Hancock: A Language for Extracting Signatures from Data Streams

Corinna Cortes Kathleen Fisher Daryl Pregibon Anne Rogers
Frederick Smith*

AT&T Labs—Research
Shannon Laboratory
180 Park Avenue
Florham Park, NJ 07932, USA

ABSTRACT

Massive transaction streams present a number of opportunities for data mining techniques. Transactions might represent calls on a telephone network, commercial credit card purchases, stock market trades, or HTTP requests to a web server. While historically such data have been collected for billing or security purposes, they are now being used to discover how customers or their intermediaries (called *transactors*) use the underlying services.

For several years, we have computed evolving profiles (called *signatures*) of the transactors in large data streams using handwritten C code. The signature for each transactor captures the salient features of his transactions through time. Programs for processing signatures must be highly optimized because of the size of the data stream (several gigabytes per day) and the number of signatures to maintain (hundreds of millions). C programs to compute signatures often sacrificed readability for performance. Consequently, they are difficult to verify and maintain.

Hancock is a domain-specific language created to express computationally efficient signature programs cleanly. In this paper, we describe the obstacles to computing signatures from massive streams and explain how Hancock addresses these problems. For expository purposes, we present Hancock using a running example from the telecommunications industry; however, the language itself is general and applies equally well to other data sources.

1. INTRODUCTION

A transactional data stream is a sequence of records that log interactions between entities. For example, a stream of stock market transactions consists of buy/sell orders for particular companies from individual investors. Likewise, a stream of credit card transactions contains records of purchases by consumers from merchants. Data mining techniques are needed to exploit such transactional data streams because these streams contain a huge volume of simple records, any one of which is rather uninformative. When the records related to a single entity are aggregated over time, however, the aggregate can yield a detailed picture of evolving behavior, in effect, capturing the "signature" of that entity.

We have analyzed data streams from the telecommunications domain for the past four years. In our initial work, we processed roughly five million (M) international call-detail records per day, generated by approximately 12M customers. In subsequent work, we have tackled larger and larger data streams, including the complete AT&T long distance data stream, which consists of approximately 300M records per day from 100M customers.

For each data stream, we compute or update signatures based on selected fields in each record in the data stream. A signature for a phone number might contain directly measurable features such as when most telephone calls are placed from that number, to what regions those calls are placed, and when the last call was placed. It might also contain derived information such as the degree to which the calling pattern from the number is "business-like" [4].

Programs to compute signatures must be highly optimized because of the size of the data stream and the number of signatures tracked. Because of its size, the entire collection of signatures cannot be kept in memory. Consequently, signature programs are very I/O intensive: they must read from and write to the signature data on disk as they process transactions.

Our initial C programs for computing telecommunication signatures were efficient, but they often sacrificed readability to obtain this efficiency. Regulatory changes force frequent modifications to these programs. Consequently, program maintenance and verification are important issues, both of which require program readability. When we started working with the complete AT&T long distance data stream, we realized that we needed software that could function at scale and yet be maintained as changes were required.

*Department of Computer Science, Cornell University.

Appears in the *Proceedings of the Sixth International Conference on Knowledge Discovery and Data Mining*, 9–17, Boston, Massachusetts, August 2000.

Hancock is a C-based domain-specific programming language that we designed and implemented in response to this need. By design, the language makes time and space efficient signature programs easy to read and write, independent of the quantity of data involved. Because Hancock manages scaling issues, it allows data analysts to prototype new signatures quickly. We introduce Hancock by describing the code necessary to compute a signature from AT&T's wireless services (*AWS*) data stream. This stream contains *call-detail* records, each of which stores information particular to one call in AT&T's wireless network.

The main goals of this paper are to discuss the computational difficulties in writing efficient signature code for massive data streams, to show how Hancock alleviates these difficulties, and to discuss the motivations behind the Hancock design. In an earlier paper [1], we presented a preliminary version of Hancock that handled only one particular form of data (call-detail records from AT&T's long distance network). The version described in this paper handles arbitrary data streams.

2. RUNNING EXAMPLE

In this section, we describe the Cell Tower signature, which we will use as a running example to describe Hancock. This signature mines information from a wireless call stream containing records used to bill customers for calls sent and received from their mobile telephones. This stream contains approximately 80M records per day from 20M customers. Although these records contain many fields, only the following few are relevant for computing the Cell Tower signature:

- Mobile Phone Number (MPN)
- Dialed telephone number
- First and last cell tower used

Our illustrative application characterizes the "diameter" of a mobile phone, *i.e.*, is the phone used exclusively in one or a few neighboring cells, or is it used in a much larger region? Such information is useful for target marketing and for developing new service offerings.

To compute this information, we designed the Cell Tower signature. For each MPN, we track the five most frequently (and most recently) used cell towers and another value that captures the frequency with which calls placed from the MPN do not involve the top five cell towers. As one might expect, the top five list is dynamic, so the signature computation includes a probabilistic bumping algorithm that allows a new cell tower to enter the top five list as its frequency of use increases. Earlier work describes how to design signatures [2, 5, 6, 7].

Given the Cell Tower signature, any number of measures of "diameter" can be computed, *e.g.*, the area of the convex hull defined by the geographical coordinates of the top five cell towers. Maintaining the list allows us to experiment with alternative measures before committing to any specific measure that might be computed directly from the call-detail stream.

As a prelude to subsequent sections in which we intersperse Hancock code with text, the following Hancock/C code describes the `profile` struct that we associate with each MPN in our Cell Tower signature:

```
#define NDIV 5
#define MAX_TOWER_NAME 12
typedef struct {
  char celltower[NDIV][MAX_TOWER_NAME];
  float count[NDIV];
  float other;
} profile;
```

The `celltower` array stores the names of the five most commonly used towers as fixed-size C strings. The parallel array `count` measures the frequency with which the corresponding tower is used. Field `other` counts how many calls are not reflected in the list of the top five towers.

3. HANCOCK

Hancock is a C-based domain-specific language designed to facilitate signature computations. Hancock makes it easier to read, write, and maintain such programs by factoring into the language the issues that relate to scale. In this section we discuss some of these issues and describe how Hancock addresses them.

Prior to designing Hancock, we studied existing signature programs to understand their structure and the techniques they employ to achieve good performance. Figure 1 illustrates the process flow for a typical signature program from the telecommunications industry. Transaction records are collected for some time period, the length of which depends on the application (*e.g.*, a day for marketing but just a few minutes for fraud detection). At the end of the time period, the records are processed to update the signatures. Before processing, the old signature data is copied to preserve a back-up for error-recovery purposes. During processing, the records are typically sorted in several ways, *e.g.*, according to the originating and then the dialed phone numbers. After each sort, a pass is made over the data stream. During a pass, the portion of each signature relevant to the given sort is retrieved from disk, updated, and then written back to disk. For example, after sorting by the originating telephone number, only the portion of a signature that tracks out-bound calling would be updated typically; after sorting by the dialed number, the portion that tracks in-bound calling would change. Sorting the stream ensures good locality for accesses to the signatures on disk and groups the information relevant to each transactor into a contiguous segment of the stream.

3.1 Logical and physical streams

The fields in a transaction record are often encoded and packed to save space. In studying the existing signature programs, we noticed that code to decode the representation of stream records was interleaved with signature processing code, which made it difficult to change the physical representation of stream records.

In Hancock, we separate the *physical* representation of the records in a data stream from the *logical* (expanded) representation on which we perform computations. This separation allows one person to understand the physical represen-

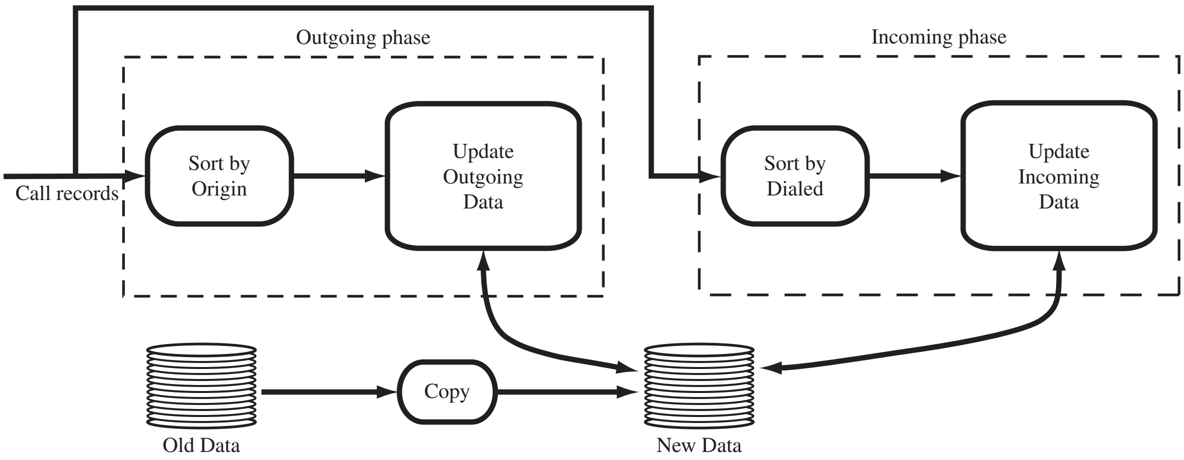

Figure 1: High-level architecture of signature computations. The processing typically consists of several phases, each sorting the data in a different order and updating a different part of the signature.

tation (the expert on that data source) but many people to use the logical representation (the consumers of that data source). This division facilitates maintenance: if the physical representation changes, only the translation from the physical to the logical representation must be modified, presumably by the expert on that data source. The consumers need not modify their programs.

To declare a new stream type, programmers use the `stream` type operator. Generally, Hancock requires one stream type per data source. As there are many fewer data sources than there are signature programs, declaring new streams is rare. There are two forms of stream declarations: a specialized form for streams whose records are stored on disk in a fixed-width binary format, and a general form for records stored in other formats. The binary form is more convenient, while the general form is more broadly applicable.

The declaration of a binary stream specifies both the physical and the logical representations for the records in the stream. It also specifies a function to convert from the encoded physical representation to the expanded logical representation.

The following declaration introduces the stream `AWS_s`:

```
stream AWS_s {
  getvalidAWS : awsPhy_t => awsLog_t;
};
```

For this stream, the C type `awsPhy_t` serves as the physical representation and `awsLog_t` serves as the logical. The identifier `getvalidAWS` names the function that specifies how to convert from the physical to the logical representation. This function, whose prototype is:

```
char getvalidAWS(awsPhy_t *pc, awsLog_t *c);
```

checks that the record `*pc` is valid and if so, unpacks `*pc` into `*c` and returns `true` to indicate a successful conversion. Otherwise, `getvalidAWS` simply returns `false`. Programmers can declare variables of type `AWS_s` using standard C syntax (for example, `AWS_s calls`).

In the general case, a stream declaration specifies a function that reads data from a file and returns a logical record. We use the term "record" to mean the logical representation of the elements in a stream, since stream definitions are the only place where the physical representation is needed.

3.2 Logical, approximate, and physical signatures

When the number of customers is in the hundreds of millions, one can maintain only very small signatures for each customer because the I/O cost to update the signatures would be prohibitive otherwise. To save space, the values of a signature are often quantized or otherwise approximated before they are stored. For example, a floating point number representing the probability that a phone number is behaving like a business might be quantized into sixteen levels; similarly, the number of daily outbound minutes may be categorized according to one of eight logarithmically spaced usage bins. These approximate signatures can be compressed conveniently into a few bytes before writing them out to disk. Thus, each signature program conceptually uses three different representations of each signature: the *logical* representation used for computation, the *approximate* representation that specifies what information to preserve, and the compressed *physical* form that is written to disk.

The original C programs for computing signatures contained routines to approximate and compress each signature before writing it to disk and routines to uncompress and expand it before computing with it. However, we found situations in which the original C code performed computations not only on the logical representations, but also on the approximate and on the compressed representations. While the code was very efficient, it was highly unreadable, making it difficult to verify and maintain.

Hancock's `view` construct provides a mechanism to specify two views of a single piece of data and the conversion between them. Signature programs use one view to describe the logical representation of each signature and another to describe the approximate representation. Hancock's `map` abstraction provides a mechanism to specify application-specific compression functions (see Section 3.3).

As an example of views, consider the following declaration that specifies approximate (bin) and logical (minute) representations of a unit of time:

```
view time(bin, minute) {
  char <=> int;
  bin(m) { return min_to_bin(m); }
  minute(b) { return bin_to_min(b); }
}
```

The line char <=> int declares that the bin view is represented as a char and the minute view is represented as an int. The bin function specifies how to convert from the logical to the approximate representation by computing the bin associated with m minutes. Similarly, the minute function converts from the approximate to the logical representation by assigning a default number of minutes to the bin b. To translate between these two views, the programmer uses the Hancock view operator ($):

```
bin b = 3;
minute m;
m = b$minute; // Convert bin b to minutes m
    ...
b = m$bin;    // Convert minutes m to
              // corresponding bin number
```

Views allow Hancock programmers to document the representation they are using in a given context. Views also insure that the definition of how to convert between their representations appear only once in the program. Both of these aspects of views make Hancock programs easier to read and maintain than the corresponding C programs.

3.3 Signature collections

The original signature programs associated values with keys using a data structure called a *map*. Maps allowed direct addressing to retrieve and store values in a customized, compressed format. Hancock kept the notion of a map, but instead of the programmer compressing the approximate representation before storing it in a map, the programmer now stores the approximate representation directly. The signatures are still compressed, but now application-specific compression functions are called directly by the Hancock map implementation. The bytes representing compressed data are no longer available to the programmer.

In the Cell Tower application, for example, we want to associate a profile, which we defined earlier, with each mobile phone number. The map declaration:

```
map cellTower_m {
  key pn_t;
  value profile;
  default {{'\0', '\0', '\0', '\0', '\0'},
           {0.0, 0.0, 0.0, 0.0, 0.0},
           0.0};
  compress ctSqueeze;
  decompress ctUnsqueeze;
};
```

creates a new map type, cellTower_m, for this purpose. The key clause indicates that this map will be indexed by values of type pn_t, a type which represents mobile phone numbers in a format required[1] by the on-disk representation of maps. The value clause of a map declaration specifies the type of data to be associated with each key. The value type may be any valid Hancock type. In the Cell Tower application, this type is a standard C struct profile, but in many other applications the value type is the approximate representation from a Hancock view type. (In this signature application, the logical and approximate representations are the same.) The default clause specifies a value to be returned if the programmer requests data for a key that does not have a value stored in the map. For maps with type cellTower_m, the default profile contains empty strings for the cell towers and zeros for the counts.

Hancock maps provide generic compression routines, but programmers may have additional domain-specific knowledge that would allow them to write significantly better custom compression routines. The optional compress and decompress clauses in a map declaration allow programmers to specify such compression functions. In the example, the identifier ctSqueeze names a C function that takes a pointer to a profile and returns a collection of bytes. The C function ctUnsqueeze performs the reverse operation. These functions are usually lossless, but Hancock does not require them to be so.

Variables of type cellTower_m can be declared using the usual C syntax. Hancock provides an indexing operator <: ... :> to access values in maps. The code:

```
cellTower_m ct;
pn_t mpn;
c = ct<:mpn:>;
    ...
ct<:mpn:> = c;
```

uses mobile phone number mpn to first read from and then write to map ct. A common idiom in Hancock

```
m<:key:>$logview
```

uses the indexing operator to get an approximate value out of a map (m) and the view operator to convert that value into the logical representation (logview).

Hancock provides a lazy map copy operator, written using the infix notation :=:. For example, the statement new_ct:=:ct initializes the map new_ct with the data from map ct.

3.4 Events

Much of the work in computing a signature is done in response to "events" in the input stream. For example, when a program sees a new mobile phone number in an AWS_s stream, it might re-initialize counters for that phone number. The original signature programs contained a hierarchy of events, illustrated in Figure 2. Such events included seeing a new area code (npa), seeing a new exchange (nxx),[2] see-

[1] Hancock maps are represented on disk as indexed files. The programmer controls the number of index levels and their composition through the key type. Restrictions on key types are described in the Hancock manual [8].

[2] An *exchange* is the first six digits of a ten digit telephone number.

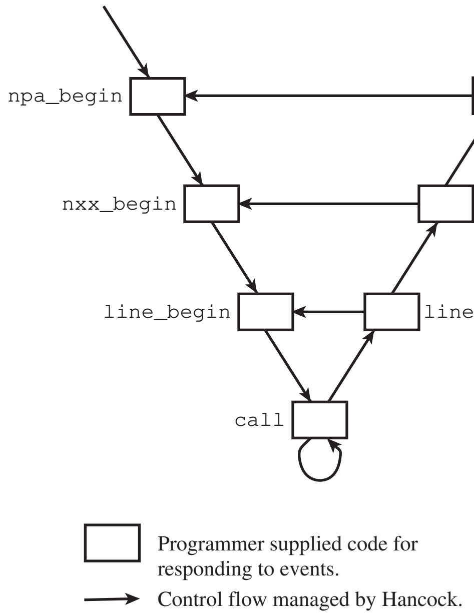

Figure 2: Hierarchical event structure.

ing a new phone number (line), seeing an individual call record, seeing the last record for a phone number, *etc.* In the diagram, these events are named `npa_begin`, `nxx_begin`, `line_begin`, `call`, `line_end`, *etc.*, respectively. When a signature program detects an `npa_begin` event in a stream, it may retrieve the time zone for the triggering area code. In response to an `nxx_begin` event, it may retrieve all the old signatures for the newly seen exchange. For a `line_begin` event, it may initialize counters that it later increments in response to `call` events. The program may store the final values for these counters when a `line_end` event occurs.

In Hancock, we divide this processing into two pieces: event detection and event response. Event detection includes defining the events of interest in a given stream and specifying how to identify them. Event response indicates what to do when an event is detected. In the example in Figure 2, event detection defines the boxes in the diagram, while event response determines the code that inhabits those boxes. The Hancock compiler generates the control flow that sequences the response code. Because this control-flow code involves deeply nested loops, Hancock programs are much easier to read and maintain than the corresponding C programs, which mix these loops with event response code. In the remainder of this section, we discuss how to describe and detect events in a stream. In the next section, we discuss how to respond to detected events.

To define events in a general fashion, we introduced a new kind of type into Hancock: a *multi-union*. A multi-union names the set of labels it may contain and associates a type with each such label. Although we designed multi-unions to describe events, they are in fact a general construct, suitable for many purposes; hence we named their constituents "labels" instead of "events." When we use multi-unions to describe events, however, we often refer to their labels as "events." As an example, consider the declaration:

```
munion line_e {: areacode_t npa_begin,
                 exchange_t nxx_begin,
                 pn_t line_begin,
                 awsLog_t call,
                 pn_t line_end,
                 exchange_t nxx_end,
                 areacode_t npa_end :};
```

This code creates a multi-union type `line_e` to describe the events from Figure 2. A value with this type contains any subset of the declared labels, including the empty set, which we write `{: :}`. Each label in the set carries a value of the indicated type. If `l` is the current phone number and `c` the current call record in a stream, then the expression

```
{: line_begin = l, call = c :};
```

creates a value with type `line_e`. This value would describe the events that occur when the first (but not the last) call record for telephone number `l` appears in the stream. If `e1` and `e2` are multi-union values with the same type, then expression `e1:+:e2` produces a new value that contains the union of the labels of `e1` and `e2`.

After describing the events of interest using a multi-union declaration, the programmer must specify how to detect such events by writing an *event-detection* function. Such a function looks at a small portion of a stream and returns a multi-union to describe the events detected in that window.

To describe a small portion of a stream, Hancock provides a *window* type. The size of the window determines how many records in the stream can be viewed at once. A window is like an array, but has the added notion of a "current" record. In specifying a window, the programmer indicates the placement of the current record in the window. For example, the declaration:

```
awsLog_t *w[3:1]
```

specifies that `w` is a window of size three onto a stream with records of type `awsLog_t`. A pointer to the current record appears in the middle slot of the window, *i.e.*, in `w[1]`. Slots with lower indices (`w[0]`) store pointers to records earlier in the stream; slots with higher indices (`w[2]`) look ahead to records appearing later in the stream. If the window overlaps either the beginning or the end of the stream (or both), the slots with no corresponding stream record are set to NULL.

An event detection function takes a window onto a stream and returns a multi-union describing the events detected in that window. The Cell Tower signature uses the following event detection function:

```
line_e originDetect(awsLog_t *w[3:1])
{ line_e b,e;
  b = beginDetect(w[0], w[1]);
  e = endDetect(w[1], w[2]);
  return b :+: {: call = *w[1] :} :+: e;
}
```

This function calls the auxiliary functions `beginDetect` and `endDetect`. The first determines whether the current record represents a new MPN by comparing the `origin` from the previous record (`w[0]`) to the `origin` of the current record (`w[1]`). The second determines whether the current record represents the last call for a MPN by comparing the `origin` for the current record to the `origin` for the next record (`w[2]`).

3.5 Consuming a stream

As in the original signature code, Hancock's computation model is built around the notion of iterating over a sorted stream of transaction records. Sorting the records groups all the data relevant to one transactor into a contiguous segment of the stream and ensures good locality for map references that follow the sorting order. Consequently, each signature program typically makes multiple passes over its data stream. During each such pass, the signature program sorts the stream in a different order and updates a different portion of each transactor's signature. We call each pass a *phase*.

In Hancock, we implement phases using Hancock's `iterate` statement. The iterate statement has the following form:

```
iterate
  (over  stream variable
   filteredby  filter predicate
   sortedby  sorting order
   withevents  event detection function)
 {
   event clauses
 };
```

The header specifies an initial stream, a set of transformations to produce a new stream, and a function to detect events in the stream. The body contains a set of event clauses that specify how to respond to the events detected in the transformed stream.

We explain each of these pieces in turn. The `over` clause names an initial stream to transform. The `filteredby` clause specifies a predicate to remove unneeded records from the stream. For example, an `AWS_s` stream may include land-to-cell calls, which are not used by the Cell Tower signature. Immediately removing such records improves the efficiency of sorting and simplifies event response code.

The `sortedby` clause describes a sorting order for the stream by listing the fields from the records in the stream that constitute the desired sorting key. For example, using the following clause:

```
sortedby origin, connecttime
```

produces a stream sorted primarily by the originating telephone number and secondarily by the time at which the call was made.

The `withevents` clause specifies an event detection function. As described in Section 3.4, such a function takes a window onto the stream and returns a multi-union describing the events detected in the given window.

The *event clauses* specify code to execute when an event detection function triggers an event. Events that occur simultaneously (*i.e.*, in the same multi-union value) are processed in the order they appear in the event clauses. Given this ordering information, Hancock generates the control-flow to sequence the response code, *i.e.*, it generates the arrows in Figure 2. The name of each event clause corresponds to a label in the multi-union returned by the event detection function. Each event clause takes as a parameter the value carried by the corresponding label. For example, the exchange that triggers an `nxx_begin` event is passed to the `nxx_begin` event clause. The body of each event clause is a block of Hancock/C code.

As an example, Figure 3 shows the outgoing phase of the Cell Tower signature, which processes the calls *made* by wireless telephone numbers. The function `out`, which encapsulates this phase, contains a single iterate statement that processes a wireless call stream. It uses the predicate function `completeCellCall` to remove incomplete and non-cellular calls from the stream. It sorts the filtered stream by the originating phone number. It uses the function `originDetect` to detect events in the sorted stream. The event clauses in lines 11 to 27 of Figure 3 specify how to respond to the detected events. Note that this phase does not use all the events defined in the `line_e` type.

```
1   void out(AWS_s calls, cellTower_m ct)
2   {
3     profile p;
4
5     iterate
6       ( over calls
7         filteredby completeCellCall
8         sortedby origin
9         withevents originDetect ) {
10
11      event nxx_begin(exchange_t npanxx) {
12        degradeBlock(ct, npanxx);
13      }
14
15      event line_begin(pn_t mpn) {
16        initProfile(&p);
17      }
18
19      event call(awsLog_t c) {
20        diversify(&p, c.cellid);
21      }
22
23      event line_end(pn_t mpn) {
24        profile mytemp;
25        mytemp = ct<:mpn:>;
26        ct<:mpn:> = update(&mytemp,&p);
27     }
28    };
29  }
```

Figure 3: Outgoing phase for the Cell Tower signature.

3.6 Putting it together

In the previous section, we explained that computing a signature may require multiple passes over the data. Hancock provides the `sig_main` construct to express the data flow between such passes and to connect command-line input to the variables in a Hancock program. The arcs between the phase boxes in Figure 1 depict this construct. The following code implements sig_main for the Cell Tower signature:

```
void sig_main(const AWS_s calls <a:>,
              exists const cellTower_m oldCT <m:>,
              new cellTower_m newCT <M:>) {
  newCT :=: oldCT;
  out(calls,newCT);
}
```

The parameters to sig_main connect command-line arguments to program variables. The calls parameter is a stream that contains the raw wireless call data. The const keyword indicates that this data is read-only. The syntax (<a:>) after the variable name specifies that this parameter will be supplied as a command-line option using the -a flag. The colon indicates that the flag takes an argument, in this case the name of the directory that holds the call files. The absence of a colon indicates that the parameter is a boolean flag. The oldCT parameter is a Cell Tower map, the name of which is specified using the -m flag. The const qualifier indicates the map is read-only, while the exists annotation requires the map to exist on disk. The newCT parameter names the Cell Tower map to hold the result of this program. The -M flag specifies the file name for this map, and the new qualifier indicates that the map must not previously exist on disk.

In general, the body of sig_main is a sequence of Hancock and C statements. In the Cell Tower application, sig_main copies the data from oldCT into newCT and then invokes the outgoing phase with the raw call stream and the new Cell Tower map as arguments. If the Cell Tower application required a second phase, *e.g.*, an incoming phase, we would call it after the call to out.

4. DISCUSSION

In the process of designing Hancock, we carefully examined signature programs that had been written by hand in C. We observed several problems with these programs. First, the persistent profile data typically could be manipulated in several different representations. It was often unclear which representation was being used at any given point in the program. Second, the code to process the stream events was difficult to decipher and maintain. Finally, the dataflow between the phases was often unclear. Despite the weaknesses of these programs, they had some significant strengths: they were efficient and they used an effective representation for their persistent data.

While designing Hancock, we tried to address the problems while preserving the strengths of the original programs. Views allow a programmer to document the relationship between two views of a type. The view operator guarantees that programmers switch between representations in a consistent and well-documented way. Maps provide an efficient implementation for the most performance critical part of signature programs. Their indexing operation provides some type-checking, which helps ensure that programmers use data consistently.

We designed Hancock's event clauses to clarify stream processing without losing efficiency. Such clauses have several advantages. First, they have the flavor of function definitions with their attendant modularity advantages, but without their usual cost because the Hancock compiler expands the event definitions in-line with the control-flow code. Second, having the compiler generate the control flow removes a significant source of bugs and complexity from Hancock programs. Finally, programmers can share information across events easily.

We designed the sig_main mechanism in Hancock to clarify the dataflow between phases, to simplify the process of parsing arguments, and to provide a way to connect the runtime representation of Hancock data types, such maps and streams, to their on-disk counterparts. The automatic generation of argument parsing code also helps programmers protect valuable data through the const, new, and exists qualifiers. The runtime system catches attempts to write to constant data and generates error messages. It detects when data annotated as new already exists or when data tagged with exists is not on disk, in each case reporting a runtime error. These data-protection features are important when it is time-consuming or even impossible to reconstruct an accidentally overwritten signature.

4.1 Language versus library

One question we are asked often is why we chose to design a language rather than a library. There are two technical reasons for choosing the language option. First, expressing Hancock's event model and the information sharing it provides proved awkward in a call-back[3] framework, the usual technique for implementing such abstractions. Second, by designing a language we could use the language's type system to provide more precise typechecking than is provided by C. For example, the natural way to implement maps using a library interface would require the programmer to cast between the actual type of a value and void *, thereby losing the benefits of typechecking. The scale of the data makes the complexity of finding and fixing bugs in signature programs substantial. Therefore, static error detection is essential.

The more compelling reason to choose a language over a library for us is sociological. The experience of writing a Hancock program is fundamentally different than writing the equivalent program in C. This difference arises in part because Hancock removes issues of scale, leaving programmers free to concentrate on the design of the individual profiles, and in part because Hancock provides a vocabulary tailored to the domain of signature design.

5. EXPERIENCES

Over the past six months we have gained experience with using Hancock to write signature programs. This section briefly describes our implementation and discusses our experiences with using the language for both production and experimental signatures.

Our implementation of Hancock consists of a compiler that translates Hancock code into plain C code. We compile and link the resulting C code with a runtime system to produce executable code. We modified CKIT [3], a C-to-C translator

[3] A call-back is a call from a function in a library "back" to a function in user code.

written in ML, to parse Hancock and translate the resulting extended parse tree into abstract syntax for plain C. During the translation, we typecheck the various Hancock forms. The runtime system, which is written in C, manages the representation of Hancock data on-disk and in memory. It converts between these representations as necessary and it mediates all access to both stream and map data.

We have implemented five production signatures that produce information that AT&T uses daily for fraud detection, customer care, and target marketing. We have implemented six additional experimental signatures, some of which we expect to move into production in the next few months.

These signatures use data from four sources: long distance calls (`callDetail_s`), wireless calls (`AWS_s`), WorldNet sessions (`WorldNet_s`), and IP packets (`IP_s`). We have written Hancock descriptions of these sources. In general, we expect a single domain expert to write and maintain such descriptions; others who are interested in a particular data source simply import these descriptions. Of the existing signatures, seven use the `callDetail_s` description, two use `AWS_s`, and the other two use `WorldNet_s` and `IP_s` . The header files that describe these streams contain roughly 250-300 lines of code each. Approximately 40% of that code describes the physical stream record types, the logical stream record types, and the translation functions. Another 40% describes the sets of stream events commonly used by applications. The final 20% describes how to translate logical telephone numbers into the format required for map keys. Writing the Hancock code to describe the wireless data source took about two hours. Much of that time was spent understanding the physical representation of the data and writing the translation function.

We often use the term "signature" to refer collectively to the header file that describes the persistent data computed by a signature program (*e.g.*, a map), the Hancock program that describes how to compute the map from a data source (*e.g.*, a stream), and a set of auxiliary programs used to query the map. The header files are very small; they range in size from 40 to 180 lines. The bulk of the header code is devoted to application-specific compression and decompression functions. The signature computation programs range in size from 100 to 600 lines of code. The complexity of the desired profile determines the size of the program. The larger programs employ a complex bumping algorithm to approximate the most common N occurrences of a feature in the data stream. The auxiliary programs are simple and small; they range in size from five lines to 200 lines, with a median size of 40 lines.

Our experience with the two wireless signature applications gives an indication of how rapidly one can build a new prototype signature. Once we had described the wireless data source, these two signature programs took roughly one hour each to write. One of these programs computes a wireless version of a signature designed for long-distance call-detail data. The programs that compute the wireless and long-distance versions are closely related: 80% of the code is identical. The differences in the remaining 20% arise largely from name changes (*e.g.*, `callDetail` to `AWS`). These two programs are the largest in our suite.

Four of the production signatures are revisions of programs that were written originally in C. The Hancock versions run in approximately the same time as their C counterparts, but the programs are much clearer. In addition, Hancock's improved map representation reduces the size of the resulting persistent data by a factor of four. The persistent data for the production signatures ranges in size from half a gigabyte for the smallest map to seven gigabytes for the largest. These maps contain roughly 350M key/value pairs each. The values in the different maps range in size from a few bytes to as many as 112 bytes. The largest of these profiles is significantly larger than any of the profiles used in the original C implementations.

6. CONCLUSIONS

Working with transactional data streams is like drinking from the proverbial fire hose: the volume is simply overwhelming. But this challenge provides an opportunity for data mining research to enter a new area. We believe that Hancock is a valuable tool for exploiting this opportunity.

Hancock has allowed us to improve our application base by replacing hard-to-maintain, hand-written C code with disciplined Hancock code. Because Hancock provides high-level, domain-specific abstractions, Hancock programs are easier to read and maintain than the earlier C programs. By careful design, these abstractions have efficient implementations, which allow Hancock programs to preserve the execution speed and data efficiency of the earlier C programs. Hancock gave domain experts the confidence to attack more challenging problems because it allowed them to concentrate on *what* to compute without worrying about *how* to manage the volume of data.

We continue to explore the boundaries of the applicability of Hancock to related areas, *e.g.*, processing ISP (Internet Service Provider) session records or IP packet-header logs. Soon, Hancock will be publicly available for non-commercial use at:

```
www.research.att.com/~kfisher/hancock
```

We hope that others will join us in exploring the language and its functionality.

7. REFERENCES

[1] D. Bonachea, K. Fisher, A. Rogers, and F. Smith. Hancock: A language for processing very large-scale data. In *USENIX 2nd Conference on Domain-Specific Languages*, pages 163–176, October 1999.

[2] P. Burge and J. Shawe-Taylor. Frameworks for fraud detection in mobile telecommunications networks. In *Proceedings of the Fourth Annual Mobile and Personal Communications Seminar*. University of Limerick, 1996.

[3] S. Chandra, N. Heintze, D. MacQueen, D. Oliva, and M. Siff. Pre-release of C-frontend library for SML/NJ. See `cm.bell-labs.com/cm/cs/what/smlnj.`, 1999.

[4] C. Cortes and D. Pregibon. Giga mining. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, 1998.

[5] C. Cortes and D. Pregibon. Information mining platform: An infrastructure for KDD rapid deployment. In *Proceedings of the Fifth International Conference on Knowledge Discovery and Data Mining*, 1999.

[6] D. E. Denning. An intrusion-detection model. In *IEEE Trans Soft Eng Vol 13, No 2*, 1987.

[7] T. Fawcett and F. Provost. Adaptive fraud detection. *Data Mining and Knowledge Discovery*, 1:291–316, 1997.

[8] K. Fisher, A. Rogers, and F. Smith. The Hancock language manual. In preparation. See `www.research.att.com/~kfisher/hancock`.