Technical Report

Data Modeling of Ubiquitous System Software

Jochen Streicher

07/2014

Part of the work on this technical report has been supported by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis", project A1.

Abstract

The multitude of events and internal data structures in complex modern system software are an excellent target for data analysis. The tools to collect the data range from low-level tracing frameworks to more sophisticated ones with specialized data collection and processing languages. However, these lack information on the relationship between different data sources and between currently and already collected data. We describe a formal data model that captures the structure of data streams in the system software as well as the relationships between them.

1 Introduction

Ubiquitous systems are often subject to resource constraints. Due to the complexity of modern system software, optimization potential regarding resource consumption is hard to discover without examining the dynamic characteristics, which heavily depend on application and user behaviour. This comprises specific analyses, intended to find bottlenecks regarding latency or throughput, but also more general analysis techniques, like the detection of utilization patterns in order to use them for resource-saving adaptations to the system software. [7] Several tools were conceived to extract relevant data from the system software. Debug output or specific tracing frameworks like *LTT(ng)* [2] or *ftrace* are restricted with respect to the available data or need recompilation to add additional data sources. More generic dynamic instrumentation tools like *kprobes* allow for greater flexibility but are tedious to operate. Finally, there are frameworks that use extensible event-action languages that allow to specify the trace events in a more declarative manner, while being generic enough to allow preprocessing (e.g., aggregation) on the collected trace data. Notable examples for these are *dtrace* [1] for Solaris and *SystemTap* [4] for Linux. *Fay* [5] extends this idea to clusters of Windows machines, and uses a variant of *language-integrated queries* (FayLINQ) to process the data during collection. MobiDAC [8] is an infrastructure to dynamically collect data from mobile devices on-demand, utilizing different instrumentation tools (e.g., SystemTap).

However, collected data has only implicit semantics in these cases and contains no information on the relationship between the different data sources. Consequentially, there is also no information that allows to link them to already collected data. PiCO QL [6] can be seen as a step into that direction. It allows to specify a mapping from UNIX kernel data structures to a relational view. Using the virtual table mechanism of *SQLite*, kernel data can then be queried via actual SQL statements. There is no instrumentation, just generated helper functions that read existing data structures and convert them into a relational view. This has the advantage of not imposing any overhead during normal operation, but the disadvantage of restricted data availability. For example, the frequency or inter-arrival times of system events or data modifications cannot be captured with PiCO QL if this is not recorded by the kernel itself. In other cases it might be available by frequent queries (polling), which is clearly less resource-efficient than instrumentation. Thus, snapshot access to system software data structures is not sufficient.

We describe a way to model the structured data streams of system software as a concise form of entity-relationship models extended by data streams. We also provide an instance of this metamodel that describes the data we collected with our MobiDAC infrastructure. [8]

The rest of this document is structured as follows: Sections 2 through 4 explain all meta levels of our data model, this is, the metamodel (M2, Sec. 2), a look on the model (M1, Sec. 3), with an example for each element defined by the metamodel, and finally an idea of actual data collected according to the model (M0, Sec. 4). Section 5 concludes this document.

2 Metamodel (M2)

The basic data-generating model elements are data *sources*, *events*, and *objects*.

Name spaces help to organize the model, which is their only function. They can contain any other model element, including other namespaces.

Sources have a data value of a certain *value type* that can be retrieved at any time. The data value may change over time. This happens either asynchronously or in conjunction with an *event*. Events can thus be used to obtain a stream of updates to a data source. Data sources may also have *implicit events* that occur exactly when the data source is updated.

Value types can be simple types like an integer, string or an *enumeration*. Another value type is the *reference* to a data source or *object*. Finally, there are *complex* types, which are simply a composition of other value types. Names may be assigned to types in order to refer to them when specifying the type for, e.g., a data source.

Events occur at specific points in time and, additionally to updating data sources, they may have *context data* of a certain type. Unlike data sources, this context data is only available for its respective event instance and cannot be retrieved any time. As such, each event generates a data stream on its own, additionally to the update streams of data sources.

Implications define the relationship between updates to mutable data sources and events. An event implies a data source if the occurence of this event implies a change to the respective data. A data source implies an event if every update to that data source implies an occurence of the respective event.

Objects are closely related to objects known from programming languages. They are meant to represent operating system objects like processes or external context like WiFi access points. In the model, they are basically instantiable namespaces. As such, they can contain own data sources and events, or even objects, with, again, their own data sources. An example would be the processes in the system, multiple instances of the same object type. They contain non-mutable data like the command line, but also mutable data sources like its execution time. Furthermore, each process can have multiple open files, which are again represented as objects. Objects may have an identifier that is unique during their lifetime (e.g., the process

Listing 1: Concrete syntax (Xtext).

```
Model: members+=Element (';' members+=Element)∗ ';'?;
Element: "const" ConstSource | Mutable | Event | PureImplication | Namespace | "type"
    NamedType;
Mutable: Object | Source;

/∗
 ∗ model elements
 ∗/
Namespace: name=QualifiedName ('{' members+=Element (';' members+=Element)∗ ';'? '}')?;
Event: name=QualifiedName '(' implications+=Implication∗ ')' (':' type = Type)?;
ConstSource: name=QualifiedName ':' type=Type;
Source: name=QualifiedName (implicitEvent?='!')? ':' type=Type;
Object: name=QualifiedName '[' (reftype = Type)? ']'  (':' general = [Object | QualifiedName])? ('{'
     members+=Element (';' members+=Element)∗ ';'? '}')? implications+=Implication∗;
PureImplication: events+=[Event | QualifiedName] (',' events+=[Event | QualifiedName])∗
    implications+=Implication+;

//  implications
Implication: type=ImplType  right=[Mutable | QualifiedName];
enum ImplType: Implied="<=" | Implies="=>" | Iff="<=>";

/∗
 ∗ type system
 ∗/
Type: "enum" Enum | Simple | '{' Complex '}' | '&' Reference | TypeRef;
Reference: referred = [Mutable | QualifiedName];

//  simple types
Simple: type = SimpleType;
enum SimpleType: Int="int" | Str="str"  | Float="float"  | Bool="bool" |  Void="void";

//  enums
Enum: values+=EnumValue (',' values+=EnumValue)∗;
EnumValue: name=ID;

//  complex types
Complex: members+=MultiTyped (';' members+=MultiTyped)∗ ';'?;
MultiTyped: names+=QualifiedName (',' names+=QualifiedName)∗ ':' type=Type;

//  named types
NamedType: name=QualifiedName ':' type=Type;
TypeRef: ref = [NamedType];
```

ID). If they are part of another object, they are only unique inside their parent instance (e.g., file descriptors). In that case, a unique identifier consists of the own identifier and the unique idenfier of the parent instance. For many types of operating system objects there are more specialized subtypes. For example, directories are a special type of file. This is covered by a generalization relation that allows to specify subtypes of objects (e.g., the directory). These subtypes may extend their generalized version (e.g., the file), by additional content.

Listing 1 shows the concrete syntax of the modeling language we used to describe the data model in the following section and the appendix. The notation of the concrete syntax is based on Xtext [3], a textual modeling framework for the *Eclipse* IDE.

3 Model (M1)

Listing 2 shows an example data model that covers all of the metamodel's elements and relations between them. An example for a simple integer *source* is battery.level in the *namespace* battery. It has an *implicit event*, which is just a shorter notation for an event that is *implied* by battery.level and can be used to capture every change to the battery level. A source with a *reference* as data type is process.current. The data consists of a reference to the currently running process, a processes.process *object*. Each process instance contains its own data: A *constant* data source (its command line cmdline) and a mutable data source (the time spent in usermode utime). It also contains own objects, namely references to open files. The *event* context_switch updates (*implies)* processes.current and also is always invoked when processes.current is updated (is *implied* by current). A directory (fs.dir) is a special type of fs.file, which contains fileref entries that consist of a name and a reference to a file. Since there is more than one event that updates the content of directories, all events necessary to get every change to directory contents (create, unlink, rename, . . .), are denoted by a multi-implication.

The appendix contains a model of the data we captured with MobiDAC. [8]

4 Relational and Stream Interpretation (M0)

The purpose of this section is to provide an idea of what collected data looks like when collected according to the model. The data model encompasses both a static relational snapshot view on the data as well as a dynamic stream view.

The current state of objects and sources can be interpreted as a relational database, similar to PiCO QL. [6] Table 1 shows relational snapshots for three OS object types. A table for an object contains its ID, and the current values of its data sources. If it is a subordinate object (i.e., contained in another object), there is also a reference to its parent instance, which might be the unique ID for the parent.

Events and updates to objects and data sources can be interpreted as a data stream. Table 2 shows data streams for an object, a source, and an event. Every tuple of a

Listing 2: Example model.

```
battery {
  level! : int;
};

fs {
  file[int]; // files
  regular[] : file { // regular files
    lcount : int; /* number of hard links to this file */
  };
  dir[] : file { // directories
    fileref[] { // directory entry
      name : str;
      file : &file;
    }
  };

  create( /* File is created */
    => dir.fileref /* Directory */
  );

  open( /* A process opens a file */
    => processes.process.of /* modifies the list of open files of a process */
  ) : &fs.dir.fileref; // the file reference used for access

  hlink(
    => dir.fileref  /* the newly created file reference */
  ) : &dir.fileref; /* an existing file reference */

  /* ... */

  hlink, unlink, create, rename, mkdir, rmdir // multi-implication
    <=> file
    <=> dir.fileref;
};

processes {
  current: &process; // currently running process

  process[int] {
    const cmdline : str; // command line
    utime : int; // time spent in user mode
    of[int] { /* files currently opened by the process */
      file : &fs.file;
    };
  };

  context_switch ( // switch between processes
    <=> current
    <=> process.utime
  ) : {from, to : &process};
}
```

processes.process				
ID	tcomm	utime	stime	. . .
1	init	1	5	. . .
5	loopd	1000	10	. . .
6	nc	20	20	. . .
. . .	. . .	. . .	. . .	. . .

fs.file			
ID	size	lcount	. . .
1232	12132	2	. . .
45721	935	1	. . .
213	128	1	. . .
. . .	. . .	. . .	. . .

processes.process.of				
&process	*ID*	file	pos	. . .
2	3	1232	4096	. . .
5	3	45721	0	. . .
6	3	20	132	. . .
. . .	. . .	. . .	. . .	. . .

Table 1: Relational snapshots of objects processes.process, fs.file, and processes.process.of.

processes.process		
time	&process	*exists*
0.271	5	true
0.281	2	false
0.425	6	true
. . .	. . .	...

processes.process.utime		
time	&process	utime
0.125	1	220
0.281	2	110
0.301	5	18
. . .	. . .	. . .

processes.context_switch			
time	*&process*	from	to
0.125	1	1	2
0.281	2	2	5
0.301	5	5	9
. . .	. . .	. . .	. . .

Table 2: Streams of object processes.process, data source processes.process.utime, and event processes.context_switch.

source, object or event stream contains a timestamp and, if it is contained in an object, a reference to its parent instance. Streams for sources additionally contain the new data values. Streams for object types additionally contain information about the appearance and disappearance of instances instead (i.e., processes that are started and terminated). Finally, the tuples of event streams additionally contain the event's context data as well as references to the modified data sources' and objects' parent instances.

5 Conclusion

We described our model of the structure and relationships of dynamic system software data. As future work, we intend to extend it by a query language that allows to declaratively collect, filter, join, and aggregate the data defined by the model. One purpose would be to easily create feature vectors that are amenable for data analysis. Whether relational snapshots are sufficient for information retrieval scenarios set aside, we plan to not only extract and analyze data, but to use it directly in the system software for context- and utilization-aware adaptation. This demands reaction to events, rather than evaluating periodic snapshots.

References

[1] Bryan Cantrill, Michael W Shapiro, Adam H Leventhal, et al. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28, 2004.

[2] Mathieu Desnoyers and Michel Dagenais. Lttng: Tracing across execution layers, from the hypervisor to user-space. In *Linux Symposium*, volume 101, 2008.

[3] Sven Efftinge and Markus Völter. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, page 118, 2006.

[4] Frank Ch. Eigler and et al. Architecture of SystemTap: a Linux trace/probe tool. `http://www.cs.ucsb.edu/~grze/papers/profile/eigler05systemtap.pdf`.

[5] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. Fay: Extensible distributed tracing from kernels to clusters. *ACM Transactions on Computer Systems (TOCS)*, 30(4):13, 2012.

[6] Marios Fragkoulis, Diomidis Spinellis, Panos Louridas, and Angelos Bilas. Relational access to unix kernel data structures. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 12:1–12:14, New York, NY, USA, 2014. ACM.

[7] Peter Fricke, Felix Jungermann, Katharina Morik, Nico Piatkowski, Olaf Spinczyk, Marco Stolpe, and Jochen Streicher. *Towards Adjusting Mobile Devices To User's Behaviour*, volume 6904 of *Lecture Notes in Computer Science*, pages 99–118. Springer-Verlag, Berlin, Heidelberg, 2011.

[8] Jochen Streicher, Nico Piatkowski, Katharina Morik, and Olaf Spinczyk. Open smartphone data for mobility and utilization analysis in ubiquitous environments. In Martin Atzmüller and Christoph Scholz, editors, *Proceedings of the 4th International Workshop on Mining Ubiquitous and Social Environments (MUSE)*, 2013.

Appendix: MobiDAC Data Model

This model describes the dataset collected with the MobiDAC infrastructure. While MobiDAC as well as the data model are extensible, this one corresponds to the dataset described in [8]. It is split into parts for Linux-specific (Kernel-internal data structures), mobile-device-specific (sensors, connectivity and communication peers), and Android-specific data.

```
/*
 * Linux-specific
 */

net {
    device[str] {
        rx_bytes : int; // number of bytes received
        rx_dropped : int; // number of dropped incoming packets
        rx_packets : int; // number of received packets
        rx_errors : int;
        tx_bytes : int; // number of transferred bytes
        tx_dropped : int; // number of dropped outgoing packets
        tx_packets : int; // number of outgoing packet
        tx_errors : int;
    };
};

processes[int] { // currently running processes (from /proc/[PID])
    cmdline : str; // the full command line that was used to start the process
    tcomm : str; // the executable, truncated to 15 characters
    state : enum sleeping, sleeping_nonint, running, zombie, stopped; // the current process state
    utime : int; // the number of jiffies (time unit) the process spent in user mode
    stime : int; // the number of jiffies (time unit) the process spent in kernel mode
    cutime : int; // the number of jiffies (time unit) the childs of this process spent in user
     mode
    cstime : int; // the number of jiffies (time unit) the childs of this process spent in kernel
     mode
     priority : int; // the process's priority
    nice : int; // the process's nice value (effects the priority )
    num_threads : int; // the number of threads belonging to the process
    start_time : int; // the time the process started (UNIX timestamp)
    vsize : int; // virtual memory size
};

cpu {
    load {
        onemin : float; // CPU load averaged over 1 minute
        fivemin : float; // CPU load averaged over 5 minutes
    }
};

memory {
    buffers : int; /* size of currently allocated buffer memory (kB) */
    cached : int; /* size of currently allocated cache memory. (kB) */
     dirty : int; /* size of dirty cache pages */
    free : int; /* free random access memory (kB) */
    total  : int; /* total  size of available  memory (kB) */
    writeback : int; /* total  size of pages currently being written  back (kB) */
};

/*
 * Mobile device specific
 */

type mac_address : int;

battery {
    health  : enum unknown, good, overheat, dead, overvoltage, failure;
    level  : int; /* charge level  */
    plugged : bool; //  plugged in to  A/C charger?
    present : bool; //  whether a battery is  present
    status  : enum unknown, charging, discharging, not_charging, full;
    temperature : float;
    const technology : str; //  usualli  Li−Ion
    voltage  : float;
};

bluetooth {
    active  : bool;
    visible  : bool;
    device[mac_address] {
        name : str; //  the name of this device
        class : int; //  the class of this  device
        bondstate : int; //  a numerical value telling  whether this device is  currently  paired with
     us
        prev_bondstate : int; //  the previous bondstate
    };
    connected[mac_address]; // Currently connected devices
};

device {
    id  : str; //  the IMEI of the device
    type devicetype: enum GSM, CDMA, SIP, unknown; // the phone type (GSM, CDMA, SIP,
     unknown)
};

type location_provider  : str; //  the location  provider (gps, network, passive, ...)

positioning [location_provider] {
    location  : {
        accuracy : float; //  accuracy of the given position  data in  meters
        altitude  : float; //  altitude  in meters
        latitude  : float; //  latitude  in degrees
        longitude  : float; //  longitude  in degrees
        bearing : float; //  heading in degrees off true  north
        speed : float; //  estimated speed in m/s
        time  : float; //  time supplied by GPS or network
    };
    update(<=> location);
};
```

```
network {
    cells { /* information about network cells */
        cid : int;
        lac : int;
        neighbors[] { /* list of neighbor cells and their signal strenghts */
            cid : int;
            rssi : float; /* signal strength of this cell */
        };
    };
    operator { /* details about the current network operator */
        id : int; /* ID (mobile country code + mobile network code) */
        name : str; /* operator name */
    };
    signal {
        cdma_dbm : float;
        cdma_ecio : float;
        evdo_dbm : float;
        evdo_ecio : float;
        gsm_bit_error_rate : float;
        gsm_signal_strength : float;
    };
    roaming : bool; /* whether the phone is roaming */
    ntype : int; /* current network type (e.g. EDGE, GSM, UMTS, ...) */
};

phone {
    const incomingNumber: str; // the phone is currently being called from that number
    line1Number : str; // the phone's first own number (usually, there is only one)
    state : enum idle, ringing, offhook; // the current phone state (idle , ringing , offhook)
};

screen {
    on! : bool; /* is it on? */
};

sim {
    state : enum absent, locked, pin_required, puk_required, ready, unknown;
    subscriber_id : str; /* the IMSI of the subscriber */
};

sensors {
    type vector : {x, y, z : float};
    acceleration : vector;
    mag : vector;
    orientation : {pitch , yaw, roll : float};
};

wifi {
    connection { /* information about the current wifi connection */
        bssid : str;
        hidden_ssid : str; /* a hidden SSID, if available */
        ip_address : int; /* the current IP address of the device in the network it is connected to
    */
        link_speed : int; /* the current connection speed */
        rssi : float; /* signal strength */
        ssid : str; /* name of the network */
    };
    supplicant_state : int; // Current state of the supplicant's negotiations
    scan[] { // List of scanned WiFi access points
        bssid : str;
        capabilities : str; // list of capabilities (regarding wireless security)
        frequency : float; // WiFi channel frequency for that AP
        level : float; // signal strength
        ssid : str; // SSID of the AP's network (Anm.: BezÃ¼gl. hashen gilt das gleiche wie oben)
    };
};

/*
 * Android specific
 */

packages {
    type name : str;
    launchable[name] { // installed packages
        visible_name : str; // the visible name of that application
    };
    running[name] : launchable { // running packages
        process : &processes;
    }
};

settings {
    airplanemode : bool; // whether the phone is in airplane mode
    screen {
        brightness : int;
        timeout : int; /* time between last user interaction and automatic screen turnoff */
    };
    media {
        const maxvolume : int; // maximum possible volume for media
        volume : int; // current media volume
    };
    notifications {
        vibrate : bool; // whether the phone vibrates on new notifications
    };
    ringer {
        const maxvolume : int; // maximum possible volume of the ring sound
        silent : bool; // whether the phone is set to silent mode
        vibrate : bool; // whether the phone is set to vibrate
        volume : int; // current volume of the telephone ring sound
    }
};
```