2009

A System for Incorporating Time-based Event-Condition-Action Rules into Business Databases

Christina Marie Steidle

Wright State University

Follow this and additional works at: https://corescholar.libraries.wright.edu/etd_all

Part of the Computer Sciences Commons

Repository Citation
https://corescholar.libraries.wright.edu/etd_all/952

This Thesis is brought to you for free and open access by the Theses and Dissertations at CORE Scholar. It has been accepted for inclusion in Browse all Theses and Dissertations by an authorized administrator of CORE Scholar. For more information, please contact library-corescholar@wright.edu.
A SYSTEM FOR INCORPORATING TIME-BASED EVENT-CONDITION-ACTION RULES INTO BUSINESS DATABASES

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science

By

CHRISTINA MARIE STEIDLE

B.S., Wright State University, 2002

2009

Wright State University

Mateen Rizki, Ph.D.
Thesis Advisor

Thomas Sudkamp, Ph.D.
Department Chair

Committee on Final Examination

Thomas Hartrum, Ph.D.

Thomas Sudkamp, Ph.D.

Joseph F. Thomas, Jr., Ph.D.
Dean, School of Graduate Studies
ABSTRACT


Human beings handle time-based events continuously; however the passage of time does not play an active part in most business systems because they are typically driven by interaction from human users or other systems. In order to take an action based upon the passage of time it is necessary to build a framework which will monitor the progression of time and a way to define what events the system should be waiting for. This thesis describes such a system, and shows that the system performs as specified. With this system business users are able to build event-condition-action rules using a simple graphical user interface. These rules are then maintained by the system as events which are updated if the source data from which they were generated is modified. When the appropriate time comes they will be activated and the action assigned by the rule will be completed.
Table of Contents

1. Introduction ............................................................................................................. 1
   1.1. Objective ........................................................................................................... 1
   1.2. Problem Definition ........................................................................................... 1
   1.3. Background ....................................................................................................... 2

2. Approach ................................................................................................................ 10
   2.1. System Overview ............................................................................................ 10
   2.2. Event Database ............................................................................................... 13
   2.3. Rule Builder .................................................................................................... 15
   2.4. Event Monitoring Service ............................................................................... 20
   2.5. Data Consistency System ................................................................................ 22

3. Testing and Results ................................................................................................ 25
   3.1. Source Database ............................................................................................. 25
   3.2. Generating Test Data ...................................................................................... 28
   3.3. Test Scenario 1: Static Test ............................................................................. 31
   3.4. Test Scenario 2: Dynamic Tests ...................................................................... 36

4. Conclusion .............................................................................................................. 43
   4.1. Lessons Learned .............................................................................................. 43
   4.2. Future Work .................................................................................................... 45

5. References ............................................................................................................. 48
List of Figures

Figure 1: The components of the system ................................................................. 12
Figure 2: The event database schema ................................................................. 14
Figure 3: Rule builder architecture ................................................................. 16
Figure 4: Rule builder user interface ................................................................. 18
Figure 5: The architecture of the event monitoring service ......................... 20
Figure 6: Database model of the source database ........................................ 26
Figure 7: Entity model of the source database .............................................. 27
Figure 8: Test data from the source database .............................................. 30
Figure 9: Source data - expected over age dependent ................................ 33
Figure 10: Results from the static test ............................................................. 35
Figure 11: Source data for comparison against event data .......................... 37
Figure 12: Results for the rule update test ..................................................... 39
Figure 13: Results from the tracking changes to the source database test .... 42
List of Tables

Table 1: System use cases ........................................................................................................... 11
Table 2: Steps for ensuring data consistency ........................................................................... 24
Table 3: Parameters used for populating the source data ...................................................... 28
Table 4: Static test setup ........................................................................................................... 31
Table 5: Test scenarios for verifying the data consistency system ........................................ 40
1. Introduction

1.1. Objective

The objective of this project is to create a system which will allow business users (i.e. non-programmers) to configure business rules that will define an action that will be triggered when a specified condition occurs. Specifically this system will be designed to handle rules that are based on the passage of time, such as recognizing the date when a child reaches the age where he or she is no longer a valid dependent on his or her parents’ health insurance. This system must also take into account the fact that the data against which the business rules are run is not static. When the source data is updated the system must determine whether the modification affects the date on which the action should be taken. If necessary, the action must be updated so that it will be triggered on the newly calculated date (or immediately if the newly calculated date is in the past), not on the originally scheduled date.

1.2. Problem Definition

The inspiration for this work comes from an existing business problem, specifically from a company that provides integration services between employers and employee benefit vendors. The primary service the company provides is to manage the enrollments entered by the customers’ employees and to make sure that the enrollment information is correctly transmitted to the various vendors in whose plans
the employees have enrolled. One of the details in this process is recognizing when an employee’s child has become over age, requiring an action to be taken either to notify the benefit provider that the dependent is a full time student and should remain covered or to remove coverage for the child. Currently the integration provider does not have an automated process for handling the detection of over age dependents. Instead, a manual auditing process is used to determine when a dependent in the system is over age. The level of work necessary to maintain this process varies by client depending on the frequency the client wants to audit for over age dependents and the number of employees the client has using the enrollment service. For example, a large client recently had hundreds of over age dependents found in an audit, causing an emergency development effort to be undertaken to avoid having to manually process each of the over age dependents. A comprehensive automated solution to this problem would not only save the integration provider’s client services department hundreds of man-hours per year, but would also keep their software engineering department’s strategic projects from being interrupted in order to create ad-hoc solutions to similar problems.

1.3. Background

There are two categories of background information for this project. The first topic is a discussion of other systems that define and use event-condition-action rules. The second section contains information on concepts and technologies that were used for this project such as object-relational mapping and database triggers.
1.3.1. Event-Condition-Action Rules

An Event-Condition-Action (ECA) rule can be generically defined as any rule that defines an action that will be performed when a certain event occurs if the condition specified evaluates to true. These kinds of rules are ubiquitous and they are accepted as commonplace. For example, Wright State will grant a master’s degree (action) when this thesis is complete (event) provided that it is approved and all of the other requirements for the degree are met (condition). Since ECA rules are so general and so intuitive it is not surprising that they have been applied to many domain areas. A few of the areas where the ECA rule concepts are being applied in new research are business process management systems, active database management systems, and in an active software support system.

Using ECA rules in business process management systems provides the major benefit of enabling business processes to operate in real-time; alerting the necessary parties or systems to changes as soon as the event occurs, instead of when someone takes the initiative to check on the process [11]. ECA rules are also commonly used for business process definition because they are easy to work with. They can be defined in a manner that is effortlessly understood by all parties, which reduces the amount of work necessary to define and maintain the business processes. The inherent ability to chain ECA rules (an action could be an event for another rule) and the ability to integrate the action of a rule with unrelated processes makes the ECA rule concept a powerful way to model business processes [2].
Several business process management systems have been built which demonstrate the use of ECA rules for defining business processes. Bry et al. [2] built a system where events defined in messages using an XML format are passed over the web. These messages are handled by ECA rules defined in a custom semi-procedural language called XChange. In contrast Schiefer et al. [11] developed their SARI (Sense and Respond Infrastructure) to allow users to define rules graphically, defining decision graphs comprised of event condition objects, event pattern objects (which allow a series of events to be recognized as a special case and handled differently than the individual events), and response events. These systems demonstrate that ECA rules can be used as a foundation for defining business rules.

Active database management systems also leverage ECA rules in order to automatically change either the schema of a database or the data contained within the database. At a basic level this is done with database triggers which will be discussed in the next section. However, researchers have built a higher level of ECA rules on top of this basic functionality such as the distributed rule management system built by Kantere et al. [6] which supports the dissemination of data between multiple databases in a networked environment. To achieve this goal, both a language for defining the ECA rules and a specific Java based system for interpreting and invoking the rules were developed. This example shows how active database management systems can be used in conjunction with application logic to provide enhanced functionality.
Daniel [3] recently published a research paper on the concept of an active application system called OES (Open ECA Server) which supports defining rules anywhere in a system, from the database level to the application level. His rule system supports monitoring databases for events, monitoring for time-based events, and monitoring for external events generated from other applications. Rules are defined using a custom language called “OpenChimera” which specifies the event(s), condition(s) and action(s) for each trigger (rule). This system is very similar to the initial idea behind the project completed for this thesis. However, OES would not be able to solve the over age dependent problem without adding a way of generating an “instant” temporal event whenever a dependent should be checked for being overage. If such an event could not be generated OES would have to constantly execute the business rules against the source database, which would be very inefficient. Also the system presented in this thesis sacrifices generality to simplify rule building, providing a way of defining rules by just filling in text boxes and not forcing the business users to write any code.

1.3.2. Database triggers

Database Management Systems (DBMS) evolved dramatically during the 1990s into the active DBMS products commercially available today [6][13][4]. An active DBMS is distinguished by the ability to automatically execute actions against the data or schema of a database [6]. Database triggers are the basis of this functionality, and allow ECA rules to be defined so that when a command is given to the database the event generated by that command may cause a trigger to execute [4]. Microsoft’s SQL Server
product is an example of an active DBMS, and SQL Server 2005 was used as a key component of this project.

In Microsoft SQL Server triggers are implemented as stored procedures that are automatically executed whenever one of the events specified in the trigger definition occurs. SQL Server supports two different types of triggers, Data Manipulation Language (DML) triggers and Data Definition Language (DDL) triggers. DML triggers are defined per table or view and can be tied to insert, update and delete events. DDL triggers are defined per database or server wide and can be tied to create, alter, drop and other database modification (as opposed to data modification) commands [8]. Only DML triggers are used in the implementation of this project since changes to the data, not the schema, are important to the system.

While incredibly powerful, the amount of complexity added to a database by using triggers often causes maintainability concerns. Diaz [4] wrote in his study of the complexity of active DBMSs that users of active systems found the interactions between triggers to make developing and maintaining active databases difficult. The use of triggers in this project does not require triggers to interact with each other and the number of actions that can cause a trigger to fire is kept to a minimum in order to use the functionality of triggers without causing undue complexity to the source database.

1.3.3. Object/Relational Mapping

In conventional data-centric business applications it is an accepted best practice to divide the application up into a layered architecture, usually along the lines of a
presentation layer, a domain logic layer and a data source layer (more commonly called a data access layer) [5]. The data access layer is commonly implemented with objects that make calls to stored procedures in a relational database and then take the results from the stored procedure call and populates data transfer objects that will be used by the other layers. The objects being populated are commonly custom classes or strongly typed datasets. In either case the data access layer usually involves a lot of code mapping the values from the stored procedure to the fields in the objects. This process resolves the differences between the type systems of the relational data store and the object oriented language while shuffling the data back and forth between the relational model and the object model. The data access layer also contains a great deal of set-up code, for example: creating database connections, setting up the parameters for stored procedures, and error handling. In general the data access layer contains a lot of tedious, but crucial code for bridging the gap (often referred to as an “impedance mismatch” [1]) between the relational model and the object oriented model.

Object/Relational mapping (ORM) frameworks are a relatively new concept created to ease the burden of handling the object relational impedance mismatch. The open source Hibernate project was started in 2002 [10] and provides a framework for mapping between relational database and Java classes. Hibernate.org also supports NHibernate, a port of Hibernate that provides mapping to .NET languages. In 2006 Microsoft announced that they were developing an ORM called “Entity Framework” [7] which was released as part of the .NET Framework 3.5 Service Pack 1. Microsoft is certainly not the first company to note the success of Hibernate and provide another
implementation; there are dozens of ORM tools available, for languages from C++ and Delphi to PHP, Ruby and Perl. Some of these implementations use Hibernate under the hood while others are completely new implementations. While Hibernate (and other ORM's derived from it) share a lot of high level features with the Microsoft .NET Entity Framework [10] the focus in this thesis will be on the .NET Entity Framework implementation since that is what was used for this project.

The .NET Entity Framework not only provides a robust object relational mapping system, but it also wraps the whole process up within Microsoft Visual Studio so that for simple scenarios a user simply selects the ADO.NET Entity Data Model template and walks through a wizard to create their entity data model from an existing database. This automatically generated model contains all of the mapping data for the selected tables in the database, and adds the connection data into the configuration file for the application without the user writing a single line of code or XML. Visual Studio also provides a viewer for the entity data model file, which displays the graphical view of the entities and allows you to modify the mapping for each entity in a property page. Behind the scenes the entity data model file is really an XML file which contains a definition for the storage model, the conceptual model, and the mappings between them. (To be completely accurate the entity data model XML also stores the position of the shapes and connectors that are displayed in the graphical view, but this will be ignored since it is not a part of the actual mapping data.) The storage model is what the Entity Framework uses to generate SQL. The conceptual model defines how the Entity Framework will generate the .NET objects that the developer will work with, and this
The model is what is updated if the user wants the domain objects or properties of the domain objects to have names different than the tables or columns in the database. The mapping section provides the necessary data for the Entity Framework to resolve the differences between the two models [9]. This dual model system is necessary to provide the flexibility necessary to span the differences between the objects and the database, enabling not just naming differences, but more complicated features such as inheritance hierarchies [10].

Using the domain objects created by Entity Framework in an application allows one to observe more of the framework’s features. Not only does Entity Framework know how to persist changes made to the domain objects back to the database, but the framework is also keeping track of what changes have been made so that it knows which entities need to be updated [7]. Entity framework also supports multiple ways of managing data concurrency so if the data inside the database has been modified between the time the domain objects were populated and the time the framework attempts to save the changes the appropriate action can be taken [7].

Microsoft .NET Entity Framework also contains a simplified process of querying data called LINQ (Language Integrated Query) to Entities. LINQ itself is an extension to the .NET languages which allows developers to build queries with compile time checking and intellisense instead of writing SQL queries as inline strings. LINQ to Entities is an implementation of LINQ for Entity Framework objects and includes the ability to sort, filter, include related objects (like a SQL join), group by a specific field, and more [7].
2. Approach

2.1. System Overview

The system designed in this project to solve the over age dependent problem is not as generic as it was initially envisioned to be. The original plan allowed the business users to develop any rule they liked and the system would be responsible for handling the SQL generation for any rule the business user created. While this grandiose solution would ideally require very few feature enhancements from the software engineering group in the future, creating such a system would be a massive undertaking as well as a huge risk because it would basically allow business users to write queries against the database. Instead a more moderate design was undertaken. The following table describes the use cases for the implemented system.

<table>
<thead>
<tr>
<th>Actor</th>
<th>Event</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developer</td>
<td>New rule type is desired</td>
<td>The developer must first analyze the conditions for the rule type. From this analysis the developer builds the rule type editor and the rule template for the rule builder component. The developer also defines and applies the trigger necessary for the new rule type to support data consistency. The developer then provides a new release of the rule builder to the business user. It is expected that new rule types will not be added very often.</td>
</tr>
</tbody>
</table>
The developer determines the desired functionality and adds a class to the standard action library to perform the new functionality. This event could potentially have occurred with the original concept of the system, however once a suitable library of actions is developed new actions should not need to be added very frequently.

The user opens the rule builder and selects the option to add a new rule. The user then enters the values necessary in the condition section of the rule and selects an action to be performed by the rule. When complete the user saves and activates the rule. This would happen frequently when new clients are being added to the system.

The user opens the rule builder and selects the rule that needs to be changed. The user deactivates the rule and then makes the necessary changes. Once the changes are complete the user saves the rule and activates it again.

The system creates events as defined by the conditions for the rule type.

The system removes all events for the deactivated rule.

If the data modified has a rule applied to it, or a new item is added to a source table that is the basis for a rule type then the system will remove the invalid event and add a new event.

<table>
<thead>
<tr>
<th>Role</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developer</td>
<td>New action desired</td>
<td>The developer determines the desired functionality and adds a class to the standard action library to perform the new functionality. This event could potentially have occurred with the original concept of the system, however once a suitable library of actions is developed new actions should not need to be added very frequently.</td>
</tr>
<tr>
<td>Analyst (business user)</td>
<td>Add a rule</td>
<td>The user opens the rule builder and selects the option to add a new rule. The user then enters the values necessary in the condition section of the rule and selects an action to be performed by the rule. When complete the user saves and activates the rule. This would happen frequently when new clients are being added to the system.</td>
</tr>
<tr>
<td>Analyst (business user)</td>
<td>Edit a rule</td>
<td>The user opens the rule builder and selects the rule that needs to be changed. The user deactivates the rule and then makes the necessary changes. Once the changes are complete the user saves the rule and activates it again.</td>
</tr>
<tr>
<td>System</td>
<td>Rule Activation</td>
<td>The system creates events as defined by the conditions for the rule type.</td>
</tr>
<tr>
<td>System</td>
<td>Rule Deactivation</td>
<td>The system removes all events for the deactivated rule.</td>
</tr>
<tr>
<td>System</td>
<td>Source data modification</td>
<td>If the data modified has a rule applied to it, or a new item is added to a source table that is the basis for a rule type then the system will remove the invalid event and add a new event.</td>
</tr>
</tbody>
</table>

Table 1: System use cases

The solution implemented for this project is comprised of five main components. The first is the event database which stores not only the events but also the rule definitions. The event database is the underlying component which unites the rule builder and event monitoring service. The second component is the rule builder which provides users with the ability to configure how they want the system to behave by
creating condition-action rules. The rule builder also contains the logic necessary to populate the events database with the initial event data set for a given rule when the user decides to activate a rule. The third component is the event monitoring service which is a windows service that polls the event database for events to process. When an event is found that should be processed the monitoring service dynamically loads the appropriate action from the action library and invokes the method specified in the action definition. The standard action library is a collection of actions and it is only used indirectly via reflection. Finally, the data consistency system provides a mechanism for updating events in the event database whenever a change is made to the source data. The source database is the database that the system operates against. This database is not considered a part of the solution. The results section will contain a more detailed discussion of the source database.

Figure 1: The components of the system
2.2. Event Database

The event database is a simple database built to store the rule definitions as well as the events. The diagram below shows the tables and relationships in the event database.
The event table is the cornerstone for the entire solution and it is used by every other component. Other than its primary key (EventID), the event table contains three crucial columns. The timestamp column stores the earliest time the event is allowed to occur and is used by the event monitoring service to determine if any events are ready to be processed. The rule ID column is a foreign key to the rule table and is used in two
different ways. The first use is to look up the action that needs to occur when the event is being processed. Secondly when a rule is modified all of the events that were created from that rule are removed and then re-created with the updated rule. The ItemID column is an unenforced reference to the source database. This column is used by the data consistency system to determine if an item that is being updated has any events bound to it.

The rule table is also a central table in the event database schema. Not only is it used to relate events to their actions, but it also stores definition of the rule. The name and rule type (a classification necessary for the rule builder) are stored in this table along with whether or not it has been activated (the “Active” column in the table). Thinking of a rule as a condition-action pair, the condition is defined by combining the condition template defined in the rule type with values stored in the rule properties table. The action for the rule is defined by ActionID, a foreign key to the action table.

The action table stores the data necessary to invoke the action using reflection. This includes the name of the .NET assembly, the name of the class to instantiate and the name of the method to invoke.

2.3. Rule Builder

The rule builder is the graphical interface that the business users employ to create the condition-action rules for generating events. The standard layered architecture was used, with each layer implemented as a separate .NET assembly. The figure below shows the layers defined for this project.
An exception to the strict layered architecture was made with the entity objects. Since Microsoft .NET Entity Framework was used for the data access layer the entity objects for the event database are contained in the RuleDesignerDataAccess assembly. In order for the entity objects to be shared between all layers the data access layer is referenced by both the presentation layer and domain layer instead of just the domain layer. However, the concept of the layered architecture is upheld because only the domain layer uses the Entity Framework context object to retrieve, update or remove items from the database. The presentation layer never instantiates the Entity Framework context object; it only uses the reference for the definitions of the entity objects that it will send and receive from the Domain Layer.

A quick note about the architecture of this component:

Using the entity objects as domain objects works fine for this application because it is running as a standalone application on a single physical tier (it would also be fine in a 2-
A tier scenario where the application is on one tier and the database is on a separate tier.

If in the future it was desired to break this application into an n-tier application and expose the domain layer through a web service so that the client application did not need to access the database there would be some difficulties. The inherently stateless nature of web service calls would destroy the ability for the Entity Framework to track changes and manage data concurrency automatically because the Entity Framework’s context object is what tracks the changes to the entities. It would not be feasible to have a static context object like the current solution does because the server would need to store a context for each connected client and handle all of the problems with adding state to a web service [12]. There is a workaround for the context problem; manually marking every property in the entity object as modified after attaching it to a new context object will allow the new context object to properly detect concurrency issues [12]. Although this application would not benefit from such a change, it is important to realize that there are challenges which must be addressed when physically distributing an application that would pass entity objects between physical tiers.

The presentation layer for the rule builder uses Windows Presentation Foundation (WPF) instead of the older windows forms components. This allows the rule builder to utilize the advanced data binding features of WPF. The following screen shot shows the user interface of the rule builder.
The main window contains all of the information that is common between different rule types such as the list of existing rules; the ability to create a new rule and edit the rule name; and the test, activate/deactivate, and save buttons at the bottom of the rule editor panel. The condition and action sections are a dynamically loaded user control that is defined by rule type. This means the developer must define an editor for each rule type and limits the creativity of the user when building rules. However the system gains stability from this restriction - it ensures that the necessary criteria for the condition is provided and allows for more in depth validations than a generic rule syntax.
could provide. For example the Dependent Age Rule template requires an employer name, which must be selected from a list of the employer names in the system, and an age which must be an integer.

Each editor in the presentation layer has a corresponding template object in the domain layer. This template object provides all of the logic for the rule. The template knows how to create rules from the individual field values from the template, and provides methods to retrieve any database data (such as the employer list in the dependent age rule example) for data binding in the presentation layer. Most importantly the template contains the logic for populating the event database when the rule is activated. In addition to template objects, the domain layer for the rule builder contains a rule manager object for populating and refreshing the list of rules, an action manager object for providing the list of available actions, and a static utilities class that contains a property to get the single instance of the event database Entity Framework model’s context object. As mentioned above it is crucial that all of the actions taken against the event database Entity Framework model are invoked against the same context object.

In order to enable the template objects to return lists of data from the source database and to generate the events for the event database, the data access layer for the rule builder is comprised of both an assembly containing the event database Entity Framework model and an assembly containing the source database Entity Framework model. While it is possible to define both of these models in the same assembly they
were separated because only the domain layer needs to reference both; the presentation layer only uses entity objects defined for the event database.

2.4. Event Monitoring Service

Once rules defined in the rule builder have been activated, which causes the event table to be populated, the event monitoring service will start handling events. The event monitoring service follows a layered architecture, with a windows service assembly in the place of a presentation layer as shown below. The event monitoring service follows a strict layering paradigm (unlike the rule builder) meaning that each layer only references the layer directly below it.

![Architecture of the event monitoring service](image)

**Figure 5: The architecture of the event monitoring service**

There is limited user interaction with the windows service that runs the event monitoring service. That interaction can be in the form of modifying the configuration file or through the services management snap-in which allows you to start and stop the service. The configuration file allows the polling interval as well as logging and
simulation flags and data to be modified without recompiling and redeploying the service. When the service is running it uses a system timer to generate a system event based on the time interval set in the configuration file. The event handler for this elapsed time event calls into the logic layer to perform the work.

The logic layer is the heart of the event monitoring service. It determines whether the service is running in simulation mode or real-time and uses the appropriate time to check for events. If events are returned they are handled by invoking action specified for the rule. The action library is completely decoupled in order to allow any new actions to be added without recompiling or redeploying the event monitoring service. However, since dynamically loading assemblies is slow when compared with using an already loaded assembly, the logic layer caches assemblies so that if the action has already been invoked for one event it will exist in the cache and the logic layer will not need to load the assembly again. The logic layer also has the ability to log the actions being handled for tracking or verification. Logging in the logic layer allows the time that was used to determine the eligibility of the event (either real time or simulation time) to be logged as well as the event information.

The data access layer for the event monitoring system was developed against the .NET framework 3.5 without service pack 1, so it is using plain LINQ to SQL instead of the Entity Framework model. There are only two tasks the data access layer performs; retrieving all events from the database that are ready to be handled and deleting these
events when they are complete. Since it is so simple there was no motivation to update it to use Entity Framework.

2.5. Data Consistency System

The data consistency system is responsible for updating events to keep the events valid whenever there is a change. There are two ways that events could become invalid, either the rule could be updated or the source database could be updated. In the first case the rule builder enforces data consistency by requiring that a rule be deactivated before changes are saved. For example, Maryland state law just recently mandated that all employers must support dependents until age 25. When the business user opens an active rule to edit it the “save” button is disabled. The user must deactivate the rule, which removes all of the events in the event table associated with the rule, before the changes to the rule can be saved. Once the changes are complete the user would reactivate the rule, causing the rule builder to populate the event table with the appropriate events for the updated rule.

The second facet of the data consistency system handles changes to the source database. This is completely invisible to the end user and requires the developer to define one or more triggers on the appropriate table(s) in the source database to ensure that when the source data is changed the event is updated. The developer needs to define these triggers by rule type, for example the system currently has only one rule type of “Dependent Age Rule” and once the triggers for this rule type are defined they do not need to be updated, regardless of how many instances of the rule type are
defined. The steps necessary for defining the data consistency triggers for a rule type are described in the table below.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Determine (based on knowledge of the rule type template) which table(s) in the database contain fields on which the rule type depends.</td>
<td>The dependent age rule type depends on the Dependent table when an insert or deletion is made and the Person table when an update is made. The dependence on both tables is necessary because the Person table contains the person’s date of birth while the Dependent table contains the actual record representing the child.</td>
</tr>
<tr>
<td>2</td>
<td>Consider how to handle deleted items – the “ItemID” key in the event table is the means for determining which item in the source database the event is based upon.</td>
<td>This is very straightforward for the dependent age rule type because the primary key on the Dependent table is both the item id for the event table and the same as the primary key on the Person table. This means that when a Dependent is deleted we simply delete the event where ItemID = PersonID.</td>
</tr>
<tr>
<td>3</td>
<td>Establish a way to select the proper rule instance from all of the rules built off of the rule type. The trigger must be able to determine which rule to use based off of the record being inserted.</td>
<td>For the dependent age rule the restriction of only one rule per employer is critical for enabling the data consistency system to determine from the source database which rule should be used to create the new event.</td>
</tr>
<tr>
<td>4</td>
<td>Once the proper rule can be selected add the ability to insert new events</td>
<td>The dependent age rule insert is fairly complex; it joins most of the tables in the source database in order to verify the dependent is a child and to find the employer of the employee to which the child belongs. The employer is the key for determining which rule to use, and the rule specifies the number of years to add to the child’s date of birth when generating the event.</td>
</tr>
<tr>
<td>5</td>
<td>Verify that the delete and insert activities will support updating (which is implemented as a delete followed by an insert)</td>
<td>In this example the delete and insert are applicable only to the Dependent table. The trigger that watches for updates needs to be applied to the Person table since that is where the date of birth will be changed. However, the trigger to handle updates applied to the Person table for the dependent age rule was able to use basically the same logic as the trigger built for the Dependent table. It is important to note that the trigger added to the person table only is fired on the update event because the trigger on the dependent table covers additions and deletions. Minimizing the number of ways that triggers will be executed is important to keep the system maintainable.</td>
</tr>
</tbody>
</table>

Table 2: Steps for ensuring data consistency
3. Testing and Results

3.1. Source Database

To test the system it was necessary to create and populate a test database. This database, referred to as the source database, represents the database that would store employee demographic information as well as enrollments in a real enrollment system. The source database created to test this project was designed to be as simple as possible. Since the only rule we are testing is for over age dependents the source database stores demographic information about dependents. Also, since over age dependent rules are defined by employer it was necessary to provide a way to relate dependents to the employer (through an employee) so that the appropriate rule can be applied. The schema for the source database is shown below.
An Entity Framework mapping for this database was created for use by the logic layer in the rule builder and re-used in the test data generator. This mapping makes use of the ability to define table-per-type inheritance to mask the normalization done in the database and make employee entity objects and dependent entity objects inherit from the person entity object instead of having to update the person as a separately. The
entity diagram below shows how the relationships were modeled after implementing table-per-type inheritance.

Figure 7: Entity model of the source database
3.2. Generating Test Data

In order to populate the source database a small application was built and using the entities discussed above for data access the database was populated with employers, employees, dependent spouses and dependent children. The following parameters were used when generating the test data:

<table>
<thead>
<tr>
<th>Entity</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employer</td>
<td>10 employers with hard coded names</td>
</tr>
<tr>
<td>Employee</td>
<td>Number of Employees per Employer: Random number in the range[500-5000]</td>
</tr>
<tr>
<td></td>
<td>Person: Adult Profile</td>
</tr>
<tr>
<td></td>
<td>Hire Date: Random date at least 16 years after date of birth</td>
</tr>
<tr>
<td>Spouse</td>
<td>Zero or one spouse per employee (66% chance of having a spouse)</td>
</tr>
<tr>
<td></td>
<td>Person: Adult Profile with gender opposite of employee</td>
</tr>
<tr>
<td></td>
<td>Dependent Type: Spouse</td>
</tr>
<tr>
<td>Child</td>
<td>Between zero and five children per employee. If the random draw for number of children is greater than 3 there is a 50% chance that the number is divided in half (truncated) to skew the distribution towards fewer kids.</td>
</tr>
<tr>
<td></td>
<td>Person: Child Profile</td>
</tr>
<tr>
<td></td>
<td>Dependent Type: Child</td>
</tr>
<tr>
<td>Adult Person Profile</td>
<td>Gender: Random (except spouse)</td>
</tr>
<tr>
<td></td>
<td>First Name: Random draw from gender specific list</td>
</tr>
<tr>
<td></td>
<td>Last Name: Random draw from list</td>
</tr>
<tr>
<td></td>
<td>Date of Birth: Random age in the range that would make the adult between 18 and 70 as of the date the data generator is executed</td>
</tr>
<tr>
<td>Child Person Profile</td>
<td>Gender: Random</td>
</tr>
<tr>
<td></td>
<td>First Name: Random draw from gender specific list</td>
</tr>
<tr>
<td></td>
<td>Last Name: Random draw from list</td>
</tr>
<tr>
<td></td>
<td>Date of Birth: Random age in the range that would make the adult between 0 and 30 as of the date the data generator is executed</td>
</tr>
</tbody>
</table>

Table 3: Parameters used for populating the source data
While this data does not contain many of the exceptions that a real employee population might have (such as domestic partners, or “other” dependents such as mentally handicapped adults) it provides sufficient variety to test the system. After running the data generator the source database had the following populations for employees and dependent children per employer:
Figure 8: Test data from the source database
3.3. Test Scenario 1: Static Test

The first test devised for this system validates whether the rule builder and event monitoring system are working properly. First a set of rules are built and the rules are activated, which populates the event database. Then the event monitoring system is run using simulation mode to simulate the passage of time at a rate of one simulated day per second. The event monitoring system logs every event that it processes and the simulation time at which it was handled. The data gathered for comparison contains all of the events scheduled for the year 2010. In order to verify that each rule is applied only to the dependents of the specified employer each rule criteria contains a different age. The table below details the rules created for this test.

<table>
<thead>
<tr>
<th>Rule Name</th>
<th>Employer Affected</th>
<th>Age at Which Children are no Longer Covered</th>
<th>Events Occurring in 2010 Pertain to Children Born in:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andy Corp – 16</td>
<td>Andy Corp</td>
<td>16</td>
<td>1994</td>
</tr>
<tr>
<td>Another Company – 17</td>
<td>Another Company</td>
<td>17</td>
<td>1993</td>
</tr>
<tr>
<td>Christina Corp – 18</td>
<td>Christina Corp</td>
<td>18</td>
<td>1992</td>
</tr>
<tr>
<td>Finish – 19</td>
<td>Finish</td>
<td>19</td>
<td>1991</td>
</tr>
<tr>
<td>My Company – 20</td>
<td>My Company</td>
<td>20</td>
<td>1990</td>
</tr>
<tr>
<td>Steidle Solns – 22</td>
<td>Steidle Solutions</td>
<td>22</td>
<td>1988</td>
</tr>
<tr>
<td>Test – 23</td>
<td>Test Employer</td>
<td>23</td>
<td>1987</td>
</tr>
<tr>
<td>VGL – 24</td>
<td>Valhalla Games</td>
<td>24</td>
<td>1986</td>
</tr>
<tr>
<td>VSL – 25</td>
<td>Visionary Simulations</td>
<td>25</td>
<td>1985</td>
</tr>
</tbody>
</table>

Table 4: Static test setup
Since the rule for each employer specifies a different age the task of gathering
the expected result set for this test was a little more involved than a simple query
against the source database. For each month in 2010, the number of events processed
is expected to be the sum of the number of child dependents born in that month, during
the year specified in the table above. The following graph shows the expected results
broken out by employer.
Figure 9: Source data - expected over age dependent
After running the test case, the following results were generated which show that the rule builder is correctly generating events when the rules are activated and that the event monitoring service handled all of the events that were generated in the appropriate timeframe.
Simulated Event Handling for the Year 2010 by Month

Figure 10: Results from the static test
The results data came from three distinct sources. The expected event count was gathered from the source database. The event count for each month was queried from the event database after all of the rules were activated. Finally, the number of events executed per month was collected from the event monitoring service log file.

3.4. Test Scenario 2: Dynamic Tests

The second test scenario exercises the data consistency system. This system is responsible for making sure that any changes made to the source data or the rule itself reflect that change to the events. First the ability to update rules is tested. One rule, “School Projects – 21” was updated multiple times to specify various ages. Each time the rule was activated after the change was made and the number of events generated was counted by month. The employer was held constant so that the expected number of events would just shift with the age change. The expected number of events was gathered by querying the source database for the number of children born each year during the range of 1986 – 1990, as shown in the graph below.
Figure 11: Source data for comparison against event data
The data collected from the event table shows that each time a rule is updated and re-activated the previous events are removed and the new events correspond to the update made to the rule. Each data series on the results graph below can be mapped to a segment on the source data graph above.
Figure 12: Results for the rule update test
The second test of the data consistency system validates that changes to the source database made between the time that the test rule is activated and the time when the event monitoring service handles the event are reflected back into the event data. A small set of dependents who would all become over age in August of 2009 were selected from the source database to be the set of test subjects. The original date of birth and original event date created by the dependent age rule were collected from the source database and event database respectively. Then these test subjects were used to test that the data consistency system can support all three modifying actions: inserting new child dependents into the source database, updating existing child dependents, and deleting child dependents from the source database. The table below shows how each modifying activity was tested and which test subjects were used for each type of modifying activity.

<table>
<thead>
<tr>
<th>Action Taken</th>
<th>Subject Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>Two new dependents were created in addition to the original set. The date of birth for these new dependents was set so that they would become overage in August 2009</td>
</tr>
<tr>
<td>Delete</td>
<td>Three of the original dependents were deleted from the source database.</td>
</tr>
<tr>
<td>Update</td>
<td>The date of birth was updated for each of the remaining original members of the test set. A random number of days in the range [-5,5) was added to the date of birth.</td>
</tr>
</tbody>
</table>

Table 5: Test scenarios for verifying the data consistency system

Once the modifying actions had been applied the event monitoring system was enabled and processed the events. The source database was queried to retrieve the
updated birth dates and the event monitoring system log was used to determine the simulated date on which the event was handled. The data gathered was used to build the following chart showing how each test subject was modified and how the data consistency system handled all three modifying actions correctly.
Figure 13: Results from the tracking changes to the source database test
4. Conclusion

This thesis demonstrates that a system which enables business users to define event-condition-action rules to trigger actions based on the passage of time can be constructed. The first test case confirms that the system developed correctly creates events and handles them at the appropriate time. The second test validated the system’s ability to manage the dynamic nature of the data to provide accurate events regardless of the changes made to the source data or the rule definitions. While the software developed for this project does not provide a generic solution for defining and managing time based business rules it does provide a framework which allows additional rule types to be added to support new functionality. The following section expands on the benefits of this decision as well as other lessons learned while completing this project.

4.1. Lessons Learned

Initially the solution for this problem was envisioned to be completely generic, allowing business users to define any rule that they could think of. The imagined system would be able to translate any rule into a query which would run against the source data to generate the events. The triggers for the data consistency system would be created automatically by the system as well. The source database itself would be decoupled from the system with the entity model dynamically loaded using reflection to provide objects for the business users to use when defining rules. A lot of time was
invested in trying to create this generic solution before it became apparent that perhaps a less generic solution would suffice. Not only was this focused solution created in a fraction of the time it would have taken to create the generic solution, it is also much easier to validate and easier for the business users to use. A completely generic system would have given the business users more freedom but at the same time would have made them responsible for creating business rules that were fully specified even though they only have a partial understanding of the source database. Specifying a UI “editor” for each rule type allows the developer to control what is used to define the rule. The rule templates enable the rule builder to enforce uniqueness criteria for each rule so that the data consistency system can determine which rule should be used to generate events when data is added to the source system. While this implementation only provides a small subset of functionality imagined the original concept it provides a first step which can be expanded upon in future iterations.

The fact that the dependent age rule type may be the only rule type that ever needs to be implemented in this system provides evidence that agile methods may provide solutions that are more aligned with business needs than the traditional waterfall method. Assuming that this implementation was the first iteration and that additional functionality could be added later enabled this project to overcome the paralysis by analysis state it was stuck in a few months ago. Another practice common to agile methodologies is the use of unit testing. Unit testing proponents explain the practice as a way of reducing the fear of making code changes by reducing the risk of breaking the existing functionality. While a full suite of unit tests was not developed for this
application the NUnit framework was used for creating and running unit tests for the logic layer of the event monitoring service. The use of unit testing was extremely beneficial in this case because it provided a way of exercising the logic layer without deploying, starting, and attaching to the windows service. It also provided a way of quickly verifying that the logic layer was still working correctly after changes were made. While the purpose of this thesis is not to examine the benefits or drawbacks of various software development processes, learning about the benefits of some of the agile practices turned out to be beneficial in the development of this solution.

4.2. Future Work

The most important addition for the next iteration of this project would be to add more variety to the standard action library. Currently there is only one test action that does not have any configuration, which means there is no variety in the final outcome of the rules. Developing a robust library of actions, and expanding the rule builder to include editors for actions so that the actions can be applied in various ways would be a vast improvement. For example a “send email” action could be built, and the rule builder would allow the business user to specify who the email should go to, either as a static email address or a variable address that would be different for each event, such as the employee’s email address. Another useful action would be an “invoke web service action” which would enable the event to kick off any processing that could be defined as a web service, including human centric business processes exposed as web services.

Another action that would be beneficial, but not exposed to the business users, is an action for “continuation”. Rules would use this action by default to reduce the number
of events that need to be added to the event database. The rule would only add the near future events to the event table and the last event added for that rule would be a continue event which would cause the next batch of events to be added. For example the current system will generate events for all underage dependent children, so if a child is two years old and will become over age at age twenty-five that event would (in theory) sit in the event table for twenty-three years (the system certainly will not be around that long) before it is needed. The continue event would make the system much more storage space efficient. However, there is a balance to be maintained - the continuation event should not occur too frequently because the whole purpose of the event table is to prevent the event monitoring system from constantly checking against the source database. Another solution to consider is specific to the over age dependent scenario. Employee benefit enrollment is usually an annual process, and benefits often change for each new plan year. It would make sense in the dependent age rule scenario to limit the events generated by the rule to only apply to dependents becoming over age during the current benefit plan year. This would also mean that a copy of the rule, or a new rule would be required for each plan year, and activating this rule would become a part of the configuration process for each open enrollment season. Perhaps an ideal implementation would be to define rules for each benefit plan year and add monthly continuation events.

A potential flaw in this system is that it has no support for use across multiple time zones. It is not a major concern for the dependent age rules because they are dealing with time on the order of days, instead of minutes, but this could cause
problems in other scenarios. For example, say this system was being used as part of an order management system to implement a rule that says all orders must be held for one hour after being submitted before being processed. If an order is submitted at 2 pm Eastern Time but the order is added to the database with a local which happens to be Pacific Time (11 am), and the event monitoring service is using Eastern Time the event monitoring service would process the event immediately because the event time would be 12 pm. Or the reverse could happen and a four hour delay could result. If this system is going to be used in conjunction with servers in other time zones for processes that are time sensitive it is important to make sure all of the times are created using Coordinated Universal Time.
5. References


5. Fowler, Martin, Patterns of Enterprise Application Architecture, Addison-Wesley (2003), pp 19-22


