5

Supporting the requirements for multi-level secure and real-time databases in distributed environments

Sang H. Son and Craig Chaney
Department of Computer Science, University of Virginia, Charlottesville, VA 22903
son@virginia.edu

Abstract

Conflicts in database systems with both real-time and security requirements can sometimes be unresolvable. We attack this problem by allowing a database to have partial security in order to improve on real-time performance when necessary. By our definition, systems that are partially secure allow security violations between only certain levels. We present the ideas behind a specification language that allows database designers to specify important properties of their database at an appropriate level. In order to help the designers, we developed a tool that scans a database specification and finds all unresolvable conflicts. Once the conflicts are located, the tool takes the database designer through an interactive process to generate rules for the database to follow during execution when these conflicts arise. We briefly describe the BeeHive distributed database system, and discuss how our approach can fit into the BeeHive architecture.

Keywords

Real-time, partial security, specification, conflicts, rules, analysis tool, deadline-miss ratio

Database Security XI T.Y. Lin and Shelly Qian (Eds)

1 INTRODUCTION

A *real-time system* is one whose basic specification and design correctness arguments must include its ability to meet its timing constraints. This implies that its correctness depends not only on the logical correctness, but also on the timeliness of its actions. To function correctly, it must produce a correct result within a specified time, called *deadline*. In these systems, an action performed too late (or even too early) may be useless or even harmful, even if it is functionally correct [16]. If timing requirements coming from certain essential safety-critical applications would be violated, the results could be catastrophic.

Traditionally, real-time systems manage their data (e.g. chamber temperature, aircraft locations) in application dependent structures. As real-time systems evolve, their applications become more complex and require access to more data. It thus becomes necessary to manage the data in a systematic and organized fashion. Database management systems provide tools for such organization. The resulting integrated system, which provides database operations with real-time constraints is generally called a *real-time database system.*

In many of these applications, security is another important requirement, since the system maintains sensitive information to be shared by multiple users with different levels of security clearance. As more and more of such systems are in use, one cannot avoid the need for integrating them. Not much work has been reported on developing database systems that support both requirements of multilevel security and real-time. In this paper, we address the problem of supporting both requirements of real-time and security, based on the notion of partial security.

1.1 Real-time Database Systems

Real-time database systems extend the set of correctness requirements from conventional database systems. Transactions in real-time systems must meet their timing constraints, often expressed as deadlines, in order to be correct. In stock market applications and automated factories, a poor response time from the database can result in the loss of money and property. In many real-time database systems, transactions are given priorities, and these priorities are used when scheduling transactions. In most cases, the priority assigned to a transaction is directly related to the deadline of the transaction. For example, in the Earliest Deadline First scheduling algorithm, transactions are assigned priorities that are directly proportional to their deadlines; the transaction with the closest deadline gets the highest priority, the transaction with the next closest deadline gets the next highest priority, and so on. One important goal of a real-time transaction scheduler is to minimize or eliminate the number of *priority inversions* -- situations where a high priority transaction is forced to wait for a lower priority transaction to complete. As we shall see below, it is this goal that comes in conflict with security requirements.

1.2 Multilevel Secure Database Systems

Multilevel secure database systems have a set of requirements that are beyond those of conventional database systems. A number of conceptual models exist that specify access rules for transactions in secure database systems. One important such model is the Bell-LaPadula model [1]. In this model, a security level is assigned to transactions and data. A security level for a transaction represents its clearance level; for data, the security level represents the classification level. Transactions are forbidden from reading data at a higher security level, and from writing data to a lower security level. If these rules are kept, a transaction cannot gain direct access to any data at a higher security level.

However, system designers must be careful of covert channels. A covert channel is an indirect means by which a high security clearance process can transfer information to a low security clearance process [7]. If a transaction at a high security level collaborates with a transaction at a lower security level, information could flow indirectly. For example, say that transaction T_a wished to send one bit of information to transaction T_b. T_a has top secret clearance, while T_b has a lower clearance. If T_a wishes to send a "1", it locks some data item previously agreed upon. (This data item could be one that is created specifically for this covert channel by T_a.) If T_a wishes to send a "0", it does not lock the data item. Then, when T_b tries to read the data item and finds it locked, it knows that T_a has sent a "1"; otherwise, it knows that T_a has sent a "0". Covert channels may use the database system's physical resources instead of specific data items.

One sure way to eliminate covert channels is to design a system that meets the requirements of *non-interference*. In such a system, a transaction cannot be affected in any manner by a transaction at a higher security level. In other words, a subject at a lower access class should not be able to distinguish between the outputs from the system in response to an input sequence including actions from a higher level subject and an input sequence in which all inputs at a higher access class have been removed [7]. For example, a transaction must not be blocked or preempted by a transaction at a higher security level.

1.3 Integration of Real-time and Security Requirements

The requirements of real-time systems and those of security systems are often in conflict [11]. Frequently, priority inversion is necessary to avoid covert channels. Consider a transaction with a high security level and a high priority entering the database. It finds that a transaction with a lower security level and a lower priority holds a write lock on a data item that it needs to access. If the system preempts the lower priority transaction to allow the higher priority transaction to execute, the principle of non-interference is violated, for the presence of a high security transaction affected the execution of a lower security transaction. On the other hand, if the system delays the high priority transaction, a priority inversion occurs. The system has encountered an unresolvable conflict. In general, these unresolvable conflicts occur when two transactions contend for the same resource, with one transaction

having both a higher security level and a higher priority level than the other. Therefore, creating a database that is completely secure and strictly avoids priority inversion is not feasible. A system that wishes to accomplish the fusion of multi-level security and real-time requirements must make some concessions at times. In some situations, priority inversions might be allowed to protect the security of the system. In other situations, the system might allow covert channels so that transactions can meet their deadlines.

There are other factors, besides security enforcement, that could degrade the timeliness of the database system. For example, transient overload or failure of certain components could impact the system performance. However, regardless of the reason for impaired timeliness, relaxing security requirements always provide a positive impact on the system performance.

1.4 Our Approach

Our approach to this problem of conflicting requirements involves dynamically keeping track of both the real-time and the security aspects of the system performance. When the system is performing well and making a high percentage of its deadlines, conflicts that arise between security and real-time requirements will tend to be resolved in favor of the security requirements more often, and more priority inversions will occur. However, the opposite is true when the real-time performance of the system starts to degrade. Then, the scheduler will attempt to eliminate priority inversions, even if it means allowing an occasional covert channel.

Semantic information about the system is necessary when making these decisions. This information could be specified before the database became operational using a specification language. In this language, users would be able to express the relative importances of keeping information secure and meeting deadlines. Specifications in this language could then be "compiled" by a pre-processing tool. After a successful compilation, the system should be deterministic in the sense that an action must be clear for every possible conflict that could arise. This action might depend on the current level of real-time performance or other aspects of the system. Any ambiguities would be caught at compile time, causing the compilation to be unsuccessful. The compilation of the specification produces output that can be understood and used by the database system.

The problem of accomplishing the union of security and real-time requirements becomes more complicated in a distributed environment. In a distributed environment, having a single entity keep track of system performance in terms of timeliness and security for the entire global database could be impractical for a number of reasons. Requiring transactions to report to this performance monitor after every execution could put more load on the network and have a negative impact on performance. The node that contained the performance monitor would be a "hotspot" and might introduce a performance bottleneck. These problems would be serious as the system got bigger, so such a solution would have a limited scalability. This brings up an interesting question: Is it better to have many performance monitors,

each responsible for a small part of the database, or to have fewer of them, each with a larger domain? In other words, what granularity of the system should the performance monitors be responsible for? In our approach, there is a performance monitor responsible for every node. One of the issues to be addressed in a system with multiple performance monitors is how to optimize the database globally with only local knowledge. In our approach, this is accomplished through communication between performance monitors at each node.

In the next section, we describe some related works in the areas of specifying real-time and secure requirements for database systems, distributed security models, and some previous work in combining the requirements of real-time and secure database systems. Section 3 describes the partial security policy, the ideas behind the specification language, and the tool to analyze the language. In Section 4, we describe how our approach will fit into BeeHive, a distributed database system with real-time, security, quality of service, and fault-tolerance requirements. Section 5 concludes the paper with a discussion of future work.

2 RELATED WORK

There has been much work on specifying security requirements. One approach is using the security constraint classification system [13], in which the authors classified a number of security constraints. Nine different categories are given, ranging from the simple, usual constraints to constraints that classify the database depending on the content or security level of data. Constraints can also depend on real-world events, information that has been previously released, and can classify associations between data, collections of data, and even other constraints. The current version of the tool reported in this paper does not provide such a flexible specification, but eventually, we need to support a complete security constraint classification for each application.

Several methods of specifying real-time requirements also exist. For example, a real-time specification method of activity and data graphs, is presented in [8]. The fundamental building block in this design is called an atomic activity. This specification system does employ some clever techniques to group and relate these atomic activities through graphs. An activity, which is defined as a set of computations, can be viewed as a transaction. Atomic activities are given a set of properties that include name, preconditions, postconditions, preemptability, state variables, importance level, timing constraints, resource requirements, and behavior. The activities are also given temporal properties, such as arrival time, ready time, scheduling deadline, completion deadline, execution time, starting time, and completion time. Our model for the specification of real-time properties was influenced by these methods, and is probably closest to the model for the atomic activities. However, some of the properties used in that model were not necessary in ours, and we needed to add a couple of properties not present in the atomic activity model.

There have been attempts to define security protocols in distributed, object-oriented environments. Two examples are Legion [15] and CORBA [3]. However, we

are not aware of any previous attempts to satisfy *both* security and real-time requirements in a distributed, object-oriented environment. George and Haritsa studied the problem of combining real-time and security requirements [5]. They examined real-time concurrency control protocols to identify the ones that can support the security requirement of non-interference. This work is fundamentally different from our work because they make the assumption that security must always be maintained. In their work, it is not permissible to allow a security violation in order to improve on real-time performance.

3 SPECIFICATION

In this section, we first outline the approach to defining partial security. We then provide the details of specifying different rules for the database system.

3.1 Partial Security

As explained above, our approach will at times call for a violation of security in order to uphold a timeliness requirement. When this happens, the system will no longer be completely secure; rather, it will only be partially secure. One of the major research questions to be addressed is to identify quantitative partial security levels and to develop methods for making trade-offs for real-time requirements. Traditionally, the notion of security has been considered binary. A system is either secure or not. A security hole either exists or not. The problem with such binary notion of security is that in many cases, it is critical to develop a system that provides an acceptable level of security and risks, based on the notion of partial security rather than unconditional absolute security, to satisfy other conflicting requirements. In that regard, it is important to define the exact meaning of partial security, for security violations of confidential data must be strictly controlled. A security violation here indicates a potential covert channel, i.e., a transaction may be affected by a transaction at a higher security level.

One approach is to define security in terms of a percentage of security violations allowed. However, the value of this definition is questionable. Even though a system may allow a very low percentage of security violations, this fact alone reveals nothing about the security of individual data. For example, a system might have a 99% security level, but the 1% of insecurity might allow the most sensitive piece of data to leak out. For serious secure database applications, a more precise metric would be necessary.

A better approach involves adapting the Bell-LaPadula security model and blurring boundaries between security levels in order to allow partial security. In this scheme, only violations between certain security levels would be allowed. As the real-time performance of the system degrades, more and more boundaries can be blurred, allowing more security violations and reducing the number of security conflicts. Since there are less conflicts, this can improve the real-time performance of the system. Additionally, with this scheme, we can still make guarantees about the

security of the data.See Figure 1 for an example. Here, we are considering a system with four security levels: top secret, secret, confidential, and unclassified. In Figure 1a, the system is completely secure. Figures 1b through 1d show systems that are partially secure, progressing from more secure to completely insecure. Solid lines between security levels indicate that no violations are allowed between the levels; dashed lines indicate that violations are allowed. For example, in Figure 1b, transactions that are at the unclassified level may have conflicts with transactions at the confidential level in accessing to unclassified data, resulting in a potential covert channel.

It is possible to combine this approach with the use of percentages to define partial security. Then, the amount of security violations between two levels for which the boundary had been blurred would be required to fall below this percentage. The above example is really a special case of this scheme, where the percentage can either be 0% or 100%. Note that no guarantees can be made between levels that have been assigned a non-zero percentage. Guarantees can still be made between levels designated as allowing 0% security violations; for the other levels, database designers can use different percentages to denote their preferences on where they would rather have the potential security violations occur.

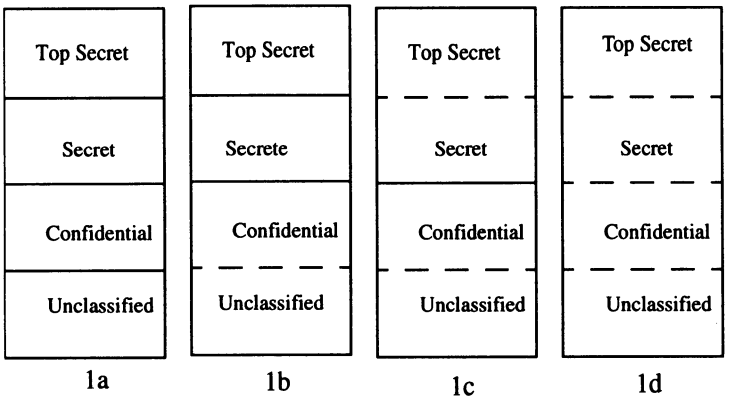

Figure 1 Partial Security Levels

For certain applications in which absolute security is required for safety-critical applications, any trade-offs of security for timeliness must not be allowed. The idea of partial security discussed in this paper cannot be used in such applications. Even if partial security is acceptable to an application, the system designer should be careful in identifying the conditions under which it might be dangerous to compromise the security. For example, some sort of denial of service attack could force the system into a condition where timeliness constraints are not satisfied. The system can limit the potential damage by setting up rules that can identify the situation and take appropriate actions, if necessary. For example, the system may audit the possible covert channels and log any activity that might be exploring the channel. The

rules can utilize the notion of encrypted profile to either look for patterns of illegal access or, alternatively, to certify a good pattern of access.

3.2 Specification Methods

Application designers should be able to specify semantic information using a specification language to express the relative importance of keeping desired level of security and meeting timing constraint requirements. A question to be addressed in that approach is the verification of the given specification. Specifications should be compiled and verified to check any inconsistency in the requirements and to clearly determine the necessary actions to be taken. We developed a specification language that allows designers to generate rules at varying levels of detail. We have also developed a tool to analyze the specification to identify any inconsistency and produce semantic information and rules that will be maintained by the database system. The approach to specifying the security and real-time requirements is a preprocessor that aids the database designer first with locating conflicts and then with denoting their preferences according to the semantics of the database. First, we will give the details of the specification language. We will then go through a few examples to illustrate the ideas.

3.2.1 Specification Language

Our specification language allows designers to create rules at varying levels of detail. In applications where much information is known about the database beforehand, designers can control security and real-time aspects of the database much more tightly than in situations where less is known beforehand or such a tight control is not required. There are three levels of detail in this specification scheme. Note that one system can use rules from all three levels if needed.

The specification consists of two parts: a *description* of the database and a set of *rules* to follow when conflicts arise. The description provides a framework for the rules. As we shall see below, the specification of both the description and the rules varies between the different levels of details. Regardless of the levels of details that are used, the first part of the specification contains facts about the database as a whole. Here, designers specify the number of data items, the number of security levels, and the number of priority levels used in the entire database.

In the first, most detailed level, designers can make rules for specific transactions. Transactions are given a number of components. Each transaction is given a readset and a writeset. These can consist of any number of data items. If no readset or writeset is given, they are assumed to be empty. The real-time requirements of a transaction are given by four variables: priority, execution time, release time, and periodicity. The periodicity of a transaction defines how often it starts executing, and the release time indicates the offset of the periodic start. Finally, transactions are given a security level.

Information about data can also be specified. Data items are specified by number, and each data item is given a security level. The specification can also contain

a default security level, which is assigned to any unspecified data items. All of this information about transactions and data belong in the description portion of the specification.

Not all of these components for transactions and data items are required. In general purpose database systems, some of the information might be hard to specify. However, in many real-time applications, most information is available, since such information is necessary for schedulability analysis of the system to support the timeliness and predictability requirements. In fact, in real-time database systems, many transactions are periodic and their access pattern is known. The only truly necessary components are the security level and the priority level. If a designer leaves out, for example, the readset or the writeset, the preprocessor tool (discussed below) cannot make any assumptions about the data accessed by this transaction, so it must assume that the transaction may conflict with every other transaction.

Next, the database designer comes up with rules that define the actions that the system must take when the transactions conflict. These rules can either be static or dynamic. Static rules apply to conflicts that are resolved in the same way every time. For example, the user might specify that a conflict between two specific transactions, or two categories of transactions, will never result in a security violation.

Dynamic rules can depend on certain run-time variables that the database keeps track of during execution. Currently, dynamic rules can be based on three different dynamic variables: security violation percentage, transaction miss percentage (the percentage of transactions that have missed their deadlines), and the number of consecutive missed deadlines. Each dynamic rule has a list of clauses and a default action. A clause contains a boolean relation (>, >=, =, <, or <=) between one of these three dynamic variables and a constant value. Each clause also contains the action (either violate security or violate priority) to be taken if the boolean relation is true. When a conflict is encountered by the database system, it checks the first clause. If that clause is true, it takes the associated action. If not, it checks the next clause. If none of the clauses turn out to be true, the database takes the default action. For example, a rule might be "If the security violation percentage is greater than 5, violate security. Otherwise violate timeliness." Here, the "otherwise" sentence represents the default action.

In a distributed environment, when conflict occur between transactions executing at different nodes, the action taken may need to depend on the performance at all the nodes. In that case, rules should be created that take into account the statistics on every nodes involved in the transaction. Every rule should have a "partner" rule, covering this contingency. This rule might also take into account the latency between the nodes at which the conflicting transactions are being executed.

The second level of specification detail replaces specific transactions with categories of transactions. Transactions are categorized by their security levels and priority levels. The designer can create any number of categories at any granularity that he or she feels is appropriate, and describes these categorizations in the description portion of the specification. Then, rules are created for conflicts

Description:

```
numDataItems 5;
numSecurityLevels 4;
numPriorityLevels 4;

data[default].security = 1;
data[3].security = 2;

ComputeProfit.readset = 1, 2, 3, 4;
ComputeProfit.writeset = 5;
ComputeProfit.periodicity = 12;
ComputeProfit.priority = 3;
ComputeProfit.security = 3;

UpdatePrice.writeset = 3;  # Two transactions access data item 3.
UpdatePrice.periodicity = 30;
UpdatePrice.security = 2;
UpdatePrice.priority = 2;
```

Rule for ComputeProfit-UpdatePrice conflict:

```
(SecViolation% >= 5) ~ violateTimeliness,
(TransMiss% > 10) ~ violateSecurity,
(otherwise) ~ violateTimeliness;

((LocTransMiss% <= 15) & (RemTransMiss% <= 10)) ~ violateTimeliness,
((LocSecViolation% < 10) & (RemSecViolation < 10)) ~ violateSecurity
(otherwise) ~ violateTimeliness;
```

Figure 2 Specification example with detail level 1

between categories of transactions. These rules are the same as the rules for the first level.

In the third level of specification, designers create a set of rules describing actions to take in case of conflicts. Here, the conflicts are not specific; the same rule set is consulted for every conflict. Conditions would depend on the characteristics of the transactions that are conflicting or the current performance statistics. Depending on the results of the comparison, the rule would mandate either a security violation or a priority violation. All of this information belongs in the rules portion of the specification; nothing is needed in the description portion.

By carefully creating the rules, database designers can implement the partial security scheme described in the previous section. As with many other aspects of designing these rules, a tool can help designers carefully model their partial security system.

Specifications are not required to solely use one of these levels of details. The descriptions and rules for these detail levels can be mixed. In this case, when the database encounters a conflict during execution, it first searches to see if a level 1 rule applies. If not, it searches the level 2 rules, and finally checks the level 3 rules.

3.2.2 Examples

Figure 2 shows an example of a system completely specified with detail level 1. This is a small example, with only two transactions. Every relevant component of these transactions has been specified. Both transactions access data item 45, and ComputeAverage writes to it, so we have a potential conflict. Since ComputeAverage has both a lower security level and a lower priority level than SampleTransaction, this conflict cannot be resolved without causing either a covert channel or a priority inversion. Had ComputeAverage been given a higher priority than SampleTransaction, we can satisfy both requirements by allowing ComputeAverage to preempt SampleTransaction. Alternatively, if ComputeAverage had a higher security level than SampleTransaction, then both requirements could be satisfied by forcing ComputeAverage to wait for Sample transaction. As will be seen in the next section, the task of locating such conflicts can be automated.

There are two rules for this conflict -- the local rule and the non-local rule. In the rule specification, `SecViolation%` indicates the percentage of security violations and `TransMiss%` indicates the percentage of deadline miss ratio. Each rules consists of a condition and a decision. The condition part of a rule is stated inside the parenthesis and followed by the decision after tilde (~). Conditions can be connected by logical AND (&) of OR (|). In the local rule, the first line represents a security crisis. If more than 5 percent of transactions have violated security, then this transaction cannot afford to, so it must violate timeliness. If the condition in the first line is false, the condition in the next line is checked. This line represents a real-time crisis. If more than 10 percent of transactions have missed their deadlines, then the real-time performance is suffering, so this transaction must violate security. Again, if the condition in this second line is false, the next line is checked. Here, this line is the "catch-all" rule. If none of the above rules apply, the database is instructed to violate timeliness.

The non-local rule operates in much the same way. The first line in this rule represents a state in which the real-time performance of the system is at an acceptable level either locally or at the remote site. If either condition is satisfied, the database is instructed to violate timeliness. If the real-time performance is not at an acceptable level, the system checks the second line of the rule to determine if the security of the system is acceptable. If so, it violates security; if not, it moves on to the third, "catch-all" line and violates timeliness.

Rule for SampleTransaction-ComputeAverage conflict:

```
(SecViolation% >= 5) ~ violateTimeliness,
(TransMiss% > 10) ~ violateSecurity,
(otherwise) ~ violateTimeliness;

((LocTransMiss% <= 10) | (RemTransMiss% <= 5)) ~ violateTimeliness,
((LocSecViolation% < 10) & (RemSecViolation < 10)) ~ violateSecurity,
(otherwise) ~ violateTimeliness;
```

Rule for HighSecurityCategory-LowSecurityCategory conflict

```
(otherwise) ~ violateTimeliness;
```

Level 3 rules:

```
(SecViolation% < 10) ~ violateSecurity,
(TransMiss% < 15) ~ violateTimeliness,
(priorityLevelDifference >= 2) ~ violateSecurity,
((TransMiss% > 10) & (SecViolation% <= 10)) ~ violateSecurity,
(otherwise) ~ violateTimeliness;
```

Figure 3 Example of mixed level specification

Figure 3 shows an example specification with mixed levels of detail (the database description is not shown). There are two transactions specified using detail level 1, but with only the bare minimum number of components specified. These transactions are the same as those used in figure 2. There are also a couple of transaction categories, relating to high and low security transactions. Also, there is an example of a level 3 rule set.

The rules for the conflict between the specific transactions is specified in the same manner as in the previous example. Here, we see the specification for conflicts between two transaction categories. These also are specified in the same manner.

This specific level 2 rule is also an example of a static rule -- every time that transactions in these two categories conflict, the database must violate priority and uphold security. Rules for violations between specific transactions and transaction categories can be specified, if the database designer so desires. Finally, we see a rule set for detail level 3. If none of the rules in level 1 or level 2 apply to a conflict encountered by the database, it determines the course of action by consulting this ruleset. Again, these are specified in the same manner, with the exception that a couple of new variables can be used. The variable priorityLevelDifference represents the difference in the priority levels of the two transactions; securityLevelDifference does the same for security levels

In Figure 4 we give an example of a rule that deals with multiple conflicts. This

rule is interpreted much like the level 3 rules. In the first line, the database has allowed a high number of security violations in the past, so the rule commands the database to execute the transaction with the lowest security in order to avoid all covert channels. The second line deals with a database that has allowed too many transactions to miss their deadlines; here, the database will execute the transaction with the highest priority. If the database does not have a real-time or security crisis, then the transaction that has been waiting the longest will execute.,

3.3 Tool Implementation

When the pre-processor executes, the description portion of the specification is read and stored in internal data structures. The processor checks for syntax errors and, if no errors are found, it analyzes the specification and finds all potential conflicts between the security and real-time requirements. For completely specified level one descriptions, in order for two transactions to conflict, the following must be true:

1.They must both access the same data item.

2.At least one of the transactions must write to the data item.

3.One transaction must be at a higher security and priority level than the other.

4.The execution times of the transactions must intersect.

Every pair of transactions that satisfy these conditions are reported to the user. Of course, in less detailed descriptions, not all of these rules apply. For example, if the readset or writeset of one of the transactions is left unspecified, then the first two rules do not apply. If the timing information is incomplete for one of the transactions, the last rule does not apply. For level 2 categories, all categories might conflict, so every possible pair of categories is reported to the designer.

The user then goes through an interactive process to create rules that capture the requirement for the databases actions when these conflicts are encountered. For each conflict, the pre-processor advises the user about the implications of violating security with regard to the scheme of partial security described above. For example, in the case of a four level secure database, if a conflict occurs between transactions at the top secret level and the unclassified level, allowing a security violation would force the database into the situation of Figure 1d.

Armed with this information, the user now creates the rules for the database to follow during execution. Rules are created as explained above. The rules for detail level 3 are also inputted now. Note that since level 3 rules do not require any entries

```
(SecViolation% >= 10) ~ executeLowestSecurity,
(TransMiss% > 10) ~ executeHighestPriority,
(otherwise) ~ executeOldestTransaction;
```

Figure 4 - Example of rule for conflicts of three or more transactions.

into the description portion of the specification, a database that contains rules only of level 3 will not use the description analyzer stage of the tool. Once the user has finished providing the rules, the pre-processor verifies that it can determine an action to take in any possible situation. If this is not the case, the tool finds and reports the weakness in the specification. When the specification has no remaining weaknesses, the pre-processor creates an output file that contains the choices of the user. This file will be referenced by the database during system execution.

4 FUNCTIONING IN A DISTRIBUTED ENVIRONMENT

In order to examine the distributed properties of this system further, we put it into the context of the BeeHive system, which is a distributed database system being designed with requirements beyond those of real-time and security. First, we give an overview of the BeeHive system, and then we present how our approach fits into the BeeHive architecture. Note that we use BeeHive as one possible distributed setting to implement our approach. The actual security subsystem of BeeHive can be different from the approach we present in this paper.

4.1 BeeHive Overview

The BeeHive project at the University of Virginia [10] is an attempt to build a global virtual database with real-time, security, fault-tolerance, and quality of service. The BeeHive system is composed of native BeeHive sites, legacy sites ported to BeeHive, and interfaces to legacy systems outside of BeeHive. For the purposes of this paper, we will focus on the native BeeHive sites.

Figure 5 shows the basic design of a native BeeHive site. At the application level, users can submit transactions, analysis programs, general programs, and access audio and video data. For each of these activities the user has a high level specification interface for real-time, QoS, fault tolerance, and security. As transactions (or other programs) access objects, those objects become active and a mapping occurs between the high level requirements specification and the object API via the mapping module. This mapping module is primarily concerned with the interface to object wrappers and with end-to-end issues. A novel aspect of the work is that each object has semantic information (also called reflective information because it is information about the object itself) associated with it that makes it possible to simultaneously satisfy the requirements of time, QoS, fault tolerance, and security in an adaptive manner. For example, the information might include rules or policies and the action to take when the underlying system cannot guarantee the deadline or level of fault tolerance requested. This semantic information also includes code that makes calls to the resource management subsystem to satisfy or negotiate the resource requirements. The resource management subsystem further translates the requirements into resource specific APIs such as the APIs for the OS, the network, the fault tolerance support mechanisms, and the security subsystem.

The resource manager of BeeHive, referred to as the "BeeKeeper", is the central

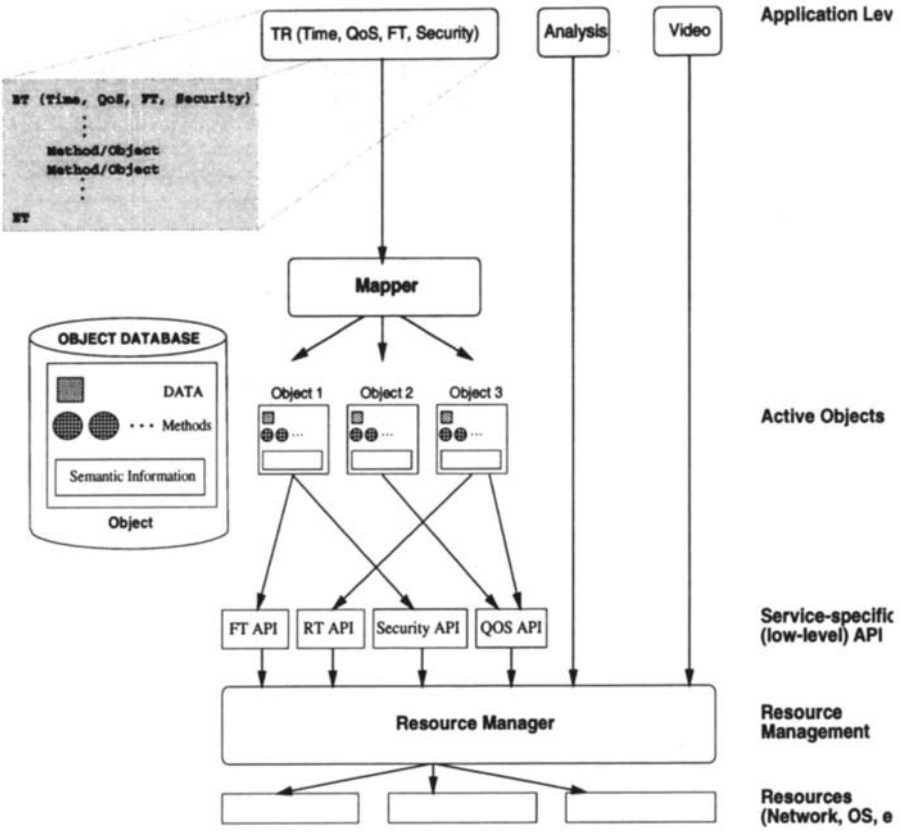

Figure 5 Design of a native BeeHive site

entity of the resource management process. The main function of the BeeKeeper is the mapping of service-specific, possibly qualitative, QoS requirements into actual, quantitative, resource requests. The BeeKeeper contains an Admission Controller, a Resource Manager, and a Resource Allocation Module. The Admission Controller decides whether BeeHive has sufficient resources to support the requirements of a new transaction without compromising the guarantees made to currently active transactions. The Resource Allocation Module is responsible of managing the interface of BeeHive to underlying resource management systems of BeeHive components. The Resource Planner attempts to globally optimize the use of resources. The Admission Controller of the BeeKeeper merely decides whether a new application is admitted or rejected. Obviously, such a binary admission control decision leads to a greedy and globally suboptimal resource allocation. The Resource Planner is a module to enhance the admission control process and to yield globally optimal resource allocations.

4.2 Supporting Real-time and Partial Security in BeeHive

Along with the security and real-time requests of the transactions, the mapper conveys the identity of the transaction and its timestamp to each of the objects that it invokes. The timestamp is necessary to identify this specific instance of the transaction. This information is stored with the other semantic information of the object, and is conveyed to the Resource Manager through the APIs.

The real-time and security APIs allow the objects being used by transactions to convey their requirements to the resource manager. In all cases besides rules dealing with detail level 1, this is all the information about the transaction needed by the resource manager to make decisions when conflicts arise. However, in detail level 1, the resource manager needs to be aware of the identity of the transaction for which the object is executing. This information can be conveyed through either the security or the real-time API.

The admission controller will be a natural choice for the agent that detects conflicts that require the violation of either real-time or security requirements; i.e., a conflict between a high priority, high security transaction and a low-priority, low security transaction. All other conflicts are easy to resolve, and can be handled by the Admission Controller. However, for these special conflicts, the decision is delegated to the Resource Planner; this is the entity that contains and executes the rules created by the database designer.

When a transaction encounters a conflict, the Admission Controller decides (perhaps after consulting the Resource Planner) which of the two transactions is allowed to continue execution and which must be delayed. The delayed transaction is placed in a queue associated with the executing transaction. Once that transaction is finished, the delayed transaction may begin execution. If two transactions are waiting on the queue, the Admission Controller consults the rule covering this conflict and allows one of the transactions to proceed. If more than two transactions are on the queue, and the rules do not support the execution of one of the transactions over all the other transactions, then the Admission Controller must consult the rule that deals with this situation.

The performance monitor fits best into the Resource Allocation Module. This module is closest to the resources that the statistics are representing. The feedback on resource usage that the Resource Allocation Module provides to the Resource Planner is useful for other BeeHive functions, such as for QoS and fault tolerance requirements. For our purposes, the Resource Allocation Module must keep track of the percentage of transactions that have committed a security violation or missed a deadline.

As we have seen, when conflicts occur between nodes, the action taken can depend on the performance at both nodes. Therefore, some sort of cooperation and exchange of statistics must occur between the resource managers of the nodes. In the BeeHive model, the resource managers at different nodes should communicate with each other; this will be necessary not only for our purposes, but also for the resource reservation necessary for QoS guarantees.

At first glance, this scheme seems to locally optimize the database, rather than globally optimizing it. However, when examined more closely, this node-by-node optimization may be preferable to a global optimization. Consider a database with ten nodes. In eight of these nodes, the security requirements have been upheld but the real-time performance has started to degrade. The opposite is true of the remaining two nodes. Now, a conflict occurs between these last two nodes. If the database is globally optimized, the resource managers might decide to violate security to help the overall real-time performance of the system. This decision will have little effect on the real-time performance of the eight nodes whose real-time performance is degrading, and further the security problems on the two nodes where security violations are a problem.

5 CONCLUSIONS

In this paper, we have presented mechanisms to allow the union of security and real-time requirements in distributed database systems. An important part of this union is the definition of partial security. The definition allows security violations in order to improve real-time performance, yet does not entirely compromise the security of the entire database system. However, database designers must be careful with violations between transactions whose security levels differ greatly. If a violation is allowed between transactions, say, at the highest and lowest security levels, no partial security remains in the system at all. In a system with many such conflicts, it may be very difficult to improve on real-time performance. However, it is essential that the system designer can specify how to manage the system security and real-time requirements in a controlled manner in real-world applications.

We have come up with a scheme that allows database designers to create rules at whatever level of detail that they feel is appropriate. These rules can then be analyzed by a tool, which allows designers to create a database and easily make conscious decisions about the partial security of the database. The tool can also automates the process of scanning through the complex dependencies of a database specification to find conflicts. It then informs the user of the consequences of violating security for each conflict.

Currently, we have a tool that can analyze transactions completely specified in detail level 1. This tool parses a database description, analyzes the dependencies and conflicts, and then goes through an interactive process with the user to create rules for all possible conflicts. Our future work includes extending this tool to handle rules and descriptions of levels 2 and 3. We are also developing a simulator to investigate the performance of a database that uses the output of this tool, analyzing the effects of different choices made by the user of the tool.

REFERENCES

[1] D. E. Bell and L. J. LaPadula. "Secure Computer Systems: Unified Exposition

and Multics Interpretation," The Mitre Corp., March 1976.

[2] P. C. Clements, C. L. Heitmeyer, B. G. Labaw, and A. T. Rose. "MT: A Toolset for Specifying and Analyzing Real-Time Systems," Real-Time System Symposium, Lake Buena Vista, FL, December 1990.

[3] CORBA Security, OMG Document no. 95-12-1, December 1995.

[4] Andre N. Fredette and Rance Cleveland. "RTSL: A Language for Real-Time Schedulability Analysis," Real-Time System Symposium, Raleigh-Durham, NC, December 1993.

[5] Binto George and Jayant Haritsa. "Secure Transaction Processing in Firm Real-Time Database Systems," ACM SIGMOD Conference, Tucson, AZ, May 1997.

[6] T. F. Keefe, W. T. Tsai, and J. Srivastava. "Multilevel Secure Database Concurrency Control," In Proceedings of the Sixth International Conference on Data Engineering, pp 337-344, Los Angeles, CA, February 1990.

[7] Butler W. Lampson. "A Note on the Confinement Problem," Communications of the ACM, Vol. 16, No. 10, pp 613-615, October 1973.

[8] Alice H. Muntz and Randall W. Lichota. "A Requirements Specification Method for Adaptive Real-Time Systems," Real-Time Systems Symposium, San Antonio, TX, December 1991.

[9] Ravi S. Sandhu and Edward J. Coyne. "Role-Based Access Control Models," IEEE Computer, vol. 29, no. 2, February 1996.

[10] John A. Stankovic, Sang H. Son, and Jorg Liebeherr. "BeeHive: Global Multimedia Database Support for Dependable, Real-Time Applications," *Computer Science Technical Report*, CS-97-08, University of Virginia, April 1997.

[11] S. H. Son, R. David, and C. Chaney, "Design and Analysis of an Adaptive Policy for Secure Real-Time Locking Protocol," Journal of Information Sciences, to appear.

[12] Douglas A. Stuart. "Implementing a Verifier For Real-Time Systems." Real-Time Systems Symposium, Lake Buena Vista, FL, December 1990.

[13] Bhavani Thuraisingham and William Ford. "Security Constraint Processing in a Multilevel Secure Distributed Database Management System," IEEE Transaction on Knowledge and Data Engineering, Vol. 7, No. 2. April 1995.

[14] Chenxi Wang and Wm A. Wulf, "A Distributed Key Generation Technique". *Computer Science Technical Report,* CS-96-08, University of Virginia, 1996.

[15] William A. Wulf, Chenxi Wang, and Darrell Kienzle. "A New Model of Security for Distributed Systems," Computer Science Department Technical Report, The University of Virginia, April 1996.

[16] IEEE Symposium on Real-Time Technology and Applications, Montreal, Canada, June 1997.

ACKNOWLEDGEMENT

This work was supported in part by National Security Agency and by Office of Naval Research.

BIOGRAPHY

Sang H. Son received the Ph.D. in Computer Science from the University of Maryland, and is currently an Associate Professor in the Department of Computer Science at the University of Virginia. His current research interests include real-time computing, database systems, and information security. He has served on numerous program committees of international conferences in those areas. He is an Associate Editor of IEEE Transactions on Parallel and Distributed Systems, and is serving as the General Chair of IEEE Real-Time Systems Symposium (RTSS'97) and Program Co-Chair of International Workshop on Real-Time Computing Systems and Applications (RTCSA'97). He has served as the Program Chair of RTSS'96 and Program Co-Chair of the International Workshop on Real-Time Database Systems, both held in 1996. He edited the book "Advances in Real-Time Systems," published by Prentice-Hall in 1995, and is a co-editor of "Real-Time Database Systems: Issues and Applications," published by Kluwer Academic Publishers in 1997.

Craig Chaney received the MS degree in Computer Science from the University of Virginia in 1996. He is now with IBM.