issue is that of clock synchronization. In the formal part of our work, we use global time to reason about events and conditions. This approach is similar to that in [Koy92]. For example, consider a rule of the form $\mathcal{E}_1 \rightarrow_5 \mathcal{E}_2$ where events E_1 and E_2 matching the event templates $\mathcal{E}_1$ and $\mathcal{E}_2$, respectively, are at different sites. This rule states that if E_1 occurs at site S_1 at 9:00:00 a.m., then E_2 must occur at site S_2 before 9:00:05 a.m., where both times refer to absolute, wall-clock time. Recall that this is a specification of an interface or strategy based on expected maximum delays. Implementing such a rule does not require any access to global time (or even a notion of time in the implementation). Thus our toolkit does not rely on global clock synchronization for implementing strategies even though we use it as a reasoning tool in our formal framework. Certain kinds of guarantees, such as the periodic guarantees of Section 6, explicitly refer to global time, and they assume global clocks. Such a scenario does not pose a problem as long as the time intervals specified in the guarantee are significantly larger than the expected skew in system clocks. In the example of Section 6, a clock skew of a few seconds (or even minutes) can be accommodated by including an error margin in the interval specified in the guarantee.

8 Conclusion

Distributed integrity constraints arise naturally when information systems inter-operate, due to interdependencies between data. Traditional constraint management techniques assume facilities like atomic transactions, locking, and global 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 and capabilites for accessing data, which also makes constraint management more difficult. Currently, constraints in heterogeneous environments are either not monitored at all, or are 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 information systems. Our framework formalizes weakened notions of consistency, which are essential in real-world loosely coupled heterogeneous scenarios, where it is not possible to guarantee strict consistency. Our framework also allows us to formally specify the interfaces each database offers, along with constraint management strategies. Our toolkit provides a set of configurable modules that enable us to monitor and enforce constraints in a uniform and useful manner.

References

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

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

[BGM92] D. Barbara and H. 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] S. Chawathe, H. Garcia-Molina, and J. 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`.

[CGMW94] S. Chawathe, H. Garcia-Molina, and J. Widom. Constraint management in loosely coupled distributed databases. Technical report, Computer Science Department, Stanford University, 1994. Available through anonymous ftp from host `db.stanford.edu`.

[Cri89] F. Cristian. A probabilistic approach to distributed clock synchornization. In *Proceedings of the Ninth International Conference on Distributed Computing Systems*, pages 288–296, Jun 1989.

[CW93] S. Ceri and J. 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] A. Elmagarmid, editor. *Special Issue on Unconventional Transaction Management,* Data Engineering Bulletin 14(1), March 1991.

[Gre93] P. 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] A. Gupta and J. 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.

[GW94] P. Grefen and J. Widom. Integrity constraint checking in federated databases. Memoranda Informatica 94-80, Department of Computer Science, University of Twente, The Netherlands, December 1994.

[Koy92] R. Koymans. *Specifying Message Passing and Time-Critical Systems with Temporal Logic.* Number 651 in Lecture Notes in Computer Science. Springer-Verlag, 1992.

[L+93] D. Luckham et al. Partial orderings of event sets and their application to prototyping concurrent, timed systems. *Journal of Systems and Software*, 21(3):253–265, June 1993.

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

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources, March 1995.

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

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

Appendix A Syntax and Semantics of Rule Language

In Section 3, we motivated and informally presented the rule language used in our framework. Many concepts were introduced by example in that section. In this section, we present the formal specification of the syntax and semantics of our rule language. Examples of how these definitions can be used to prove guarantees of consistency can be found in [CGMW94].

Appendix A.1 Events, Templates and Rules

We first define events and event templates, which are the building blocks of our rule langage. The syntax and informal semantics of the rule language are presented next. (Formal semantics of the rule language are in the next section.)

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.

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 the following set of descriptors. (This set can be expanded by adding new templates and their semantics.)

$$\{W_s(x, \_), W_g(x, \_), RR(x), N(x, \_), WR(x, \_)\}$$

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 below.

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.

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.

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 $W_s(X, \_, b)$.

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

$$\mathcal{E}_0 \wedge C_0 \rightarrow_\delta C_1?\mathcal{E}_1, C_2?\mathcal{E}_2, \ldots, C_k?\mathcal{E}_k$$

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.[7] 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.

Appendix A.2 Valid Executions

We define the semantics of our rule language using the concept of valid executions over a system of databases. 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 and Appendix A.1.

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

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[8] 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 $E_i.old$.

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.

[8] 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'.

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 $E_i \rightsquigarrow^R E_j$.

   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 $E_1@t_1 \rightsquigarrow^{R_1} E_2@t_2$ and $E_3@t_3 \rightsquigarrow^{R_2} E_4@t_4$, where R_1 are R_2 are related rules, then $t_1 < t_3$ iff $t_2 < t_4$.

Using the above specification of the semantics of our rule language, in [CGMW94] we derive proof rules and present proofs of some consistency guarantees.