Making Sense of Healthcare Benefits

Jonathan Bnayahu, Maayan Goldstein, Mordechai Nisenson, Yahalomit Simionovici
IBM Haifa Research Lab
Haifa University Campus, Mount Carmel,
Haifa, Israel 31905
{*bnayahu,maayang,motinis*}*@il.ibm.com, syally@gmail.com*

***Abstract*—A key piece of information in healthcare is a patient's benefit plan. It details which treatments and procedures are covered by the health insurer (or payer), and at which conditions. While the most accurate and complete implementation of the plan resides in the payers claims adjudication systems, the inherent complexity of these systems forces payers to maintain multiple repositories of benefit information for other service and regulatory needs. In this paper we present a technology that deals with this complexity. We show how a large US health payer benefited from using the visualization, search, summarization and other capabilities of the technology. We argue that this technology can be used to improve productivity and reduce error rate in the benefits administration workflow, leading to lower administrative overhead and cost for health payers, which benefits both payers and patients.**

I. Introduction

Health insurers (or payers) were early innovators in using computers to automate tasks that had generally been reliant on complex human judgment. Inside of their claim processing systems, they created domain-specific rules-based languages and data structures that could evaluate claims against contractual rules for coverage, eligibility, and complex liability and payment structures. Those languages enabled early adopters to drive down their costs to process claims by reducing the dependence on human claim evaluation processes. Most of these domain-specific languages resemble languages currently used by rules engines. However, because they were specialized languages developed by many system vendors and insurers, they are islands of "dead language" that have locked insurers into their legacy systems. Because the languages were designed for efficient processing of hundreds of thousands of rules, they are extremely terse and significantly less "readable" than traditional programming languages. No language exists in more than a few dozen systems, or is well understood by more than a few dozen specialists - benefits coders - at any health insurer.

The combination of the obscurity of the language and the complexity and volume of the benefits coded is a huge source of pain. It takes years to train a coder, setting up a single contract can take three months or more to code and test, with a typical insurer processing tens or hundreds of thousands of contracts.

In addition to implementing benefit rules, health insurers typically need to maintain additional representations of essentially the same contents, in different formats and levels of abstraction. For example:

- Client friendly summaries of benefits, which are generally provided along with the policy documents to the subscribers.
- Legal or regulatory benefit summaries, which are submitted to regulatory bodies (e.g., the states).
- Eligibility summaries, to support Electronic Data Interchange (EDI) with other external systems.
- Benefit summaries for use by member services portals and call centers.

Each of these different artifacts could be considered a different "view" on what the policy actually covers, whereas the policy itself provides the most detailed specification of the terms and conditions of the insurance. Since these are created and maintained by people (and typically by different people), there are difficulties in ensuring that each of these views is consistent. The implementation of the benefit rules in the claims adjudication system is unique amongst these different views in that it is the only one that determines what actually occurs when claims are processed. While the policy document or a summary may state that "out-patient surgery[1] is covered," there may be specific exceptions, such as elective or experimental procedures, which do not appear in either of them. Thus, even when they appear to be clear, these other sources of information may in fact be ambiguous. Furthermore, each of these views must be setup and maintained, which beyond adding to operating costs (and thus likely affecting subscriber premiums) is non-trivial and could result in incorrect or unreliable information being given to patients and providers.

Medical providers and facilities are turning to consultants for assistance with claim editing, tracking and management [1]–[3]. Karen Bowden from ClaimTrust [1] says "Most hospitals still lose as much as six percent of their hard-earned revenue to preventable denials that are never appealed... In fact, for many claims, most payer organizations cannot define a clean claim or even know if a particular claim will be denied or not by their adjudication system." We believe that extraction of benefit rules could give a clear and

[1]Out-patient surgery is a surgery that does not require an overnight hospital stay.

concise representation of policies and therefore improve the providers' responsiveness to patients' claims, increase their revenue and lower the premiums payed by the patients.

We identified two major challenges in the extraction of benefit rules from claim adjudication systems. One challenge is that each system uses different, and often proprietary semantics for setting up benefits. These include, for example, the terminology used, the granularity of the rules, their expressiveness, as well as the system's architecture and platform. The second challenge is the need to transform and abstract benefit rules into human-readable summaries. The abstraction requires selecting the appropriate segments of the implemented benefit rule, translating these segments into proper English sentences and aggregating them into coherent benefit summaries.

Both of these challenges make it difficult for payers to use the rules implementation as the absolute source of truth for benefit details, both internally and when required to provide the information externally.

In this paper we present a technology to overcome these challenges. The technology combines information retrieval algorithms [4]–[11] with static analysis techniques [12]–[15] to provide a number of functionalities: a capability to extract the control flow and data of a benefit rule; a capability to visualize a rule; a capability to search benefit rules and a capability to summarize benefit rules.

The remainder of this article is organized as follows. Section II details the technological challenges we identified in the area of benefit rules extraction. Section III presents our methodology and solution architecture. Section IV pauses briefly to provide technical background for our algorithms and proceeds with the description of how the suggested solution works. Section V describes how it was validated on a large customer system. Section VI describes related work in this field, whereas Section VII gives our conclusions and discusses future directions.

II. CHALLENGES

Our work focused on extracting benefit information from existing claim adjudication systems. The two major challenges identified above can be further expanded into the following:

1) *Lack of standardization*: there is a wide variety of proprietary languages and paradigms for benefits coding.
2) *Level of detail*: the code has significantly more detail than can easily be understood by a non-coder.
3) *Terminology inconsistency*: the implementation of claims processing systems uses formal semantics as opposed to natural language description.
4) *Code complexity*: dead code, "spaghetti coding" and other poor programming practices may complicate the code and make it difficult to comprehend.

We now take a deeper look at each of these challenges in turn.

A. The Lack of Standardization Challenge

There is no one standard for benefits systems. The common state is for insurance companies (or payers) to have multiple proprietary systems. This results from payers having developed their own systems, customizing acquired systems, and from mergers and acquisitions between payers. While all systems have the same end result of determining how much the payer will cover, they may do so in different fashions. These changes go beyond the trivial and to be expected cases of differing internal representations of standard concepts such as service type and diagnosis class. Indeed, the programming paradigm itself may be completely different from one system to the next. Some are written using imperative style domain-specific languages, others use table-based rule systems, still others use pre-configured rules written in COBOL. In order to extract information from these systems it is necessary to be able to cope with these different paradigms and the accompanying differences in storage formats and semantics.

B. The Level of Detail Challenge

Assuming that we are able to "translate" the code into English, the next challenge deals with the fact that the code may be overly detailed for the task at hand. Thus, for example, there will be code dealing with different places of treatment, specific procedures, different cases of patient history, and even state mandates. Indeed, dozens of distinct cases may be handled by a single "function" in the code. It is necessary to present a digestible version of this code to the end-user. The challenge here is how to determine which details shouldn't be presented. Clearly, this question is sensitive to the end-user's context. For example, is the question what is the coinsurance[2] in general for a surgery or what is it for a specific procedure at a specific location?

C. The Terminology Inconsistency Challenge

When patients call customer service with an inquiry, the customer service representative must be able to answer their questions in an accurate and timely manner. For this purpose, databases are often created, which contain summaries of patient benefits. These summaries are written in a natural language (usually English), and are often a mix between tables showing how deductibles apply and coinsurance amounts for different types of treatments and provider networks, and free-form text covering exceptions and special cases. The summaries are created manually and need to be updated every time there is a change in the code that processes the claims. To assure consistency, we need to replace these manually-created summaries with automatically-generated ones. This requires sophisticated mapping mechanisms to convert benefits written in some

[2]Co-insurance is a percentage of the total cost that insured person may pay for a procedure.

form of programming language, which is understandable to the computer, and perhaps to a specially-trained coder (who develops the benefits code), to a form comprehendible by an average person.

D. The Code Complexity Challenge

Often code may suffer from various poor programming practices. Copying code from a similar policy and then editing it may result in dead code being present. Spaghetti coding, where the code is disorganized structurally can also occur as a result of copying and editing or as a result of patches done to fix errors. Introduction of any new treatment or procedure that needs to be covered by the policy frequently results in inserting the relevant code into multiple places in the system, without reorganizing and restructuring it to improve its readability and design. Merely providing a more legible translation of the code does not sufficiently relieve the burden of understanding the code. Tool support to remove dead code and disentangle code paths is necessary.

In this paper we focus on the first three challenges. However, techniques used for the analysis of the claim processing systems' code can help coping with the code complexity challenge as well.

III. Solution Architecture

In this section we describe the technology we designed and developed to cope with the aforementioned challenges. The technology is used to translate benefit rules into human readable descriptions. This capability can be used by coders that need to modify a specific part of the system and need to understand the meaning of it. It can also be used by patients that want to learn which benefits they are entitled to. Another important functionality of the technology is the summarization capability which allows to present general description of what is covered by a policy without getting into specifics on different limitations. Furthermore, the support for visualization of control flow of benefit rules is most advantageous for the coders who use this capability to investigate the code. Finally, a search engine is used to look up parts of policies and to get information on, for example, coverage of specific treatment for a specific client.

A. Information Extraction and Mapping

There are two main steps in benefit analysis and processing. The first step is an off-line step that is performed on system documentation and code that the insurance company maintains to create a repository of benefits. During this step, the entire legacy system is formatted and indexed to allow fast and efficient access to the data. The second step is an online step that is performed repeatedly by different users of the system such as coders and patients.

Figure 1 presents the architecture of the claims processing engine of the technology. It consists of four main components, whereas the first three are in use during the off-line processing step and the last one is used during the online step:

1) An interface for specification of the claim adjudication system setup. This component is responsible for extracting control flow graphs of claim processing rules and for mapping parameters used within those rules into natural language terms.
2) A customizable engine for extracting system-specific benefit data. The engine is responsible for extracting benefit rules based on specific implementation of policies that are made by coders developing the claims processing system.
3) A common benefit repository. This repository stores data required during any of the aforementioned capabilities such as summaries and control flow graph representation, and search queries.
4) An analytical engine for producing benefit summaries. This engine creates summaries of benefits in natural language and is described in details in Section IV.

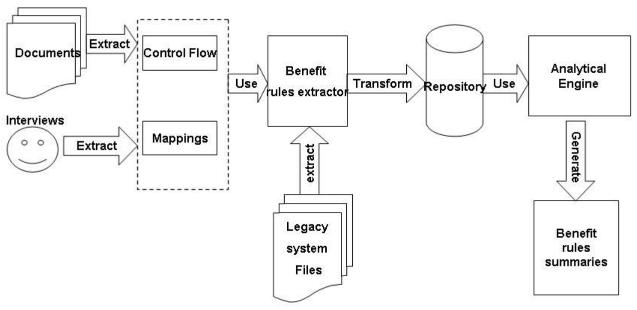

Fig. 1. Analysis flow in the system.

Each claim adjudication system configures the benefit rules differently. For example, imperative style Domain Specific Language (DSL) systems may define two types of commands: *decision making commands*, e.g. "is the patient's age over 18?", and *action commands*, e.g. "the resulting copay[3] for a prescription medicine is $50". For this type of languages, the control-flow of a given benefit rule is the sequence of decision making commands which the claim applies to. The resulting action commands represent the resulting coverage.

A second kind of system sets up benefit rules using a sequence of tables. Usually each table contains different category of details, such as a membership table, a medical service table, a provider table, etc. For examples, Table I shows a snippet of a benefit rule as it appears in one of the tables, with slight modifications made for confidentiality reasons. The rule described by the table is translated to "Go to subtable 36 if at least one historical claim was found for

[3]Co-payment is the amount that the patient must pay before the health insurer pays for a particular service.

the same service period, where first date of service (FDS) is between January 1, 2001 (010101) and December 31, 2020 (201231) and procedure table (PT) is FLU SHOTS (817) and network indicator (NETI) is In".

Tab. I
SNIPPET FROM A BENEFIT RULE CODED IN A PROPRIETARY SYSTEM.

Qualifier	Begin	End	Subtable
GOTO			36
FDS	010101	201231	
PT	817	817	
NETI	In	In	

In the example shown in Table I the control flow is both inter-table (i.e. there is dependency in the decision making between fields in the table) and intra-tables (i.e. one table decisions leads to another). Therefore, one needs to define the control flow rules of the system in terms of tables and fields inside the tables. In some cases the rules are more generic (for example in DSL systems). In other cases, it is required to specify how each coverage element is computed separately.

Consequently, the process of extracting control flow graphs representing claim processing rules is done by first manually inspecting the design of the insurance company claim processing system and then implementing adaptors that convert the inspected artifacts (e.g., tables) into control flow graphs. As a result of this process, a repository which holds the extracted control flow graphs of the benefit rules for each client is generated.

The insurance company is then asked to provide accurate mappings of the claim system's internal representations to external representation which is normally a natural-language. This mapping is then used by our technology to calculate a natural-language summary of the benefit rules. There are in fact two types of mapping that need to be performed: internal to external codes mapping and internal to external terms mapping. The first type of mapping is required as each claim adjudication system defines it's own medical services, places of treatment, providers, etc. In order to provide standard representation of the benefit rules, we must obtain a mapping between the internal representations and the external (standardized) ones. For example, the code *14* representing a "broken leg" diagnosis in the insurance company's system needs to be mapped to code *127* representing the same diagnosis in a hospital's system.

The second type of mapping is required to translate the claims encoded in a way that is clear to the machines processing them, but not necessarily the patients trying to understand what their policy covers. This mapping can provide a translation from machine code to natural language. For example, the term *gen* in one system can be mapped into the patient's "gender", and the term *F* can then be mapped to "female". It may also provide a semantical meaning for a specific term. For example, the terms *TR586* and *TR587* may represent drugs used to cure the common flu. When presented to the end user, it might be useful to specify that these terms are drugs for a specific decease without providing exact explanation on what they are.

B. Overall Architecture

The high level architecture of the technology we have developed is presented in Figure 2. The end users access the tool via a web based interface. This interface is a part of a web application running on a Jetty Web server [16]. The user invokes one of the capabilities shown in ellipses. As the result, our tool accesses the relevant data stored on a server in the benefits repository or as retrieved by the search engine. The results are then presented back to the user.

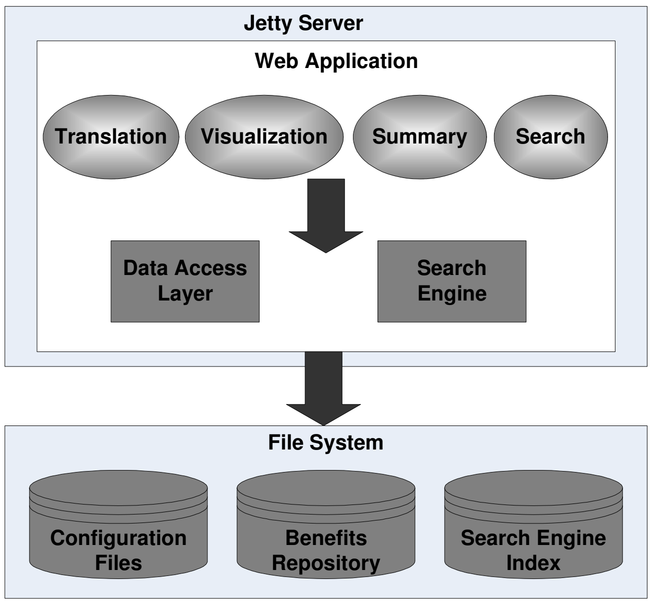

Fig. 2. The high level architecture of our technology.

The benefits repository is used to store and provide access to the logical entities (such as a procedure) and, depending on the analytical capabilities needed, to meta-data such as dependencies between entities. The base requirements for data access are fairly simple; all entities of a certain type are to be retrieved, or a specific entity is to be retrieved based on a single key. These requirements are easy to meet even when only using a file-system. However, more advanced capabilities such as impact analysis would require more varied ways of accessing an entity and relational databases are then a good fit. Due to the fact that the system does not need to provide updates back to the source system, our design allows the run-time to be read-only. Thus, even if several systems are to be deployed to ensure availability and to scale to large numbers of users, a local or embedded database can be used when these more advanced capabilities are needed.

The Apache Lucene search engine [17] is used as the underlying index and search mechanism for various features such as locating code for a specific policy by the account holder's name. It is also used to support advanced search features enabling both structured and free text searches for code reuse. It is accessed through its Java APIs and generates files which are stored on the disk (represented as the Search Engine Index in Figure 2).

The configuration files store various configuration data for the system, such as the web server configuration, where input data can be found, and how users should be authenticated.

IV. Implementation

This section takes a deep dive into the implementation of benefit summarization and search capabilities. We first cover some background on dependency graphs and slicing. This background is prominent for understanding how the benefit rules summarization algorithm works. We then present the summarization algorithm. Finally, we explain how the benefits search engine was implemented.

A. Benefit Rules Summarization

The summarization capability is based on aggregation of rules from the common repository. The level of aggregation depends on the required level of detail. In some cases, it may be necessary to generate a very low level rule which correspond to a specific procedure, diagnosis, etc. In this case, there is no aggregation and the summarized rule is simply the rule derived from a specific entry in the repository. However, in most cases it is required to aggregate the benefit rules which address the same medical service.

Despite being implemented in different forms and using different semantics, all benefit rules we examined share a common logical control flow graph. A control flow graph is a graph based notation of all paths that might be traversed during execution of a program. Consider, for example, the following scenario: Mr. Smith has some pain in his knee and his doctor wants to have an MRI scan of the knee. A control flow graph representing the relevant benefit rule is demonstrated in Figure 3. The control flow is comprised of a series of conditions on the attributes of a claim, and results in either specific coverage values or a rejection of the claim. The graph shows that the MRI scan coded as service with type *37*, is covered with relevant copay and coinsurance values, assuming Mr. Smith is a grown up and has not had another MRI scan during the last year.

The control flow graph in figure 3 is quite simple and easy to comprehend. However, in many cases, there may be dozens and even hundreds of conditions representing various limitations and exceptions for a specific procedure coverage or when the copay and coinsurance change. Therefore, it is important to be able to present a general statement saying whether a procedure is covered or not, taking, for example, into consideration where the procedure is to be performed. The reason for that is quite obvious, the patient wants to know how much he will have to pay if he has the procedure in one hospital or another (hospitals that have a contract with the insurance company are considered "in-network" providers and will normally have a lower coinsurance). But it may be less important to have the age limitation presented to the patient.

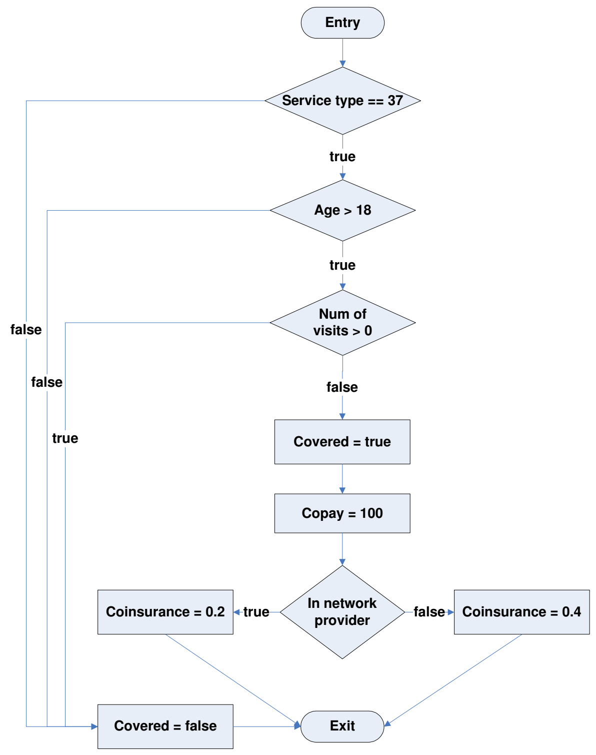

Fig. 3. An example of a benefit rule control flow graph.

The benefit summarization algorithm therefore works as follows: it traverses the control flow graphs representing different rules using slicing techniques [12]–[15], [18] and aggregates the conditions relevant to, for example, copay and coinsurance. The algorithm then applies a list of heuristics that simplify and re-organize the conditions to produce the benefit summary.

Let G be a control flow graph of a program (or a benefit rule's code). *Slicing* is a set of program statements that (potentially) affect the variable values that are referenced at some locations in the program's code. A slice could be readily computed for a given vertex $v \in G$ based on *control dependence paths* of a program [15]. Intuitively, node n is *control-dependent* on a node $m \in G$ if m determines whether n is executed or not. Control dependence paths $CDPaths(n)$ can then be defined as a collection of all paths between nodes $m_i \in G$, such that n is control dependent on them.

A method for computing control dependence edges for arbitrary programs is presented in [18]. We used a simplified version of this approach as in the specific cases we analyzed there were no loops and the control flow graphs could be traversed to determine control paths in linear time.

Let $CD(n) = m_i$ be all the nodes such that the node n is control dependant on them. The control dependence paths $CDPaths(n)$ for node n can then be computed as shown in Figure 4.

```
1.  $CDPaths(n, G)$ {
2.    // $n$ – node for which control paths are calculated
3.    // $G$ – control flow graph representation of benefits claim rule
4.    // Initialization:
5.    $paths(n) \leftarrow new\ node[][]$;
6.    // Main loop:
7.    $\forall m_i \in CD(n)$ {
8.      $paths = CDPaths(m_i, G)$;
9.      if !$isUsed(m_i)$ {
10.       $\forall path \in paths$
11.           $path \leftarrow path \cup m_i$
12.     }
13.     $paths(n) \leftarrow paths(n) \cup paths$
14.   }
15.   $isUsed(n) \leftarrow true$
16.   return $paths(n)$;
17. }
```

Fig. 4. Pseudo-code for computing control dependence paths for a node.

This is a recursive algorithm that traverses the control flow graph in the reverse direction starting from node n. At each step it tries to compute the control paths of the nodes that n is control dependent on them. The algorithm terminates when all nodes that n is control dependant on are analyzed and accumulated within the computed paths.

In order to compute summaries of rules with respect to specific variables, such as copay and coinsurance, we need to first determine what are the vertices we wish to compute a slice for. In the example shown in Figure 3 assume that we want to learn what is the coinsurance and copay that Mr. Smith will have to pay. The vertices we are interested in are those that assign values to copay and coinsurance. We compute the slice for those vertices as shown in Figure 5.

For each node n that we chose to focus on, the algorithm iterates over the control dependance paths for such node and constructs a slice of all the nodes (and edges connecting those node) that affect n. Once the slice is computed, the processing engine translates the result into natural language, using the mappings defined prior to the analysis. It also applies some heuristics to simplify the summary. For example, one of the heuristics specifies that the age condition (*Age* $>$ *18*) that appears in the control flow graph is not important for the summary, as this purely means that the person for whom we present the summary is an adult. Note that in other cases it may be important to take the age condition into consideration as it may, for instance, determine the amount of specific drug that should be given to a child or an infant. This gives us a natural language description of the benefit rule. In case of Mr. Smith, the result looks like this:

```
1.  $Slice(N, G, CDPaths)$ {
2.    // $N$ – set of nodes for which slice is calculated
3.    // $G$ – control flow graph representation of benefits claim rule
4.    // $CDPaths$ – dependence paths for nodes in $G$
5.    // $S, W$ – sets of nodes in G
6.    // Initialization:
7.    $W \leftarrow N$; $S \leftarrow \varnothing$;
8.    // Main loop:
9.    while $W \neq \varnothing$ do {
10.     Select a node $n$ from $W$ and remove it
11.     $S = S \cup n$;
12.     $\forall m_i \in CDPaths(n)$ {
13.       if !$(m_i \in S)$
14.           $W = W \cup m_i$;
15.     }
16.   }
17.   return $S$ and all the edges from $G$ that connect nodes in $S$
18. }
```

Fig. 5. Pseudo-code for computing a slice for a set of nodes.

MRI Scan
- In network provider: copay: \$100, coinsurance: 20%
- Out of network provider: copay: \$100, coinsurance: 40%
- Maximum number of visits a year: 1

An important manipulation that we apply to the slice prior to its translation to a natural language is related to the type of the service provider that significantly impacts the copay and coinsurance values. During the slice computation, when a condition related to the type of provider is detected (*In network provider*), we split the final result into two sets of nodes and edges. Each set of nodes and edges is a subset of the slice's nodes and edges and refers to either *true* or *false* value of the condition. Consequently, two summaries are created: one for in network provider and another one for out of network provider.

Note that the condition *Num of visits* $>$ *0* turned into a condition on the maximum number of visits a year. This is one of the transformations we apply while translating the condition into a natural language. Clearly, this only makes the condition more readable without changing its actual meaning. Another heuristic allows us to separate the condition from the two summaries related to in (and out of) network providers. It may seem natural that the computation of the copay may also be separated out, as it is the same for both in and out of network providers. However, in some scenarios, the copay can be different for the different types of providers. For the sake of uniformity, we did not apply any further simplifications on the summary.

B. Benefit Search

An additional capability provided by our technology is Benefit Search. The search process is illustrated in Figure 6. Here, a user wishes to discover code that implements parts of a new policy. A policy can be thought of as a collection of specifications, where each specification consist of a coverage or payment result and under what conditions such a result occurs. For example, under what conditions (such as treatment, patient age, and service location) is there a deductible. This provides a fairly natural way for the user to specify a query; a set of results are given (such as there being a deductible or there not being any coinsurance), and a set of conditions when the result should occur. Since there may be several different ways to code such a specification, the user may wish to use plain English instead of language keywords when performing the search. We note that each such specification (a result and its corresponding conditions) corresponds directly to commands implementing the result (such as take a copayment) and the control paths for that command. Thus, the control paths found can be of use. Of course, this information must be preprocessed and indexed in order to enable the user to locate it.

Note that a single rule in the code can contain many different control paths. Each such control path corresponds to a set of conditions that all must exist for a specific result to occur. We transform each such path and a result into a "search document." A search document consists of several different "fields", where each field can contain processed natural language text or metadata. Thus, in each search document we transcribe the result, and the conditions. Each condition is broken down based on whether it is positive or negative (e.g. "the procedure code is" vs. "the procedure code is not"), and its keyword in the programming language in order to specify which field it is to be stored in. Values given in the condition as well as the English translation of the condition are then stored in this field. Additionally, the translations of the language keywords are indexed; this allows a user to input a free text which can then be mapped to several keywords. For example, the user could specify the condition "procedure is a vaccine" and "procedure" could map to several language keywords, such as whether the procedure code is in a table, or whether the procedure code matches a specific value.

Once the keywords are identified, these serve to provide the fields in which the search is to be performed; in our example, the word "vaccine" would be searched for within these fields. To put the whole process together, the user gives several specifications. Each such specification is used to find and score control paths. These scores are then aggregated for paths belonging to the same benefit rule, thus providing a base score for each benefit rule.

Obviously, a benefit rule which only contains what the user specified and nothing else is preferable to one which contains more logic which is not desired by the user. Thus, benefit rules are penalized for paths that do not match any of the user's specifications. The results are presented to the user together with the scores. It was found through user testing that the scores would primarily be used as a means of deciding which benefit rules to look at; for example if the top three scores were in the range of 20 to 25 and then the scores were all 5 or less then only the top three results would need to be examined. Similarly, benefit rules which did not match all of the specifications were completely removed from the results.

V. Customer Validation

We validated our technology in collaboration with a major healthcare payer in the US. Our main objectives were to learn how this technology impacts productivity of the different users performing their daily tasks, and how it affects the required skill set of benefit coders. While we are unable to disclose quantitative measurements of the system, we can, however, disclose aggregate qualitative feedback received from business stakeholders and senior benefit coders indicating that our system increases productivity, more pronouncedly for less experienced coders. It also gives business stakeholders visibility into the internals of the claims adjudication system which was previously non-existent.

The overall size of the benefit rules in the payer's mainframe based, proprietary claims processing system was nearly 1.4 million procedures over $3.6GB$ source data. To facilitate the deployment in the payers' production environment, certain non-functional requirements had to be met:

- Data is isolated between the six different data owners and an access control mechanism is put in place.
- Tool is available to multiple concurrent users, with speedy response time.
- No client-side installation is required.

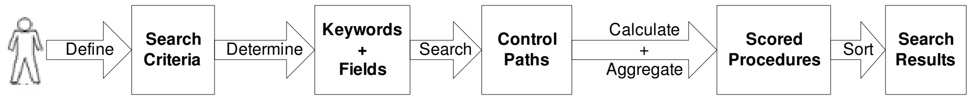

Fig. 6. The search process.

Figure 7 shows how the tool was deployed across multiple machines in production to ensure high availability and data isolation. There are four blades (modular servers optimized for usage in high density data centers), each hosting three clients. Each client is uses two blades balanced by a load balancer. A user management mechanism and login screens were created to address access control. Other than a web browser, no client-side installation was needed.

To achieve speedy response times, some of the analysis was done in a series of preprocessing steps. Data was extracted from the mainframe for each data owner and transferred for processing on a quad-core machine with $8GB$ memory. Batch, or offline, processing takes an average of $165ms$ wall-clock time / procedure (using four cores) This includes parsing, storage, offline analysis and indexing (for the search capabilities). This speed means that processing a full extract can take 24 to 36 hours. Incremental updates can also be applied and processed in a time corresponding linearly with their size. After processing the data is transferred to the online production machines which provide access to the users as previously described.

VI. Related Work

To the best of our knowledge, there is neither prior work specifically for benefit rules summarization out of claim adjudication systems, nor for business rule extraction from domain specific languages and configurations. However, the issue of eligibility and benefit accuracy is an important one for participants in the healthcare system. This can be seen by the rise of organizations and consultant practice that deal with just these issues [1]–[3].

CORE (Committee On Operating Rules for Information Exchange) defined an eligibility and benefits data content rule (270/ 271) in order to improve electronic eligibility and benefits verification [19]. CORE assumes that if eligibility and benefits are accurately known to healthcare providers, all the associated electronic transactions that follow will be more effective and efficient.

The CORE Phases I and II eligibility and benefits data content (270/271) rules primarily outline a set of requirements for health plans to return base patient financial responsibility related to the deductible, co-pay and co-insurance for a set of services in the 271 eligibility response transaction, and for vendors, clearinghouse and providers to transmit and use this financial data.

It is hoped that the entities that exchange eligibility information will work to develop and exchange standard formats within the health care industry and among their trading partners.

There are many studies in the area of generic methods of reverse engineering for business rules extraction out of legacy code [20]–[24].

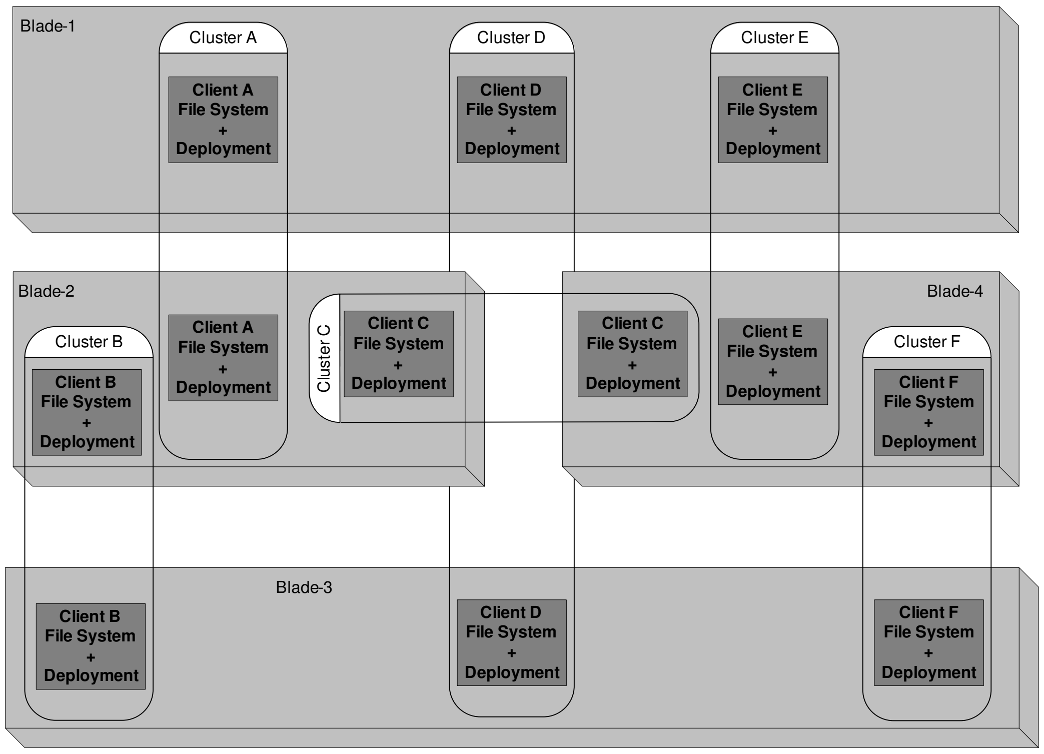
Blade-1
Cluster A
Cluster D
Cluster E
Client A File System + Deployment
Client D File System + Deployment
Client E File System + Deployment
Blade-2
Blade-4
Cluster B
Cluster C
Cluster F
Client B File System + Deployment
Client A File System + Deployment
Client C File System + Deployment
Client C File System + Deployment
Client E File System + Deployment
Client F File System + Deployment
Blade-3
Client B File System + Deployment
Client D File System + Deployment
Client F File System + Deployment

Fig. 7. The deployment environment.

Sneed and Erdos [20] present a method for identifying and extracting business rules by means of data output identification and program stripping, implemented in a reverse engineering tool for COBOL programs. The rules representation is very low-level, represented as if-then sentences of formulas of the Cobol identifiers. This representation is too detailed compared to ours.

Wang et al. [21], [24] present a framework for extracting business rules from large legacy systems. Their framework is composed of five steps: program slicing, identifying domain variables, data analysis, presenting business rules and business validation. They managed to successfully extract business rules from a C/C++ securities trading system and C++ finance system and concluded that their tool successfully extracted the business rules from the systems. Our approach is similar to the one proposed by the authors. However our focus was on a specific domain of claim adjudication systems and was applied to proprietary languages that have not been widely researched.

Gang [23] improves upon Wang's framework [21] and presents a method that uses dependence-cache slicing technique to extract business rules. They experimented on a C++ program and concluded that their technique improves slicing precision. Future versions of our solution will explore whether they can benefit from this enhancement.

Huang et al. [22] present a customized technique which combines variable classifications, program slicing, and hierarchical abstraction for business rules extraction from legacy code. They demonstrate how their tool extracts rules from Cobol programs. The heuristics the authors used to determine candidate variables and criteria to compute slices for can be used in our implementation when the variables of interest are not known in advance.

Ramsey and Alpigini [25] present a mathematical framework for two different rule extraction approaches for an arbitrary program. These approaches are based on the mathematical assertions that programs are composed from language structures, and that extractable business rules can be functionally defined in terms of specific language structures and elements. They present a simple C language example of rule extraction using each approach. This research uses an approach very different from our own and it still needs to be validated on large scale applications and other languages.

VII. Conclusions and Future Work

We believe that extracting benefit rules directly from claims adjudication systems is the only way to guarantee accurate benefit data is available to patients, providers, payers and regulatory bodies. The capabilities described in this paper, and the experiments we have performed lead us to the conclusion that the level of detail challenge can be addressed by providing summaries of benefit rules and searching for benefits according to various criteria. Thus, the benefit data becomes more accessible, and the labor-intensive, complex interpretation of benefit rules is significantly reduced.

In this paper we focused on a specific health care insurer's system. However, in the course of our work we have examined a number of different claims processing systems, used by multiple healthcare payers. While these are proprietary and are their implementations vary, we identified common characteristics in the way benefit rules are coded. These commonalities are in the input values, output values and high-level structure of the rules. The idea of leveraging these commonalities and forming a canonical representation is of great interest and remains an avenue for future research. Such a representation will assist in coping with the standardization and the terminology inconsistency challenges. It will also serve as a basic block for policy validation algorithms for new legislation.

Furthermore, during our engagements with payers and providers we learned that a generic framework for control flow visualization and benefits summarization is called for. This would allow insurers to provide standardized summaries of their policies as required by the latest legislation [26]. It will allow patients to compare between the different policies and choose the policy that best suits their needs. Finally, standard representation of policies and their summaries will allow government officials to validate that indeed the insurers provide coverage for treatments required by the law. Developing the algorithms and methodologies for such a framework is an exciting proposition for future research.

We believe that the solution described in the paper could be parallelized and distributed, for example using the *MapReduce* framework [27]. However, this still remains to be verified and is a subject of further research.

References

[1] B. Karen, "Blueprint to clean claims," http://www.claimtrust.com/hospital-results/blueprint-to-clean-claims.html, 2011.

[2] "Healthcare information systems," http://www.valuelabs.com/healthcare-information-systems-development.php, 2011.

[3] "Mdremedi," http://mdremedi.com/claims.html, 2011.

[4] A. Abadi, M. Nisenson, and Y. Simionovici, "A traceability technique for specifications," in *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, ser. ICPC '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 103–112.

[5] A. Globerson, N. Tishby, I. Guyon, and A. Elisseeff, "Sufficient dimensionality reduction," *Journal of Machine Learning Research*, vol. 3, pp. 1307–1331, 2003.

[6] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Trans. Softw. Eng.*, vol. 28, pp. 970–983, 2002.

[7] T. Hofmann, "Probabilistic latent semantic indexing," in *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, ser. SIGIR '99. New York, NY, USA: ACM, 1999, pp. 50–57.

[8] G. Salton, A. Wong, and C. S. Yang, *A vector space model for automatic indexing*. Morgan Kaufmann Publishers Inc., 1997, pp. 273–280.

[9] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 125–135.

[10] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, vol. 41, no. 6, pp. 391–407, 1990.

[11] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Recovering traceability links in software artifact management systems using information retrieval methods," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 4, 2007.

[12] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 1, pp. 26–60, 1990.

[13] V. P. Ranganath, T. Amtoft, A. Banerjee, J. Hatcliff, and M. B. Dwyer, "A new foundation for control dependence and slicing for modern program structures," *ACM Trans. Program. Lang. Syst.*, vol. 29, 2007.

[14] F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, pp. 121–189, 1995.

[15] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," *SIGPLAN Not.*, vol. 19, pp. 177–184, April 1984.

[16] "Jetty web server," http://www.eclipse.org/jetty/, 2011.

[17] "Apache lucene," http://lucene.apache.org, 2011.

[18] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, pp. 319–349, 1987.

[19] "Core 260 phase ii eligibility & benefits (270/271) data content rule," http://www.caqh.org/pdf/260.pdf, 2011.

[20] H. M. Sneed and K. Erdos, "Extracting business rules from source code," in *WPC '96: Proceedings of the 4th International Workshop on Program Comprehension (WPC '96)*. Washington, DC, USA: IEEE Computer Society, 1996.

[21] X. Wang, J. Sun, X. Yang, Z. He, and S. Maddineni, "Business rules extraction from large legacy systems," *Software Maintenance and Reengineering, European Conference on*, vol. 0, p. 249, 2004.

[22] H. Huang, "Business rule extraction from legacy code," *Computer Software and Applications Conference, Annual International*, vol. 0, p. 0162, 1996.

[23] X. Gang, "Business rule extraction from legacy system using dependence-cache slicing," *Information Science and Engineering, International Conference on*, vol. 0, pp. 4214–4218, 2009.

[24] C. Wang, Y. Zhou, and J. Chen, "Extracting prime business rules from large legacy system," *Computer Science and Software Engineering, International Conference on*, vol. 2, pp. 19–23, 2008.

[25] F. V. Ramsey and J. J. Alpigini, "A simple mathematically based framework for rule extraction from an arbitrary programming language," *Computer Software and Applications Conference, Annual International*, vol. 0, p. 763, 2002.

[26] "U.S. Congress: The patient protection and affordable care act (2010)," http://www. gpo.gov/fdsys/pkg/PLAW-111publ148/pdf/PLAW-111publ148.pdf, 2011.

[27] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.