Hancock:
A Language for Processing Very Large-Scale Data

Dan Bonachea* Kathleen Fisher Anne Rogers Frederick Smith†

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

`bonachea@cs.berkeley.edu`
`{kfisher,amr}@research.att.com`
`fms@cs.cornell.edu`

Abstract

A signature is an evolving customer profile computed from call records. AT&T uses signatures to detect fraud and to target marketing. Code to compute signatures can be difficult to write and maintain because of the volume of data. We have designed and implemented Hancock, a C-based domain-specific programming language for describing signatures. Hancock provides data abstraction mechanisms to manage the volume of data and control abstractions to facilitate looping over records. This paper describes the design and implementation of Hancock, discusses early experiences with the language, and describes our design process.

1 Introduction

There are many families of programs whose members have a high degree of commonality; such a family is called a *domain*. When the commonality is inherently complex, a domain-specific programming language may help domain programmers develop better software more quickly by factoring the complexity into the language. Recent examples of domain-specific languages and their domains include Envision [SF97] for computer vision, Fran [Ell97] for computer animation, GAL [TMC97] for video card device drivers, Mawl [ABB+97] for dynamic web servers, and Teapot [CRL96, CDR+97] for writing coherence protocols. In each case, the domain-specific language moved the burden of writing intricate domain code from the programmer to the compiler (or interpreter). We have designed and built a domain-specific language, called Hancock, to handle the complexity that arises from the scale of *signature* computations [CP98]. As with the earlier languages, Hancock programs are easier to write, debug, read, and maintain than the equivalent programs written in general-purpose programming languages.

Signatures are profiles of customers that are updated daily from information collected on AT&T's network. They are used for a variety of purposes, including fraud detection and marketing. For example, AT&T uses signatures to track the typical calling pattern for each of its customers. If customers far exceed their usual calling levels, the fraud detection system raises alerts. In contrast, a sudden drop in their calling levels signals that they may have chosen another long-distance carrier, triggering a marketing alert. Cortes and Pregibon describe signatures and their uses in more detail [CP98].

At a high level, the computation of a signature is straightforward: process a list of call records and update data in a file based on those records. Unfortunately, performance requirements that arise from the volume of call records and the amount of stored profile data complicate matters substantially. Efficiently coping with the data requires a more

*Now in the EECS department at the University of California at Berkeley.

†Now in the CS department at Cornell University

complex system architecture. Furthermore, the scale pressures programmers to conserve instructions, which tends to reduce program readability. The complexity of the architecture and the code complicates testing, which is already difficult because of long testing cycles.

The scale of signature computations arises from both the number of calls and the number of customers. On a typical weekday, there are more than 250 million calls made on AT&T's network. The call records that are used for signature computations contain only 32 of the more than 200 bytes of data that are collected for each call. To get a sense of the scale of this data, it takes about 15 minutes to read through one weekday's call data (approx 6.5GB) and about two hours to compute a typical signature on one processor of a 16 processor SGI Origin 2000. A typical signature tracks several hundred million phone numbers. Even if we store only two bytes of data per customer, the files are very large, requiring a minimum of 500MB to hold the data. Such files also require significant additional space to provide appropriate indexing.

This scale complicates the architecture of signature programs because at such levels, I/O presents a significant bottleneck. To reduce this bottleneck, signature programs must be structured to minimize I/O. In particular, signature programs sort the call records that they process to improve locality of reference to their signature files, and they cache parts of these files to reduce the number of disk accesses. Both techniques trade improved performance for increased code complexity.

In addition to affecting the high-level architecture, the amount of data complicates the low-level code structure. Programmers writing signature programs have tended to respond to performance pressures by avoiding function calls and thereby writing less modular code. The deeply nested code that results is difficult to decipher, which is a serious problem because it is often the only documentation of the signature it implements. Another complication comes from the fact that the size of the signature files limits the number of bytes we can store per phone number. As a result, we cannot store exact information for each phone number; instead we must approximate. Managing the translation between the representation that we can afford to store and the representation that we want to compute with complicates the code.

Hancock is a C-based domain-specific language that reduces the coding effort of writing signatures and improves the clarity of the resulting code by factoring into the language all the issues that relate to scale. In particular, Hancock programmers can focus on the signature they want to compute rather than the scaffolding necessary to compute it. The language includes constructs for describing the overall system architecture, for specifying work that should be done in response to events in the call stream, and for specifying the representation of signature data. The Hancock compiler uses these constructs to generate the boiler-plate code that makes hand-written signatures difficult to maintain, without sacrificing efficiency. The Hancock runtime system supports external sorting and efficient access to signature data. Hancock does not address directly the problem of testing; instead, it reduces opportunities for bugs by relieving programmers of the burden of writing intricate code.

This paper describes the design and implementation of Hancock. Section 2 describes the domain in more detail and gives an example signature. Sections 3-5 present Hancock's data and control abstractions. We discuss the implementation of Hancock's compiler and runtime system in Section 6, early experiences with Hancock in Section 7, and our design process in Section 8. We conclude in Section 9.

2 Signature domain

Signatures are a way to associate information with individual telephone numbers. For each phone number, a typical signature contains information about the characteristics of outgoing calls from that number and incoming calls to it. The outgoing data is often split into sub-categories based on the type of call (for example, toll-free, international, intra-state, and other). This section describes the process of computing signatures, discusses the representation of signature data, and presents an example signature.

We first define some basic terminology. Signatures maintain information about phone

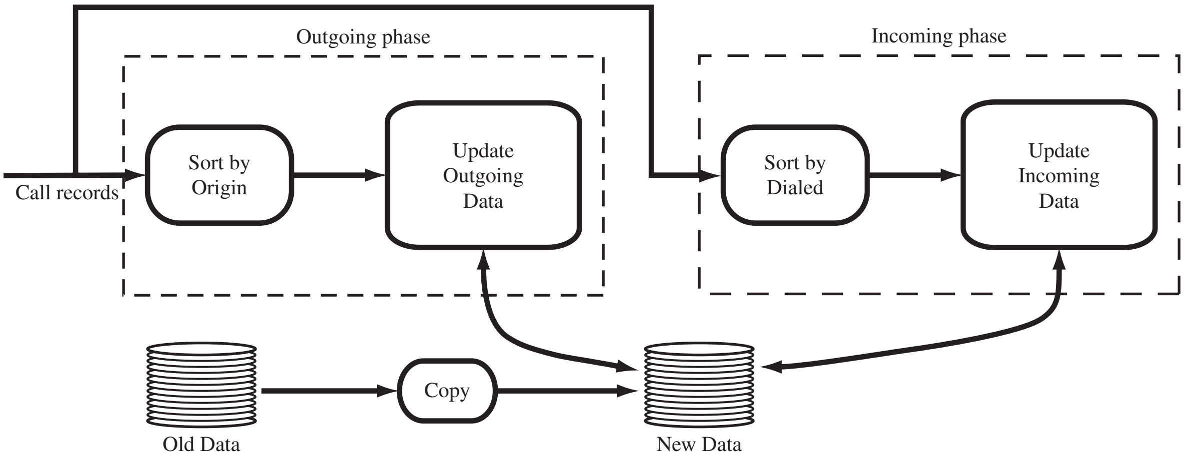

Figure 1: High-level architecture of signature computations

numbers rather than customers. A second database can be used to match signature data with customer information, but that process is outside of Hancock's domain. A telephone number (973 555 1212) contains an *area code* (973), an *exchange* (555), and a *line* number (1212). We often use the term exchange to mean the first six digits of a phone number (973 555) and the term line to mean the whole phone number. Hancock supports a few basic types for phone numbers, dates, and times: area_code_t, exchange_t, line_t, date_t, and time_t. It also supports a type for call records:

```
typedef struct {
   line_t origin;
   line_t dialed;
   date_t connectTime;
   time_t duration;
   char isIncomplete;
   char isIntl;
   char isTollFree;
   ...
} callRec_t;
```

Each call record contains two phone numbers: the originating number and the dialed number. Each record also stores the time the call was connected (`connectTime`) and the duration of the call in seconds (`duration`). Call records also contain boolean flags describing the type of call: `isIncomplete`, `isIntl`, `isTollFree`, *etc.*

2.1 Signature computation

We update signature data daily from the records of the calls made the previous day.

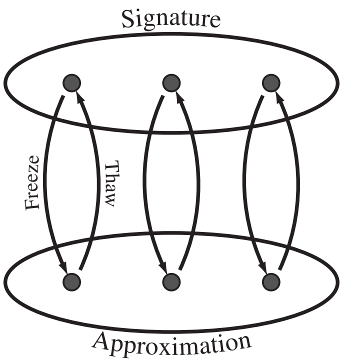

Figure 2: Signature data representation

Figure 1 shows a graphical depiction of the typical architecture we use for such computations. We first sort the call records by the originating phone number and then update the outgoing portion of the signature for each phone number that made a call. We then repeat this process, sorting the call records by the dialed number and updating the incoming portion of the signature for each phone number that received a call. This architecture reduces the I/O needed to process a signature by ensuring good locality for references to the stored signature data at the cost of a pair of external sorts.

2.2 Signature representation

As mentioned earlier, the size of signature files limits the number of bytes we can keep for each phone number. As a result, we cannot keep exact information for each line. Instead, we need to approximate. Conceptually, in each signature we have two views for our data: a precise form called the *signature* view and an approximate form called the *approximation* view. We compute with the signature, but store the approximation. The choice of these two views is application specific, as is the method for converting between them. We call the process of converting from the signature to the approximation view *freezing*; the converse process, *thawing*. (See Figure 2.) Note that `thaw(freeze(v))` does not necessarily equal `v` because freezing is often lossy.

Having two views for the data allows programmers to compute with the natural representation while saving disk space, but it comes at the cost of having to manage conversions between the two views.

2.3 Usage signature

This section describes the *Usage* signature, which we use as a running example. Usage approximates cumulative daily call duration for incoming calls, outgoing calls, and outgoing calls to toll-free numbers. Usage uses the same structure to track all three types of calls. In particular, the signature type for each is seconds; the approximation type, a bucket number between zero and fifteen. The buckets represent non-uniform ranges of durations. Bucket zero corresponds to very short calls and serves as the default value for lines with no recorded activity, while bucket fifteen corresponds to long calls. Thawing converts bucket numbers to seconds by associating a default duration with each bucket. Freezing identifies the bucket with the appropriate range of times.

There are two parts to the daily Usage computation. First, we accumulate the precise usage in seconds for a particular phone number for a particular type of call, and then we *blend* that data with the existing signature for that type of call. The code for blending is:

```
blend(new, old) = new*lambda +
                  old*(1-lambda)
```

Blending of this form is common in signature computations.

Pseudo-code for computing part of the Usage signature appears in Figure 3. For brevity, we include only the code for computing the outgoing portion of the signature

```
outgoingUsage(origin, calls)
  int cumTollFree = 0;
  int cumOut = 0;

  uApprox = Get usage data for origin
  uSig = Convert uApprox from buckets
         to seconds

  for c in calls do
     if c.isTollFree then
        cumTollFree += c.duration
     else
        cumOut += c.duration
     endif
  done

  uSig.outTollFree =
     blend(cumTollFree, uSig.outTollFree)
  uSig.out = blend(cumOut, uSig.out)

  uApprox = Convert uSig from seconds
            to buckets.
  Record new uApprox data for origin
```

Figure 3: Pseudo-code for computing part of the Usage signature

for a single line, called `origin` in the pseudo-code. A detailed version of Usage that tracks some additional information can be found in Appendix A.

3 Data model

This section describes Hancock's data model, which includes a model for collections of call records and a model for profile data.

3.1 Call stream

We model a collection of call records as a stream in Hancock. Programmers use the `stream` type operator to declare a new stream type. Such a declaration names the new type and specifies both the *physical* and the *logical* representations of the records in the stream. Intuitively, the physical representation describes the (highly encoded) structure of the records as they exist on disk, while the logical representation describes an expanded form convenient for programming. The declaration specifies a function to convert from encoded physical to expanded logical records. For example, the following code declares a stream type `callStream`:

```
stream callStream {
  getvalidcall : PCallRec_t =>
                         callRec_t;
}
```

For this stream, the physical type is `PCallRec_t`, the logical type is `callRec_t`, and the conversion function `getvalidcall` constructs a logical record from a physical one. Function `getvalidcall` has type

```
char getvalidcall(PCallRec_t *pc
                        callRec_t *c)
```

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

We represent streams on disk as a directory that contains binary files. Hancock's wiring-diagram mechanism, which we discuss in Section 5, provides a way to match the name of a directory to a stream.

3.2 Signature data

Hancock provides two mechanisms for describing signature data. Programmers use the `record` declaration to specify the format of a profile and the `map` declaration to specify the mapping between phone numbers and profiles.

Records are designed to capture the relationship depicted in Figure 2. They specify the types for the signature and approximation views of a profile, as well as the freeze and thaw expressions for converting between these types. We use the following simple record to introduce the pieces of a `record` declaration.

```
record uField(ufSig, ufApprox){
  int <=> char;
  ufSig(b) = bucketToSec[b];
  uApprox(s) = secToBucket(s);
}
```

This declaration introduces three types:

- `uField`: the type of the record,
- `ufSig`: the type of the left-hand view (`int`), and
- `ufApprox`: the type of the right-hand view (`char`).

The `ufSig(b) = ...` portion of the record declaration specifies how to thaw `ufApprox b` to produce a `ufSig` value. Similarly, `ufApprox(s) = ...` specifies how to freeze a `uSig s` to obtain a `uApprox` value. In this application, `uApprox(s)` uses the function `secToBucket` to convert the seconds stored in integer `s` into a bucket number, and `uSig(b)` uses the array `bucketToSec` to convert a bucket stored in `b` into the corresponding mean number of seconds for that bucket. Together, these expressions are an example of where `thaw(freeze(s))` does not equal `s`.

Records can have more than one field (in which case the fields are named), and they can be included in other record declarations. For example, a second `record` declaration that appears in the Usage signature, `uLine`, has the following form:

```
record uLine(uSig, uApprox){
  uField in;
  uField out;
  uField outTF;
}
```

As in our earlier example, this declaration introduces three types: `uLine`, `uSig`, and `uApprox`. The type `uLine` is the type of the record. The types `uSig` and `uApprox` are equivalent to C structures constructed from the left and right types of `uField`:

```
typedef struct {
   int in;
   int out;
   int outTF;
} uSig;

typedef struct {
   char in;
   char out;
   char outTF;
} uApprox;
```

Because the record `uLine` does not include explicit freeze and thaw expressions, Hancock constructs them automatically from the freeze and thaw expressions of the record's fields. For this record, the compiler constructs the following freeze function:

```
uApprox freeze(uSig s){
  uApprox a;
  a.in  = secToBucket(s.in);
  a.out = secToBucket(s.out);
  a.outTF = secToBucket(s.outTF);
  return a;
}
```

The thaw function is constructed similarly.

Although this example record only contains fields with record types, fields may also have regular C types. In this context, C types can be thought of as records with the same left- and right-hand type and the identity function for freezing and thawing.

To convert between views, Hancock provides the view operator ($). The expression `ua$uSig` converts `uApprox ua` to a `uSig`, using the conversion specified implicitly in the `uLine` declaration. Expression `us$uApprox` behaves analogously.

Hancock's `map` declaration provides a way to associate data with keys. Typically a map does not contain data for every possible key. Consequently, Hancock supports the notion of a *default value*, which is returned when a programmer requests data for a key that does not have a value stored in the map. For example, in the following `map` declaration, the keys have type `line_t`, the data are structures of type `uApprox`, and the default value is the constant `uApprox` structure consisting of all zeros.

```
map uMap {
   key line_t;
   value uApprox;
   default {0,0,0};
}
```

Defaults may also be specified as functions that use the key in question to compute an appropriate default for that key. For example, the `map` declaration below specifies a function, `lineToDefault`, to call with the line in question when a default record is needed.

```
map uMapF {
   key line_t;
   value uApprox;
   default lineToDefault;
}
```

A common use for this mechanism is to construct defaults by querying another data source.

The identifier `uMap` names a new map type. Variables of this type can be declared using the usual C syntax (for example, `uMap usage`). Hancock provides an indexing operator <:...:> to access values in a map. The code:

```
line_t pn;

u = usage<:pn:>;
...
usage<:pn:> = u;
```

gives an example of reading from and writing to a map. The usual idiom for accessing map data combines the indexing and view operators as in:

```
us = usage<:pn:>$uSig;
```

Hancock also provides an operator `:=:` to copy maps. In particular, the statement

```
new_usage :=: usage;
```

causes `uMap` map `new_usage` to be initialized with the data from `usage`.

3.3 Discussion

By providing the programmer with appropriate abstractions, Hancock reduces the intellectual burden of writing signatures. Although programmers were freezing and thawing their data prior to Hancock, they had not abstracted this idea. The result was numerous bugs caused by confusing the types of the two views. The structure enforced by records eliminates many of these bugs by requiring programmers to document the relationship between the two views, and to apply the view operator to convert between them explicitly. As an added benefit, records simplify signature code by generating conversion functions automatically from record fields when possible.

Maps provide an efficient implementation for the most performance critical part of signature programs. The index operation is more convenient than a library interface, and it provides stronger type-checking.

4 Computation Model

Hancock's computation model is built around the notion of iterating over a sorted stream of calls. Sorting call records ensures good locality for references to the signature data that follow the sorting order. Off-direction references may not have good locality, however. For example, if we sort the call records by the originating number, then updating the usage for that number would have good locality, while updating the usage for the dialed number would suffer from bad locality. Consequently, signature computations are typically done with multiple passes over the data, each sorting the data in a different order and updating a different part of the profile data. We call each such pass, represented in Figure 1 as a dashed box, a *phase*.

A phase starts by specifying a name and a parameter list. The body of a phase has three pieces: an iterate clause, a list of variable declarations, and a list of event clauses. The following pseudo-code outlines the outgoing phase of the usage signature:

```
phase out(callStream calls, uMap usage) {
    iterate clause
    variable declarations
    event clauses
}
```

This code defines a phase `out` that takes two parameters: a stream of calls and a usage map. The iterate clause specifies an initial stream and a set of transformations to produce a new stream. Variables declared at the level of a phase are visible throughout that phase. The event clauses specify how to process the transformed stream. The next two subsections describe these clauses.

4.1 Iterate Clause

Through the iterate clause, Hancock allows programmers to transform a stream of logical call records into a stream of records tailored to the particular signature computation. The iterate clause has the following form:

```
iterate
   over  stream variable
   sortedby  sorting order
   filteredby  filter predicate
   withevents  event list
```

We explain each of these pieces in turn. The `over` clause names an initial stream to transform. The `sortedby` clause specifies a sorting order for this stream. For example, the calls could be sorted by the originating phone number or by the connect time. At present, the allowable sorting orders are hardcoded into Hancock. The `filteredby` clause specifies a predicate that is used to remove unneeded records from the stream. For example, a call stream may include incomplete calls, which are not used by the Usage signature. Removing unneeded call records before processing the stream simplifies the processing code.

The `withevents` clause specifies which events are relevant to this signature. We call the occurrence of a group of calls in the stream an *event*. Depending on the sorting order, different groups of calls can be identified in the call stream. For example, if the calls are sorted by originating number, the possible groups are:

- calls for the same area code,
- calls for the same exchange,
- calls for the same line, or
- a single call.

Within an event there are two important sub-events: the beginning of the block of calls and its ending. The possible events and sub-events are related hierarchically; calls are nested with lines, lines within exchanges, and exchanges within area codes.

Putting all these pieces together, the iterate clause for the outgoing phase of Usage has the following form:

```
iterate
   over calls
   sortedby origin
   filteredby noIncomplete
   withevents line, call;
```

This code specifies that `calls` is the initial stream, that it should be sorted by the originating number, that it should be filtered using function `noIncomplete`, which removes incomplete calls from the stream, and that two events, `line` and `call`, are of interest.

4.2 Event Clauses

The iterate clause specifies the events of interest in the stream, but it does not indicate what to do when an event is detected. The *event clauses* of a phase specify code to execute in response to a given event. We illustrate this structure in Figure 4. Programmers supply event code for the boxes, while Hancock generates the control-flow code that corresponds to the arrows.

Each event has a name that corresponds to the event name listed in the iterate clause. Each event takes as a parameter the portion of the call record that triggered the event.

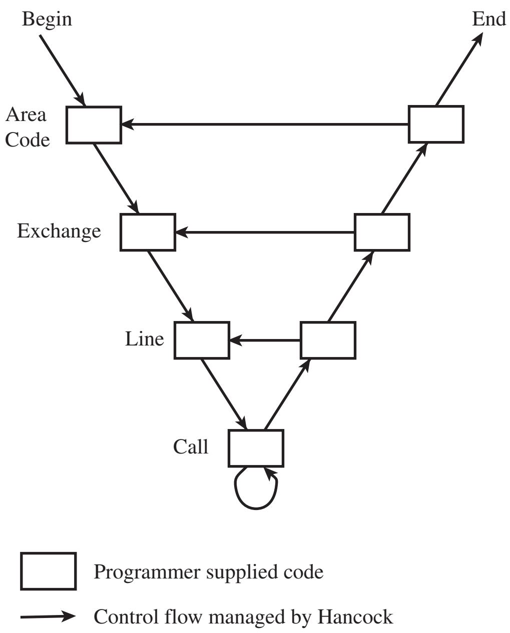

Figure 4: Hierarchical event structure

For example, an area code event is passed the area code shared by the block of calls that triggered the event. The body of each event has three parts: a list of variable declarations, a begin sub-event, and an end sub-event. Variables declared at the level of an event are shared by both its sub-events and are available to events lower down in the hierarchy, using a scoping operator (`event name::variable name`). A common pattern is for the programmer to declare a variable in the code for a line event, to initialize it in the begin-line sub-event, to accumulate information using that variable while processing calls, and then to store the accumulated information during the end-line sub-event code.

A sub-event is a kind (`begin` or `end`) followed by a C-style block that contains a list of variable declaration and Hancock and C statements. Variables declared in a sub-event are visible only within that sub-event. The code in Figure 5 implements the line and call events for the outgoing phase of Usage.

Note that the `call` event does not have sub-events. We call such events *bottom-level* events. The lowest event in any list of events is a bottom-level event. For example, Frequency, a signature that we discuss later, does not collect summary information for a set of calls. It tracks only the existence of at least one call, so `line` is its bottom-level event.

```
event line(line_t pn) {
  uSig cumSec;

  begin {
    cumSec.out = 0;
    cumSec.outTF = 0;
  }

  /* process calls */

  end {
    uSig us;

    us = usage<:pn:>$uSig;
    us.outTF =
         blend(cumSec.outTF, us.outTF);
    us.out = blend(cumSec.out, us.out);
    usage<:pn:> = us$uApprox;
  }
}

event call(callRec_t c) {
  uSig line::cumSec;

  if (c.isTollFreeCall)
    cumSec.outTF += c.duration;
  else
    cumSec.out +=  c.duration;
}
```

Figure 5: Event code for Usage signature

4.3 Discussion

Hancock's event specifications have several advantages. First, the specifications 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 use the variable-sharing mechanism to share information across events.

A common question when thinking about designing a domain-specific language is whether or not a library would suffice. We rejected the library option largely because of Hancock's control-flow abstractions. In particular, expressing Hancock's event model and the information sharing it provides proved awkward in a call-back framework, the usual technique for implementing such abstractions.

5 Wiring Diagram

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 passes and to connect command-line input to the variables in the program. The arcs between the phase boxes in Figure 1 depict this construct. The following code implements `sig_main` for Usage:

```
void sig_main(
            const callStream calls <c:>,
     exists const uMap y_usage <u:>,
     new          uMap usage <U:>) {
   usage :=: y_usage;
   out(calls, usage);
   in(calls, usage);
}
```

There are three parameters to the Usage signature. The first is a stream that contains the raw call data. The `const` keyword indicates that this data is read-only. The syntax (<c:>) after the variable name `calls` specifies that this parameter will be supplied as a command-line option using the `-c` flag. The colon indicates that this flag takes an argument, in this case the name of the directory that holds the binary call files. The absence of a colon indicates that the parameter is a boolean flag. The Hancock compiler generates code to parse command-line options. The second parameter is a Usage map, the name of which is specified using the `-u` flag. The `const` qualifier indicates the map is read-only, while the `exists` annotation indicates the map must exist on disk. The final parameter names the Usage map used to hold the result of this signature computation; the `-U` flag specifies the file name for this map. The `new` qualifier indicates that the map must not exist on disk.

In general, the body of `sig_main` is a sequence of Hancock and C statements. In Usage, `sig_main` copies the data from `y_usage` into `usage` and then invokes Usage's outgoing and incoming phases with the raw call stream and the Usage map under construction as arguments.

5.1 Discussion

The wiring diagram clarifies the dataflow between phases. For example, some signatures need to make off-direction references to signature data. A question that arises is: are these references referring to data computed in a previous phase or to data computed the previous day? This question can be answered by looking at `sig_main`. If the parameters to the phase do not include the original input map, then all references must be to the partially computed map.

The automatic generation of argument parsing code is convenient and removes a source of tedium, but its real benefit is that it connects Hancock variables to their on-disk counterparts. It 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.[1] It detects when data annotated as `new` already exists or when data tagged with `exists` is not on disk, in each case reporting a run-time error. These data-protection features are important when it is time-consuming or even impossible to reconstruct an accidentally overwritten signature.

6 Implementation

Our implementation of Hancock consists of a compiler that translates Hancock code into plain C code, which is then compiled and linked with a runtime system to produce executable code. We modified CKIT, a C-to-C translator written in ML[SCHO99] to parse Hancock and translate the resulting extended parse tree into abstract syntax for plain C. The compiler generates code for the various Hancock operators and clauses and for the `main` routine. 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 the data.

We were able to build Hancock relatively quickly by leveraging other people's software. In particular, we built the compiler on top of CKIT, an existing C-to-C translator[SCHO99] written for the purpose of building compilers for C-based domain-specific languages. We also used a collection of libraries: msort[Lin99, MMB92], sfio[KV91], and vmalloc[Vo96]. Msort pro-

[1] We intend to check for writes to `const` data at compile time eventually.

Table 1: Example signatures.

Signature	Description
Usage	Average daily usage
Frequency	Calling frequency
Activity	Days since last seen
Bizocity	"Business-likeness"

vides an external sorter and sfio supports 64-bit files, making these two libraries particularly useful in addressing issues of scale.

7 Early experiences

Table 1 briefly describes four signatures: Usage, Frequency, Activity, and Bizocity. These signatures are computed daily from call records. In this section, we discuss how these signatures use Hancock's data and control-flow mechanisms.

The maps used by these signatures all have index types of `line_t` and value types that are records. Usage, Frequency, and Activity use constant defaults. Bizocity uses a default function that queries a secondary map, which indicates whether the phone is a known residence, a known business, or unknown.

The records used in these signatures vary based on the application, but they all have a common form: the desired profile contains several fields that have the same underlying structure. To express this structure in Hancock, we use two records: one to describe the basic fields and a second to group these fields into a profile.

Table 2 describes the basic fields for the sample signatures and indicates how many such fields are contained in the profile record. In all these examples, the approximation type is a range that can be represented with a C `char` type. The signatures use different approximation techinques. *Bucketing* divides the range of signature values into disjoint buckets and associates a default value with each such bucket. With this technique, freezing converts a signature value into the containing bucket, whereas thawing returns the default value for a bucket. Bucketing can use either fixed-width or variable-width buckets. *Clamping* converts values above the range of signature values to the largest value in the range and values below the range to the lowest value in the range.

In all four signatures, the amount of Hancock code needed to describe the data is small. The largest, Bizocity, takes fewer than 30 lines.

In terms of control-flow, the example signatures share the same high-level structure, each containing two phases: one to compute information for outgoing calls and another for incoming calls. The event structures for the signatures are different, however. Frequency tracks only the existence of a call for a given number, so its bottom-level event is the line event. Activity and Usage do work at both call and line events. Bizocity uses these events and does significant computation at the exchange level.

The Hancock code that implements these phases is small: the smallest, Frequency, takes 40 lines of code; the largest, Bizocity, takes 300 lines, more than 100 of which are for processing exchange events.

In all, these examples indicate that Hancock data descriptions are compact and that the event processing code is modest in size. Ideally, we would like to compare the Hancock implementations with hand-written C implementations. Unfortunately, this comparison is very hard to do fairly. The only C implementation of Usage, Activity, and Bizocity available to us is a program that combines the computation of all three signatures and has code to manage the on-disk representations of the signature files embedded in it. This program is about 1500 lines of code.

8 Design Process

This section describes the process that we used in the design of Hancock and discusses the lessons we learned from our experience. Designing a domain specific language involves developing a model of the domain and then embodying that model in a language. The language should capture the commonalities of the domain effectively and let the domain experts worry about the details specific to an individual application. Figure 6 depicts the process we followed; it can be viewed as an elaboration of the first box of the FAST process [GJKW97].

Table 2: Record structure for example signatures

Signature	Signature type	Approximation type	Approximation method	Number of fields
Usage	int	(0-15)	variable-width buckets	4
Frequency	double	(0-255)	fixed-width buckets	2
Activity	int	(0-39)	clamping	3
Bizocity	char	(0-15)	fixed-width buckets	2

First, we alternated talking with domain experts about sample signatures and constructing models of the domain that captured the common elements of these signatures. By iterating, we got feedback on the models and developed a common vocabulary.

Once we had a good model for signatures, we applied this model by hand to construct a few sample applications. This experience led us to replace our model with one based on more primitive abstractions because the original abstractions were too high-level to serve as the basis for a programming language. We eventually used these hand-written signatures to establish that the code we expected to generate automatically would perform adequately and to evaluate whether a library-based solution was sufficient.

After we had revised our model based on the hand-written signatures, we translated it into an actual language design. This translation was not straightforward because although the model revealed what concepts needed to be expressible in the language, it did not give much guidance as to how they should be expressed. Then we wrote sample applications using our design, evaluated the resulting programs, and revised the design as appropriate. We found that we needed to iterate through this process many times. We discussed these sample applications with the domain experts to confirm that we had captured the essential elements of the domain.

Finally, we implemented a compiler and runtime system for Hancock and evaluated the signatures written in Hancock. This evaluation led us to refine many aspects of the original design, including the stream model, the view operator, the `sig_main` annotations `new` and `exists`, support for user-supplied compression functions, the `record` construct, and the implementation of maps.

8.1 Lessons Learned

Our goal in this project was to design and implement a language that would be adopted by signature researchers. While this project is not yet finished and so we cannot claim complete success at this time, we believe that we are on the right path because the signature researchers have become advocates for Hancock. We believe that three things that we did were responsible for our success and could be useful to others who want to design domain specific languages.

First, we were open to constantly revising our models based on many different kinds of input. Figure 6 highlights the many sources of feedback that we employed in our design process. In addition to consulting with the domain experts at many points in the process, we also built artifacts at several intermediate stages. These artifacts proved useful even though they are not the final product of our work.

Second, we worked closely with domain experts and valued their input. This point deserves elaboration, as it seems quite obvious that language designers should heed their experts. The fact that domain experts may not be expert programmers leads to a temptation to dismiss their comments. Sample implementations they provide may be written badly. Consequently, it is easy to leap to the erroneous conclusion that they do not know what they are doing. For example, signature researchers had developed a map representation that drew a lot of criticism from others outside the domain. When we investigated their representation, we learned that although their implementation had some problems, the basic structure provided very good random access time. Such access is crucial to their clients, but it had been ignored by other onlookers. As language designers, we

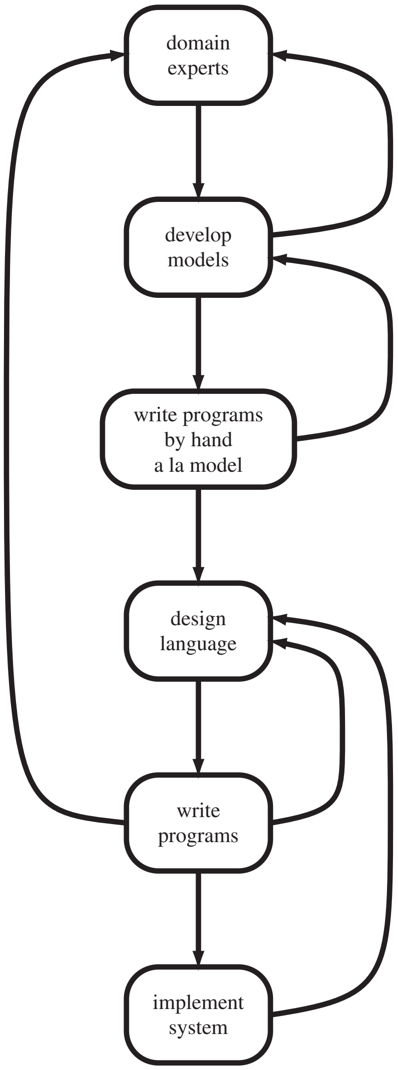

Figure 6: Design Process

had to be very careful to separate crucial domain constraints from irrelevant details in the existing implementations.

Finally, gaining credibility with domain experts was essential because they are the potential users for the language. By designing concrete models in response to our discussions with them, we established that we were serious about helping them and that we understood their domain. It also gave us a focus for our discussions. By handwriting signatures in C, we established that we could satisfy their performance requirements. By choosing C as the basis for Hancock, we kept Hancock close to their usual programming environment. By using signature representations consistent with the ones the domain experts had designed, we demonstrated that Hancock programs could manipulate their existing data without difficulty. Most importantly, by consulting with the domain experts at every point, we improved the design and got them excited about using Hancock.

9 Conclusions

Hancock handles the scale of the data used in signature computations, thereby reducing coding effort and improving the clarity of signature code. The language, compiler, and runtime system provide the scaffolding necessary to compute signatures, leaving programmers free to focus on the signatures themselves. In particular, Hancock's wiring diagram makes the relationships among phases clear. Its event model allows programmers to specify the work needed to compute a signature without having to write complicated control-flow code by hand. Finally, Hancock's data model makes writing signatures less error-prone by handling multiple data representations automatically.

We plan to extend Hancock in two ways. First, we intend to enrich the set of operations that Hancock provides for streams. For example, we plan to add a reduction operation that would allow programmers to preprocess streams to combine related records. Second, we intend to broaden the class of data that can be processed using Hancock, for example, to include Internet protocol logs or billing records. To accomplish this goal, we need to provide a mechanism for describing data streams. Such a description must include how such streams can be sorted and how to detect events based on the sorting order.

10 Acknowledgments

We would like to thank Corinna Cortes and Daryl Pregibon for their help in understanding the signature domain, Glenn Fowler, John Linderman, and Phong Vo for their assistance with the run-time system, Nevin Heintze and Dino Oliva for their help in using CKIT, Dan Suciu and Mary Fernandez for discussions which helped refine the stream model, and John Reppy for producing the pictures.

References

[ABB+97] Atkins, D., T. Ball, M. Benedikt, G. Bruns, K. Cox, P. Mataga, and K. Rehor. Experience with a domain specific language for form-based services. In *Proceedings of the USENIX '97 Conference on Domain-Specific Languages*, 1997.

[CDR+97] Chandra, S., M. Dahlin, B. Richards, R. Y. Wang, T. E. Anderson, and J. R. Larus. Experience with a language for writing coherence protocols. In *Proceedings of the USENIX '97 Conference on Domain-Specific Languages*, 1997.

[CP98] Cortes, C. and D. Pregibon. Giga mining. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, 1998.

[CRL96] Chandra, S., B. Richards, and J. R. Larus. Teapot: Language support for writing memory coherence protocols. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, 1996.

[Ell97] Elliott, C. Modeling interactive 3D and multimedia animation with an embedded language. In *Proceedings of the USENIX '97 Conference on Domain-Specific Languages*, 1997.

[GJKW97] Gupta, N. K., L. J. Jagadeesan, E. E. Koutsofios, and D. M. Weiss. Auditdraw: Generating audits the fast way. In *Proceedings of the Third IEEE Symposium on Requirements Engineering*, 1997.

[KV91] Korn, D. G. and K.-P. Vo. SFIO: Safe/fast string/file IO. In *Proc. of the Summer '91 Usenix Conference*. USENIX, 1991, pp. 235–256.

[Lin99] Linderman, J. Msort. Private communication, 1999.

[MMB92] McIlroy, M. D., P. M. McIlroy, and K. Bostic. Engineering radix sort. *Technical Memorandum 11260-920902-23TMS*, AT&T Bell Labs, Murray Hill, NJ, September 1992.

[SCHO99] Siff, M., S. Chandra, N. Heintze, and D. Oliva. Pre-release of C-frontend library for SML/NJ. See http://cm.bell-labs.com/cm/cs/what/smlnj/index.html., 1999.

[SF97] Stevenson, D. E. and M. M. Fleck. Programming language support for digitized images or, The monsters in the closet. In *Proceedings of the USENIX '97 Conference on Domain-Specific Languages*, 1997.

[TMC97] Thibault, S., R. Marlet, and C. Consel. A domain specific language for video device drivers: From design to implementation. In *Proceedings of the USENIX '97 Conference on Domain-Specific Languages*, 1997.

[Vo96] Vo, K.-P. Vmalloc: A general and efficient memory allocator. *Software—Practice and Experience*, **26**, 1996, pp. 1–18.

A The Usage signature

```
#define NUMBINS 16
int bucketToSec[NUMBINS]= { ... };
char secToBucket(int v) { ... };

record uField(ufSig, ufApprox) {
  int <=> char;
  ufSig(b) = bucketToSec[b];
  ufApprox(s) = secToBucket(s);
}

record uLine(uSig, uApprox) {
  uField in;
  uField out;
  uField outTF;
  uField outIntl;
}

map uMap {
  key line_t;
  value uApprox;
  default {0,0,0};
}
```

```
#define LAMBDA .15
#define blend(new, old) \
  (((new) * LAMBDA) + \
          ((old)*(1 - LAMBDA)))
```

```
#include calls.h
int getvalidcall(PCallRec_t *pc,
                   callRec_t *c){...}
stream callStream
  {getvalidcall: pCallRec_t =>
                      callRec_t}

char noIntlOrIncomplete(callRec_t *c) {
  return !(c->isIncomplete) &&
         !(c->isIntl);
}

char noIncomplete(callRec_t *c) {
  return !(c->isIncomplete);
}
```

```
phase out(callStream calls,
          uMap usage) {
  iterate
    over calls
    sortedby origin
    filteredby noIncomplete
    withevents line, call;

  event line(line_t pn) {
    uSig cumSec;

    begin {
      cumSec.outTF = 0;
      cumSec.outIntl = 0;
      cumSec.out = 0;
    }

    end {
      uSig us = usage<:pn:>$uSig;
      us.outTF =
          blend(cumSec.outTF, us.outTF);
      us.outIntl =
          blend(cumSec.outIntl, us.outIntl);
      us.out =
          blend(cumSec.out, us.out);
      usage<:pn:> = us$uApprox;
    }
 }

  event call(callRec_t c) {
    uSig line::cumSec;

    if (c.isTollFree)
      cumSec.outTF += c.duration;
    else if (c.isIntl)
      cumSec.outIntl += c.duration;
    else
      cumSec.out += c.duration;
  } /* end call event */
}/* end out phase */
```

```
phase in(callStream calls,
         uMap usage){
  iterate
    over calls
    sortedby dialed
    filteredby noIntlOrIncomplete
    withevents line, call;

  event line(line_t pn) {
    uSig cumSec;

    begin {
      cumSec.in = 0;
    }

    end {
      uSig us = usage<:pn:>$uSig;
      us.in =
          blend(cumSec.in, us.in);
      usage<:pn:> = us$uApprox;
    }
 }

  event call(callRec_t c) {
    uSig line::cumSec;

    cumSec.in += c.duration;
  } /* end call event */
}/* end in phase */
```

```
void sig_main(
         const callStream calls <c:>) {
  exists const uMap y_usage <u:>,
  new          uMap usage <U:>

   usage :=: y_usage;
   out(calls, usage);
   in(calls, usage);
}
```