Working Group 2: Developing a Data Standard for Computing Education Learning Process Data (DATASTAND)

Description: A key outcome of the recent CS Education Infrastructure Workshop in Pittsburgh (see www.cssplice.org) was that our community needs to develop a data standard for learning process data generated by students' activities in integrated development environments, learning management systems, and other learning environments. Since the developers of a variety of computing education learning environments will be present at this workshop, we think the workshop serves an ideal venue for jump-starting the development of this standard. This working group will take up that challenge.

Group members: John Stamper (leader), Stephen Edwards, Andrew Petersen, Thomas Price, Ian Utting

Key RQs (mined from questionnaires):

1. Why might we want to capture log data, and how does the answer to that question influence the format in which we capture it?

2. How can log data be integrated with source code snapshot data?
3. What data standards presently exist for log data? Can we leverage them?
4. What is the best level (snapshots, keystrokes, etc.) at which to capture programming data? Which levels should the data standard support?
5. What features should the log data format have to allow for it to be synchronized with or tagged with more abstract or synthesized events that span a longer timespans?
6. What log data format will be universally applicable to any IDE used in computing education?
7. How can we capture learners' assigned programming tasks/problem-solving activities alongside their programming activities, in order to provide context for those activities?
8. Can and should we log learners' visitation of external sites during the programming progress?
9. Where should log data be stored?

Presentation
https://docs.google.com/presentation/d/1aJtIGUD1zLGeOMaoDkI3lWqZZTjNpVOJwczaJcCl7Wo/edit#slide=id.p

Executive Summary of Data Format

The format serves as a method for data *providers* to get information to data *consumers* in a common, structured way. At the core of the data format is the events table:

Events Table: Each row represents an event that occurred and each column represents some attribute of that event. The table is represented as a large, sparse table, with many columns that may be blank for many rows. The columns are defined as follows:

- At a minimum, each event has a provider-unique ID, a type, and a timestamp.
- A final standard should define a number of standardized columns for each event type with agreed-upon meanings. These columns are blank for non-applicable rows.
- Each row can give a ProjectVersionID, which points to an entry in the projects store, representing the state of a code project at some point in time.
- Additionally, each row has a number of *grouping key columns*, such as SessionID, SubjectID, AssignmentID, etc., which can *optionally* point to an entry in an additional table with more information on the grouping attribute.
- Each row may also have a ParentEventID, which points to an earlier event to which it belongs. For example, Compile events may have CompileOutcome child events.

In addition, there are a number of additional data sources that can be provided as separate tables, or through an ID lookup service/API:

Project Store: Each entry is identified by a provider-unique ID and represents a programming project at some point in time. This can be provided as a folder of source files.

- Source files should be identified by some (locally) unique identifier, which is kept invariant across versions as best as possible. This may be a relative file path.
- Understandably, the invariant nature may fail if a file is renamed, but this should be identified by a FileRename event.
- If provided as an API, this may additionally support retrieving individual files (from an ID) from a version of a project, or even a diff between versions.

Additional Tables: Data providers may define any number of additional tables, identified by *grouping keys*. Examples include Subjects, Assignments, Courses, Sessions, etc.

- The final standard should define some common tables and columns for these tables.
- Table entries are *immutable*, such that they should only have columns that don't change over time. For example, a participants table should not include "current course" as a column, since that can change. Instead, Event table rows can point to a courseID in a Courses Table.
- In lieu of a table, a *grouping key* may just be a URI that points to an external resource, e.g. an assignment page on a course website.

How to get data: Data can be accessed in two ways. First, a data provider can create an API for an *existing* data store that allows data to be accessed according to this specification. Second, the data can be exported to a set of tables. A provider could provide a mix-and-match approach as well, e.g. with the events table being exported and the project store being queried over an API.

- If provided as an API, the Events Table should allow for some basic SQL-like querying vocabulary, e.g. SELECT and WHERE. These may not be supportable by all providers, but they will return *at least* the columns and rows requested.

Proposed Event Table Format

The log file is a flat table of events. Entries in the table are immutable: the intention is that rows can be added to the table but rows may not be updated.

Each event contains a set of *grouping keys* that can be used to group related events and, optionally, may be used to query the provider for additional information (i.e., it is a key to another table).

Type Definitions

ProjectVersionID: Provider-unique identifier that may be used to request the state of the code.

FileID: With a ProjectVersionID, may be used to request a specific version of a file. Invariant across ProjectVersionIDs modulo renaming.

Universal Fields

EventType *string*
Records the type of event. Required.
Programming: (File.Edit, Compile, Compile.Error, Compile.Warning, Submit, Run.Program, Run.Error, Run.Warning, Run.Test, Run.Test.Result, …)
Debugging: (...)
Project Management: (Project.Open,Project.Close,File.Open,File.CloseFile,File.Rename, …)
Sessions: (SessionStart, SessionEnd)
Resources: (Resource.View, …)
Interventions: (Intervention, …)

EventID *key*
Column-unique identifier. Invariant across data requests. Required for hierarchical events.

ParentEventID *key*
Required for children of hierarchical events.

Timestamp *datetime*
Required for ordering. The source of the timestamp should be documented.
There may be Timestamp.Server, Timestamp.Client, etc. fields to specify sources.

Grouping Keys

All grouping keys are nullable (not required).

SubjectID *key*

The identifier for the user/subject/student/person associated with the event. Each unique subject must have its own identifier that is distinct from other identifiers in the same column that correspond to different subjects.

TeamID *key*
The identifier for the team/group of users associated with the event, if any. Each unique team must have its own identifier that is distinct from other identifiers in the same column that correspond to different teams of subjects.

ToolInstanceID *key*
The identifier for the tool instance/installation associated with the event. Each unique tool instance must have its own identifier that is distinct from other identifiers in the same column that correspond to different tool instances.

SessionID *key*
An identifier for the session containing this event. Each unique session must have its own identifier that is distinct from other identifiers in the same column that correspond to different sessions. Can be the EventID of the SessionStart event that initiated the session, or an external identifier into an auxiliary session table.

CourseID *key*
The identifier for the academic course associated with the event. Each unique course must have its own identifier that is distinct from other identifiers in the same column that correspond to different courses.

CourseSectionID *key*
The identifier for the course section associated with the event. Each unique course section must have its own identifier that is distinct from other identifiers in the same column that correspond to different course sections.

TermID *key*
The identifier for the academic term associated with the event. Each unique term must have its own identifier that is distinct from other identifiers in the same column that correspond to different terms.

AssignmentID *key*
The identifier for the assignment associated with the event. Each unique assignment must have its own identifier that is distinct from other identifiers in the same column that correspond to different assignments.

LearningObjectID *key*

The identifier for the learning object associated with the event. Each unique learning object must have its own identifier that is distinct from other identifiers in the same column that correspond to different learning objects.

Programming Events

Compile

CodeState *ProjectVersionID*
The state of the code at the end of the compile event. The is a reference that can be used to look up the project code state via a separate API or table.

Sources *List(FileID)*
The list of input files. Each file identifier is a reference used to indicate a specific file, relative to the corresponding CodeState associated with the event. A file's identifier, together with the corresponding ProjectVersionID, should be usable to retrieve the contents of the file via a separate API, table, or store.

Compiler *string*
The name, including version, of the toolchain used.

Initiator *string*
{'User', 'Tool', …}

Result *string*
{'Success', 'Warning', 'Failure', 'TBD'}

{Compile,Run}.Error

...

Session Events

SessionStart

Initiates a session, which indicates when a provider begins to collect related events from a particular user (or team) as part of some notion of a "session" of actions. The provider should document how they use sessions.

Provider *string*
The name and version of the provider.

SessionEnd

Indicates when a provider no longer expects related events from a particular user's (team's) session. Frequently unknowable or omitted.

Really hard problems we thought valiantly about

- Timestamps
    - Is the canonical timestamp client or server?
        - We recommend server.
    - Should we have an additional client timestamp field?
        - Yes, optionally.
    - Should we have a duration/end time as well? Or a separate end event? Does the timestamp imply the start of an event with duration?
        - We're not sure. Probably should have an optional duration field that needs to be well documented by the provider.
    - What about events without a timestamp (e.g. classroom test taken).
        - We force providers to make them up. No timestamp no service.
- Snapshots
    - Should Snapshots be a column or a separate API?
        - An API. We may want to query diffs or whole source files or projects.
    - How do we identify source files within a snapshot?
        - With an ID that is unique to a snapshot, invariant across snapshots.
        - Renames are challenging, and we hope they get logged, but it does make it impossible to guarantee the invariance of source IDs between snapshots.
    - How do we identify the source-type of the file (e.g. Java, Python, resource)
- Compiles
    - How do we relate compile events to their inputs and outputs?
        - We have parent/child events. The compile output list the compile event as a parent. In this sense we have a very minimal hierarchical structure.
    - Do we list "success/failure" on compile events?
        - Yes, because it's useful, but optionally because compiles are asynchronous and their output isn't always know when initiated.
    - Will successful compile events have out-of-the-box compilable code?
        - *Not necessarily*: plenty of projects have code fragments or library dependencies that will make this very difficult.
- Users
    - Does user info and identifier info go in *every* row (stateless) or do we include it one and have to look it up (stateful). Should users be a separate table, like code? There's a real tension between easy use and reducing redundant info.
        - General intuition: every row should contain grouping identifiers (e.g. studentID, courseID, assignmentID). These IDs *may* be pointers to rows in *other* tables/API-endpoints with more attributes
        - These entries are *immutable*, in the sense that we only record things as attributes that don't change for the object.
- The Event Table

    - Stateful/stateless:
        - We really want the events table to be as stateless as possible. So there's redundancy of "grouping" values (e.g. studentID, courseID, etc).
    - Flatness
        - Our event table is flat, meaning that it can be exported to a giant table. Rows are meaningful in isolation.
        - Caveat: there is a minimal hierarchical structure in which events can have parent events (e.g. compiles have compile result events, tests have test result events).
        - This was very important to us because we wanted to export to a format people actually use in analysis.
- What about non-textual languages/interfaces, e.g. block-based, Stride
    - We don't have a comprehensive solution, but we generally punt to the data provider to decide how to use columns in light of their data. Largely we depend on the documentation of the provider to be clear about how they interpret.
- Where do we store the data?
    - Want our format to be both an API specification and a data-format. For the most part, we expect that people already have data, and we want to enable them to allow access to it in a common format. It also allows for easy export into a set of CSV files, which could easily be stored in a repository.

Ideas we're kicking around

- How to use existing standards: xAPI
- How to make a database that can fit multiple input sources
    - How to balance flexibility and common fields
- Will people be willing to conform to a data standard?
    - Will they translate their data into the standard? Rewrite their logger?
    - Would it be appealing if there was common tooling?
    - Will people make tools?
- What is our "data model" - the specific format for programming data that sets it apart
- How do we share analysis tools, when context is so important?
- One advantage of having a format: we can write scripts for other people's data because we know what it will look like
- How do we describe courses and assignments?
    - Terms differ around the world for courses, assignments, credits, etc.
- What about when we don't know the student/assignment/etc at the outset? How do we organize data around these fields?
    - They can start as null and get filled in.
- Should this support post-hoc analysis or stream.
    - It almost has to be streaming, so we can make interventions that operate on this format.
- Is this a data format, where data is translated (duplicated) to match it? Or do we write an API and let people implement it for whatever datastore they already have.
- Do we want to limit to IDE data?
    - Advantage: we narrow our focus and make this more feasible
    - Downside: we rule out a lot of valuable data and context
    - What about things like an e-book? Is that different than a IDE?
- What about non-editing programming, e.g. Parsons problems
- How do we deal with collaboration and pair-programming and code sharing?
- Flat or hierarchical data

Reasons we want to collect programming log data

- We want to answer questions that require more than submitted code artifacts.
    - The process of creating code artifacts.
    - The learning process across a course.

Reasons we want to have a common data standard

- Replicability: I want to be able to run your study on my dataset with less work
    - Of course there's still a lot of work to deal with context.
- Comparing solutions to the same problem (e.g. EQ/WATWIN/NPSM)
    - If you claim X > Y, let me test that.
- Sharing metrics, analyses and code
- Making a common codebase for logging (and/or data analysis) that would make life easier
    - But do we really believe this is possible? Probably not.

Pre-Workshop Thoughts on RQs

1. Why might we want to capture log data, and how does the answer to that question influence the format in which we capture it?
    a. **Thomas**: Possible reasons include 1) experimental/controlled studies to evaluate an intervention/hypothesis, 2) post-hoc data mining to identify programming patterns, strategies, misconceptions and their relationship to outcomes, 3) to model student programming knowledge, 4) to generate data-driven support, 5) to help teachers and curriculum designers make better decisions. While each of these have certain requirements and constraints, I think they largely require the same information, with the general goal of being able to reconstruct a student's problem-solving process.
2. How can log data be integrated with source code snapshot data?
    a. **Thomas**: From my experience, the snapshots are the most interesting piece of the log data, and the rest of the logs are mostly helpful for putting the snapshots in context. I would advocate for a source-code-centric logging perspective. For clarification, we may want to more clearly define "snapshot," since it may imply different things to different people (e.g. is the snapshot frequency per edit/save/compile/run?).
    b. Andrew: I agree with Thomas that the snapshots are what we currently find interesting, but I've felt a need for context to help interpret them. I would find value in snapshots -- however they are defined -- embedded within a timeline or activity stream defined by the log.
3. What data standards presently exist for log data? Can we leverage them?
    a. **Thomas**: I'm not aware of any ISO-style standards. The PSLC Datashop gives a good example of a log data format we could build on. Industry has many best practices, but they may also have slightly different objectives (e.g. catching bugs, market analysis). Basic practices I'm aware of are: a) log every student interaction with the UI, b) ensure that you can reproduce the state of the IDE at any point in the logs, c) test your logging *and* do a dry-run of your analysis before a deploy, so you figure out if you're missing important data.
4. What is the best level (snapshots, keystrokes, etc.) at which to capture programming data? Which levels should the data standard support?
    a. **Thomas**: I would log every "action" the student makes. Some IDEs have a discrete notion of an action (e.g. in Snap, every time a block is moved), but for most we have to define it somewhat arbitrarily. My hunch is that for text-based IDEs, a good approach would be to log a snapshot every time a student changes lines, stops typing/editing for X seconds (e.g. 2s), or uses the UI to change their code (e.g. redo/undo). The format itself should support any level of granularity. This should be straightforward to do if we log edits as diffs, so a keystroke-level log would contain many very small diffs, and a compile-level log would contain a

few large diffs. The diff-based approach is used by BlueJ's Blackbox for its massive data collection.

   b. **Andrew**: The logs should also record actions the system takes in response to the user. For example, knowing that a timeout event (or warning) has been produced would be useful. The appearance of a debugging message, as opposed to a success message, is also useful, though those might be included as the response to a user action.

      In general, I think that the more fine-grained the data, the better. It will require significantly more space to record, but I'd rather that the standard allow for data to be acquired at as fine a level of detail as the tool designer wishes, rather than imposing an arbitrary level.

5. What features should the log data format have to allow for it to be synchronized with or tagged with more abstract or synthesized events that span a longer timespan?
   a. **Thomas**: Giving each user, project and session a GUID allows you to easily pair data with other tables, e.g. student grades/outcomes, pair-programing pairs, etc.
6. What log data format will be universally applicable to any IDE used in computing education?
   a. **Thomas**: There are really two questions here. What are common properties of *logs* and what are common properties of *source code snapshots*. For the former, the PSLC Datashop approach is define common columns that *most* loggers should produce (e.g. timestamp, assignment ID, student ID, action label, etc.) marking them as optional or required. Importantly, it also allows loggers to define their own custom columns for data that might be important for only a few IDEs. For source code formatting, I think a separate but related goal could be to define some universal (or at least very common) attributes of source code that could be logged consistently across IDEs (e.g. [source] lines of code, # of variables/procedures, is it compilable, # of compiler warnings, etc.), along with an AST representation of the code.
   b. **Andrew**: Focusing on the "properties of logs" aspect of Thomas's answer, I'd argue for an extensible format that will allow IDEs to log the events its designers think are important. Each event would simply be tagged with an event name, timestamp, guid, sessionid (?), and event-specific data. Then, we talk about event names (and associated event-specific data) that are likely to be relevant to multiple IDEs, so that tools can export relevant data from logs.
7. How can we capture learners' assigned programming tasks/problem-solving activities alongside their programming activities, in order to provide context for those activities?
   a. **Thomas**: This is important and probably challenging. It's easy to log an assignment ID with a student's logs, but it would be nice if there was some universal(ish) way of representing that assignment. Could we define a format for this as well (e.g. name, plain text description, language, concepts taught, type [HW, in-class, optional], sample solutions)?

8. Can and should we log learners' visitation of external sites during the programming progress?
    a. **Thomas**: This would be interesting, challenging, and probably hard to do without explicit student consent. We could limit it to specific sites (e.g. StackOverflow), but knowing their browser is being logged would probably concern students and alter their behavior.
9. Where should log data be stored?
    a. **Thomas**: While I appreciate what existing repositories do (e.g. Datashop, LearnSphere, Blackbox), nothing exists that is designed specifically for code that is generic enough for multiple IDEs. Could we get funding to roll our own? Could we alter an existing framework (e.g. Datashop) to save work?
    b. Andrew: I think individual systems should be logging their own data. The effort, I think, will be in providing export tools that allow data from two different sources to be compared.

Feedback on IUSE Proposal

- Thomas' notes here.