SUPPORTING DATA CONSISTENCY IN CONCURRENT PROCESS EXECUTION WITH ASSURANCE POINTS AND INVARIANTS

by

Andrew Courter, B.S.

A Thesis in

COMPUTER SCIENCE

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Committee Members:

Dr. Susan Urban

Dr. Michael Shin

Dr. Susan Mengel

December 2010

ACKNOWLEDGEMENTS

I would like to thank my academic advisor, Dr. Susan Urban for her guidance and continued support throughout this research.

I would also like to thank Le Gao for all of his support. Thanks also to my committee members, Dr. Michael Shin, and Dr. Susan Mengel, for taking the time to review my research. Thanks also go to Mary Shuman from the University of North Carolina, Charolette and a participant in the Texas Tech NSF Research Experience for Undergraduates Site Program for her supporting work with implementation of the Invariant Evaluation Web Service.

This work is dedicated to my parents and everyone who has encouraged me to excel.

*This research has been supported by NSF Grant No. CCF-0820152. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation

Table of Contents

ABSTRACT

This research has developed the concept of invariants for monitoring data in an environment that allows concurrent data accessibility with relaxed isolation. The invariant approach is an extension of the assurance point concept, where an assurance point is a logical and physical checkpoint that is used to store critical data values, to express a post-condition for completed service, and to express a pre-condition for the next service to execute. Invariants provide a stronger way of monitoring constraints and guaranteeing that a condition holds for a specific duration of execution as defined by starting and ending assurance points, using the change notification capabilities of Delta-Enabled Grid Services (DEGS). This thesis outlines the specification of invariants as well as the invariant monitoring system for activating invariants, evaluating and re-evaluating invariant conditions, and deactivating invariants. Algorithms are also presented for the delta analysis agent of the system, which is responsible for filtering data changes from DEGS against the monitored objects of the active invariants. The system is supported by an invariant evaluation web service that uses materialized views for more efficient re-evaluation of invariant conditions. This research includes a performance analysis of the invariant evaluation Web Service, illustrating the benefits of using materialized views. The strength of the invariant technique is that it provides a way to monitor data consistency in an environment where the coordinated locking of data items across multiple service executions is not possible, thus providing better support for reliability and maintenance of user-defined correctness conditions among concurrent processes.

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Web Services and service-oriented computing have challenged the development of distributed applications over the Internet in the past several years. In service-oriented computing, business processes are composed by executing distributed Web Services (Martens, 2005), where services run collaboratively to achieve global goals. Although each Web Service itself is autonomous and self-contained, composing business processes and achieving a correct global solution is still a difficult and sometimes error-prone task, especially in the context of concurrently executing processes that access shared data.

In traditional distributed transaction systems, the two-phase commit (2PC) protocol (Elmasri & Navathe, 2010) has been used to support the ACID properties of atomicity, consistency, isolation, and durability. In service-oriented computing, however, it is generally not feasible to support ACID properties by coordinating the commit time of all services that are part of a global process because of the loosely-coupled, autonomous, and heterogeneous nature of services. Moreover, in traditional transaction processing, the concept of serializability is supported by using locking protocols (Elmasri & Navathe, 2010). In service-oriented computing, however, it is not practical to require constituent services to lock data for the entire duration of a global process. This is especially true for long-running processes, causing processes to execute using a relaxed form of isolation in between service executions. As a result, the correctness of a process might be affected by another concurrently running process if both processes are accessing shared data. Insuring the consistency of data in a service-oriented environment with relaxed isolation is a challenging task.

The solution to this problem is not simple and has been the focus of the Decentralized Data Dependency Analysis for Concurrent Process Execution (D3) Project. The concept of Delta-Enabled Grid Services (DEGS) was developed for monitoring data changes made by a service and using this information about data

changes to analyze data dependencies among concurrent processes (S. D. Urban, Xiao, Y., Blake, L., & Dietrich, S. W., 2009) (Blake, 2005). Building upon this concept, a decentralized system to monitor these changes was made in (Liu, 2009). However, these elements of the overall research focus on recovering processes instead of helping to detect failures inside the processes before they cause problems.

The research topic discussed in this thesis builds on the concepts defined in (Shrestha, 2010), where the concept of an *assurance point (AP)* is defined. An AP is used to support greater flexibility in the recovery process and to also provide a way of specifying user-defined correctness conditions in the form of pre- and post-conditions. As defined in (Shrestha, 2010), an AP is a logical checkpoint created in between the service calls of a process, defining a named point that can be used to store critical data values, to express a post-condition for completed service, and to express a pre-condition for the next service to execute. Rules can be used to define how to react to failed conditions, and recovery actions can refer to named assurance points for forward recovery actions. APs together with conditions and rules can improve constraint checking and recovery procedures in an environment where ACID properties cannot be guaranteed. In some applications, however, stronger condition checking techniques may be needed to ensure data consistency.

This thesis presents an investigation of an extension to APs known as *Invariants*. An invariant is a condition that must be true during process execution between two different APs. An invariant is designed for use in processes where 1) isolation of data changes in between service executions cannot be guaranteed (i.e., critical data items cannot be locked across multiple service executions), and 2) it is critical to monitor constraints for the data items that cannot be locked. The data monitoring functionality provided by the work with DEGS makes it possible to declare and monitor invariant conditions.

Whereas the work with APs defined in (Shrestha, 2010) allows data consistency conditions to be checked at specific points in the execution, invariants provide a stronger way of monitoring constraints and guaranteeing that a condition

holds for a specific duration of execution without the use of locking. Using the invariant technique, a process declares an invariant condition when it reaches a specific AP in the process execution, also declaring an ending AP for monitoring of the invariant condition. When a concurrent process modifies a data item of interest in an invariant condition, the process that activated the invariant is notified by a monitoring system built on top of Delta-Enabled Grid Services. If the invariant condition is violated during the specified execution period, the process can invoke recovery procedures as defined in (Shrestha, 2010). The strength of the invariant technique is that it provides a way to monitor data consistency in an environment where the coordinated locking of data items across multiple service executions is not possible, thus providing better support for reliability and maintenance of user-defined correctness conditions among concurrent processes.

To investigate the invariant concept, this research involved the design of an invariant monitoring system. The system includes an Invariant Agent and a Delta Analysis Agent. The Invariant Agent begins the monitoring process by registering the invariant condition and is a key component in the re-evaluation of the condition. The Delta Analysis Agent filters through delta notifications from the DEGS system and determines if and when an invariant condition must be re-evaluated. The work is supported by a Web Service for evaluating invariants. Since an invariant may need to be evaluated several times between Assurance Points, the Web Service was designed to make use of materialized views for more efficient re-evaluation of invariant conditions. This research includes a performance analysis of the invariant evaluation Web Service, illustrating the benefits of using materialized views.

In the remainder of this thesis, Chapter 2 covers related work on transactional issues for Web Services, and other related research using various methods to avoid strict data locking in distributed and concurrent environments. Chapter 3 then presents background information about research on service composition and recovery with Assurance Points. Chapter 4 introduces the Invariant Monitoring System and also presents motivational examples as well as the overall structure of the system. Chapter

5 elaborates on the functionality of the Invariant Monitoring System and describes a prototype of the system. Chapter 6 presents an evaluation of the Invariant Evaluation Web Service, followed by a summary of the work and future research in Chapter 7.

CHAPTER II

RELATED WORK

Traditional transactions use the properties of atomicity, consistency, isolation and durability (ACID), together with serializability to ensure data consistency among concurrent transactions. ACID properties and serializabitilty, however, are often difficult to achieve in distributed processes found using service composition. This section outlines the challenges for using transactions in a service-oriented environment and provides an overview of solutions that address extended and relaxed transaction models for use with Web Services.

2.1 Transactional Issues for Web Services

Due to the nature of Web Services, traditional ACID properties are difficult to enforce. These properties do not work well because Web Services are loosely-coupled, meaning that they have few interdependencies. Longer durations of transactions, complex interactions between multiple components, and multiple possibilities of failure are all usual actions of Web Services (Papazoglou, 2003). A single transaction manager can have difficulty coordinating transactions between multiple services because of different guidelines between separate businesses. Finding where errors occur or where certain transactions fail can be challenging due to system complexity.

In distributed transaction processing, two-phase commit is often used to support atomicity. Two-phase commit is a type of atomic commitment protocol in which everything or nothing is committed (Elmasri & Navathe, 2010). The two phases of the algorithm are the commit-request phase and the commit phase. In the commit-request phase, the coordinator process attempts to prepare all the transaction's participating processes (participants) to take the necessary steps for either committing or aborting the transaction. Then in the commit phase, each participant votes to either commit or abort. The coordinator decides whether to commit (only if all vote commit)

or abort the transaction, and notifies the participants of the result. The participants then follow with the needed actions (commit or abort) with their transactional resources and their respective portions in the transaction's output. Dealing with the atomicity requirement would require having all Web Services of a process either commit fully or not at all. The autonomous nature of Web Services makes this difficult to achieve. Isolation is also difficult to achieve especially if data must be locked across several Web Service invocations. As a result, concurrent processes cannot be completely isolated from each other, potentially causing problems with data consistency. Instead of conforming to these ACID properties, the research presented in this thesis focuses on extending the relaxation of ACID properties to create an optimistic execution environment.

2.2 Transactional Workflows

Past research with transactional workflows first investigated the need to relax ACID properties for long running workflow activities (Worah & Sheth, 1997). The Saga transaction model was proposed as a base model for long-running activities and defines a chain of transactions as a unit of control (Garcia-Molina & Salem, 1987). The Saga model relaxes the requirement of the entire transaction as an atomic action by releasing a resource before it completes without sacrificing the consistency of the database. Using Sagas, the long running transaction is divided into smaller subtransactions. To accomplish this, every subtransaction requires a compensating transaction to provide a backward recovery process in case of failure. However, the Saga model still had its limitations and other extended transaction models were later developed to overcome them by supporting more general and powerful sets of control flow descriptions and allowing those descriptions to grow and become more complex.

Models similar to the Saga model are called Advanced Transaction Models (ATMs). These models provide functionality to relax ACID properties and have provided solutions to problems in correctness, consistency, and reliability in transaction processing and database management environments. Transactional workflows are an extension to the Saga transaction model and were developed because

extending ATMs to model workflow transactions was seen to be inefficient and exceed the ATMs capabilities. Transactional workflows are activities that involve coordinated execution of multiple related tasks on distributed, heterogeneous, and autonomous information systems and support selective use of transaction properties at individual task and workflow levels (Worah & Sheth, 1997).

A model that has been used to define and study transactional workflow is the ConTracts model (Reuter & Wächter, 1991). In the ConTracts model, a ConTract contains a set of predefined actions called steps and a clearly specified execution plan called a script. The steps can be sequential programs instead of just transactions and a ConTract must be forward-recoverable, meaning the state of the ConTract must be restored after a failure to allow execution to continue. This fundamental functionality is the basis of recovery mechanisms and is an integral part of the Assurance Points discussed later.

Transactional workflows try to provide a relaxed requirement of transactions so that they can be used in a loosely-coupled environment (Reichert & Dadam, 1998). One of the main components that transactional workflows provide is forward recovery, allowing execution to continue as was explained above in the ConTract model. Other transactional workflow examples are the Transaction Specification and Management Environment (TSME) and the METEOR models. The TSME model consists of constituent transactions corresponding to workflow tasks (Georgakopoulos, Hornick, Krychniak, & Manola, 1994). Each workflow has an execution structure that is defined by an ATM which defines the correctness criteria for the workflow. Because TSME supports various ATMs it can cover a larger variety of workflow processes and support correctness and reliability in those processes. The METEOR workflow model is a collection of many different workflow models (W. Jin, Rusinkiewicz, Ness, & Sheth, 1993). There are many tasks in a METEOR workflow and each task can be heterogeneous. The execution behaviors of tasks are defined using task structures and can be composed of multiple tasks. This grouping of tasks helps to define the execution and define the workflow execution.

These are just a few examples of the different approaches to achieving the goal of defining application specific and user-defined correctness, reliability and functional requirements within workflow executions. However, transactional workflows do not support all ACID transactions and are more prone to problems when dealing with failures in multiple databases. Therefore, other techniques must be considered to compose a more reliable system.

2.3 Promises

One of the recent research projects addressing a non-standard approach to transactional issues for Web Services is the Promises project (Greenfield, Fekete, Jang, & Kuo, 2003). A goal of the Promises project is to make sure that certain values are not overwritten or changed by concurrently executing Web Services. This can be realized by making promises to hold information instead of locking data and creating problems with long runtimes. A promise is an agreement between a client application and a service or 'promise maker'. When a promise request is accepted by the 'promise maker', the promise value guarantees that some set of conditions will be maintained over a set of resources for a specified period of time. The approach discussed in (Greenfield, et al., 2003) defines the 'promise maker' to be a Promise Manager that records promises and the main functionality of the Promise Maker is to handle promise making, check on resource availability, and ensure that promises are not violated, instead of having services handle that information. Client applications can send the promise manager what resources they need in order to complete successfully and express them as a set of predicates. These predicates are simply Boolean expressions over the resources. There are no limitations on the form of the predicates or on the way promise managers implement the predicates to guarantee that they remain true. This allows the promise managers and resource managers more flexibility and the freedom to use the best isolation mechanism for that specific type of data. The request for a promise will be examined by the promise manager and it will either grant or reject the request. Once a promise request is granted, the client

application is isolated from the effects of concurrent execution and can complete successfully.

Although the client is granted a promise request, there is a period of time agreed upon for which a promise will be valid as part of the process and requests can expire. If a promise request expires the promise manager will return a 'promiseexpired' error to the client. The main advantage to using promises is to allow applications to be "promised" resources and then continue processing without having to recheck that those resources are still available. Traditional locking mechanisms are very strong and guarantee that no one will change or modify the data that is being locked. However, this stops all other Web Services from accessing it and creates a wait. Promises are a weaker form of locking but are more specific and allow other Web Services to access the data so that any wait is avoided. The predicates that are contained within a promise specify the resources a client needs and allows other promises covering the same resources to be granted at the same time as long as they do not conflict with any already granted promises. Promises are general requests for resources and must be general so that a single resource can be promised to the client application.

One method that has been used to implement promises is the concept of 'soft locks'. This method uses a field in the database record to show whether an item has been allocated already for a client. The record is not completely locked from other applications but instead applications read this field when looking for available resources and ignore any record that has been already allocated. Promises separate the isolation technique from the logic so that programmers do not have to be concerned about concurrent processing. This approach provides flexibility and reliability for concurrent Web Service execution. However, implementing this system would be difficult because of the centralized approach of the Promises Manager. If every Web Service had to request all resources from the Promise Manager then there would be a large amount of waiting for clients.

Promises are similar to the Invariant Monitoring System in that the ACID properties are relaxed to provide a more optimistic environment where hard locks are not used. However, Promises still use 'soft locks' and so not all concurrent processes can see all the data at any certain time. The Invariant Monitoring System never hides or locks information and allows any process to modify data. Instead, the Invariant Monitoring System monitors data changes, re-evaluates constraint conditions affected by data changes, and invokes recovery actions if necessary.

2.4 Reservation-Based Techniques

In addition to promises, there is another similar method to temporarily lock data in a concurrent environment. This is called the reservation-based approach. This approach 'reserves' resources that meet the criteria of what the Web Service has requested. The reservation-based approach has many advantages over two phase commit or the optimistic two phase commit protocols. First, database records are locked, both physically and logically, only for the duration of local transactions. Second, only the required amount of a resource is reserved, rather than locking the database record or the entire resource for an extended period of time. And third, the reservation-based protocol provides better response times for the clients and better throughput and completion times at the server (Zhao, Moser, & Melliar-Smith, 2009).

The reservation-based protocol is implemented on both the client and server and each task is executed as two steps. In the first step, the resource involved in the task is reserved in a single traditional transaction. In the second step, the reservation is either confirmed or cancelled according to the business rules. Each of these steps is executed as a separate traditional short-running transaction, as in the sagas strategy (Zhao, et al., 2009). Because the resource that the application requests is reserved in the first step, the application has the choice and freedom of either continuing processing or backtracking to make sure everything is as expected. When other applications ask the server for resources they only see the resources that have not already been reserved by other applications. However, the application that makes the reservation is not allowed to actually use the reserved resource until it has changed the

status of the resource to "committed". In traditional locking any of the participants can decide to abort and rollback the entire transaction, but in the reservation-based protocol only the coordinator can decide this. One of the main advantages of using the reservation protocol over using traditional locks is that if a resource is reserved and another transaction wants to access it, the transaction can acquire a lock on the resource and the requesting application can immediately be informed of the state of the resource instead of having to wait. To restrict the reservation time, "fees" can be charged by the resource provider to deter applications from making reservations for extended periods of time. Using reservations is another way to avoid locking resources from other applications and is a viable alternative to the traditional locking while still maintaining the ACID properties.

The reservation-based technique provides a way to avoid hard locks on shared data and provides a notification to waiting service calls but still locks data for a period of time. As previously discussed, the Invariant Monitoring System provides an optimistic execution environment by avoiding global locks and allowing all processes to modify data at any time.

2.5 Transactional Attitudes

Transactional Attitudes are used as a framework to handle the transactional reliability issue in Web Services. Transactional Attitudes establish a separation of transactional properties from other aspects of a service description. In (Mikalsen, Tai, & Rouvellou, 2002), the WSTx framework uses transactional attitudes which makes Web Service providers declare their individual transactional capabilities and semantics, and Web Service clients declare their transactional requirements. Using Provider Transactional Attitudes (PTAs) and Client Transactional Attitudes (CTAs), transactional attitudes can be defined so that Web Services can remain autonomous without having to worry about reliability. PTAs are a mechanism for Web Service providers to explicitly describe their specific transactional behavior. PTAs use WSDL extension elements to annotate Web Service provider interfaces for web transactions, according to well defined transactional patterns. The PTA also includes the name of

an abstract transactional pattern with any port-specific information needed to make the pattern concrete.

There are three types of PTAs that are defined by the WSTx framework to handle different Web Service behavior's pending-commit, group pending-commit, and commit-compensate. The pending-commit is described as a transactional port of a single Web Service where a single forward operation invocation can be held in a pending state. The Web Service waits until another event occurs and then either accepts or rejects the effect of the operation. For the group-pending-commit a single Web Service waits for the effects of a group of forward operation invocations. Just like the pending-commit, the group-pending-commit either accepts or rejects the effects of the operations at the end. The commit-compensate is when a Web Service accepts the effect of a single forward operation invocation but can reverse the operation later using an associated compensation operation. CTAs are described in terms of well-defined WSDL port types and outcome acceptance criteria (Mikalsen, Tai, & Rouvellou, March 2002). A client's transactional attitude is established by its use of a particular WSDL port type to manage. The client executes one or more named actions within the scope of a web transaction, where each action represents a provider transaction that executes within the context of the larger web transaction. The WSTx framework uses both PTAs and CTAs to define attitudes for Web Services transaction compositions and provide reliability during execution.

PTAs and CTAs could be combined with the Invariant Monitoring System to define their transactional behavior and provide greater reliability. Combining the two would also allow Web Services to stay autonomous and not have to worry about locking data in a concurrent environment.

2.6 Tentative Holding

Tentative holding is allowing a tentative, non-blocking hold or reservation to be requested in data by a business resource (Limthanmaphon & Zhang, 2004). When the owners of the resource receive these requests they grant non-blocking reservations on their services, preserving control of their resources while allowing many potential

clients to place their requests. Placing these holds instead of locking the resources minimizes the need for clients to cancel transactions. To accomplish this, there are several states that are used to determine the progress of a hold. These states include responding to an initial Request, In Process to show that the request has been received and is being processed, Active to show a hold is active, and Inactive to show that a hold is no longer valid. The initial requests are sent to the client coordinator, who checks the status of any previously granted holds and provides the client the ability to ask about other queries, cancel an existing hold, query logged holds, and modify an existing hold. The resource coordinator is responsible for verifying the expiration times of holds and grants holds to requesting client coordinators. Additionally, the resource coordinator notifies the client coordinator if a resource becomes unavailable. This technique is similar to the Promises and Reservation-Based techniques because resources are held temporarily in a non-blocking mechanism. The client coordinator and resource coordinator act similarly to the Promise Manager that was previously described but use states to coordinate the holding or releasing of records and are separate instead of one entity. This technique would be difficult to implement and would also have similar difficulties when trying to implement this in a large scale system.

This technique is similar to Promises and Reservation-Based techniques, except that tentative holding uses more states to coordinate the state of the locked data. In contrast, the Invariant Monitoring System monitors data changes and re-evaluates user-defined constraints as needed.

2.7 Monitoring Extensions to BPEL

Due to the many inadequacies of BPEL, several research projects have incorporated monitoring techniques inside the BPEL processes to make sure everything runs smoothly. The work in (Baresi, Ghezzi, & Guinea, 2004; Baresi & Guinea, 2005) uses monitoring rules woven inside the WS-BPEL process to dynamically control the execution during runtime. The monitoring rules are annotated in the source code using assertion languages, such as Anna (Annotated Ada)

(Luckham, 1990) and JML (Java Modeling Language)(Baker, Leavens, & Ruby, 2005). User-defined constraints are blended with the WS-BPEL process at deployment time and are defined externally to allow separation of the different functionalities. The constraints are defined using WS-CoL (Web Service Constraint Language), a special purpose language that borrows its roots from JML but has additional constructs to gather data from external sources. Because the WS-BPEL and the monitoring rules are separately defined, the BPEL preprocessor combines the two together. Pre and post conditions are added between service invocations so that constraints can be checked and a BPEL exception is made if one is violated. External Web Service invocations are routed to the Monitoring Manager which makes the invocation and then relays back to the WS-BPEL process if the invocation was successful or not. Invariants between Web Services can also be monitored and are handled similarly by making a call to the Monitoring Manager. The Monitoring Manager is the key component of this work and instead of calling a Web Service the information is passed to the Monitoring Manager. Upon receiving this information the Monitoring Manager checks the pre-conditions if there are any, then invokes the Web Service if there is no violation in the pre-conditions. After invoking the Web Service the Monitoring Manager then checks any post-conditions and if there is no violation, returns the information from the Web Service back to the WS-BPEL process.

This monitoring extension is similar to the Assurance Point in that pre and post conditions are checked at specific points in a business process. However, recovery techniques are not incorporated into the monitoring extension and additional checks are not present in case other conditions must be met. Additionally, the Invariant Monitoring System described in this thesis monitors data conditions over a certain period of time to provide a more reliable *and* optimistic environment.

2.8 Aspect-Oriented Workflows

Current workflow languages do not allow techniques to properly modularize concerns that cut across process boundaries such as security and data validation. Therefore, the process code is spread across several workflow process specifications

and mixed in with the process code addressing other concerns. This makes understanding and having to change process specifications difficult. Aspect-oriented process logic software development has been investigated to address these modularity problems by introducing new programmatic constructs. The work presented in (Charfi & Mezini, 2006), proposes using aspect-oriented concepts to address the modularity issues in workflow languages. A prototype extension to BPEL using aspect-oriented workflow concepts (AO4BPEL) (Charfi & Mezini, 2007) was developed to validate their work.

Aspect-oriented programming (AOP) (Irwin et al., 1997) introduces a new concept of modularizing several different concerns called an aspect. A well known aspect-oriented programming language AspectJ (Kiczales et al., 2001), uses three key concepts: join points, pointcuts, and advice, to support the aspect portion of the aspect-oriented workflows and AO4BPEL described in (Charfi & Mezini, 2006). Join points are points in the execution of a program similar to check points where method calls, constructor calls, and other functions can be used to check conditions. A pointcut is a way to identify whether join points are related to each other. Advice is the code executed when a join point in the set identified by a pointcut is reached. This code may be executed before, after, or instead of the join point. An aspect contains several pointcut and advice declarations and is a module that encapsulates a non-functional crosscutting concern. Using the aspect, the workflow can separate the workflow logic and the non-functional concerns into the process module and aspect modules, respectively. This methodology helps to separate the different elements in a process workflow and make it easier to modify and reuse the same workflow. However, modularizing a workflow is time-consuming and requires many tools for integrating the aspects with the workflow processes (weaving).

Join points are similar to the assurance points used in this thesis, where assurance points focus on constraint checking to support dynamic recovery techniques. The Invariant Monitoring System, however, has the capability to monitor data conditions in between the occurrence of Assurance Points.

An extension to Aspect Oriented Programming to enable monitoring of shared data between composite Web Services is proposed in (Wu, Wei, Ye, Zhong, & Huang, 2010). This research extends the original design by adding monitoring parameters, local variable declaration, Historycut to specify the desired behavior property, a list of symbol pointcuts, and advice to be executed when there is a constraint violation. The Historycut and Advice properties are explained in more detail in (Wu, Wei, & Huang, 2008) and are built upon in the research extension. The list of symbol pointcuts "defines the lexeme of the declared behavior constraints, capturing individual events in an execution trace" (Wu, et al., 2010).

Using these extra features in the Aspect Oriented Programming, a monitoring instance is created for each set of parameters that need to be monitored. Because thousands of monitoring instances could possibly be required to monitor parameters of each process instance, the goal is to make the monitoring instance as efficient as possible. To handle the creation of monitoring instances, a Monitor Manager is used to save the start events of each deployed aspect. When receiving a trace monitor creation request from the aspect manager, the monitor manager will check whether the event is a start event. If so, it will create a new trace monitor and return to the aspect manager. Otherwise, it will discard this request.

After creating a trace monitor to monitor a set of parameter bindings, there needs to be a way to find the trace monitor again. (Wu, et al., 2010) proposes using an inverted list structure based on the work in (Moffat & Zobel, 2006), which has widely been used in the area of large scale search and information retrieval for its impressive scalability properties. The inverted list is shown to perform with little overhead in the prototype implementation. However, it is difficult to understand how to set up this system and not clear if it is feasible for several thousand or even hundreds of thousands of process instances.

2.9 Summary

This related work section has described several past and ongoing research projects that support non-blocking techniques, relaxing the ACID properties to provide

an optimistic approach to data consistency. However, using these approaches, specific conditions cannot be monitored during a specific execution period in the process. The focus of the research presented in this thesis is to present a system to extend the Assurance Point architecture to allow monitoring of critical data conditions at specific execution periods in a process. Any violation of this condition will invoke corresponding actions. Providing this capability will allow data inconsistencies to be more quickly recognized and allow a more dynamic approach to process recovery.

CHAPTER III

BACKGROUND RESEARCH FOR THE USE OF INVARIANTS

The research presented in this thesis is part of the ongoing Decentralized Data Dependency (D3) Analysis Project at Texas Tech University which addresses issues with data dependency analysis and failure recovery in a service-oriented environment using a decentralized (Liu, 2009) and rule-based approach. To present research with the Invariant Monitoring System, it is first necessary to provide background on Delta-Enabled Grid Services, or DEGS, and Assurance Points, which are used to realize the functionality of the Invariant Monitoring System. DEGS support the ability to invoke constraint checking actions autonomously after a change in the source database. Assurance Points provide the framework inside of the business process to activate invariant conditions and to define the time frame for condition monitoring.

3.1 Delta-Enabled Grid Services

The Delta-Enabled Grid Service is a Grid Service that has been enhanced with an interface that stores the incremental data changes, or *deltas*, that are associated with service execution in the context of globally executing processes. The DEGS uses the OGSA-DAI Grid Data Service for database interaction. The database captures deltas using capabilities provided by most commercial database systems. The DEGS functionality was first used to determine what processes were dependant on each other based on delta values. As described in (S. D. Urban, Xiao, Y., Blake, L., & Dietrich, S. W., 2009), each execution site would send all deltas related to the source database to a central Delta Repository. If a process failure occurred, then the Delta Repository would be used to determine what processes changed the data and notify them to recover to allow the preservation of consistent data between multiple sites (S. D. Urban, Xiao, Y., Blake, L., & Dietrich, S. W., 2009).

The DEGS functionality is demonstrated in Figure 1. A DEGS uses Oracle Streams and triggers inside of a source database to allow automatic notification of new

deltas in the delta repository. Oracle Streams is an Oracle tool that can be used to forward information from the database log file to a separate database or table. Triggers are database rules that can be used to monitor and react to data changes. When a change to the source database is made by a Grid service, then the delta is automatically created by Oracle Streams or triggers and inserted into the delta repository. A Grid service was used to modify the source database so that information from the external process can be related to the data changed in the source database. The Delta Repository contains the name of each table in the source database and has a separate table for inserts, deletes, and updates to that source database table. This allows inserts, deletes, and updates to be easily recognized and information about each type of change to be kept separate. Additionally, a table mapping each delta to information about the Grid service that made the change is kept.

After creating the delta, a Java Stored Procedure deployed in the source database is automatically called to notify a listening Grid service that there are new deltas in the table that was just modified. The listening Grid service then looks for new deltas in the forwarded table name, looking in each of the insert, delete and update delta repository tables. These deltas are then compiled into an XML format and relayed to any other systems that use the delta for recovery or dependency determination. The overall functionality of DEGS is modeled in Figure 1 where the XML format of the deltas is forwarded to the DeltaGrid.

To make use of the DEGS capabilities in the Invariant Monitoring System, changes had to be made. In particular, to monitor user-defined data constraints, all columns or attributes of a table would need to be returned in a delta instead of just one or a few. Additionally, the DEGS forwarded all received deltas to delta event process to handle other process recovery actions. Since the Invariant Monitoring System only needs deltas related to the invariants that are currently being monitored, the DEGS was modified to forward deltas to extract the deltas for condition constraint checking.

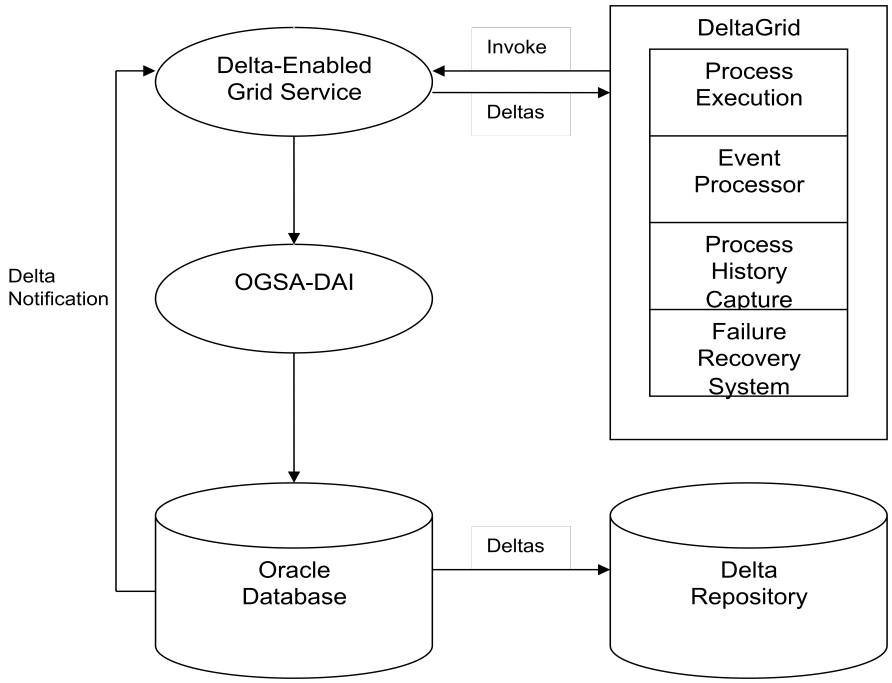

Figure 1. Delta-Enabled Grid Service (Blake, 2005)

3.2 Service Composition and Recovery with Assurance Points

The work in (S. D. Urban, Xiao, Y., Blake, L., & Dietrich, S. W., 2009) presents service composition models that provide the foundation for the D3 project on which this thesis is based. The service composition model did not originally provide a way to handle constraints and respond to exceptions that might arise during the normal execution of a process (Shrestha, 2010). Assurance Points (APs) with different types of rules provide a more complete notion of user-defined correctness in the context of exception handling. An AP is a logical and physical checkpoint for storing data and using rules to check pre and post conditions at critical points in the execution of a process. During normal execution, APs invoke integration rules that check pre-conditions, post-conditions, and other application conditions. The action of an integration rule is used to invoke recovery procedures or to initiate alternative execution paths. An AP is used in forward execution for validation of critical correctness conditions, where post-conditions are used to validate completed services and pre-conditions are used to validate necessary conditions prior to the next service

execution. Post-conditions and pre-conditions are expressed in the AP model extensions as integration rules (IR) based on work with using rules to interconnect software components as originally defined (S. D. Urban et al., 2001). When a specific AP is reached during execution, the condition of the IR is checked. The action of the AP is then based on the evaluation of the IRs inside the AP. An AP can also be used for backward recovery.

Three different forms of backward recovery are introduced in (Shrestha, 2010), with the different forms supporting either full backward recovery or a combination of backward and forward recovery. APRetry is used when the running process needs to be backward recovered to a previously-executed AP. After backward recovery through the use of compensation to a specified AP, the pre-condition defined in the AP is rechecked. By default, the first AP reached during backward recovery will be the target AP, but a specific AP can be specified. APRollback is used when the overall process has some severe errors and must be recovered back to the beginning of the process. APCascadedContingency is a hierarchical backward recovery that continues to compensate backwards, checking each AP that is encountered for a possible contingent procedure that can be used to correct an execution error.

Given that concurrent processes do not execute as traditional transactions in service-oriented environments, inserting APs at critical points in a process is important for checking consistency constraints and potentially reducing the risk of failure or inconsistent data. The most basic use of an AP together with integration rules is shown in Figure 2. Figure 2 shows a process with three composite groups and an AP between each composite group. The shaded box on the right shows the functionality of an AP using AP2 as an example. The post-condition integration rule, the pre-condition integration rule, and any conditional rules are checked sequentially when AP2 is reached. If the pre-condition (pre-IR), or the post-condition (post-IR), is violated then one of the Recovery Actions is invoked. However, if both the pre and post conditions are not violated then the AP will use the condition-IR and invoke the Conditional Operation.

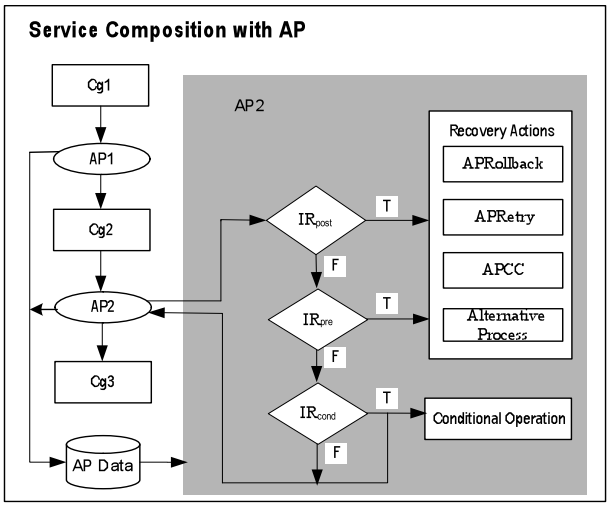

Figure 2. Basic Use of Assurance Points and Integration Rules (Shrestha, 2010)

The Invariant Monitoring System extends the functionality of Assurance Points by adding an additional *inv* rule. This new rule allows the definition of an invariant that can be monitored over several APs. IRs for pre and post conditions can check conditions at certain points in the business process but cannot make sure that a condition holds for a specified period of time. The Invariant Monitoring System provides the capability to monitor conditions between APs and provide a more secure execution environment without having to lock data from other processes using the new rule and other functionality.

CHAPTER IV

OVERVIEW OF THE INVARIANT MONITORING SYSTEM

The research described in this thesis is an extension of the Assurance Point concept described in (Shrestha, 2010). This chapter begins in Section 4.1, with an overview of the functionality and design of the Invariant Monitoring System. Section 4.2 then discusses the format used to define rules inside of the APs. Next, Section 4.3 gives the new rule structure for extending the AP. Two examples are presented in Sections 4.4 and 4.5 to show how the Invariant Monitoring System can be used and how to define the rules for invariants. Lastly, Section 4.6 gives a summary of the chapter.

4.1 The Invariant Monitoring System

Building on top of the research described in (Shrestha, 2010), the Invariant Monitoring System allows process designers to define a constraint over specific data items as well as the time period to monitor this condition. These conditions are defined similarly to the ECA rule structure that is used to define integration rules. Inside each AP, one or many Invariants can be defined to allow for constraint monitoring.

Figure 3 presents the basic use of the Invariant Monitoring System. As an example, assume the condition to be monitored is Tuple1.A + Tuple2.B > 10 by Process 1 where the initial values for Tuple1.A and Tuple2.B are shown in the delta repository of Figure 3. Attribute A from Tuple 1 of Table 1 is 10 and attribute B from Tuple 2 from Table 1 is 12, so the condition is initially satisfied. The numbers in the figure indicate the execution sequence, where T_i through T_n are timestamps for operations in Process 1 and Process 2. At timestamp T_i, the Invariant is activated, meaning the starting AP has been reached and so the Invariant condition, Monitored Objects, and activation state have been sent to the Invariant Agent in step 1. After receiving this information, the Invariant Agent checks the Invariant condition and if

there are no violations, stores the Invariant and Monitored Objects into a local database. It is assumed that an outside Web Service is called when checking and rechecking the Invariant condition.

After activating and registering the invariant condition, step 2 shows the Monitored Objects being passed from the Invariant Agent, to the Delta Analysis (DA) Agent so that the DA Agent knows what to look for in the delta objects. The DA Agent receives the new Monitored Objects and stores them in a container to look through upon receiving a new delta notification from the Delta Repository. At T_j, the Tuple1.A is changed to 5 by Process 2 in step 3, where the change is captured and propagated to the delta repository in DEGS (Steps 4-6). This propagation is done automatically using Oracle Streams as was described earlier in this thesis. Next, the DA Agent is notified that Tuple1.A is changed and as a result, the DA Agent recognizes that deltas related to a Monitored Object have been found and so the Monitored Objects are forwarded to the Invariant Agent, which acquires the related Invariant condition and then rechecks the Invariant condition (Steps 7-8). In this check, Tuple1.A is 5 and Tuple2.B is 12, so there is no violation because their sum is greater than 10. Later, Tuple2.B is changed at T_k, the DA Agent again forwards the related Monitored Objects to the Invariant Agent and the Invariant condition is rechecked again (Steps 9-14), but this time the condition is violated because Tuple1.A is 5 and Tuple2.B is 2, which is less than 10. Step 15 indicates that notification will be sent to Process 1 to indicate violation of the Invariant condition, where Process 1 can then invoke recovery procedures.

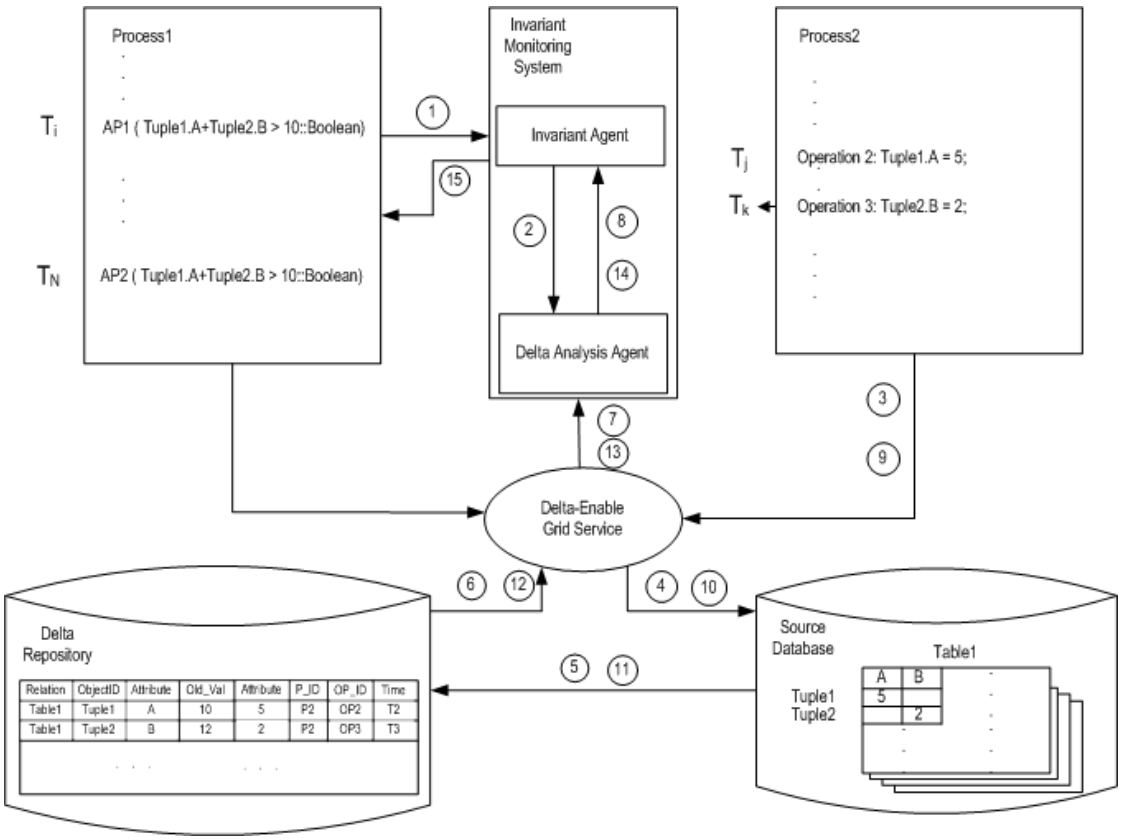

Figure 3. Invariant Monitoring System

4.2 Invariant Specification

The previous work on Assurance Points uses Integration rules in the Event-Condition-Action (ECA) specification to define the different types of Integration rules. These ECA rules are based on previous work with using integration rules to interconnect software components (Y. Jin, 2004; S. D. Urban, et al., 2001). Figure 4 gives an example from the original Integration Rule Structure of APs. The CREATE RULE defines the rule name and what type of rule is being defined. The EVENT clause expresses what Assurance Point this rule is related to and what parameters, if any, are required for the Assurance Point. Next, the CONDITION specifies the condition to check to determine whether the rule is violated or not. If a rule is violated then the ACTION is used to specify what is to be done. If an error occurs a second time, the on RETRY action is executed.

CREATE RULE	ruleName::{pre | post | cond}
EVENT	apId(apParameters)
CONDITION	rule condition specification
ACTION	action 1
[ON RETRY	action 2]

Figure 4. Integration Rule Structure (Shrestha, 2010)

Figure 5 shows the revised rule structure to accommodate the expression of invariants. The CREATE RULE statement contains the Invariant_identifier and the inv invariant rule tag. The Invariant_identifier is used to differentiate between invariants. The EVENT statement contains the startAP as well as the endAP and any parameters needed for the rule condition specification. The startAP and endAP are used to activate and deactivate the invariants during execution in the AP environment. In the CONDITION section of the ECA structure the rule condition specification is expressed as NOT EXISTS(select * from …), where the select statement returns the tuples that satisfy the invariant condition. If the select statement returns tuples then NOT EXISTS evaluates to false and no recovery actions are triggered. However, if the SQL condition returns no tuples, then the NOT EXISTS will return true, indicating that the invariant condition is not satisfied. In this case, the process is notified and the recovery procedure in the ACTION is invoked.

CREATE RULE:	Invariant_identifier::inv
EVENT:	startAP (endAP, Parameter1, Parameter2….)
CONDITION:	rule condition specification
ACTION:	recovery procedures
[ON RETRY]:	additional/alternative recovery procedures

Figure 5. Structure of an Invariant

4.3 Hotel Room Monitoring Example

An example, a subprocess is modeled in Figure 6, where the HotelRoomMonitoring invariant is defined between AP1 and AP2. The process represents a travel planning process, where the process is scoping out available hotel

and airline options before finalizing the plans. Figure 6 shows an invariant that checks a specific hotel for the availability of a seaside room that is less that a specified price, where the hotel and price are passed as parameters from the BeginTravelPlanning AP. Expression of the AP allows the process to continue checking the availability of other travel options, such as airline reservations, but to be notified if the room availability changes. The condition is expressed as an SQL query, preceded with the not exists clause. Therefore, according to the SQL condition defined, if there is not a room that meets the criteria, then the select condition will return no tuples, making the not exists clause true, which triggers recovery action 1. If the query returns tuples that satisfy the SQL condition, then the process continues and the status of the SQL query is monitored using the DEGS capability and the invariant monitoring system. If the process reaches the ReadyToBook AP and the desired room type and price are still available, then the process continues past the ReadyToBook AP, making the appropriate reservations after deactivating the HotelRoomMonitoring invariant. If at anytime between the BeginTravelPlanning AP and the ReadyToBook AP the room is no longer available, the invariant monitoring system will notify the process, which will execute one of the recovery actions.

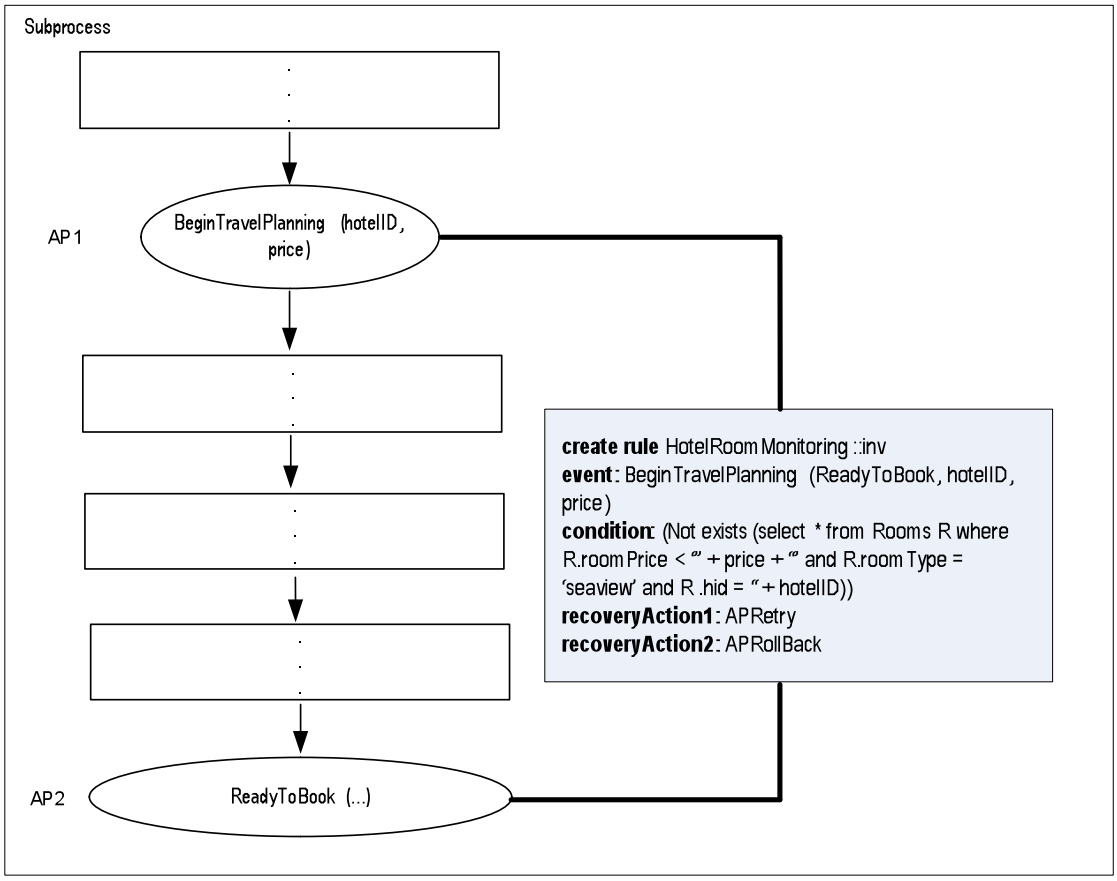

Figure 6. Hotel Room Monitoring

4.4 Bank Loan Application Monitoring Example

As another example consider the subprocess in Figure 7, where the LoanAmountMonitoring invariant is defined between AP1 and AP2. The process represents a loan approval process, where the process is creating a loan application for a customer at a bank that already has an account at that bank. Figure 7 shows an invariant that checks to make sure the loan applicant has a tenth of the requested loan amount in their account, where the amount and customerid are passed as parameters from the LoanAppCreation AP. The condition is expressed as an SQL query, preceded with the not exists clause. Therefore, according to the SQL condition defined, if there is not an account balance that meets the criteria, then the select condition will return no

tuples, making the not exists clause true, which triggers recovery action 1. If the query returns tuples that satisfy the SQL condition, then the process continues and the status of the SQL query is monitored using the DEGS capability and the invariant monitoring system. If the process reaches the LoanCompletion AP and the applicant's account balance still meets the necessary criteria, then the process continues past the LoanCompletion AP, completing the loan application after deactivating the LoanAmountMonitoring invariant. If at anytime between the LoanAppCreation AP and the LoanCompletion AP, the applicant's account balance falls below the necessary criteria, the invariant monitoring system will notify the process, which will execute one of the recovery actions.

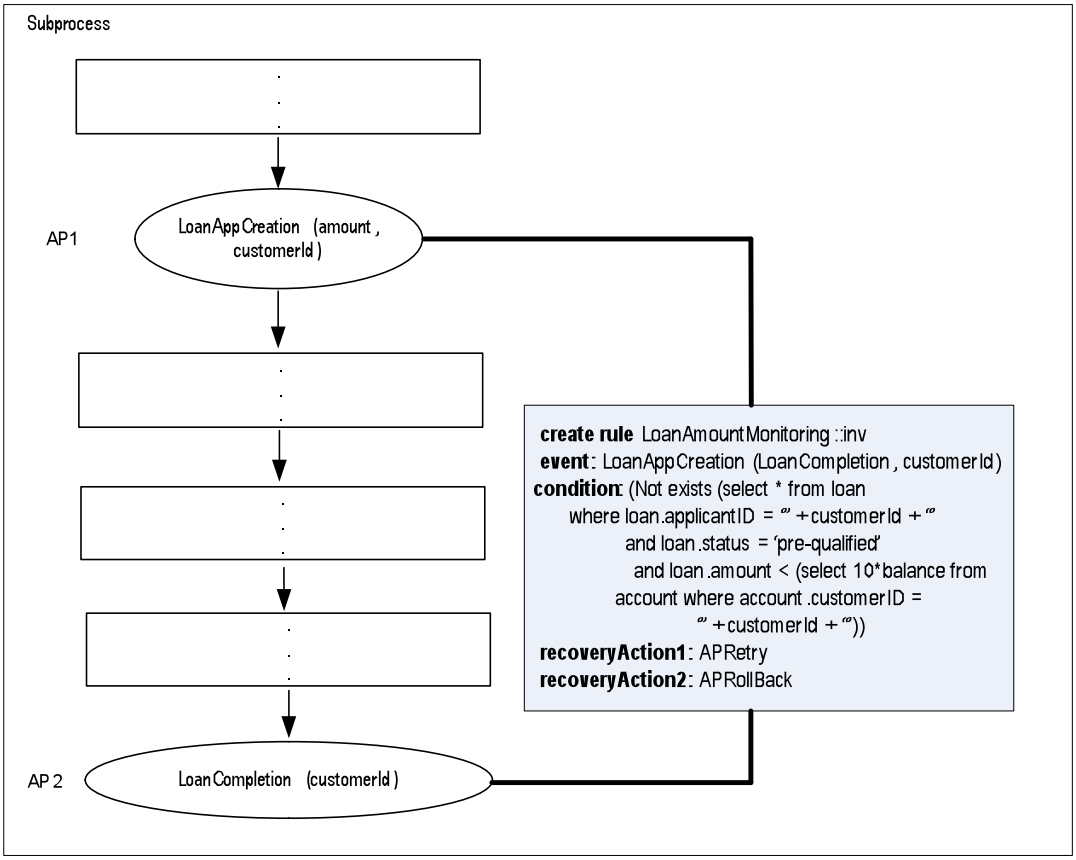

Figure 7. Loan Amount Monitoring

4.5 Summary

These examples illustrate the basic functionality of invariants and how they can be used with assurance points to monitor critical data. In the next chapter, a prototype of the Invariant Monitoring System is presented.

CHAPTER V

A PROTOTYPE OF THE INVARIANT MONITORING SYSTEM

This research has prototyped an execution environment to model the capability of monitoring critical data conditions between Assurance Points.

In this chapter, Section 5.1 provides a more detailed description of monitored objects. Section 5.2 then describes the representation of invariants. Section 5.3 describes the process of invariant registration, while Section 5.4 describes the invariant evaluation Web Service. Section 5.5 presents relevant extensions to Delta-Enabled Grid Services. The filtering process of the Delta Analysis Agent is then outlined in Section 5.6.

5.1 Monitored Objects

As previously shown in Figure 5, the structure of an Invariant rule is defined in the ECA rule format similarly to the other rule types. This rule can be parsed and processed to extract the SQL condition and the Monitored Objects from the Invariant rule definition. The Monitored Objects identify the attributes of interest in the deltas from the DEGS System. For example, if the loan processing example is monitoring data changes from a customer with "customerid = 456" then the monitoring process would ignore a withdrawal or deposit made by a customer with "customerid = 654".

Monitored Objects are acquired from the SQL condition by extracting the table names together with the attributes and relevant conditions. Changes to these extracted Monitored Objects can affect the result of the query. The Invariant Monitoring System may need to re-evaluate the SQL condition when it detects a change in the Monitored Objects.

As an example, consider the SQL query from Figure 6. The table in this query is the `Rooms` table. The monitored objects are built from the table name, attribute name, relation, and set value of the conditions in the query. There are three conditions in the `where` clause of the SQL query and so there are three monitored objects. The

conditions to store as monitored objects include "roomPrice < price", "roomType = 'seaview'", and "hid = hotelID". For the first condition, the table name is Rooms, the attribute is roomPrice, the relation is "<", and the set value is price. For the second condition, the table name is Rooms, the attribute is roomType, the relation is "=", and the set value is 'seaview'. And for the last condition, the table name is Rooms, the attribute is hid, the relation is "=", and the set value is hotelID.

As another example, consider the SQL query from Figure 7. The two tables in this query are the Loan table and the Account table. There are three conditions in the where clause of the base SQL query and so there are three monitored objects from this table. The conditions to store as monitored objects include "applicantId = customerId", "status = 'pre-qualified'", and "amount < (select ...)". To simplify the extra select query, "amount < (select ...)" is converted into "amount < calc" since multiple tables cannot be analyzed during the delta filtering. Therefore, the calc value is used to signify that this is a calculated value and must be re-evaluated. So for the first condition, the table name is Loan, the attribute is applicantId, the relation is "=", and the set value is customerId. Note that customerId is a parameterized value that is inserted by the parameters of the AP. For the second condition, the table name is Loan, the attribute is status, the relation is "=", and the set value is 'pre-qualified'. And for the last condition, the table name is Loan, the attribute is amount, the relation is "<", and the set value is calc.

The second table, Account, has one condition in the where clause, "customerId = customerId". The table name of this monitored object is Account, the attribute is customerId, the relation is "=", and the set value is customerId, which will be instantiated with a parameterized value. This query also illustrates a relevant monitored object in the select clause. The last monitored object is on the balance of the Account table. Balance is set as a calculated value because if this attribute changes then it will change the result of the select and could violate the invariant condition. So for this invariant there are a total of five monitored objects over two separate tables.

To acquire the monitored objects, the select queries used for the invariant conditions must be analyzed to extract attributes and the conditions on those attributes, as well as any calculated values. This process allows the Invariant Monitoring System to compare information in the monitored objects to incoming delta notifications.

5.2 XML Representation of Invariants

Figure 8 shows an example rule structure that is defined in the XML file with all of the AP rules. It is assumed that the XML structure is generated as a result of parsing the ECA rule form. The XML structure starts with the inputs tag which includes the invariantName, startAP and endAP. These variables are used to keep track of the invariant and to make sure that the invariant is placed in the correct AP so that monitoring can be activated or deactivated accordingly. In this prototype only one invariant is allowed per AP, therefore, only one set of inputs can be used per invariant definition.

The next section includes any parameters that are defined in the process that are used in the SQL condition (SQLVariable) or in any of the monitoredObjects. Multiple parameters can be expressed so that those values can be extracted from the process and used for monitoring the correct SQL condition during runtime. Only the name is needed to specify a parameter and this name should match the variable name specified in the process.

Next, the invariant is specified first with the name of the condition and the information needed to invoke the Invariant Agent. The Invariant Agent will be called to register the invariant rule. Additionally, the monitoredObjects are defined so that the Invariant Agent can store this information and also relay it to the Delta Analysis Agent for delta monitoring. Multiple monitoredObjects can be related to one Invariant condition and each table within a monitoredObject can have multiple attributes. Each attribute has an attributeName, setValue and the relation between the attributeName and the setValue. Lastly, up to two recovery actions can be defined so that if there is a violation or a reoccurring violation, it can be handled by the process appropriately.

This definition identifies the details of the activation and deactivation of monitored objects.

```
<rules>
<event type="ap" ap="loanAppCreation ">
      <inv>
        <ecaRule>
         <inputs>
            <input invariantName="loanAmountMonitoringInv " startAP="loanAppCreation " endAP="loanCompletion "/>
         </inputs>
         <parameters >
            <parameter name ="customerId"/>
         </parameters >
             <condition name="loanAmountMonitoring ">
                 <invoke
                   serviceName="SQLEvaluation " serviceLocation="http://localhost:8080/wsrf/services/examples/core/sqlEval/
sqlEvalService"
                   operation ="main"
                   SQLVariable="(Not exists(select * from loan where loan .applicantID = '" + customerId + "' and loan .status =
'pre-qualified' and loan .amount < (select 10*balance from account where account .customerID ='" + customerId + "'))"
                   curRecovVariable="currentRecovery "
                   outputVariable="result">
                   <monitoredObjects tableName ="loan ">
                       <attributes>
                          <attribute attributeName="status" setValue="pre-qualified " relation="equals "/>
                          <attribute attributeName="applicantID" setValue=customerId relation="equals "/>
                          <attribute attributeName="amount" setValue="calc" relation="lessthan"/>
                       </attributes>
                   </monitoredObjects >
                   <monitoredObjects tableName ="account ">
                       <attributes>
                          <attribute attributeName="applicantID" setValue=customerId relation="equals "/>
                          <attribute attributeName="balance" setValue="calc" relation="equals "/>
                       </attributes>
                   </monitoredObjects >
                   </invoke>
             </condition >
             <actions>
                <action name="APRetry" targetAP= "loanAppCreation "/>
                <action name="APRollback"/>
             </actions>
        </ecaRule>
      </inv>
   </event>
<rules/>
```

Figure 8. XML Representation of Invariant Rules

Figure 9 presents a more conceptual view of the information being captured by Invariant rules in the context of a process and its Assurance Points. The top portion of Figure 9 shows the metadata associated with a process. The lower portion shows the runtime information.

The metadata of the Invariant Monitoring System consists of the Invariant, InvariantParameters, InvariantCondition, Table, and Attribute classes. Each Invariant has a startap and an endap, and 0…* InvariantParameters. InvariantParameters are used to include information in the invariant that is present at runtime. The two APs that are

related to the Invariant determine when the invariant is activated and deactivated. Each Invariant can have only one InvariantCondition because an invariant can only monitor one SQL condition at a time. The InvariantCondition is broken down into monitored objects which have 0…* tables and each Table can have 0…* Attributes.

The runtime information of the Invariant Monitoring System includes the Invariant, InvariantParameters, and InvariantCondition classes. Each Invariant has a invariantName, startap and an endap, and 0…* InvariantParameters. InvariantParameters are related to a single Invariant and contain the paramName and the invarId of the Invariant it is related to. Each Invariant can have only one InvariantCondition because an invariant can only monitor one SQL condition at a time. The InvariantCondition contains the invariantId, SQLCondition, MObjList, tupleCount, numViolations, and singleTable. The invariantId is the Invariant that the InvariantCondition is related to. The SQLCondition is the SQL query that the Invariant forwards to the evaluation Web Service to query the source database. Next is the MObjList which contains a list of all monitored objects that are related to this Invariant. Each monitored object contains the table name, attribute name, relation, and set value. The tupleCount and numViolations are used to keep track of the number of tuples returned from the last SQL evaluation and how many potential violations have occurred since the last evaluation. The singleTable attribute is used store whether the Invariant is monitoring one or multiple tables. Use of these values in the monitored process will be explained in the following sections that describe the filtering algorithms.

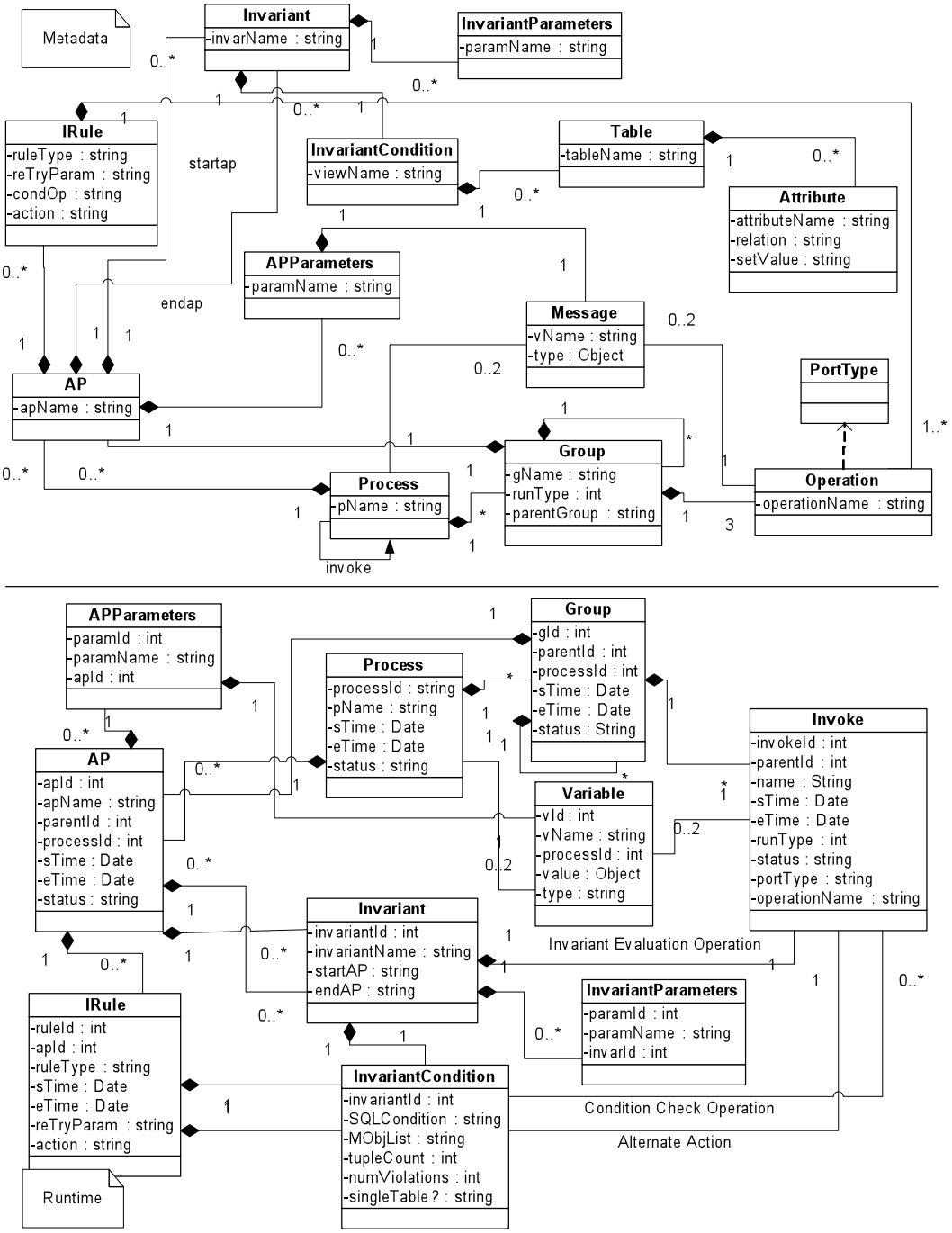

Figure 9. Process Metadata and Runtime Information

5.3 Registration of Invariants

Within the AP structure, an invariant object is used to forward the invariant information to the Invariant agent, where the invariant will be registered. The invariant object contains the SQL condition, the list of monitored objects, the siteID, and the current recovery action. The siteID is the machine id combined with the process id on that machine and is forwarded to the Invariant agent from the executing process. The Invariant agent receives the invariant object and begins by checking the SQL condition using an SQL evaluation Web Service (Shuman, 2010). The location of this Web Service is received by the Invariant Agent from the XML rule structure and stored in the Invariants table. If the SQL condition is satisfied then the SQL condition and current recovery action are inserted into the Invariants table along with the siteID if the invariant is not already in the database. A procedure is used in the database so that a new invariant id is created when a new invariant is inserted. If the invariant is already in the database then the recovery action is updated. Next, the monitored objects are inserted into the MonitoredObject table if they are not already stored there. Another procedure is used in the database to create a new monitored object id and ensure a unique identifier.

Figure 10 shows a high level view of the relationship between the MonitoredObject table and the Invariants table. As can be seen in Figure 10, there can be many MonitoredObjects related to many Invariants. If an Invariant no longer needs to be monitored, then it is deactivated and deleted from the Invariants table and if the objects related to that invariant are not also related to another invariant, then they will also be removed. The combination of the siteID and unique identifiers of the invariant and monitored objects allow relationships to be easily seen and know which invariant relates to each monitored object. Additionally, this structure allows the system to keep track of what information should still be monitored and what properties to check.

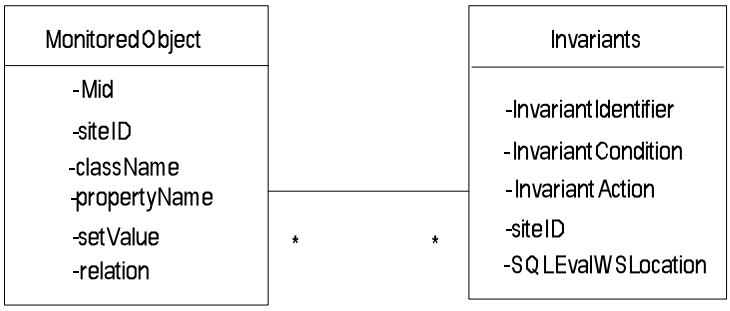

Figure 10. Invariant Table Relationship

5.4 Invariant Evaluation Web Service

An important component of the Invariant Monitoring System is the Invariant Evaluation Web Service (Shuman, 2010). The Web Service is used to initially evaluate the SQL query of an invariant to determine if the condition is satisfied. Since the invariant may need to be re-evaluated several times between the starting and ending APs, the Web Service was designed to make use of materialized views to provide a more efficient way of checking the status of the invariant.

A materialized view is a database object that contains the results of a query and is automatically updated after a table that is in its SQL query is changed. Therefore, simply selecting the number of tuples from the materialized view is faster and more efficient than re-executing the SQL query.

Figure 11 presents an example structure for creating a materialized view. The name of the materialized view is the invariant identifier with "inv" preceding it. Since we always want to refresh the materialized view after a commit of any changes to relevant tables, the "REFRESH FORCE" option is used. If a "FAST" refresh is not possible then the materialized view will execute a "COMPLETE" refresh. A "FAST" refresh only applies data changes from recent data changes and is the most efficient way to refresh the materialized view. A "COMPLETE" refresh starts from scratch and completely rebuilds the materialized view, which takes longer than a "FAST" refresh but still refreshes the materialized view. The next line allows the materialized

view to rewrite the query to optimize it and make the query more efficient. The last line in Figure 11 is the actual SQL that the materialized view executes.

```
CREATE MATERIALIZED VIEW inv123
REFRESH FORCE ON COMMIT
ENABLE QUERY REWRITE AS
<select statement for invariant>
```

Figure 11. Materialized View Example

Because the "FAST" refresh requires logs on each table of the SQL query, the invariant evaluation Web Service checks to make sure that logs have been created for each table in the SQL statement. Figure 12 describes the functionality of a Web Service to evaluate the invariant. Instead of simply re-executing the SQL query to re-evaluate the SQL, an invariant evaluation service creates a materialized view and then rechecks the materialized view. After creating any necessary logs, the invariant evaluation Web Service checks if the materialized view exists. If the view does not exist the view is created and the number of tuples is queried from the view. If the view already exists then the number of tuples are queried from the view.

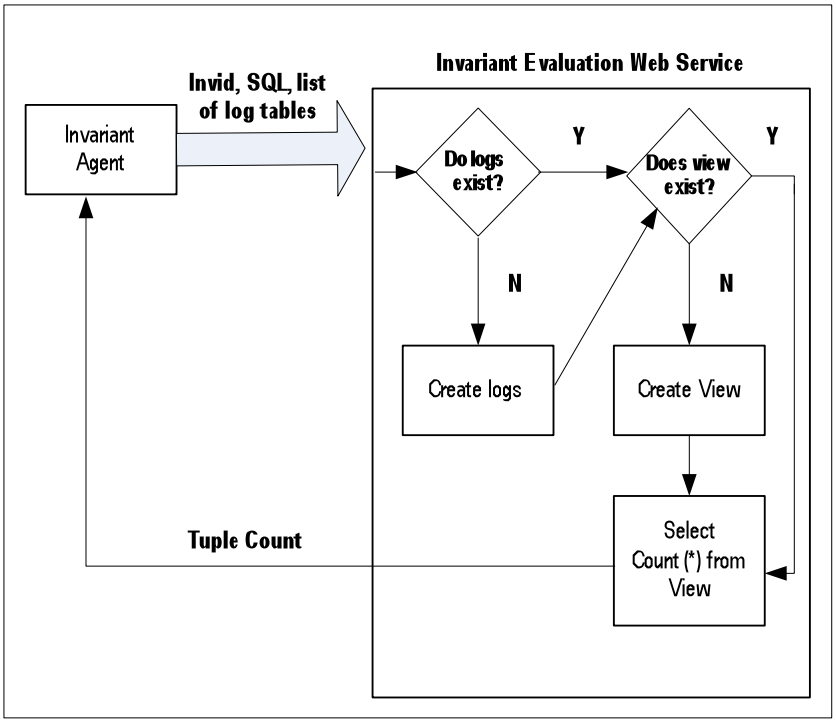

Figure 12. Evaluation Web Service Functionality

5.5 Extensions to DEGS

To allow the monitoring of invariants, changes had to be made to the existing DEGS System. Instead of only gathering information about certain attributes in a table, all attributes of a table are forwarded to the DEGS. Figure 13 describes the format of the delta after modifying the DEGS. The original DEGS only gave the old and new values of the items that had changed. Therefore in an update, only one value would be stored in the delta. In order to monitor data conditions, more information was needed to narrow down deltas that related to monitored objects. The DEGS was modified to capture all fields so that all information could be parsed in the Delta Analysis Agent. If a column or attribute is unchanged then the old and new values in the delta would be the same value. This new structure gives the Delta Analysis Agent the necessary information it needs to filter through deltas and determine if any violations have occurred.

transactionID	sourcerowID	Deltacreation time	oldAttr1	...	oldAttrN	newAttr1	...	newAttrN	sentToEvent Service
24243523	3422	11:11:14	'X123'	...	5	'X123'	...	15	0

Figure 13. New Delta Representation

In addition to the changes in the delta structure, the Delta Analysis Agent is also now incorporated into the DEGS to allow the Delta Analysis Agent to filter out specific deltas to check for violations to invariants currently being monitored.

5.6 The Delta Analysis Process

The Delta Analysis Process invokes the filtering of delta information against the monitored objects. This section first outlines the storage of information about monitored objects. The remainder of the section outlines the algorithm for filtering objects and rechecking invariant conditions.

5.6.1 Invariant Storage Container

To support the delta filtering process, a storage container for the monitored objects is required. Figure 14 shows the Delta Analysis Agent (DAA) Invariant Storage Container which consists of two hashtables. The first hash table is the table/attribute hashtable containing a vector of invariant identifiers that have monitored objects containing the same table/attribute combination as the key. For example, if an invariant is monitoring the price attribute in the orders table then the key would be orders/price and the invariant identifier of that invariant would be inserted into the container of that key in the table/attribute hashtable. The second hashtable or invariant hashtable uses the invariant identifier as the key and relates that key to a container of Monitored Objects of that invariant. The first monitored object in the container contains information about the number of tuples that the last evaluation of the invariant found, the current number of violations found against that invariant identifier, and the invariant identifier. The rest of the container holds the monitored objects that are related to that invariant so that all conditions related to that invariant can be checked at the same time.

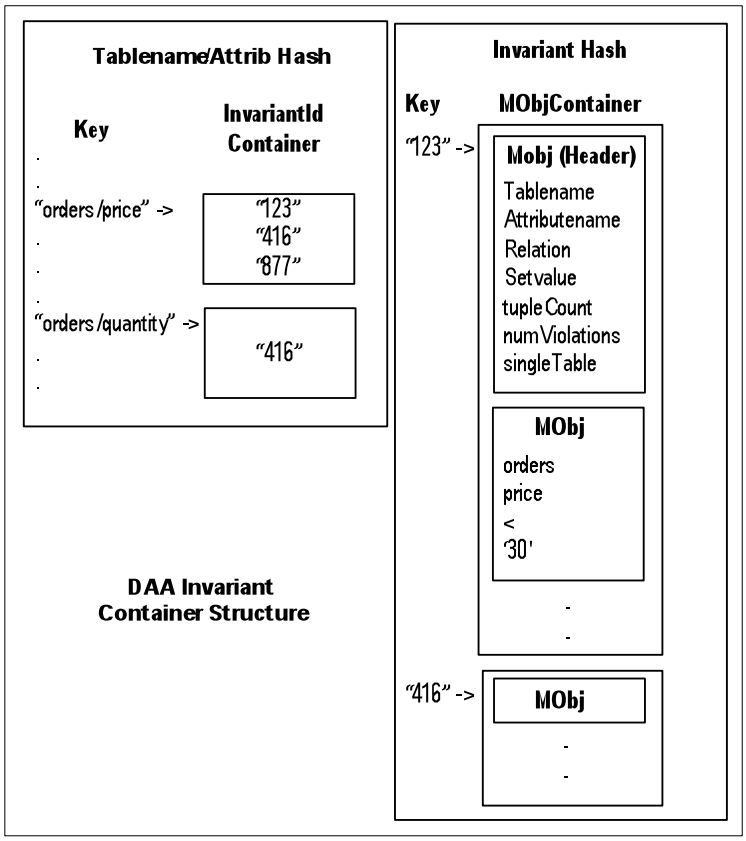

Figure 14. Delta Analysis Agent Storage Structure

Figure 15 introduces the algorithm for updating the DAA structure with new invariant information. The Invariant Agent sends a string containing the invariant identifier and the list of monitored objects to the DAA. The list is tokenized using a string tokenizer and while there are more tokens the function continues to build upon the temporary monitored object container to be inserted into the DAA structure. The first token is always the invariant identifier. This token is stored in a temporary monitored object and then inserted into the temporary monitored object container. The algorithm then loops through the rest of the monitored objects that have been tokenized and capture the tablename, attributename, relation, and setvalue in the temporary monitored object. This information is then inserted into the temporary monitored object container. If there is already a table/attribute key in the table/attribute hashtable, then the algorithm adds the new invariant identifier to the invariant container. If the table/attribute key does not already exist, a new entry is

created. After looping through all the monitored objects, the algorithm inserts the temporary monitored object container into the Invariant hashtable using the invariant identifier as the key. Lastly, the algorithm checks to make sure the hashtable has been updated accordingly and returns true if the size is greater than zero and false otherwise.

```
public boolean updateMonitoring  (String MObjectList)
{
        temporary MObj and MObjContainer are created
        tokenize input MObjList
        while( there are more tokens  )
        {
                if( first token )
                {
                        set invID in temp MObj
                        set other attributes to empty
                        insert temp MObj into temp MObjContainer
                        increment token counter
                }

                set tablename , attribute name , relation and setvalue in temp MObj
                insert temp MObj into temp MObjContainer

                //Note: all keys are lower case using the .toLowerCase() function
                //Check if we are already monitoring this table/attr combination
                if(table/attribute hashtable contains tablename and attribute name in temp MObj  )
                {
                        get the container containing the invariant IDs
                        if( the invID is not already in this container)
                        {
                                add the invID to the container
                                update tablename/attribute key and container in table/attribute hashtable
                        }
                }
                else
                {
                        create new integer container
                        add the invID to the container
                        insert tablename/attribute key and container into table/attribute hashtable
                }
                increment token counter by  4
        }
        //Finished getting all the monitored objects
        insert tempMObjContainer into the Invariant hashtable

        if( table/attribute hashtable's size is0 )
        {
                return false; //no objects to monitor
        }
        else
        {
                return true ; //objects to monitor
        }
}
```

Figure 15. UpdateMonitoring Algorithm

Figure 16 demonstrates the removal of invariants and monitored objects. This algorithm is used when an invariant that is being monitored needs to be deactivated and removed. Similar to the update monitoring algorithm, a string containing the invariant identifier and the list of monitored objects is sent from the Invariant Agent when an invariant is being removed or deactivated. The string is tokenized and the while loop continues while there are more tokens. The invariant identifier is extracted from the first token. Then the following tokens are checked to see if they are a key in the table/attribute hashtable. If a token is a key, then the container of invariant identifiers is captured and the invariant identifier of the invariant being removed is removed from the container. If the last or only invariant left in that container was just removed then the algorithm removes the entry in the table/attribute hashtable. After removing the invariant identifier from all entries in the table/attribute hashtable, the invariant is removed from the Invariant hashtable. If there are no errors then the function returns true and the invariant has been successfully removed from both hashtables.

```
public boolean removeMonitoring  (String MObjectList)
{
        temporary InvContainer is created
        tokenize input MObjList
        while( there are more tokens  )
        {
                if( first token )
                {
                        get MObjectList invID
                        increment token counter
                }


                //Note: all keys are lower case using the  .toLowerCase() function
                //look for this table/attr combination
                if(table/attribute hashtable contains tablename and attribute name in temp MObj  )
                {
                        get the container containing the invariant IDs
                        for( each invID in the container )
                        {
                                if( current invID == MObjectList invID )
                                {
                                        remove the invID from the container
                                }

                                if( InvContainer.size() > 0)
                                {
                                        update container in table /attribute hashtable
                                }
                                else
                                {
                                        remove key from table/attribute hashtable
                                }
                        }
                }
                else
                {
                        return error ; //invID not found
                }
                increment token counter by  4
        }
        //Now remove the invID from the invToMObjContainer
        remove invID from Invariant hashtable

        return true ; //remove successfull
}
```

Figure 16. RemoveMonitoring Algorithm

5.6.2 Overview of the Delta Filtering Process

To process delta notifications from the Oracle source database, the Delta Filtering Process was used. To begin, the DAA receives the table name from a java stored procedure in the Oracle database when a table has been modified. Next, each delta table related to that received table name is queried first for inserts, then updates, and lastly deletes. When a delta is found it is sent to the checkForViolations function to

check against the monitored objects currently being monitored. If enough violations are found then the Delta Process Filtering forwards the invariant identifier to the Invariant Agent to re-evaluate the invariant condition.

To handle inserts, deletes and updates in a delta notification, two algorithms were developed. One algorithm handles insert and delete operations. The other algorithm handles updates, where a delta for an update contains old and new values for each attribute in the tuple.

To allow a more efficient method of determining when to re-evaluate the invariant for single tables, a variable containing the number of tuples returned from the SQL query was introduced. When monitoring a single table, if the number of tuples equals the number of violations found then there are no more tuples left that satisfy the invariant condition and so the Delta Analysis Agent can know that the condition has been violated without re-evaluating the condition. Using the number of tuples and number of violations, calling the Invariant Agent to re-evaluate the condition can be eliminated, saving unnecessary re-evaluation. For a single table invariant, when an insert is received the Delta Filtering Process checks if the newly inserted tuple satisfies all of the conditions in the monitored objects. If the tuple satisfies all of the conditions then the number of tuples is incremented by one. If the tuple does not satisfy all of the conditions then it is ignored. When a delta containing a delete operation is received monitoring a single table the Delta Filtering Process checks tuple against all of the monitored object conditions of the invariant. If the tuple satisfies all of the conditions then the number of violations is incremented by one. If the tuple does not satisfy all of the conditions then it is ignored.

When monitoring multiple tables a threshold is used instead of comparing the number of tuples and the number of violations found. Invariants that involve join conditions and, therefore, multiple tables, require rechecking the invariant condition. A tuple from one table can join with multiple tuples from another table. As a result, an insert, delete, or update can cause multiple tuples to enter or leave the result of the invariant. Furthermore, depending on the number of tuples in the invariant result, these

changes do not necessarily violate the invariant condition. It is not desirable, therefore, to check the invariant after each change to a relevant table. A threshold value is used as a way to periodically initiate a re-evaluation. A threshold value is a percentage of the number of tuples that determines when to re-evaluate the invariant condition. This research has used a threshold value of 25% of the invariant tuples. To more accurately determine what threshold to set, statistical analysis of the joins involved in the invariant should be used so that re-evaluations occur often enough to catch when there are a small number of tuples left and wait long enough between re-evaluations to reduce overhead as much as possible. This statistical analysis is beyond the scope of this research and is left for future research.

5.6.3 Delta Filtering Algorithms

Figure 17 presents the algorithm that handles a delta that corresponds to an insert or delete operation. The algorithm begins by iterating through each delta received that corresponds to either an insert or delete operation in the source database. Since the delta contains information about each column in the table, each column or attribute in the delta is checked. The table and attribute combination is checked to see if it is a key in the table/attribute hashtable. If the key exists then the algorithm extracts the container of invariant identifiers that relate to this table/attribute.

For each invariant identifier, the algorithm gets the monitored object container using the Invariant hashtable. If the algorithm has not already checked this invariant identifier then the algorithm checks all the monitored objects inside the monitored object container that relate to the table the delta came from. If this insert or delete violates a condition of a monitored object and is not a calculated value, then checking the rest of the monitored objects is not necessary. The algorithm stops checking because it is looking for changes that affect the amount of tuples that satisfy the SQL condition of the invariant. If the condition of the insert or delete does not match those of the monitored object, then this delta did not affect the amount of tuples.

If there is no violation in the entire monitored object container and the delta contains a delete operation, then the numViolations variable is incremented in the

monitored object container. If there is no violation in the entire monitored object container and the delta contains an insert operation, and the invariant is over a single table, then the tupleCount variable is incremented in the monitored object container. Inserts for multiple tables are ignored because the algorithm cannot keep record of changes across multiple tables. After incrementing the numViolations, the threshold is calculated by multiplying the number of tuples by the threshold percentage if the invariant is monitoring multiple tables. In an invariant monitoring multiple tables, if the number of violations is greater than the threshold then the algorithm forwards the invariant identifier to the Invariant Agent to re-evaluate the SQL condition. Instead of re-evaluating the SQL condition after every violation, re-evaluation only occurs after the number of violations affect a certain threshold or percentage of tuples. This minimizes the number of re-evaluations to only happen when an increased amount of activity has been seen in tuples related to the SQL condition. If the invariant is monitoring a single table then the algorithm compares the numViolations to the tupleCount. If these values are equal then there are no more tuples and the algorithm forwards the invariant identifier to the Invariant Agent to re-evaluate the SQL condition.

The following cases illustrate different aspects of the filtering process.

Case 1: Single table insert
Invariant: "select r.price from room r where r.price < '30' and r.roomType = 'seaview' and r.hotelid = '234'"
Monitored Objects: [(room, price, <, '30'), (room, roomType, =, 'seaview'), (room, hotelid, =, '234')]
Number of Satisfying Tuples: 1
Case Description: If a tuple satisfying all of the monitored object conditions is inserted into the room table, then the number of tuples is incremented by one. If even one of the monitored object conditions is violated by the tuple then the number of tuples is not incremented.

Case 2: Single table delete

Invariant: "select r.price from room r where r.price < '30' and r.roomType = 'seaview' and r.hotelid = '234'"

Monitored Objects: [(room, price, <, '30'), (room, roomType, =, 'seaview'), (room, hotelid, =, '234')]

Number of Satisfying Tuples: 1

Case Description: If the one tuple is deleted by an external process then the number of violations will be incremented and the number of tuples will equal the number of violations. Therefore, notification will be sent to the process monitoring the invariant condition, the invariant will be removed from the monitoring process, and the process will be informed of the violation.

Case 3: Multiple table insert

Invariant: "select r.price from room r, hotel h where r.price < '30' and r.roomType = 'seaview' and r.hotelid = h.hotelid and h.state = 'Texas'"

Monitored Objects: [(room, price, <, '30'), (room, roomType, =, 'seaview'), (hotel, state, =, 'Texas')]

Number of Satisfying Tuples: 25

Threshold: 25%

Case Description: All inserts into multiple table invariants are ignored. Inserting tuples can potentially increase the size of the number of tuples that satisfy the invariant condition, but will not cause a violation.

Case 4: Multiple table delete

Invariant: "select r.price from room r, hotel h where r.price < '30' and r.roomType = 'seaview' and r.hotelid = h.hotelid and h.state = 'Texas'"

Monitored Objects: [(room, price, <, '30'), (room, roomType, =, 'seaview'), (hotel, state, =, 'Texas')]

Number of Satisfying Tuples: 25

Threshold: 25%

Case Description: If 7 tuples from the room table satisfying the invariant condition are deleted one after another by an external process then the number of violations will be incremented each time and after the seventh deletion the number of violations will be greater than the threshold(7 > .25*25). Therefore, the invariant condition will be re-evaluated and because tuples are found, the invariant will update the number of tuples, reset the number of violations to zero and continue monitoring. If the process continues and after another re-evaluation no more tuples were found, notification would be sent to the process monitoring the invariant condition and the invariant would be removed.

If a tuple in the hotel table that is in 'Texas' is deleted by an external process and this deletion triggers 7 rooms related to that hotel to also be deleted. Then, after the seventh deletion from the room table the number of violations will be greater than the threshold (7 > .25*25). Therefore, the invariant condition will be re-evaluated and because tuples are found, the invariant will update the number of tuples, reset the number of violations to zero and continue monitoring. If the process continues and after another re-evaluation no more tuples were found, notification would be sent to the process monitoring the invariant condition and the invariant would be removed.

```
for( each delta )
{
	for(int i=1;i<=numberColumnsInDelta;i++)
	{
		//check if we are monitoring this column of the delta
		if(hashtable1 contains "tablename/column(i)")
		{
			//get invariant container from hashtable 1
			for( each invariant in the invariant container  )
			{
				if( we have already checked this invariant )
					//exit, we do not want to check an invariant for the same delta twice
				//get the MObjContainer for the invariant from hashtable 2
				for( each MObj in the MObjContainer  (j) )
				{
					for( each column in the delta  (k) )
					{
						if( MObj (j) contains attrib (k) )
						{
							if( attribute (k) violates the condition being monitored by MObj  (j)
and is not a calculated value )
							{
								//stop checking this MObjContainer because a filter has
been violated
							}
						}
					}
				}
				if( entire MObjContainer satisfies all object constaints && operation is a delete )
				{
					//increment number of violations
					//calculate threshold (number of tuples * threshold percentage )
					if(numViolations > threshold && invariant is monitoring multiple tables )
						//forward invID to Invariant Agent
					else if( numViolations >= numTuples && invariant is monitoring one table  )
							//forward invID to Invariant Agent
				}
				else if(entire MObjContainer satisfies all object constaints  && operation is a insert  &&
invariant monitoring one table )
					//add one to tupleCount
			}
		}
	}
}
```

Figure 17. Checking Insert or Delete Delta Algorithm

Figure 18 presents the algorithm that handles a delta that corresponds to an update operation. The algorithm begins by iterating through each delta received that corresponds to an update operation in the source database. Since the delta contains information about each column in the table, each column or attribute in the delta is checked. The table and attribute combination are checked to see if they are a key in the table/attribute hashtable. If the key exists then the algorithm extracts the container of invariant identifiers that relate to this table/attribute.

For each invariant identifier, the algorithm gets the monitored object container using the Invariant hashtable. If the algorithm has not already checked this invariant identifier then it continues to check all the monitored objects inside the monitored object container that relate to the table the delta came from. An update operation can have one or more values that have been changed and one or more values that stay the same. Therefore, for the attributes that have been changed in this delta, if the attribute is a calculated value or if the change violates the condition being monitored by the current monitored object in the monitored object container, the algorithm flags this attribute for a violation.

If the attribute has not been changed, the attribute value must be the same as the condition specified in the current monitored object. If the unchanged attribute violates the condition specified in the current monitored object then this delta does not relate to the current monitored object and the algorithm stops checking the monitored object container. If at least one violation is found and a violation is not encountered for unchanged values in the monitored object container then the algorithm increments the number of violations. After incrementing the numViolations, the threshold is calculated by multiplying the number of tuples by the threshold percentage if the invariant is monitoring multiple tables. In an invariant monitoring multiple tables, if the number of violations is greater than the threshold then the algorithm forwards the invariant identifier to the Invariant Agent to re-evaluate the SQL condition. If the invariant is monitoring a single table then the algorithm compares the numViolations to the tupleCount. If these values are equal then there are no more tuples and the algorithm forwards the invariant identifier to the Invariant Agent to re-evaluate the SQL condition.

The following cases illustrate the filtering process described in the algorithm.

Case 1: Single table update

Invariant: “select r.price from room r where r.price < ‘30’ and r.roomType = ‘seaview’ and r.hotelid = ‘234’“

Monitored Objects: [(room, price, <, '30'), (room, roomType, =, 'seaview'), (room, hotelid, =, '234')]
Number of Satisfying Tuples: 25
Case Description: Assume a tuple satisfying the invariant condition is updated by an external process to now have a price of '60' and the roomType to 'noview'. Because the tuple used to satisfy the invariant before the change in price and roomType, a violation is found and the number of violations is incremented. But, because the number of violations is not equal to the number of tuples, no action is taken. If the external process continues to update tuples, changing any or all of the attributes that are being monitored by the monitored objects until the number of violations is 25 then the number of violations and the number of tuples will equal the number of violations. Therefore, notification will be sent to the process monitoring the invariant condition and the invariant will update the number of tuples, reset the number of violations to zero, and notify the process if the number of tuples is zero. If the number of tuples equals zero then the invariant and monitored objects will also be removed.

Case 2: Multiple table update
Invariant: "select r.price from room r, hotel h where r.price < '30' and r.roomType = 'seaview' and r.hotelid = h.hotelid and h.state = 'Texas'"
Monitored Objects: [(room, price, <, '30'), (room, roomType, =, 'seaview'), (hotel, state, =, 'Texas')]
Number of Satisfying Tuples: 25
Threshold: 25%
Case Description: If a tuple in the room table satisfying the invariant condition is updated by an external process to now have a price of '60' and the roomType to 'noview'. Because the tuple used to satisfy the invariant before the change in price and roomType, a violation is found and the number of violations is incremented. But, because the number of violations is not greater than to the threshold(1 < .25*25), no action is taken. If the external process continues to update tuples, changing any or all

of the attributes that are being monitored by the monitored objects until the number of violations are greater than the threshold. Therefore, the invariant condition will be re-evaluated and because tuples are found, the invariant will update the number of tuples, reset the number of violations to zero and continue monitoring. If the process continues and after another re-evaluation no more tuples were found, notification would be sent to the process monitoring the invariant condition and the invariant would be removed.

If a tuple in the hotel table that is in 'Texas' is updated by an external process and this update triggers 7 rooms related to that hotel to also be updated. Then, after the seventh update from the room table the number of violations will be greater than the threshold ($7 > .25*25$). Therefore, the invariant condition will be re-evaluated and because tuples are found, the invariant will update the number of tuples, reset the number of violations to zero and continue monitoring. If the process continues and after another re-evaluation no more tuples were found, notification would be sent to the process monitoring the invariant condition and the invariant would be removed.

```
for( each delta )
{
	for(int i=1;i<=numberColumnsInDelta ;i++)
	{
		//check if we are monitoring this column of the delta
		if(hashtable1 contains "tablename /column(i)")
		{
			//get invariant container from hashtable 1
			for( each invariant in the invariant container  )
			{
				if( we have already checked this invariant )
					//exit, we do not want to check an invariant for the same delta twice
				//get the MObjContainer for the invariant from hashtable 2
				for( each MObj in the MObjContainer  (j) )
				{
					for( each column in the delta  (k) )
					{
						if( MObj (j) contains attrib (k) )
						{
							if( attribute (k) has been changed in this delta  )
							{
								if( attribute (k) is a calculated value )
								{
									//flag for violation
								}
								else if( the change in attribute  (k) violates the condition
being monitored by MObj  (j) )
								{
									//flag for violation
								}
							}
							else //attribute (k) has not been changed in this delta
							{
								if( attribute (k) violates the condition being monitored by
MObj (j) )
								{
									//stop checking this MObjContainer because a filter
has been violated
								}
							}
						}
					}
				}
				if( entire MObjContainer satisfies all object constaints and violates at least one condition )
				{
					//increment number of violations
					//calculate threshold (number of tuples * threshold percentage )
					if(numViolations > threshold  && invariant is monitoring multiple tables )
						//forward invID to Invariant Agent
					else if( numViolations >= numTuples  && invariant is monitoring one table  )
							//forward invID to Invariant Agent
				}
			}
		}
	}
}
```

Figure 18. Checking Update Delta Algorithm

Figure 19 describes the updateTupleInv function that is called after evaluating the invariant condition. The Invariant Agent receives the number of tuples from the SQL evaluation Web Service and forwards the invariant identifier and the number of

tuples to the updateTupleInv function. The function begins by separating the invariant identifier from the number of tuples using a String tokenizer. After extracting the invariant identifier, the algorithm checks to make sure it is monitoring this invariant identifier in the first hashtable. If the invariant identifier is not found in the hashtable then there is an error and something did not execute correctly. Next, the Monitored Object Container related to the invariant identifier is retrieved and the tupleCount is updated to the received value of the tuple count. The number of violations is then reset back to zero and the update is successful. However, if there are no more tuples then monitoring of this invariant identifier is removed because a violation has occurred. The algorithm iterates through the monitored objects in the Monitored Object Container and builds the String to send to the removeMonitoring function that was previously described. The removeMonitoring function will take care of removing the invariant and its Monitored Object Container so that the Delta Analysis Agent will no longer be monitoring those data conditions. After finishing, the function returns true if there were no errors and the number of tuples and number of violations has been successfully updated.

```
public boolean updateInvTupleCount (String invariantInfo)
{
        //create tempMObjContainer

        //tokenize invariantInfo, extracting the invID and the number of tuples

        if( we are monitoring this invID)
        {
                //get the MObjContainer of this invID and put it into tempMObjContainer
                //update the tupleCount with the received number of tuples value
                //set the number of violations to zero

                if( there are no more tuples ) //we want to remove the monitoring
                {
                        for( each monitored object in tempMObjContainer )
                        {
                                //add monitored object to string to send to removeMonitoring function
                        }
                        //send entire string of monitored objects to the removeMonitoring function
                }
        }
        else
        {
                return false;
        }
        return true;
}
```

Figure 19. UpdateInvTupleCount Algorithm

CHAPTER VI

TESTING AND EVALUATION OF THE INVARIANT MONITORING SYSTEM

To evaluate the prototype of the Invariant Monitoring System, a testing environment and test cases were created and initialized. The primary focus of the evaluation was on the performance of the Invariant Evaluation Web Service to determine if the use of materialized views improves the performance of the re-evaluation process.

Section 6.1 describes the test environment and the setup of the test environment. Section 6.2 and its sub-sections describe specific test cases and Section 6.3 provides results of the evaluation.

6.1 Testing Environment Setup

The testing example used was the Hotel monitoring example that was previously described. This example required two tables, a Hotel table and a Room table. Figure 20 shows the tables and their column names used for this example.

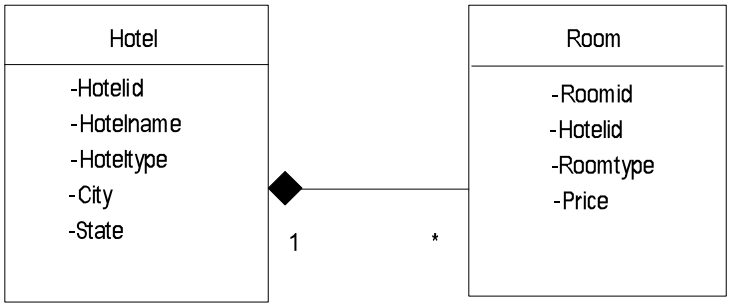

Figure 20. Example Table Structure

Next, the DEGS was setup to monitor changes to all columns of both the Hotel table and the Room table. After setting up the delta monitoring in the source database, a process with Assurance Points was created for activating and deactivating different test invariants. Another concurrent process was also needed to modify the monitored data in the source database. As a result, an additional Web Service was created to

insert, update and delete data in the source database and was deployed along with a client to pass information to the Web Service.

To test the example, the capture and apply procedures for the Oracle Streams features were started in the source database to make sure delta notifications are received when an operation was done on either the hotel or room tables. Next, the DEGS and Invariant Monitoring System were deployed inside of the container of all the Grid Services using the OGSA-DAI framework. After deploying the DEGS and Invariant Monitoring System, the process with assurance points was executed, calling the Invariant Monitoring System to activate the invariant upon reaching the startAP. This setup was used to test cases to evaluate the functionality of the Invariant Monitoring System can be investigated.

6.2 Test Cases

To test the performance of the Invariant Evaluation Web Service, two groups of test cases similar to those previously explained in Chapter 5 were used. Both test cases used were over multiple tables so that the Invariant Evaluation Web Service would be called. Since re-evaluation occurs primarily in the context of multiple table invariants, the focus of the evaluation was on invariants that involve join conditions. The first test case updated 25 tuples, with changes ranging from satisfying the invariant condition to not satisfying the invariant condition. These updates triggered the invariant condition to be re-evaluated, but the test was design so that the invariant condition was still satisfied and, as a result, the invariant was not removed. The second test case updated all of the tuples with the changes ranging from satisfying the invariant condition to not satisfying the invariant condition. This test group was designed so that the invariant was violated and, as a result, monitoring of the invariant was removed. The performance of both test cases is described in the next section.

6.3 Performance of Invariant Evaluation

The primary focus of this performance evaluation is based on the performance of the Invariant Evaluation Web Service and calling the Web Service from the

Invariant Agent and the Delta Analysis Agent. The performance of the Invariant Evaluation Web Service is crucial to the Invariant Monitoring System because the invariant condition must be evaluated often. The more efficient the Web Service is, the more efficient the Invariant Monitoring System can be. Table 1 describes different measurements that were taken and the times that each took. Using the multiple table test cases described above, the time it took to execute tasks in the Invariant Monitoring System were observed.

The measurements taken include creating the materialized view, total time of the evaluation Web Service, evaluating from the Invariant Agent, evaluating from the Delta Analysis Agent, selecting from the materialized view, and executing the SQL instead of creating a materialized view. Creating the materialized view is done when the view does not already exist and the time it takes includes creating the view and extracting the number of tuples from the newly created view. The total time of the evaluation Web Service includes checking and creating any logs, and either creating and querying from the materialized view or just querying from the materialized view if it already exists. Evaluating from the Invariant Agent is the time is takes to call and receive feedback from the re-evaluation function in the Invariant Agent for evaluating the invariant condition the first time. Evaluating from the Delta Analysis Agent is the time is takes to call and receive feedback from the re-evaluation function in the Invariant Agent and the time taken can also include the time it takes to remove the invariant condition if there are no more tuples. The time measurement for selecting from the materialized view is the time required to extract the number of tuples from a view that already exists. Executing the SQL is the time it takes to re-execute the invariant condition instead of creating a materialized view and querying from it.

Both multiple table test cases were executed 25 times and an average time in microseconds was recorded for all measurements. During testing, the Oracle database used had at least 100 tuples that satisfied the invariant condition on the initial evaluation. The machine used for testing was a Dell Precision T3400 with 2.99GHz

Intel Core 2 Extreme processor and 4Gb of RAM, running Microsoft Windows XP Professional x64 Edition.

Creating the materialized view in both multiple table test cases took about the same time as well as the total time the evaluation Web Service took. As can be seen from Table 1, the values that are significantly different in the test cases are the times for evaluating from the Delta Analysis Agent. This time difference is because the invariant condition is completely removed from the Invariant Agent and Delta Analysis Agent before returning back from the Delta Analysis Agent evaluation.

A key observation from Table 1 is that the time difference between creating the materialized view and just querying from the materialized view is significantly different and affects the invariant evaluation time. The time it takes to query from an existing materialized view is also much less than using the SQL to repeatedly query the source database from the evaluation Web Service. If the process is long running between the starting and ending APs of an invariant and might potentially re-execute the SQL query of the invariant often, then creating the materialized view is beneficial, otherwise using the SQL would be a better choice for shorter process.

In addition to using the most efficient way of re-evaluation, fine tuning the SQL statement to narrow down the search space can potentially eliminate having to re-evaluate the SQL statement more than once, saving even more execution time. Figure 21 presents two invariants that are monitoring similar attributes from the room table. Invariant 1 is looking for a room with a price less than 30, with a roomType of 'seaview' and in 'Texas'. This is a general select statement and because multiple tables are involved a join must happen between the hotel and room table. Invariant 2 is looking for a room with a price less than 30, with a roomType of 'seaview' and in the hotel with hotelid '234'. This query is more specific and is querying only one table as opposed to two. Therefore there are no joins in invariant 2 and this invariant condition will execute faster than invariant 1. Simplifying the SQL results in less overhead and saving execution time.

Invariant 1

"select r.price from room r , hotel h where r.price < '30' and r.roomType = 'seaview' and r.hotelid = h.hotelid and h.state = 'Texas'"

Invariant 2

"select r.price from room r where r.price < '30' and r.roomType = 'seaview' and r.hotelid = '234'"

Figure 21. Invariant SQL Comparison

Table 1. Performance of Multiple Test Cases

Measurement Description	Avg Time (Microsec) with Invariant Removal	Avg Time (Microsec) without Invariant Removal
Creating Materialized View	**249492**	**269974**
EvalWS Total Time	**371234**	**371025**
Evaluating from Invariant Agent	**575150**	**603457**
Select from Existing Materialized View	**452**	**443**
EvalWS Total Time (without creating materialized view)	**107844**	**117455**
Evaluating from Delta Analysis Agent	**618913**	**292364**
Executing Select Query	**2143**	**2079**

CHAPTER VII

SUMMARY AND FUTURE RESEARCH

To investigate the invariant concept, this research involved the design of an invariant monitoring system. The system included an Invariant Agent and a Delta Analysis Agent. The Invariant Agent begins the monitoring process by registering the invariant condition and is a key component in the re-evaluation of the condition. The Delta Analysis Agent filters through delta notifications from the DEGS system and determines if and when an invariant condition must be re-evaluated. The work is supported by a Web Service for evaluating invariants. Since an invariant may need to be evaluated several times between Assurance Points, the Web Service was designed to make use of materialized views for more efficient re-evaluation of invariant conditions.

Future research should include more efficient methods of checking invariant conditions. Using the DEGS to monitor the created materialized views could provide a more immediate and less complex solution to monitoring multiple tables. The key to monitoring the materialized views would be to determine when a tuple has been inserted, deleted or updated. The DEGS relies on the insert, delete or update commands for transaction logs to determine what action to take. The transaction logs, however, do not include changes in a materialized view. If changes to a materialized view could be monitored by the Oracle Streams system, then invariant condition checking would be more efficient. More statistical analysis of the threshold value should also be done to fine tune the number and frequency of re-evaluations. The analysis should look into how often the tables in join conditions are changed and how many changes are made over certain periods of time. Additionally, the evaluation Web Service could be modified to add a push feature to notify the Invariant Agent when the number of tuples in a materialized view has reached zero. If the DEGS system were to still be a key part in the Invariant Monitoring System, then more

research should be done into how to handle a heavy load of delta notifications, possibly using time stamping to handle a busy execution environment.

Further performance testing should also be done by executing concurrent processes that have APs and invariants and observing the performance under different conditions related to the overlap of the data that they access and modify. Future research should also include an investigation of the methodology for using invariants to determine the types of applications that are more appropriate for the expression of invariants.

REFERENCES

Baker, A., Leavens, G., & Ruby, C. (2005). *Preliminary Design of JML: A Behavioral Interface Specification Language for Java.* TR 98-06-rev27, Iowa State University.

Baresi, L., Ghezzi, C., & Guinea, S. (2004). Smart Monitors for Composed Services. *Proceedings of the 2nd International Conference on Service Oriented Computing*.

Baresi, L., & Guinea, S. (2005). Towards Dynamic Monitoring of WS-BPEL Processes. *Service-Oriented Computing-ICSOC 2005,* vol 3826, pp. 269-282.

Blake, L. Y. (2005). *Design and Implementation of Delta-Enabled Grid Services.* M.S. Thesis, Arizona State University.

Charfi, A., & Mezini, M. (2006). Aspect-Oriented Workflow Languages. *Lecture Notes in Computer Science, 4275*(183).

Charfi, A., & Mezini, M. (2007). AO4BPEL: An Aspect-Oriented Extension to BPEL. *World Wide Web, 10*(3), pp. 309-344.

Elmasri, R. Y., & Navathe, S. B. (2010). *Fundamentals of Database Systems* (6th ed.): Addison-Wesley Longman Publishing Co., Inc.

Garcia-Molina, H., & Salem, K. (1987). Sagas. *Proceedings of the ACM SIGMOD Conference*, pp249-259.

Georgakopoulos, D., Hornick, M., Krychniak, P., & Manola, F. (1994). Specification and Management of Extended Transactions in a Programmable Transaction Environment. *Proceedings of the International Conference of Data Engineering,* IEEE, Los Alamitos, Ca,(USA), 1994, pp. 462-473.

Greenfield, P., Fekete, A., Jang, J., & Kuo, D. (2003). Compensation is Not Enough. *Proceedings of the 7th International Enterprise Distributed Object Computing Conference (EDOC)*.

Irwin, J., Kickzales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., et al. (1997). Aspect-Oriented Programming. *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*.

Jin, W., Rusinkiewicz, M., Ness, L., & Sheth, A. (1993). Concurrency Control and Recovery of Multidatabase Work Flows in Telecommunication Applications. *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pp. 456-459.

Jin, Y. (2004). *An Architecture and Execution Environment for Component Integration Rules*: Ph.D Dissertation, Arizona State University.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., & Griswold, W. (2001). An overview of AspectJ. *Lecture Notes in Computer Science*, pp327-353.

Limthanmaphon, B., & Zhang, Y. (2004). Web Service Composition Transaction Management. *Proceedings of the Fifteenth Australian Database Conference,* vol. 27 of CRPIT, pp. 171-179.

Liu, Z. (2009). *Decentralized Data Dependency Analysis for Concurrent Process Execution.* M.S. Thesis, Texas Tech University.

Luckham, D. C. (1990). Programming with Specifications: An Introduction to Anna, A Language for Specifying Ada Programs. *Texts and Monographs in Computer Science*.

Martens, A. (2005). Analyzing Web Service Based Business Processes. *M. Cerioli, Editor, Proceedings of 8th International Conference on Fundamental Approaches to Software Engineering FASE 2005, Lecture Notes in Computer Science vol. 3442*, pp. 19-33.

Mikalsen, T., Tai, S., & Rouvellou, I. (2002). Transactional Attitudes: Reliable Composition of Autonomous Web Services. *Workshop on Dependable Middleware Based Systems*.

Mikalsen, T., Tai, S., & Rouvellou, I. (March 2002). Transactional attitudes: Reliable composition of autonomous Web services. *in Workshop on Dependable Middleware Based Systems*.

Moffat, A., & Zobel, J. (2006). Inverted Files for Text Search Engines. *ACM Computing Survey, 38*(2).

Papazoglou, M. P. (2003). Web Services and Business Transactions. *World Wide Web,* vol. 6(1), pp49-91.

Reichert, M., & Dadam, P. (1998). ADEPT Flex Supporting Dynamic Changes of Workflows Without Losing Control. *Journal of Intelligent Information Systems,* vol. 10(2), pp. 93-129.

Reuter, A., & Wächter, H. (1991). The Contract Model *A. Elmagarmid (ed.), Database Transaction Models for Advanced Applications*: Morgan Kaufmann Publishers.

Shrestha, R. (2010). *Using Assurance Points and Integration Rules for Recovery in Service Composition.* M.S. Thesis, Texas Tech University.

Shuman, M. (2010). *A Database Service for Checking Invariants*. Technical Report, Department of Computer Science, Texas Tech University.

Urban, S. D., Dietrich, S. W., Na, Y., Jin, Y., Sundermier, A., & Saxena, A. (2001). The IRules Project: Using Active Rules for the Integration of Distributed Software Components. *Proceedings of the 9th IFIP Working Conference on Database Semantics: Semantic Issues in E-Commerce Systems*, pp. 265-286.

Urban, S. D., Xiao, Y., Blake, L., & Dietrich, S. W. (2009). Monitoring Data Dependencies in Concurrent Process Execution through Delta-Enabled Grid Services. *5*(1), pp85-106.

Worah, D., & Sheth, A. (1997). Transactions in Transactional Workflows *Advanced Transaction Models and Architectures,* edited by S. Jajodia and L. Kershberg: Springer.

Wu, G., Wei, J., & Huang, T. (2008). Flexible pattern monitoring for WS-BPEL through stateful aspect extension. *ICWS '08, Beijing, China*.

Wu, G., Wei, J., Ye, C., Zhong, H., & Huang, T. (2010). Detecting Data Inconsistency Failure of Composite Web Services through Parametric Stateful Aspect. *2010 IEEE International Conference on Web Services*.

Zhao, W., Moser, L. E., & Melliar-Smith, P. M. (2009). A reservation-based coordination protocol for Web Services. *In Proceedings of 3rd IEEE International Conference on Web Services (ICWS'05)*.