This version is generated on October 13, 2007.

CAYUGA USER MANUAL

MINGSHENG HONG

1. Overview

This document describes how to use Cayuga as an end user. We assume familiarity with concepts in Cayuga. A general introduction to Cayuga can be found in [4]. The Cayuga system architecture is described in [3].

Currently Cayuga is supported on the following platforms: Windows with Visual Studio 2005, Mac OS X and Linux with GCC 4.0. Other Unix like platforms have not been tested.

1.1. Roadmap.

Section 2 introduces the directory structure of the code base, as well as the third-party libraries used by Cayuga. Section 3 and 4 provides guidances for compiling and running Cayuga. Section 5 serves as a reference to the Cayuga config options. Section 6 explains the input/output system components in Cayuga and the message formats they use. Section 7 describes the regression testing functionality in Cayuga. Finally, Section 8 describes how to generate HTML documentation for Cayuga source code.

2. CVS Repository

The CVS Repository of this project is named `cayuga-system`. Here are the descriptions of the top level directories.

`Cayuga` stores the code for Cayuga engine.

`CayugaQL` stores the code for the CEL (Cayuga query language) compiler.

`extlib` stores the header and library files of the third party libraries Cayuga depends on. A description of these libraries can be found in Section 2.1.

`Receiver` stores the code for receivers in Cayuga, such as Event Receivers.

`Sender` stores the code for senders in Cayuga, such as Client Notifiers.

`scripts` stores the script files used in regression testing.

2.1. Required Library/Component.

The third party libraries we are currently using include Antlr, dbgroup-utils (a portable thin layer of OS services developed at Cornell Database Group), and Xerces.

We included all third party libraries and header files into the CVS repository, so that an end user of Cayuga can compile the source code checked-out from CVS right away without having to manually install any third party libraries.

3. Compile and Run Cayuga

Currently, the *working directory* of Cayuga from which the system should start will have to be set to the top-level CVS directory `cayuga-system`. The reason is that when the system starts up, it will need to locate the dynamic library for xerces, which is stored under `cayuga-system`.

Date: October 13, 2007.

3.1. **Windows.** The compilation of Cayuga on Windows is through Visual Studio 2005.

To compile Cayuga, open the solution file in `Cayuga/platforms/win32/Cayuga.sln`, and click `Build Project`. The generated executable `Cayuga.exe` is stored in `Cayuga/platforms/win32/{Debug, Release}`.

If you want to start Cayuga within Visual Studio environment, follow the steps below.

- Right click `Cayuga` project in the `Solution Explorer` window of Visual Studio, and click `Set as Startup Project`.
- Again right click the above project, and choose `Properties` this time. In the pop-up window titled `Cayuga Property Pages`, choose `Configuration Properties -> Debugging` tab, and set the value of `Working Directory` to the location of `cayuga-system` in your hard drive. For example, it could be `C:\research\code\cayuga-system`.

  Also, set the value of `Command Arguments` to a Cayuga config file that you would like Cayuga to load when it starts up. Its path is relative to the top-level directory. For example, setting it to `Config.xml` will let Cayuga load `cayuga-system/Config.xml` when it starts up. The Cayuga config options will be described in Section 5.
- Press `F5` or `Ctrl+F5` to start Cayuga.

If you want to start Cayuga from a command-line console, copy the Cayuga executable `Cayuga.exe` to the top-level directory, and invoke it from there with a config file name as the input parameter. For example, `Cayuga.exe Config.xml` with start Cayuga with `cayuga-system/Config.xml`.

3.1.1. *Mac OS X and Linux.* Now we focus on how to build the system on Unix like systems with GCC and GNU auto tools.

To build Cayuga, follow the steps below.

(1) Go to `cayuga-system/extlib/inc`, and expand the zipped header files for third party libraries into that same directory.
(2) Go to `cayuga-system`. This will be the top-level directory for building Cayuga.
(3) Run `libtoolize`
(4) run `aclocal`
(5) run `autoheader`
(6) run `autoconf`
(7) run `automake -a`
(8) run `./configure`. If you want to run this script for a second time, you can speed it up by putting `-C` as a command line argument to use the cached setting.
(9) run `make`
(10) If things go smoothly, the Cayuga executable named `cayugaServer` should be generated under the current directory.

Finally, the Xerces library that Cayuga uses will be loaded at Cayuga run-time. If you already have a copy of Xerces libraby on your computer at known place, it should be loaded automatically. Alternatively, we also included this library in the top-level directory of Cayuga. You need to include the top-level directory in the environmental variable **LD_LIBRARY_PATH**, in order for the OS to correctly locate the dynamic library.

Now you can invoke Cayuga from the top-level directory with a config file name as the input parameter. For example, `./CayugaServer Config.xml` with start Cayuga with `cayuga-system/Config.xml`.

By default Cayuga is compiled with no optimization flag being set with GCC, and with **-g** being set for debug purpose. You could mdify the flags in `configure.ac` if you like.

4. Cayuga Examples

There are plenty of examples of Cayuga uses in the CVS directory. Before you continue to read the following sections, you are welcome to start playing with these examples and get a general feel for how Cayuga works.

The example config files can be found in `Cayuga/test/examples/config`. You could use any one of them to start Cayuga. They point to what query files and input event stream file to load when the system starts.

Example Cayuga automaton queries (queries in low-level automaton specification) can be found in `Cayuga/test/examples/query`. They provide good resources for learning AIR, the language for specifying Cayuga automata.

Example Cayuga streams and their schemas can be respectively found in `Cayuga/test/examples/stream` and `Cayuga/test/examples/schema`.

Finally, example Cayuga CEL queries (queries in high-level SQL-like syntax) can be found in `Cayuga/test/CayugaQL/tests`. They provide good resources for learning CEL, the declarative query language for Cayuga. Their corresponding automaton queries, produced by the CEL Compiler, can be found in `Cayuga/test/CayugaQL/results`.

5. Cayuga Config Options

- Query related options
  - `QueryInputMode`
    This variable indicates *how* to read queries. Currently queries can be read from a single file (FILE), or from all files of a given directory (DIR). If the query input mode is DIR, we require that the names of all the XML files within that directory are prefixed with "AIR_", followed by the ID of that query, starting from 0. By default it is FILE.
  - `QueryInputName`
    This variable indicates *where* to read queries. If the query input mode is FILE, the file names of one or multiple queries are stored here, separated by ;. If the mode is DIR, the directory name is stored here.
    For example, in `Config.xml`, the value of this option `AIR_XML_test.xml`, which is also included in the same directory as the configuration file. By default it is "".
  - `QueryNumber`
    This variable indicates the number of queries to read. This is only used if the query input mode is DIR. So files of name from "AIR_0.xml" to "AIR_n.xml", where n is QueryNumber-1, will be read. By default it is 0.
  - `AirQuery`
    This bool variable indicates whether the system is to load queries in AIR format or in high level query language. By default it is true, meaning input queries are in AIR format.
- Document related options
  - `DocInputMode`
    This variable indicates *how* to read documents. Currently documents can be read from a single file (FILE), from all files of a given directory (DIR), or from TCP sockets (NETWORK). If the document input mode is DIR, we require that the names of all the XML files within that directory are prefixed with "doc_", followed by the ID of that document, starting from 0. If the input mode is NETWORK, see Section 6.3 for the event message formats and how event sources should interact with Cayuga system.
    By default it is FILE.
  - `DocInputName`
    This variable indicates *where* to read documents. If the document input mode is FILE, the file name is stored here. If the mode is DIR, the directory name is stored here. By default it is "".
    For example, in `Config.xml`, the value of this option is `doc_XML_test.xml`, a file stored in the same directory as the configuration file.

If the document input mode is NETWORK, this value is not used.
- `DocNumber`
  This variable indicates the number of documents to read. This is only used if the document input mode is DIR. So files of name from "doc_0.xml" to "doc_n.xml", where n is DocNumber-1, will be read. By default it is 1.
  If the document input mode is NETWORK, this value is not used.
- `XMLStream`
  This bool variable indicates whether the input stream is XML. If so the system will invoke a special event parser called CaxParser instead of the regular relational event parser. By default it is false.
  If the document input mode is NETWORK, this value is not used, and currently we only read relational events from NETWORK.
- `ERPort`
  This variable indicates whether the input events will be received from TCP sockets. If so, it stores the TCP socket port the network event receiver uses. Otherwise it is 0. By default it is 0.

• GC related options
- `GCSize`
  This variable stores the size of the front and to heap spaces managed by the Cayuga copying garbage collector. It is 32(MB) by default.
- `GCSizeUnit`
  This variable stores the unit of GC-managed memory pool size. Its value could be drawn from {BYTE, KB, MB}. MB is the default value.

• CEL Compiler
- `InlineQuery`
  This variable indicates whether the CEL compiler should try to inline the generated AIR queries if possible. For example, if the second input expression of a binary operator such as NEXT is a unary expression, then instead of generating a separate automaton for it, we could inline it with the automaton corresponding to the binary operator, by pulling the predicates in the unary expression to the filter edge of that unary operator. By default it is true.
- `MergeStates`
  This variable indicates whether the prefix of automaton states should be merged when possible. By default it is true.

• Debugging, profiling, logging
- `Verbose`
  This variable indicates whether system should print detailed debug information for the internal execution of Cayuga engine. By default it is false.
- `DebugMessageDestination`
  This variable stores the destination of debug messages. If it is "", debug messages will be printed to screen. Otherwise, the value of this variable should be the composition of the name of a path relative to `cayuga-system`, and the name of a file which stores the debug messages. An example would be "log/debugMsg.txt". Please use / instead of other characters such as \ to specify the path name separator. By default it is ".".
- `PrintFrequency`
  This variable stores the frequency of printing some repetitive messages, such as loading each query, and processing each event. By default it is 1000.
- `WitnessLogDir`
  This variable indicates whether the output events will be logged into a disk file named `witnesses.txt`. If it is "", witnesses will not be logged. Otherwise, it specifies the name of a

directory relative to the top-level directory `cayuga-system` in which the witness file will be stored. By default it is ".", which means the witness file will be stored under the top-level directory.

- `RecordTrace`
  This variable indicates whether system should generate traces of internal state information and dump they into disk files. By default it is false.
  Traces can be read by Visualizer to visualizer Cayuga automata-based event processing. Also, it can be potentially used by Cayuga engine for crash-recovery (not yet implemented).
  See Section 6.2 for the details in generating traces.
- `TracePort`
  This variable indicates whether system should generate traces of internal state information and send them to Visualizer client via TCP. If so, it stores the port value. Otherwise, it is 0. By default it is 0.
- `OutputEventPort`
  This variable indicates whether the output events will be sent out to a TCP socket. If so, it stores the TCP socket port the output event sender uses. Otherwise it is 0. By default it is 0.
- `compiledAIRLogDir`
  This variable stores the name of directory in which the compiled AIR queries by CEL compiler will be stored. By default it is "", in which case the compiled AIR queries will not be logged.
- `CommandLogDir`
  This variable stores the name of directory in which the commands that are received at run-time will be stored. By default it is "".
  Also, when the system starts up, an XML AIR file named `queries_loaded.xml` will be created, which stores the set of queries loaded into the engine when system start time. For each new query added at run time, an query file will be created in that directory, with the naming convention `Query_i_AIR.xml` or `Query_i_CEL.txt`, depending on whether that query is in AIR or CEL. Here i is an integer sequence number for each new query addition.
- `Measure`
  This variable indicates whether the Cayuga measure manager should be turned on to continuously produce measures, including event processing and garbage collection time costs, number of NFA instances in the system, and heap space consumption. By default it is false.
  The output formats of the continuous measures are described in Section 6.1.
- `MeasurePort`
  This variable indicates whether information of cayuga measures should be sent across the network. If so, it stores the TCP socket port the measure manager uses. Otherwise it is 0. By default it is 0.
- `EventWindowLen`
  This variable stores the length of the event window with which the event processing costs are aggregated. It is measured by the number of consecutive input events Cayuga processes. By default it is 0xffff.
- `CheckPointFrequency`
  This variable stores the (time) frequency of checkpointing the system. If the value is n, it means for every period of n timeunits, the entire state of the system will be checkpointed. A special value 0 means no checkpoints will be taken. By default it is 0.
  See Section 6.2 for how to generate checkpoints.
- `CheckPointAndTraceDir`
  This variable stores the name of directory in which the check points and trace messages will be stored. By default it is `.`. This directory will be located under the top-level directory `cayuga-system`. Note that in order for Cayuga to correctly write trace files, this directory must exist before the system starts.
- `Strict`
  This variable indicates whether Cayuga should tolerate any run-time errors, such as in new query addition. It is true if it does not tolerate any errors. By default it is false.

- Other options
  - `AttrDelimiter`
    This variable stores the character delimiter used in input stream file (for FileER mode), output witness file and output trace file for automata instances. By default it is ','.
  - `CommandPort`
    This variable indicates whether the Cayuga Command Server should be on. If so, it stores the TCP socket port the command server uses. Otherwise it is 0. By default it is 0.
    See Section 6.4 for the description of Command Server.

6. Cayuga System Components and Message Formats

First we desribe a general TCP message format used by Cayuga: each TCP message starts with a 4-byte (binary) integer in network ordering stating the number the bytes in this message (excluding the 4 bytes themselves), followed by the actual message content in ASCII text. In the following, a message may be stored both in a disk file and sent to a TCP socket. In that case, the message stored in the disk file is identical to the ASCII body part of its corresponding TCP message.

6.1. **Continuously Monitoring Cayuga Engine Status.**

Cayuga system has a sub-system called *Measure Manager* to continuously produce measures of Cayuga engine. These measures can be written to disk files as well as to a TCP socket specified in the config file.

6.1.1. *Disk Files.* Currently we can store three types of measures as disk files: the time cost of processing each event in the engine, the time cost of each garbage collection invocation, and the number of instances in the system after each epoch. They are respectively stored in three text files `ep_cost.txt`, `gc_cost.txt`, and `inst.txt`. The first two files contain two columns, where the first column is the event ID (assigned by Cayuga engine according to the number of events processed so far), and the second column is the time cost measured in microseconds. Note that the cost of garbage collection will only be recorded when garbage collection is actually performed.

The third file has $2 + k$ columns, where k is the number of intermediate or end automaton states in Cayuga engine. The first column is event ID, and the second column is the total number of instances. The next k columns are respectively the number of instances under each intermediate or end state. The sum of these k values is always equal to the value in the second column.

6.1.2. *TCP Socket.* When the configuration parameter `MeasurePort` is set to non-zero, the measures can be sent to a TCP Socket Client.

Currently we send four types of measures: event processing cost, heap consumption, number of instances, and number of unprocessed events (backlog length of Cayuga). The config parameter `EventWindowLen`, whose value is denoted as W here, controls the frequency of sending out these messages: to save TCP communication, one status message will be sent after the engine processes every W input events. The status message format is 11‘STATUS‘CTIME‘t‘EP_COST‘c_1‘HEAP_SIZE‘c_2‘NUM_INST‘c_3‘EQ_LEN‘c_4‘. Those fields above with upper case letters are fixed strings denoting the attribute names. t denotes the Cayuga engine time when this status message is generated. c_1, c_2, c_3 and c_4 are four ASCII numbers denoting the measure values.

6.2. **Generating Cayuga Traces and Checkpoints.**

6.2.1. *Trace Messages.* Cayuga traces consist of five types of messages. For human-readability, the trace messages are in textual format. Each message is a record of variable number of fields. Each field is delimited by a character delimiter, which is ` by default. A message can span multiple lines (i.e., it can contain \ r and \ n characters). However it cannot contain the delimiter in its content. The last field of each message is followed by a character delimiter.

For each message, the first field is always an integer (in ASCII) representing the number of fields to follow. The second field is always the type of that messasge, taking string values from AIR, EVENT, INSTANCE, WITNESS and EOE. The remaining fields are message-dependent, described as follows.

An AIR type of message encodes one AIR XML message, which may correspond to one automaton query or multiple ones merged together by the engine. The third field of an AIR message is the XML string of the AIR query. An AIR message thus looks like `2‘AIR‘<NFA> description of automaton </NFA>‘`.

An EVENT type of message encodes one input event. Starting from the third field, there is a sequence of attribute, value pairs for this event. Then there are three more fields storing the stream name, the start and end timestamps of this event in this order. For example, an event from stream `Stock` with point timestamp 1 could look like `8‘EVENT‘Name‘IBM‘Price‘25.5‘Stock‘1‘1‘`.

An INSTANCE type of message encodes one new instance, referred to as the destination instance, created under some automaton state, referred to as the destination state. It also records which instance (referred to as the source instance) under which state (referred to as the source state) caused the creation of this new instance. Starting from the third field, there is a sequence of attribute, value pairs for this instance. Then there are six more fields storing the source state ID, source instance ID, destination state ID, destination instance ID, start timestamp and end timestamp in this order. For example, `11‘INSTANCE‘Name‘IBM‘Price‘25.5‘0‘100‘1‘101‘10‘10‘` encodes an instance with ID 101 created under state 1 with point timestamp 10. Its creation is caused by a source instance with ID 100 at source state 0.

There are a few specially cases of an INSTANCE type of trace message. First, if the source state of this instance is a start state, the source instance ID will be -1. Next, if an instance is expired by a duration predicate on the filter edge on the state with which the instance is associated, this instance is marked for deletion. In this case, we will generate a special instance message will no instance content, and source node ID = destination node ID = destination instance ID = -1. The source instance ID is set to the ID of this instance. The start and end timestamps of this message will be set respectively to the start time of this instance, and the end time of the event currently being processed. For example, `7‘INSTANCE‘-1‘8‘-1‘-1‘0‘1‘` represents an instance of ID 8, which has start time 0, and is to be marked for deletion when processing the current event ending at time 1.

A WITNESS type of message encodes one output event/witness. Its format is identical to that of EVENT. For EVENT, the third-to-last field is the *input* stream name of the event; for WITNESS, the third-to-last field is the *output* stream name of that witness.

An EOE type of message denotes an end-of-epoch tick. Its content is always `1‘EOE‘`.

6.2.2. *Checkpoints.* Similar to trace messages, checkpoints can be generated by the engine. At checkpoint at time n contains the following two pieces of information. First, the NFA queries in the engine as of time n (exclusive[1]) are stored in AIR format in a file named `checkpoint_nfa_n.xml`. Second, the set of instances in the system as of time n (exclusive) are stored in the format of INSTANCE type of trace messages in a file named `checkpoint_instance_n.txt`. Note that each instance in a checkpoint file always has source node ID and source instance ID set to -1.

6.2.3. *Visualizing Event Processing.* If you set `RecordTrace` to true, the traces genereated by Cayuga can be opened by the Java based Automaton Visualizer to show the Cayuga automata structures and their execution.

To start visualizer, go to `cayuga-system/AutomatonVisualizer`, and compile it in a Java compiler of version 5.0 or higher. Now run `java -cp . visualizer.Visualizer` to start the visualizer. On the `File` menu, click `connect`. For the connection to established, we require that at least one Cayuga trace file has been written and closed. Afterwards, you can click the step forward or backward button to display Cayuga event processing one step at a time; or you could click the play button, to display the trace execution continuously at a fixed pace. You could speed up or slow down the pace with the slide bar control to the right. Clicking the play button one more time will pause the continous play.

6.3. **Receiving Events from TCP Sockets.** The format of an input event message is similar to that of the EVENT type of trace message in Section 6.2 starting from the third field there, without the first two fields in the EVENT type of trace message, and without the fields corresponding to attribute names. Also, for backward compatility, the last

[1]The system state as of time n *exclusive* means this is the system state before processing any events of timestamp $\geq n$.

field is not followed by a delimiter. For example the textual part of the example event in Section 6.2 is as follows `IBM‘25.5‘Stock‘1‘1`.

Each event source needs to be a TCP Socket Client.

6.4. **Processing Commands at Run-Time.** Cayuga has a sub-system called *Command Server* which allows users to submit command messages to Cayuga engine to be processed at run-time. The commands include adding new queries to the engine.

The format of a command message is very similar to that of a trace message.

The first command type query addition. Its format is as follows. 3‘QUERY‘CEL‘cel_query_string‘ or 3‘QUERY‘AIR‘air_query_xml_string‘. Note that ‘ is the field delimiter. 3 denotes the number of fields to follow in this message, not including itself. If the third field in the message is CEL, then the following field is a string encoding a CEL query. If the third field is AIR, then the following field is an XML string encoding an AIR query.

The second command type stream schema addition. Its format is as follows. 2‘SCHEMA“schema_in_sir_format‘.

One test example can be found in `cayuga-system/Receiver/CR/test`, which demonstrates a simple command client of Cayuga.

Note that each command sender needs to be a TCP Socket Client.

7. Regression Test Functionality

7.1. **Cayuga Test Cases.** Each Cayuga test case consists of a set of Cayuga config file, query file(s), stream file, and schema file as inputs, and a witness file as the correct output. Some example Cayuga test cases can be found at `Cayuga/test/Engine/testCase`. By convention, the witness file corresponding to a test case is named after that test case, appended with `_witnesses`. For example, The witness file corresponding to test case `StockQueryMShape.xml` is `StockQueryMShape_witnesses.txt`.

`scripts/regression/CayugaTestCase/regressionTest.pl` reads a regresstion test configuration file, such as `RTestConfig.xml` or `RFullTestConfig.xml`, which specifies the set of test cases to run. This script then compares the outputs produced by the engine on these test cases with the stored witness files to check correctness.

7.2. **Regression Test with YFilter.** Simple XPath queries can be expressed in Cayuga. We wrote a regression test program between Cayuga and YFilter on randomly generated query and stream workload, which can be found in `scripts/regression/YFilter`.

We provided the binary Java classes for YFilter in a zipped file `scripts/regression/YFilter/YFilter.zip`. To use the regression test functionality, first unzip this file into the *current* directory (do not create new directory for the unzipped files).

In invoke the regression test, use the script `regressionTest.pl` in that directory.

Alternatively, you could manually go through this regression test as follows. First, use `randomQueryAndDataGen.pl` to generate a random workload. After being invoked, this script will output two directories as follows. `output_doc` contain one single XML document. `output_queries` output N query files in AIR format, as well as the same query set in YFilter format, in the file `xpath_queries.txt`. Parameters used by random query and document generators can be set at the beginning of `randomQueryAndDataGen.pl`.

Next, to run Cayuga on the generated workload, use the configuration file `ConfigXPath.xml` in the same directory.

To run YFilter on the generated workload, invoke “java -cp .;dtdparser113.jar;java_cup.jar edu.berkeley.cs.db.yfilter.Run output_doc/doc.xml output_queries/xpath_queries.txt ¿YWitnesses.txt” to store the YFilter witnesses into `YWitnesses.txt`.

After running both Cayuga and YFilter, you could invoke `compareWitness.pl` to compare the witnesses produced by both systems.

8. CAYUGA SOURCE CODE DOCUMENTATION

Cayuga source code is documented in Doxygen format [1]. DOT tool can be used along with Doxygen to generate graphs such as directory dependency graphs and function call graphs [2]. The paths to these executables should be set in the environmental variable PATH.

To run doxygen, go to directory `Cayuga/doc/Doxydoc`, and run `doxygen Doxygen.conf`. A directory named `doc` with be generated, and `doc/html/index.html` is the start page of the documentation.

REFERENCES

[1] The doxygen project. http://www.doxygen.org.
[2] Graphviz - graph visualization software. http://www.research.att.com/sw/tools/graphviz/.
[3] A. Demers, J. Gehrke, M. Hong, B. Panda, M. Riedewald, V. Sharma, and W. White. Cayuga: A general purpose event monitoring system. *Proc. CIDR*, 2007.
[4] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *Proc. EDBT*, 2006.