PADS:
A domain-specific language for processing ad hoc data

Kathleen Fisher
AT&T Labs Research
180 Park Ave., E244
Florham Park, NJ
kfisher@research.att.com

Robert Gruber[*]
Google
1600 Amphitheatre Pkwy
Mountain View, CA
gruber@google.com

ABSTRACT

PADS is a declarative data description language that allows data analysts to describe both the physical layout of ad hoc data sources and semantic properties of that data. From such descriptions, the PADS compiler generates libraries and tools for manipulating the data, including parsing routines, statistical profiling tools, translation programs to produce well-behaved formats such as XML or those required for loading relational databases, and tools for running XQueries over raw PADS data sources. The descriptions are concise enough to serve as "living" documentation while flexible enough to describe most of the ASCII, binary, and Cobol formats that we have seen in practice. The generated parsing library provides for robust, application-specific error handling.

1. INTRODUCTION

Vast amounts of useful data are stored and processed in ad hoc formats. Traditional databases and XML systems provide rich infrastructure for processing well-behaved data, but are of little help when dealing with ad hoc formats. Examples that we face at AT&T include call detail data [12], web server logs [23], netflows capturing internet traffic [2], log files characterizing IP backbone resource utilization, wire formats for legacy telecommunication billing systems, *etc.* Such data may simply require processing before it can be loaded into a data management system, or it may be too large or too transient to make such loading cost effective.

Processing ad hoc data can be challenging for a variety of reasons. First, ad hoc data typically arrives "as is": the analysts who receive it can only say "thank you," not request a more convenient format. Second, documentation for the format may not exist at all, or it may be out of date. A common phenomenon is for a field in a data source to fall into disuse. After a while, a new piece of information becomes interesting, but compatibility issues prevent data suppliers from modifying the shape of their data, so instead they hijack the unused field, often failing to update the documentation in the process.

[*]Work carried out while at AT&T Labs Research.

Third, such data frequently contain errors, for a variety of reasons: malfunctioning equipment, race conditions on log entry [23], non-standard values to indicate "no data available," human error in entering data, unexpected data values, *etc.* The appropriate response to such errors depends on the application. Some applications require the data to be error free: if an error is detected, processing needs to stop immediately and a human must be alerted. Other applications can repair the data, while still others can simply discard erroneous or unexpected values. For some applications, errors in the data can be the most interesting part because they can signal where two systems are failing to communicate.

A fourth challenge is that ad hoc data sources can be high volume: AT&T's call-detail stream contains roughly 300 million calls per day requiring approximately 7GBs of storage space. Although this data is eventually archived in a database, analysts mine it profitably before such archiving [13, 14]. More challenging, the Altair project at AT&T accumulates billing data at a rate of 250-300GB/day, with occasional spurts of 750GBs/day. Netflow data arrives from Cisco routers at rates over a gigabyte per second [15]! Such volumes mean it must be possible to process the data without loading it all into memory at once.

Finally, before anything can be done with an ad hoc data source, someone has to produce a suitable parser for it. Today, people tend to use C or PERL for this task. Unfortunately, writing parsers this way is tedious and error-prone, complicated by the lack of documentation, convoluted encodings designed to save space, the need to produce efficient code, and the need to handle errors robustly to avoid corrupting down-stream data. Moreover, the parser writers' hard-won understanding of the data ends up embedded in parsing code, making long-term maintenance difficult for the original writers and sharing the knowledge with others nearly impossible.

The PADS system makes life easier for data analysts by addressing each of these concerns.[1] It provides a declarative data description language that permits analysts to describe the physical layout of their data, *as it is*. The language also permits analysts to describe expected semantic properties of their data so that deviations can be flagged as errors. The intent is to allow analysts to capture in a PADS description all that they know about a given data source and to provide the analysts with a library of useful routines in exchange.

PADS descriptions are concise enough to serve as documentation and flexible enough to describe most of the data formats we have seen in practice, including ASCII, binary, Cobol, and mixed data formats. The fact that useful software artifacts are generated from the descriptions provides strong incentive for keeping the descriptions current, allowing them to serve as living documentation.

[1]PADS is short for Processing Ad hoc Data Sources.

Given a PADS description, the PADS compiler produces customizable C libraries and tools for parsing, manipulating, and summarizing the data. The core C library includes functions for reading the data, writing it back out in its original form, writing it into a canonical XML form, pretty printing it in forms suitable for loading into a relational database, and accumulating statistical properties. An auxiliary library provides an instance of the data API for Galax [17, 5], an implementation of XQuery. This library allows users to query data with a PADS description as if the data were in XML without having to convert to XML. In addition to these libraries, the PADS system provides wrappers that build tools to summarize the data, format it, or convert it to XML.

The declarative nature of PADS descriptions facilitates the insertion of error handling code. The generated parsing code checks all possible error cases: system errors related to the input file, buffer, or socket; syntax errors related to deviations in the physical format; and semantic errors in which the data violates user constraints. Because these checks appear only in generated code, they do not clutter the high-level declarative description of the data source. The result of a parse is a pair consisting of a canonical in-memory representation of the data and a parse descriptor. The parse descriptor precisely characterizes both the syntactic and the semantic errors that occurred during parsing. This structure allows analysts to choose how to respond to errors in application-specific ways.

With such huge datasets, performance is critical. The PADS system addresses performance in a number of ways. First, we compile the PADS description rather than simply interpret it to reduce run-time overhead. Second, the generated parser provides multiple entry points, so the data consumer can choose the appropriate level of granularity for reading the data into memory to accommodate very large data sources. Finally, we parameterize library functions by *masks*, which allow data analysts to choose which semantic conditions to check at run-time, permitting them to specify all known properties in the source description without forcing all users of that description to pay the run-time cost of checking them.

Given the importance of the problem, it is perhaps surprising that more tools do not exist to solve it. XML and relational databases only help with data already in well-behaved formats. Lex and Yacc are both over- and under- kill. Overkill because the division into a lexer and a context free grammar is not necessary for many ad hoc data sources, and under-kill in that such systems require the user to build in-memory representations manually, support only ASCII sources, and don't provide extra tools. ASN.1 [16] and related systems [1] allow the user to specify an in-memory representation and generate an on-disk format, but this doesn't help when given a particular on-disk format. Existing ad hoc description languages [9, 26, 4] are steps in the right direction, but they focus on binary, error-free data and they do not provide auxiliary tools.

In building the PADS system, we faced challenges on two fronts: designing the data description language and crafting the generated libraries and tools. In the rest of the paper, we describe each of these designs. We also give examples to show how useful they have been at AT&T.

2. EXAMPLE DATA SOURCES

Before discussing the PADS design, we describe a selection of data sources, focusing on those we use as running examples. Figure 1 summarizes some of the sources we have worked with. They include ASCII, binary, and Cobol data formats, with both fixed and variable-width records, ranging in size from relatively small files through network applications which process over a gigabyte per second. Common errors include undocumented data, corrupted data, missing data, and multiple missing-value representations.

2.1 Common Log Format

Web servers use the Common Log Format (CLF) to log client requests [23]. Researchers use such logs to measure properties of web workloads and to evaluate protocol changes by "replaying" the user activity recorded in the log. This ASCII format consists of a sequence of records, each of which has seven fields: the host name or IP address of the client making the request, the account associated with the request on the client side, the name the user provided for authentication, the time of the request, the actual request, the HTTP response code, and the number of bytes returned as a result of the request. The actual request has three parts: the request method (*e.g.*, `GET`, `PUT`), the requested URI, and the protocol version. In addition, the second and third fields are often recorded only as a '-' character to indicate the server did not record the actual data. Figure 2 shows a couple of typical records.

2.2 Provisioning data

In the telecommunications industry, the term *provisioning* refers to the steps necessary to convert an order for phone service into the actual service. To track AT&T's provisioning process, the Sirius project compiles weekly summaries of the state of certain types of phone service orders. These ASCII summaries store the summary date and one record per order. Each order record contains a header followed by a sequence of events. The header has 13 pipe separated fields: the order number, AT&T's internal order number, the order version, four different telephone numbers associated with the order, the zip code of the order, a billing identifier, the order type, a measure of the complexity of the order, an unused field, and the source of the order data. Many of these fields are optional, in which case nothing appears between the pipe characters. The billing identifier may not be available at the time of processing, in which case the system generates a unique identifier, and prefixes this value with the string "no_ii" to indicate the number was generated. The event sequence represents the various states a service order goes through; it is represented as a new-line terminated, pipe separated list of state, timestamp pairs. There are over 400 distinct states that an order may go through during provisioning. The sequence is sorted in order of increasing timestamps. Figure 3 shows a small example of this format.

It may be apparent from these paragraphs that English is a poor language for describing data formats!

3. PADS DESIGN

A PADS description specifies the physical layout and semantic properties of an ad hoc data source. The language provides a type-based model: basic types describe atomic data such as integers, strings, dates, *etc.*, while structured types describe compound data built from simpler pieces.

The PADS library provides a collection of broadly useful base types. Examples include 8-bit unsigned integers (`Puint8`), 32-bit integers (`Pint32`), dates (`Pdate`), strings (`Pstring`), and IP addresses (`Pip`). Semantic conditions for such base types include checking that the resulting number fits in the indicated space, *i.e.*, 16-bits for `Pint16`. By themselves, these base types do not provide sufficient information to allow parsing because they do not specify how the data is coded, *i.e.*, in ASCII, EBCDIC, or binary. To resolve this ambiguity, PADS uses the *ambient* coding, which the programmer can set. By default, PADS uses ASCII. To specify a particular coding, the description writer can select base types which indicate the coding to use. Examples of such types include ASCII 32-bit integers (`Pa_int32`), binary bytes (`Pb_int8`), and EBCDIC characters (`Pe_char`). In addition to these types, users

Name & Use	Representation	Size	Common Errors
Web server logs (CLF): Measuring web workloads	Fixed-column ASCII records	≤12GB/week	Race conditions on log entry Unexpected values
AT&T provisioning data (Sirius): Monitoring service activation	Variable-width ASCII records	2.2GB/week	Unexpected values Corrupted data feeds
Call detail: Fraud detection	Fixed-width binary records	˜7GB/day	Undocumented data
AT&T billing data (Altair): Monitoring billing process	Various Cobol data formats	˜4000 files/day, 250-300GB/day	Unexpected values Corrupted data feeds
IP backbone data (Regulus) Monitoring network performance	ASCII	≥ 15 sources ˜15 GB/day	Multiple missing-value representations Undocumented data
Netflow Monitoring network performance	Data-dependent number of fixed-width binary record	≥1Gigabit/second	Missed packets

Figure 1: Selected ad hoc data sources. We will use the bold data sources in our examples.

can define their own base types to specify more specialized forms of atomic data.

To describe more complex data, PADS provides a collection of structured types loosely based on C's type structure. In particular, PADS has **`Pstruct`**s, **`Punion`**s, and **`Parray`**s to describe record-like structures, alternatives, and sequences, respectively. **`Penum`**s describe a fixed collection of literals, while **`Popt`**s provide convenient syntax for optional data. Each of these types can have an associated predicate that indicates whether a value calculated from the physical specification is indeed a legal value for the type. For example, a predicate might require that two fields of a **`Pstruct`** are related or that the elements of a sequence are in increasing order. Programmers can specify such predicates using PADS expressions and functions, written using a C-like syntax. Finally, PADS **`Ptypedef`**s can be used to define new types that add further constraints to existing types.

PADS types can be parameterized by values. This mechanism serves both to reduce the number of base types and to permit the format and properties of later portions of the data to depend upon earlier portions. For example, the base type `Puint16_FW(:3:)` specifies an unsigned two byte integer physically represented by exactly three characters, while the type `Pstring(:' ':)` describes a string terminated by a space. Parameters can be used with compound types to specify the size of an array or which branch of a union should be taken.

Figure 4 gives the PADS description for CLF web server logs, while Figure 5 gives the description for the Sirius provisioning data. We will use these two examples to illustrate various features of the PADS language. In PADS descriptions, types are declared before they are used, so the type that describes the totality of the data source appears at the bottom of the description.

`Pstruct`s describe fixed sequences of data with unrelated types. In the CLF description, the type declaration for `version_t` illustrates a simple **`Pstruct`**. It starts with a string literal that matches the constant `HTTP/` in the data source. It then has two unsigned integers recording the major and minor version numbers separated by the literal character **`'.'`**. PADS supports character, string, and regular expression literals, which are interpreted with the ambient character encoding. The type `request_t` similarly describes the request portion of a CLF record. In addition to physical format information, this **`Pstruct`** includes a semantic constraint on the `version` field. Specifically, it requires that obsolete methods `LINK` and `UNLINK` occur only under HTTP/1.1. This constraint illustrates the use of predicate functions and the fact that earlier fields are in scope during the processing of later fields, as the constraint refers to both the `meth` and `version` fields in the **`Pstruct`**.

`Punion`s describe variation in the data source. For example, the `client_t` type in the CLF description indicates that the first field in a CLF record can be either an IP address or a hostname. During parsing, the branches of a **`Punion`** are tried in order; the first branch that parses without error is taken. The `auth_id_t` type illustrates the use of a constraint: the branch `unauthorized` is chosen only if the parsed character is a dash. PADS also supports a *switched* union that uses a selection expression to determine the branch to parse. Typically, this expression depends upon already-parsed portions of the data source.

PADS provides **`Parray`**s to describe varying-length sequences of data all with the same type. The `eventSeq_t` declaration in the Sirius data description uses a **`Parray`** to characterize the sequence of events an order goes through during processing. This declaration indicates that the elements in the sequence have type `event_t`. It also specifies that the elements will be separated by vertical bars, and that the sequence will be terminated by an end-of-record marker (**`Peor`**). In general, PADS provides a rich collection of array-termination conditions: reaching a maximum size, finding a terminating literal (including end-of-record and end-of-source), or satisfying a user-supplied predicate over the already-parsed portion of the **`Parray`**. Finally, this type declaration includes a **`Pwhere`** clause to specify that the sequence of timestamps must be in sorted order. It uses the **`Pforall`** construct to express this constraint. In general, the body of a **`Pwhere`** clause can be any boolean expression. In such a context for arrays, the pseudo-variable `elts` is bound to the in-memory representation of the sequence and `length` to its length.

Returning to the CLF description in Figure 4, the **`Penum`** type `method_t` describes a collection of data literals. During parsing, PADS interprets these constants using the ambient character encoding. The **`Ptypedef`** `response_t` describes possible server response codes in CLF data by adding the constraint that the three-digit integer must be between 100 and 600.

The `order_header_t` type in the Sirius data description contains several anonymous uses of the **`Popt`** type. This type is syntactic sugar for a stylized use of a **`Punion`** with two branches: the first with the indicated type, and the second with the "void" type, which always matches but never consumes any input.

Finally, the **`Precord`** and **`Psource`** annotations deserve comment. The first indicates that the annotated type constitutes a record, while the second means that the type constitutes the totality of a data source. The notion of a record varies depending upon the data encoding. ASCII data typically uses new-line characters to delimit records, binary sources tend to have fixed-width records, while COBOL sources usually store the length of each record before the

```
207.136.97.49 - - [15/Oct/1997:18:46:51 -0700] "GET /tk/p.txt HTTP/1.0" 200 30
tj62.aol.com - - [16/Oct/1997:14:32:22 -0700] "POST /scpt/dd@grp.org/confirm HTTP/1.0" 200 941
```

Figure 2: Tiny example of web server log data.

```
0|1005022800
9152|9152|1|9735551212|0||9085551212|07988|no_ii152272|EDTF_6|0|APRL1|DUO|10|1000295291
9153|9153|1|0|0|0|0||152268|LOC_6|0|FRDW1|DUO|LOC_CRTE|1001476800|LOC_OS_10|1001649601
```

Figure 3: Tiny example of Sirius provisioning data.

```
Punion client_t {
  Pip       ip;      /- 135.207.23.32
  Phostname host;    /- www.research.att.com
};

Punion auth_id_t {
  Pchar unauthorized : unauthorized == '-';
  Pstring(:' ':) id;
};

Pstruct version_t {
  "HTTP/";
  Puint8 major; '.';
  Puint8 minor;
};

Penum method_t {
    GET,    PUT,  POST,  HEAD,
    DELETE, LINK, UNLINK
};

bool chkVersion(version_t v, method_t m) {
  if ((v.major == 1) && (v.minor == 1)) return true;
  if ((m == LINK) || (m == UNLINK)) return false;
  return true;
};

Pstruct request_t {
  '\"';   method_t       meth;
  ' ';    Pstring(:' ':) req_uri;
  ' ';    version_t      version :
                  chkVersion(version, meth);
  '\"';
};

Ptypedef Puint16_FW(:3:) response_t :
         response_t x => { 100 <= x && x < 600};

Precord Pstruct entry_t {
         client_t       client;
   ' ';  auth_id_t      remoteID;
   ' ';  auth_id_t      auth;
   " ["; Pdate(:']':)   date;
   "] "; request_t      request;
   ' ';  response_t     response;
   ' ';  Puint32        length;
};

Psource Parray clt_t {
  entry_t [];
}
```

Figure 4: PADS description for web server log data.

```
Precord Pstruct summary_header_t {
  "0|";
  Puint32       tstamp;
};

Pstruct no_ramp_t {
  "no_ii";
  Puint64 id;
};

Punion dib_ramp_t {
  Pint64     ramp;
  no_ramp_t  genRamp;
};

Pstruct order_header_t {
       Puint32             order_num;
 '|';  Puint32             att_order_num;
 '|';  Puint32             ord_version;
 '|';  Popt pn_t           service_tn;
 '|';  Popt pn_t           billing_tn;
 '|';  Popt pn_t           nlp_service_tn;
 '|';  Popt pn_t           nlp_billing_tn;
 '|';  Popt Pzip           zip_code;
 '|';  dib_ramp_t          ramp;
 '|';  Pstring(:'|':)      order_type;
 '|';  Puint32             order_details;
 '|';  Pstring(:'|':)      unused;
 '|';  Pstring(:'|':)      stream;
 '|';
};

Pstruct event_t {
  Pstring(:'|':) state;   '|';
  Puint32        tstamp;
};

Parray eventSeq {
  event_t[] : Psep('|') && Pterm(Peor);
} Pwhere {
  Pforall (i Pin [0..length-2] :
           (elts[i].tstamp <= elts[i+1].tstamp));
};

Precord Pstruct entry_t {
  order_header_t  header;
  eventSeq        events;
};

Parray entries_t {
  entry_t[];
};

Psource Pstruct out_sum{
  summary_header_t  h;
  entries_t         es;
};
```

Figure 5: PADS description for Sirius provisioning data.

actual data. PADS supports each of these encodings of records and allows users to define their own encodings. By default, PADS assumes records are new-line terminated. Before parsing, however, the user can direct PADS to use a different record definition.

More information about the PADS language may be found in the PADS manual [7].

4. GENERATED LIBRARY

From a description, the PADS compiler generates a C library for parsing and manipulating the associated data source. We chose C as the target language for pragmatic reasons: there were libraries that made building the compiler and run-time libraries easier, our target users are comfortable with C, and it can serve as a lingua franca in that essentially all languages have provisions for calling C libraries. Nothing about the PADS language mandates compiling to C, however, and we envision eventually building alternate bindings.

From each type in a PADS description, the compiler generates

- an in-memory representation,
- a mask, which allows users to customize generated functions,
- a parse descriptor, which describes syntactic and semantic errors detected during parsing,
- parsing and printing functions, and
- a broad collection of utility functions.

To give a feeling for the library that PADS generates, Figure 6 includes selected portions of the generated library for the Sirius `entry_t` declaration.

The C declarations for the in-memory representation, the mask, and the parse descriptor all share the structure of the PADS type declaration. The mapping to C for each is straightforward: **`Pstruct`**s map to C structs with appropriately mapped fields, **`Punion`**s map to tagged unions coded as C structs with a tag field and an embedded union, **`Parray`**s map to a C struct with a length field and an embedded sequence, **`Penums`** map to C enumerations, **`Poptions`** to tagged unions, and **`Ptypedef`**s to C typedefs. Masks include auxiliary fields to control behavior at the level of a structured type, and parse descriptors include extra fields to record the state of the parse, the number of detected errors, the error code of the first detected error, and the location of that error.

The parser takes a mask as an argument and returns an in-memory representation and a parse descriptor. The mask allows the user to specify which constraints the parser should check and which portions of the in-memory representation it should fill in. This control allows the description-writer to specify all known constraints about the data without worrying about the run-time cost of verifying potentially expensive constraints for time-critical applications.

Appropriate error-handling can be as important as processing error-free data. The parse descriptor marks which portions of the data contain errors and characterizes the detected errors. Depending upon the nature of the errors and the desired application, programmers can take the appropriate action: halting the program, discarding parts of the data, or repairing the errors. If the mask requests that a data item be verified and set, and if the parse descriptor indicates no error, then the in-memory representation satisfies the semantic constraints on the data.

Because we generate a parsing function for each type in a PADS description, we support multiple-entry point parsing, which allows us to accommodate larger-scale data. For a small file, programmers can define a PADS type that describes the entire file and use that type's parsing function to read the whole file with one call. For larger-scale data, programmers can sequence calls to parsing functions that read manageable portions of the file, *e.g.*, reading a record at a time in a loop. The parsing code generated for **`Parrays`** allows users to choose between reading the entire array at once or reading it one element at a time, again to support parsing and processing very large data sources.

The ratio of the size of the data description to the size of the generated code gives a rough measure of the leverage of the declarative description. For the 68 line Sirius data description, the compiler yields a 1432 `.h` file and a 6471 `.c` file. This expansion comes from the extensive error checking in the generated parser and the number of generated utility functions.

We discuss details of the generated library in the following section as we describe its uses.

5. PADS IN PRACTICE

Because the PADS language is declarative, we can leverage PADS descriptions to build additional tools besides the core parsing and printing library. As a result, PADS provides direct support for a number of different uses: computing with the data, developing a high-level understanding of the data through statistical summaries, converting the data into a more standard form, and querying. In this section, we give a survey of some of these uses.

5.1 General computation

5.1.1 Normalizing data

We start by showing in Figure 7 a simple use of the core library to clean and normalize Sirius data. After initializing the PADS library handle and opening the data source, the code sets the mask to check all conditions in the Sirius description except the sorting of the timestamps. We have omitted from the figure the code to read and write the header. The code then echoes error records to one file and cleaned ones to another. The raw data has two different representations of unavailable phone numbers: simply omitting the number altogether, which corresponds to the `NONE` branch of the **`Popt`**, or having the value `0` in the data. The function `cnvPhoneNumbers` unifies these two representations by converting the zeroes into `NONE`s. The function `entry_t_verify` ensures that our computation hasn't broken any of the semantic properties of the in-memory representation of the data.

5.1.2 Hancock streams

Another programmatic use of the core library is to define Hancock streams. Hancock is a domain-specific language for building persistent profiles of entities described in transaction streams [12]. Data analysts at AT&T use Hancock programs over streams of call-detail records to build profiles of phone numbers, capturing behaviors like the frequency with which a given number makes a phone call. They uses these profiles to detect fraud. Defining the input streams turned out to be one of the most difficult parts of writing Hancock programs because the parsing code had to handle erroneous values and decipher complex encodings. Also, the analysts wanted to have one stream description for their many applications, but each application could only afford to check for the errors immediately relevant to it. Hence they parameterized the stream descriptions by flags to toggle various error checking code. This application provided the initial motivation for the PADS project in general, and for masks in particular.

5.2 Statistical summaries

Before using a data source, analysts must develop an understanding of both the layout and the meaning of the data. Because documentation is usually incomplete or out-of-date, this understanding must be developed through exploring the data itself. Typical

```
typedef struct {
  Pbase_m compoundLevel;   // Struct-level controls, eg., check Pwhere clause
  order_header_t_m h;
  eventSeq_t_m events;
} entry_t_m;

typedef struct {
  Pflags_t pstate;         // Normal, Partial, or Panicking
  Puint32 nerr;            // Number of detected errors.
  PerrCode_t errCode;      // Error code of first detected error
  Ploc_t loc;              // Location of first error
  order_header_t_pd h;     // Nested header information
  eventSeq_t_pd events;    // Nested event sequence information
} entry_t_pd;

typedef struct {
  order_header_t h;
  eventSeq_t events;
} entry_t;


/* Core parsing library */
Perror_t entry_t_read (P_t *pads,entry_t_m *m,entry_t_pd *pd,entry_t *rep);
ssize_t entry_t_write2io (P_t *pads,Sfio_t *io,entry_t_pd *pd,entry_t *rep);


/* Selected utility functions */
void entry_t_m_init (P_t *pads,entry_t_m *mask,Pbase_m baseMask);
int entry_t_verify (entry_t *rep);


/* Selected accumulator functions */
Perror_t entry_t_acc_init (P_t *pads,entry_t_acc *acc);
Perror_t entry_t_acc_add (P_t *pads,entry_t_acc *acc,entry_t_pd *pd,entry_t *rep);
Perror_t entry_t_acc_report (P_t *pads,char const *prefix,char const *what,
                             int nst,entry_t_acc *acc);


/* Formatting */
ssize_t entry_t_fmt2io (P_t *pads,Sfio_t *io,int *requestedOut,
                        char const *delims,entry_t_m *m,entry_t_pd *pd,entry_t *rep);


/* Conversion to XML */
ssize_t entry_t_write_xml_2io (P_t *pads,Sfio_t *io,entry_t_pd *pd,
                               entry_t *rep,char const *tag,int indent);


/* Galax Data API */
PDCI_node_t *entry_t_node_new (PDCI_node_t *parent,char const *name,void *m,
                               void *pd,void *rep,char const *kind,char const *whatfn);
PDCI_node_t *entry_t_node_kthChild (PDCI_node_t *self,PDCI_childIndex_t idx);
```

Figure 6: Selected portions of the library generated for the `entry_t` declaration from Sirius data description.

```
P_t                 *p;
entry_t             entry;
entry_t_pd          pd;
entry_t_m           mask;
    ...
P_open(&p, 0, 0);
P_io_fopen(p, "dibbler/data/2004.11.11");
    ...
entry_t_m_init(p, &mask, P_CheckAndSet);
mask.events.compoundLevel = P_Set;
    ...
while (!P_io_at_eof(p)) {
  entry_t_read(p, &mask, &pd, &entry);
  if (pd.nerr > 0) {
    entry_t_write2io(p, ERR_FILE, &pd, &entry);
  } else {
    cnvPhoneNumbers(&entry);
    if (entry_t_verify(&entry)) {
      entry_t_write2io(p, CLEAN_FILE, &pd, &entry);
    } else {
      error(2, "Data transform failed.");
    }
  }
}
```

Figure 7: Code fragment to filter and normalize Sirius data.

questions include: how complete is the description of the syntax of the data source, how many different representations for "data not available" are there, what is the distribution of values for particular fields, *etc.* PADS addresses these kinds of questions with the notion of an accumulator. For each type in a PADS description, accumulators track the number of good values, the number of bad values, and the distribution of legal values. Selected functions from this portion of the library appear in Figure 6.

We can of course use these functions by hand to write a program to compute the statistical profile of any PADS data source. However, ad hoc sources are often simply a sequence of records, perhaps prefixed by a header, so we can create a complete accumulator program from minimal extra information. Both the web server log and the Sirius data sources exhibit this pattern.[2] Consequently, given only the names of the optional header type and the record type, the PADS system will generate an accumulator program. The accumulator report for the length field of the web server data looks as follows when run on a data set used in several studies of web traffic [24, 25]:

```
<top>.length : uint32
+++++++++++++++++++++++++++++++++++++++++++
good: 53544   bad: 3824    pcnt-bad: 6.666
min: 35  max: 248591  avg: 4090.234
top 10 values out of 1000 distinct values:
tracked 99.552% of values
 val:  3082 count:  1254  %-of-good:  2.342
 val:   170 count:  1148  %-of-good:  2.144
 val:    43 count:  1018  %-of-good:  1.901
 val:  9372 count:   975  %-of-good:  1.821
 val:  1425 count:   896  %-of-good:  1.673
 val:   518 count:   893  %-of-good:  1.668
 val:  1082 count:   881  %-of-good:  1.645
 val:  1367 count:   874  %-of-good:  1.632
 val:  1027 count:   859  %-of-good:  1.604
 val:  1277 count:   857  %-of-good:  1.601
. . . . . . . . . . . . . . . . . . . . . .
 SUMMING    count:  9655  %-of-good: 18.032
```

[2]Note that a data format that can be read in one bulk read also fits this pattern.

By default, accumulators track the first 1000 distinct values seen in the data source and report the frequency of the top ten values. In this particular run, 99.552% of all values were tracked. When generating the accumulator program (or when using the library directly), PADS users can specify the number of distinct values to track and the number of values to print in the report.

Perhaps surprisingly, the report shows that 6.66% of the length fields contained errors. A glance at the error log generated by the program (which contains all records flagged as errors) reveals that web servers occasionally store the '-' character rather than the actual number of bytes returned, a possibility not mentioned in the documentation [23].

Accumulators can also be used to profile data sources automatically. Indeed, this application motivated the initial design of accumulators. AT&T's Altair project receives roughly 4000 data files per day in various Cobol formats. This volume makes looking at each file by hand prohibitively expensive. However, accumulator profiles can be used to automatically determine which profiles have high percentages of errors and which have significantly different statistical profiles than earlier versions of the same file. To support this usage, we built a tool that automatically translates Cobol copybooks into PADS descriptions.

In practice at AT&T, accumulators have proven themselves very useful for data exploration. The Regulus project uses PADS accumulator programs to find all the different representations of "data not available," typical examples of which include `0`, a blank, `NONE`, and `Nothing`. An accumulator program revealed the two representations of missing phone numbers in the Sirius data that the example program in Section 5.1.1 repaired. Accumulators also often serve as a quick tool for iteratively refining a PADS description until only genuine errors remain.

5.3 Format Conversion

Another common need is to convert ad hoc data into a more well-behaved format, such as delimited fields suitable for loading into a spreadsheet or relational database, or into XML.

5.3.1 Formatting

To support converting ad hoc data into a delimited format, the PADS library generates a formatting function for each type. This function, an example of which appears in Figure 6, takes a delimiter list as an argument. At each field boundary, it prints the first delimiter. At each nested type boundary, it advances the delimiter list unless the list is exhausted, in which case it reuses the last delimiter. The mask argument allows the user to suppress printing of portions of the data. Programmers can use the library directly to write formatting programs by hand. However, as in the accumulator case, PADS can generate a formatting program for commonly occurring data patterns given only the header type (optional), record type, and a delimiter string. Users can further customize the generated program by specifying an output format for dates and mask values. Given the delimiter string `"|"` and the output date format `"%D:%T"`, the generated web server log formatting program yields the output shown in Figure 8 when applied to the sample data in Figure 2. To support customization, PADS allows users to provide their own formatting functions for any type.

AT&T's Regulus project uses generated formatting programs to convert various data sources into a format suitable for loading into a relational database.

5.3.2 XML Generation

PADS also supports converting ad hoc data into XML by providing a canonical mapping from PADS descriptions into XML. This

```
207.136.97.49|-|-|10/16/97:01:46:51|GET|/tk/p.txt|1|0|200|30
tj62.aol.com|-|-|10/16/97:21:32:22|POST|/scpt/dd@grp.org/confirm|1|0|200|941
```

Figure 8: Formatted CLF records.

mapping is quite natural, as both PADS and XML are languages for describing semi-structured data. One interesting aspect of the mapping is that we embed not just the in-memory representation of PADS values, but also the parse descriptors in cases where the data was buggy. This choice allows users to explore the error portions of their data sources, which can be the most interesting parts of the data. Given a PADS specification, the PADS compiler generates an XML Schema describing the canonical embedding for that data source. As an example, the following is the portion of the generated XML Schema for the `eventSeq` type in the Sirius data description.

```
<xs:complexType name="eventSeq_pd">
<xs:sequence>
<xs:element name="pstate" type="Pflags_t"/>
<xs:element name="nerr" type="Puint32"/>
<xs:element name="errCode" type="PerrCode_t"/>
<xs:element name="loc" type="Ploc_t"/>
<xs:element name="neerr" type="Puint32"/>
<xs:element name="firstError" type="Puint32"/>
<xs:element name="elt" type="Puint32"
    minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>

<xs:complexType name="eventSeq">
<xs:sequence>
<xs:element name="elt" type="event"
    minOccurs="0" maxOccurs="unbounded"/>
<xs:element name="length" type="Puint32"/>
<xs:element name="pd" type="eventSeq_pd"
    minOccurs="0" maxOccurs="1"/>
</xs:sequence>
</xs:complexType>
```

The PADS compiler generates a `write_xml_2io` function for each type, an example of which is shown in Figure 6. Given a specification of the top level type, PADS can also automatically generate a conversion program, the output of which conforms to the generated XML Schema.

5.4 Queries

Our final use-scenario is querying data. Given a data source, a natural desire is to ask questions about the data, a desire which led to SQL and its many variants for relational data and XQuery for XML data [10]. Analysts working with ad hoc data would also like to query their data, but the lack of tools generally means they code their queries in an imperative fashion in languages such as AWK, PERL, or C. Indeed, the analyst working with the Sirius data took this approach. He coded queries such as "Select all orders starting within a certain time window," "Count the number of orders going through a particular state," and "What is the average time required to go from a particular state to another particular state" in a mixture of AWK and PERL. He was able to get the answers to his questions, but he had to code the queries explicitly, and the query-related code ended up embedded in his already-brittle parsing code.

We wanted to support declarative querying over ad hoc sources, but we didn't want to invent an entirely new query language, which led us to examine existing languages. Because XQuery is designed to manipulate semi-structured data, its expressiveness matched our data sources well. We were able to code all the Sirius-related queries in XQuery. For example, the XQuery

```
$sirius/sirius/order[event[1]
   [timeStamp >= xs:date("2002-04-14") and
    timeStamp <= xs:date("2002-05-25") ]]
```

asks for all orders starting within the given time window.

Happy with XQuery's expressiveness, we worked with the designers of the Galax [17] open-source implementation of XQuery to define a data API [5]. This API presents the source as a tree to Galax. With this architecture, Galax can incorporate any data source accessible through an instance of the data API. We then extended the PADS system to produce such instances. We were able to define the bulk of the API generically, having to generate on a per type basis only a handful of functions. Figure 6 contains the key generated functions for the `entry_t` type from the Sirius data. The `node_new` function creates a node in the tree representation of the data, storing the supplied name, mask, parse descriptor, and in-memory representation. It makes the argument node the parent of the newly created node. The `node_kthChild` function takes a tree corresponding to an `entry_t` node and a child index and returns the appropriate child. For the `entry_t` type, possible children are the header, the event sequence, or a parse descriptor. At the moment, it is possible to use the resulting system to query ad hoc data sources that can be loaded entirely into memory, and a version that allows the data to be read lazily is well underway. How best to optimize Xqueries over ad hoc data sources is an open research area.

6. IMPLEMENTATION

From a PADS description, the PADS compiler generates `.h` and `.c` files that together implement the data structures and operations necessary for manipulating the types declared in the source file. The PADS run-time library implements all shared functionality, including file operations, regular expression manipulation, memory management, the provided base types, and the generic portions of the Galax data API. PADS generates a recursive descent parser that makes it easy to provide multiple entry points.

We used the CKIT library [11] to implement the PADS compiler. This library greatly facilitated the construction of the compiler as it provides a framework for extending C, typechecking the extension, and then pretty printing the result. Because of CKIT, we were able to have a working version of the PADS compiler very quickly. Currently, our compiler consists of 10,000 lines of SML/NJ code layered on top of CKIT.

The PADS run-time library comprise approximately 30,000 lines of C code, built on top of the AST [18] and SFIO [22] libraries. These libraries provide support for regular expressions, container data types, a date parsing library, and various I/O routines.

To make the collection of base types user-extensible, the compiler reads all base type specifications from files. At compile time, the user can provide a list of such files to augment the provided base types. A base type specification declares the names of the in-memory representation, mask, and parse descriptor data structures and the names of the functions that implement the parsing, writing, formatting, *etc.* operations. Support for accumulators is optional,

pads_vet	perl_vet.pl
1619.1	3310.8
1600.7	3300.9
1627.5	3205.2

pads_select	perl_select.pl
426.4	548.7
419.8	518.4
418.4	492.1

Figure 10: Elapsed time in seconds of PERL and PADS vetting and selection programs. We report the times for all three runs.

as is support for the Galax data API. The user is responsible for providing a C library with implementations of all the named types and functions. More information about user-defined base types appears in the PADS manual [7].

The PADS implementation is currently going through AT&T's software release process. When the process is finished, hopefully in the next few weeks, the source code will be available from the PADS website with a non-commercial use license:

```
http://www.research.att.com/projects/pads
```

7. PERFORMANCE

To measure the performance of PADS, we compare the parser PADS generates for the Sirius data description against a hand-written PERL program, as PERL is the language that our user base has typically used. We compare the performance of the two approaches on two different tasks: vetting Sirius data in a fashion similar to the filter program of Figure 7 and collecting the order number of all records that ever pass through a particular state. For the first task, we check all the specified properties on the data, including the constraint that the timestamps in the events appear in sorted order. For the second task, we turn off all error checking and simply output the desired order numbers on standard out. We pass as input to the selection programs the cleaned data file produced by the vetter programs. We attempted to write the equivalent PERL programs in as efficient a manner possible, given the specified tasks. For each record, the PERL vetter uses the built-in `split` operator to produce an in-memory array of the pipe-separated fields. The PERL selection program uses PERL's regular expression pattern matcher to find lines with the desired state in any position after the 13th field. Figure 9 shows the relevent regular expression, looking for orders going through state `$STATE`. PERL compiles this pattern and applies the compiled pattern to each line. The PERL vetter is 323 lines of well-commented PERL code, while the selection program is 66 lines. The PADS vetter is 153 lines of well-commented C code, while the PADS selection program is 120 lines.

We used a Sirius data file to exercise the programs. This 2.2GB file contained 11,773,843 records. The minimum number of states for an order was one, the maximum number was 156, and the average was 5.5. One of these records violated the expected sorting order on event timestamps, and 53 of them contained a syntax error. (These statistics are courtesy of the generated PADS accumulator program, a nice side-benefit of the PADS description.)

We conducted our experiments on one processor of an SGI Origin 2000 running Irix 6.5. The processor, a 500 Mhz R14000, has split 32KB instruction and data caches and an 8MB secondary cache. The machine has more than 20GB of main memory. We used PERL version 5.6.1 to execute the PERL scripts, and `gcc` version 3.2.2 with optimization level O2 to compile the PADS programs. We ran each program three times. Figure 10 reports the elapsed running times using the Unix `time` utitlity. For comparison, a PERL program that simply counts the number of records takes on average 124 seconds. The corresponding PADS program takes 81 seconds.

The PADS vetter was more than twice as fast as the PERL vetter, while the PADS selection program outperformed the PERL selection program by a smaller margin. We have not yet devoted significant time to optimizing the generated parser or the base type library, so we expect improvements are possible.

8. RELATED WORK

There are many tools for describing data formats. For example, ASN.1 [16] and ASDL [1] are both systems for declaratively describing data and then generating libraries for manipulating that data. In contrast to PADS, however, both of these systems specify the *logical* representation and automatically generate a *physical* representation. Although useful for many purposes, this technology does not help process data that arrives in predetermined, ad hoc formats.

Lex and yacc-based tools generate parsers from declarative descriptions, but they require users to write both a lexer and a grammar and to construct the in-memory representations by hand. In addition, they only work for ASCII data, they do not easily accommodate data-dependent parsing, and they do not provide auxiliary services.

More closely related work includes ERLANG's bit syntax [4] and the PACKETTYPES [26] and DATASCRIPT languages [9], all of which allow declarative descriptions of physical data. These projects were motivated by parsing protocols, TCP/IP packets, and JAVA jar-files, respectively. Like PADS, these languages have a type-directed approach to describing ad hoc data and permit the user to define semantic constraints. In contrast to our work, these systems handle only binary data and assume the data is error-free or halt parsing if an error is detected. Parsing non-binary data poses additional challenges because of the need to handle delimiter values and to express richer termination conditions on sequences of data. These systems also focus exclusively on the parsing/printing problem, whereas we have leveraged the declarative nature of our data descriptions to build additional useful tools.

Recently, a standardization effort has been started whose stated goals are quite similar to those of the PADS project [3]. The description language seems to be XML based, but at the moment, more details are not available.

9. FUTURE WORK

PADS already defines a rich collection of tools for ad hoc data processing. However, there are a number of directions for future research and possibilities for extension.

Language expressiveness. We intend to add bit-field and overlay constructs to PADS to support binary data sources better, in a fashion similar to that already supported in the DATASCRIPT and PACKETTYPES languages. We also plan to generalize the switched union construct to permit arbitrary lookahead to better support parsing for complex ASCII formats such the HTTP packet specification [6]. Further, we need to design a mechanism for specifying character encodings so that we may support Unicode [8] data sources.

Generated artifacts. For research purposes, it would be very useful to be able to generate random data that conforms to a given specification, or deviates from it in specified ways, particularly when the real data is proprietary and cannot be exposed outside of AT&T. Given a PADS specification and a characterization of the desired distribution, it should be possible to generate such data. We also plan to augment the statistical profiling library with functions that use randomized and approximate techniques to create small summaries such as histograms [21, 19], wavelet summaries [19],

```
qr/^(\d+)\|(?:[^|]*\|){12}(?:[^|]*\|[^|]*\|)*$STATE\|/;
```

Figure 9: Regular expression at the heart of the PERL selection program.

or quantile summaries[20]. Another useful tool we would like to generate from PADS descriptions is a graphical binary data editor.

Semantics. We are in the process of developing a formal semantics for the PADS language so that we may have a declarative specification of what a PADS specification means.

Application-specific customization. Ideally, PADS specifications describe everything about a given data source, but nothing about how the data is to be used. This division allows a single description of a given source to be used for multiple applications. However, different applications have different performance, verification, and error-handling needs. Currently, PADS allows users to customize the behavior of library functions at run-time by supplying appropriate masks and setting various error handlers. However, this leaves a run-time overhead. We would like to design a mechanism whereby users can specify application-specific information. The PADS system would then use this information to generate an application-specific instance of the library. Conceptually, this would amount to partially evaluating the current PADS library by specifying mask arguments at binding time.

Language bindings. Nothing about the PADS language is specific to C. It would be very interesting to develop a binding for PADS in a higher-order functional language because the pattern-matching constructs of such languages are extremely adept at expressing transformations, the natural next step after parsing data.

10. CONCLUSIONS

PADS is a declarative data description language that allows its users to describe ad hoc data sources as they are, capturing layout information and semantic constraints. The language is expressive enough to describe almost all of the ASCII, Cobol, and binary data formats we have seen in practice at AT&T, while being concise enough to serve as living documentation for those data sources.

The advantages of a declarative language for describing data are significant. First, the user is spared from having to write the parser in the first place, which is an inherently tedious process. Second, because the parser is machine generated, it can check all the error conditions necessary to guard against corrupting downstream data without cluttering user code. Third, because of its conciseness, the PADS description can serve as documentation for a data source. Given that the parser is generated from the description, there is strong incentive to maintain the description and hence the documentation as the data evolves. Finally, because we have a declarative specification, we can generate not just a parser, but also a validator, a printer, a statistical profiler, various formatting tools, query support, *etc.* In other words, we can leverage the declarative nature of the specification to build a large number of useful tools.

PADS has already been quite useful at AT&T, and as the collection of generated tools grows, it will only become more so. We look forward to developing a larger user base to get more feedback to further improve the system.

11. ACKNOWLEDGMENTS

We would like to thank Mary Fernández and Jérôme Siméon for developing the Galax data API and our summer students Ricardo Medel and Yitzhak Mandelbaum for their work on implementing the XML-related portions of the PADS system. Mary Fernández, Yitzhak Mandelbaum, David Walker, Rick Schlichting, and John Launchbury made many helpful comments on an earlier draft.

12. REFERENCES

[1] Abstract syntax description language. `http://sourceforge.net/projects/asdl`.

[2] Cisco netflow. `http://www.cisco.com/warp/public/732/Tech/nmp/netflow/index.shtml`.

[3] DFDL project. `http://forge.gridforum.org/projects/dfdl-wg`.

[4] Erlang bit syntax. `http://www.erlang.se/euc/99/binaries.ps`.

[5] Galax user manual. `http://www.galaxquery.org/doc.html#manual`.

[6] Hypertext transfer protocol – HTTP/1.1. `http://www.w3.org/Protocols/rfc2616/rfc2616.html`.

[7] PADS user manual. `http://www.research.att.com/projects/pads/index.html`. To appear.

[8] Unicode home page. `http://www.unicode.org/`.

[9] G. Back. DataScript - A specification and scripting language for binary data. In *Proceedings of Generative Programming and Component Engineering*, volume 2487, pages 66–77. LNCS, 2002.

[10] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0 An XML Query Language, W3C Working Draft, Aug 2004. `http://www.w3.org/TR/xquery`.

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

[12] C. Cortes, K. Fisher, D. Pregibon, A. Rogers, and F. Smith. Hancock: A language for analyzing transactional data streams. *ACM Trans. Program. Lang. Syst.*, 26(2):301–338, 2004.

[13] C. Cortes and D. Pregibon. Giga mining. In *KDD*, 1998.

[14] C. Cortes and D. Pregibon. Information mining platform: An infrastructure for KDD rapid deployment. In *KDD*, 1999.

[15] C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: High performance network monitoring with an SQL interface. In *SIGMOD*. ACM, 2002.

[16] O. Dubuisson. *ASN.1: Communication between heterogeneous systems*. Morgan Kaufmann, 2001.

[17] M. F. Fernández, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: The Galax experience. In *VLDB*, pages 1077–1080. ACM, 2003.

[18] G. Fowler, D. Korn, S. North, and P. Vo. The AT&T AST opensource software collection. In *Proceedings of the FREENIX Track 2000 Usenix Annual Technical Conference*, pages 187–195, 2000.

[19] A. C. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *STOC*, pages 389–398, 2002.

[20] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. How to summarize the universe: Dynamic maintenance of quantiles. In *VLDB*, page 454465, 2002.

[21] S. Guha, P. Indyk, S. Muthukrishnan, and M. Strauss. Histogramming data streams with fast per-item processing. In *ICALP*, pages 681–692, 2002.

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

[23] B. Krishnamurthy and J. Rexford. *Web Protocols and Practice*. Addison Wesley, 2001.

[24] B. Krishnamurthy and J. Wang. On network-aware clustering of web clients. In *Proceedings of SIGCOMM 2000*. ACM, 2000.

[25] B. Krishnamurthy and C. Wills. Improving web experience by client characterization driven server adaptation. In *Proceedings of WWW 2002*. ACM, 2002.

[26] P. McCann and S. Chandra. PacketTypes: Abstract specification of network protocol messages. In *ACM Conference of Special Interest Group on Data Communications (SIGCOMM)*, pages 321–333, August 1998.