Data Flow in Babe

Blake R. Matheny

This document describes the data flow of events within Babe. There is an accompanying diagram at the end of this document that you may wish to skip to. As much as possible, I have tried to split this document into the section by the six major modules that comprise Babe.

1 Input

1. An input module collects data. Each instance of an input method gets its own thread, and that thread (as of now) does not get restarted. An input method should sit in a tight loop looking for data, preferably using a method like select() or poll() to be less resource intensive. Each input module has access to it's own unique (previously allocated) BABE_eventQueue, which is passed to the module at initialization.

2. When new data has arrived, an input module needs to add the new data to the event queue. A start and a stop time for the instance should be assigned to each new piece of eventData. eventData should not be added to the queue until the entire transaction has been assembled.

3. The core continuously loops through the list of input modules, looking for new data on the queue for each instance of the input. When new data is found on the eventQueue, it is passed to the appropriate format module.

At this point, new data has been acquired by an input module. The new data (after the sequence has been assembled) is added to the eventQueue by the input module.

2 Formats

4. Each instance of an input module has a format associated with it. For instance, a string or xml format may be associated with an input module. This makes logical sense, since an input is taking data from a data generation point, which is only generating data in one format. I mean to say, that a single program rarely generates consumer output in multiple data formats. The appropriate format module is passed the newly acquired data.

Now, the appropriate format module has the eventData that just came in from some input module. Each format module has a 'format map' file, that describes how to map data from the original format into an intermediary format. This may be some kind of grammar, but is undecided at this point. The important point is that the symbols in the intermediary language are associated with the symbols in the original input, such that the person in charge of administering the system has control over the semantic interpretation of the information. For instance, if the initial data was in the following format:

```
<destination>
 <ip_address>10.10.10.10</ip_address>
</destination>
<source>
 <ip_address>192.168.1.1</ip_address>
</source>
```

and came from the input module 'socket' at time 't', it might be transformed into the following intermediary format:

```
((destination_ip = 10.10.10.10)
 (source_ip = 192.168.1.1)
 (babe_input = socket)
 (babe_timestamp = t))
```

This mapping becomes very important in the way that event descriptions are created. There are several reserved symbols (including babe_input and babe_timestamp) that are reserved, the total number of reserved symbols has not yet been decided, but they may be used by the event descriptions (described later). These symbols must not be specified in the format map, and while reserved for use by babe, they may be used in the EDL.

5. After the transformation has taken place (this is where I imagine the most CPU intensive activity will take place), the intermediary representation of the data is passed back to the core. In the format module, the original should not be modified, since it may be needed later for forensic or proxying purposes.

In steps 3-5, the core has taken new data and passed it to an appropriate data transformation module which has passed back the intermediary representation of the original. There are now two copies of the data, the original and the intermediary representation.

3 Identification

6. The intermediary representation has to be identified, as to what event this data represents. For instance, say that you had described an event ev1 as follows:

```
ev1 = {
        destination_ip = "10.10.10.*";
        source = any;
        babe_input = "socket";
};
```

in which case ev1 is the set of all events where the destination ip address is 10.10.10.* and the source is ANY. The advantage of this type of abstraction is it has nothing to do with the original data format. I see this as an advantage because relevant information to some event may become available in many different original formats. By having an intermediary representation, events can be described in terms of _information_ and not in terms of _data_. If the original input (where the input came from) is important to the event, this can be specified in the event description using 'babe_input'. One should note that correlation between symbols in the intermediary representation and the event description. For more information on the event descriptions, see the paper I worked on. There are also several predicates available for time comparison as well as for the predicates as described by McCarthy.

7. So when the intermediary representation (see 4) is passed to the identification module, the ID module returns the identifier(s) associated with the event(s) this data represents. So in this case the list returned would simply contain 'ev1'. This information is returned to the core.

In steps 6-7, the event represented by the intermediary representation has been identified and returned to the core. An implementation detail we need to figure out is what what to do when no event is matched. Also, what to do in the case where an event requires information a,b,c and some intermediary representation has information a,b,c,d, what do we do with d? What do we do with 'fuzzy' matches such as this?

4 Storage

8. The core now knows what event this represents, and can build a SQL query to insert the event data into the table associated with this event. There are several advantage of using a SQL like storage for the backend. One, SQL has built in boolean operators which we don't have to code ourselves. Two, SQL implementations offer things such as date comparison, which we don't have to code ourselves. Three, SQL helps facilitates offline finite state analysis. Four, queries from some Babe instance to another Babe instance are uniform. Five, basic regular expressions are supported but can be augmented with PCRE if needed.

One thought, is that each event description gets its own table. So in our example from 6, there would be a table called 'ev1' that would contain every instance of this event. This is nice because it makes checking for some types of aggregate events quite simple, and helps facilitate checking the COUNT() of certain events. The disadvantage to this is that potentially we are dealing with a large amount of tables. I think that the advantages outweigh the disadvantages, especially since checking for the existence of some event would be really pretty quick. Something that needs to be thought about is some type of 'archive' flag, so that the original data can be stored for later forensic analysis.

5 State Analysis

9. The list of events acquired in step 7 is now passed to a state analysis module. The state analysis module checks to see if the addition of these events to the system completed any state machines for aggregate events.

10. If a machine is completed, return which event(s) are now 'complete' to the core. A problem here is going to be, how do we segment off completed events? Basically we need to keep track of which events already were counted in completing a state machine, so that they are not reused in the state machine. We may need to do look ahead in the state analysis to determine if the completed event is part of any yet to be completed aggregate event.

In steps 8-10, the new data for events is inserted into a database. After this is completed a state analysis module checks to see if the new events helped complete any state machines. Any newly completed state machines are returned, represented as a list of event identifiers.

6 Output

11. The core now checks the event chains to see if any of these events returned in step 10 need to be sent to some output.

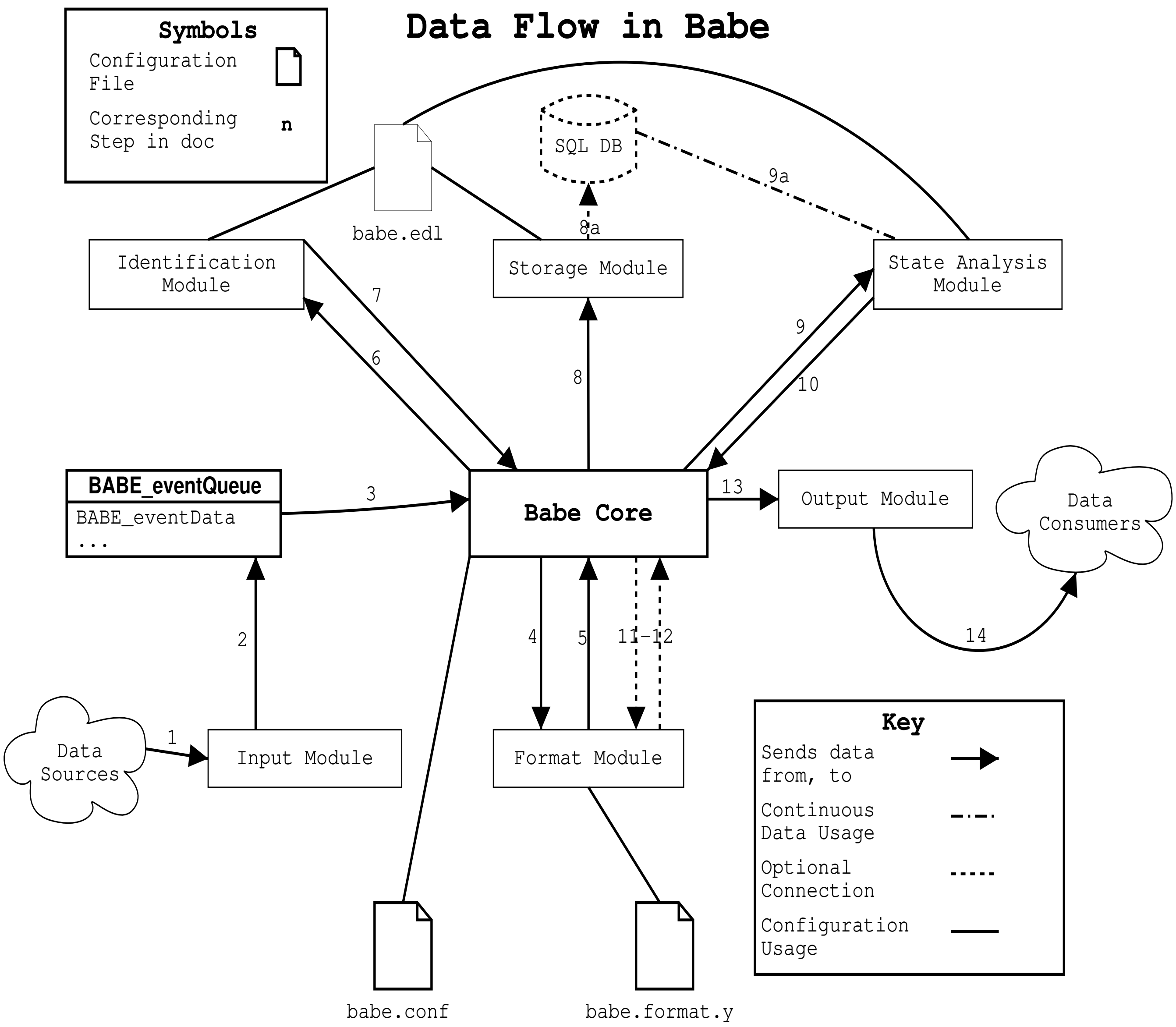
12. If an event needs to be output, the data format type for that event is looked up and the events intermediary representation is passed to the appropriate transformation/format module. The format is passed back to the core

13. Pass the newly formatted data to the appropriate output module

14. The appropriate output module sends the data on to its destination.

15. When there are no new events in the list to be output, go back to step 1

In steps 11-14, the events are sent to an output as specified in the configuration and this process starts over when there are no new events to process. This is a synchronous process, so this isn't exactly a cycle but this documents the process in general.


Data Flow in Babe
Symbols
Configuration File
Corresponding Step in doc
n
SQL DB
babe.edl
9a
8a
Identification Module
Storage Module
State Analysis Module
7
6
8
9
10
BABE_eventQueue
BABE_eventData
...
3
Babe Core
13
Output Module
Data Consumers
2
4
5
11-12
14
Data Sources
1
Input Module
Format Module
Key
Sends data from, to
Continuous Data Usage
Optional Connection
Configuration Usage
babe.conf
babe.format.y