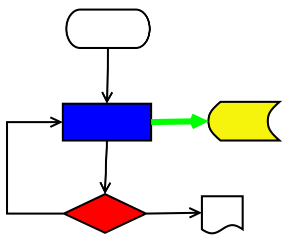
VICI



Visual Chart Interpreter

Design of libvici

Publication History

Date	Who	What Changes
29 July 2014	Brenton Ross	Initial version.
1 December 2016	Brenton Ross	Updates after implementation. Includes notes on file descriptor handling.

Table of Contents

1 Introduction

This is part of the system design document for the VICI project.

1.1 Scope

This document covers the detailed design of the interpreter library. This component is responsible for the execution of a VICI script.

The document will cover the Application Design and the User Interface Design.

1.2 Overview

The detailed design includes:

- Interface Stubs: A framework of facade classes for the modules.
- Use Case Descriptions: A description of how a user is expected to interact with the application.
- Application Design: The classes and their relationships.
- User Interface Design: The design and layout of the graphical components of the system.
- Persistent Storage Design: The specifications for the XML files used to store configuration and scripts.

1.3 Audience

This document is intended to be used by the designers and developers, and later the maintainers, of the VICI project.

2 Overview

2.1 Responsibilities

This component executes the scripts by determining which commands to execute based on the exit status of the previous commands. It is used by both the vici program and the vici-editor program. It addresses the following responsibilities:

T13.1: Construct an execution model for the script based on the diagram.

T13.2: Create thread objects for each thread of control in the script.

T13.3: Thread objects may wait for user input before executing next command.

T13.4: Thread objects may wait for a specified interval before executing next command.

T13.5: Thread objects can be repositioned to other locations in the execution model.

T13.6: The execution model can be tagged with break points that stop the thread objects from proceeding until they have user input.

T13.7: Create new thread objects when starting a background function or command.

T13.8: Determine which command to execute next according to the execution model and the exit status of the previous command.

T13.9: The thread is to start one or more process objects according to the execution model.

T13.10: Terminate all threads when processing completes.

T19.1: Provide a cd command that changes the working directory of a thread.

T19.2: Provide a “for each line” command.

T19.3: Provide a “for each” command.

T19.4: Construct and remove named pipes.

T19.5: Provide a signal command that sends a signal to another process.

T20.1: Allow a pipe to be connected to a named pipe.

T20.2: Allow a pipe to be connected to a file in either append or over-write mode.

T21.1: Expand variables embedded in in-line file.

T22.1: Remove temporary files.

T24.1: Assign the output of commands to variables.

T24.2: Expand command lines with the current value of variables.

T24.3: Perform arithmetic operations on variables.

T24.4: Provide built in variables containing process ids of background threads.

T24.5: Provide built in variables containing the exit status of the previous command.

T24.6: Provide thread specific built in variables that are passed to functions as parameters.

T24.7: Provide variables that represent the text field for drag-n-drop.

T24.8: Provide mutex variables.

T24.9: Provide semaphore variables that increments on receipt of a signal.

2.2 Design Approach

The component has a static aspect and a dynamic aspect. The static aspect mirrors the flow chart diagram with classes for each concept – Command, Variable, File, etc. The dynamic side represents the running processes.

The original idea was to use Qt's QProcess class as the basis for running the processes started by the interpreter. However its handling of the output channels is a bit awkward, requiring the controlling program to set which channel it wants to read from. The basis was therefore moved to the underlying fork – exec function calls and our own Process class connected by ordinary operating system pipes.

Two pipes were introduced to the fork – exec processing, one to ensure the calling program can save the PID of the child before it exits, otherwise it is possible for the child to run to completion and raise a SIGCHLD signal before the caller gets any run time and therefore gets a signal for child it knows nothing about. The second for passing messages back from the child to the parent process – this is necessary since there are some awkward problems when mixing threads and forking which makes the use of malloc and printf a potential point of failure.

The state information is passed between processes using a Context class. This is potentially accessed by multiple threads so it must use mutex locking to prevent collisions.

The interpreter needs to manage a lot of file descriptors, which is a problem for C++ code since it can be difficult to know which C++ object is responsible for a file descriptor. This then leads to file descriptors being closed twice or not closed at all. The approach is to use fcntl(fd, F_DUPFD_CLOEXEC, 1) to duplicate file descriptors so that each C++ object manages its own set of file descriptors.

The design uses threads as this makes it simpler to handle blocking reads. (The alternative is an event driven model using the select system call.) Unfortunately the thread model is complicated and will probably remain a point of failure into the future.

A thread pool is used with new threads being created if none are free when the request is received. This will reduce the overhead associated with creating threads and obviates the need to handle the shut-down of threads with join calls. The thread pool is passed a ThreadFunction object which it executes.

The design does create quite a few threads – the main program, a QProcess thread to interface to the Qt signal-slot system, a thread for each running function, and a thread for each built-in command process.

2.3 Interfaces

2.3.1 IF09 Command Execution

This is the interface between the VICI interpreter and the underlying operating system.

Transport Medium: Memory.

Protocol: The exec system call.

Content:

1. Command (O)
2. Option string list (O)
3. Environment strings. (O)
4. Standard input pipe (O)
5. Standard output pipe (I)
6. Standard error pipe (I)
7. Signals (O)
8. Exit status (I)

2.3.2 IF39 vici-editor using libvici

This is the interface that allows the editor user to test their scripts from the editor interface.

Transport Medium: Memory

Protocol: C++ function calls.

Content:

1. Script to execute. (O)
2. Variables (I/O)
3. Files (I)
4. Thread details (I/O)

2.3.3 IF45 libvici using libQtCore

The Qt library is used to provide a wrapper for system processes.
Note: This interface has been replaced by IF46.

Transport Medium: Memory

Protocol: C++ function calls.

Content:

1. Command and options (I)
2. Pipes (I/O)
3. Signals (I)
4. Exit status (O)

2.3.4 IF46 libvici using libcfi

This interface allows the libvici component to read the script file.

Transport Medium:	Memory
Protocol:	C++ function calls.
Content:	1. Read an XML file 2. Get nodes from the document 3. Get properties from the document node.

This interface allows libvici to manage child processes using the ChildProcessMgr class in libcfi.

Transport Medium:	Memory
Protocol:	C++ function calls.
Content:	1. Command and options (I) 2. Pipes (I/O) 3. Signals (I) 4. Exit status (O)

2.3.5 IF48 libvici using libsecure

This is the interface that allows the vici interpreter to determine if it should open a script.

Transport Medium:	Memory
Protocol:	C++ function calls.
Content:	1. Script file name (I) 2. Verification of signature. (O)

2.3.6 IF49 vici program using libvici

This is the interface that allows the user to interact with their scripts.

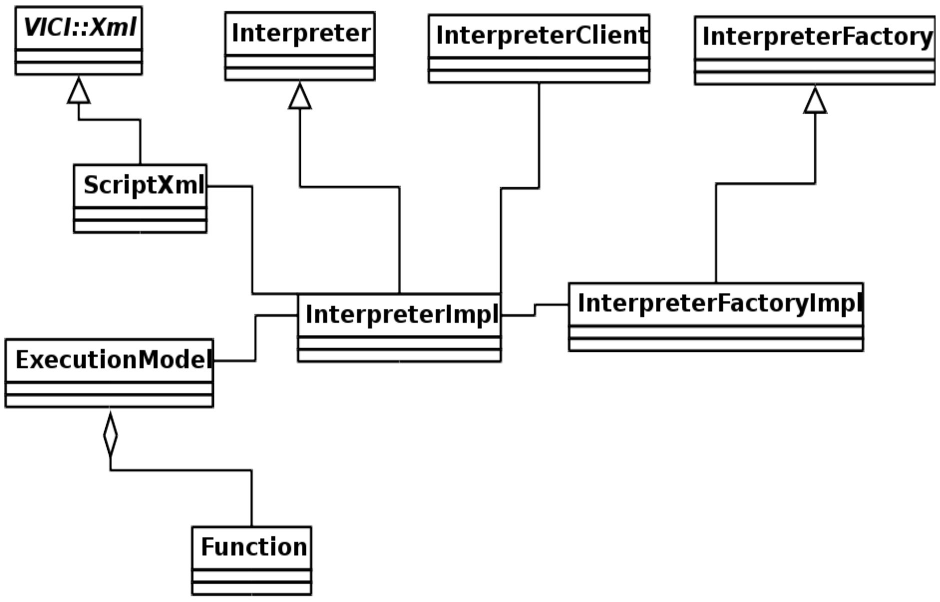
Transport Medium:	Memory
Protocol:	C++ function calls.
Content:	1. Script to execute. (I) 2. Menu actions. (I) 3. Text input (I) 4. Text output (O) 5. Error messages (O) 6. Status information (I)

3 Application Design

This section describes the overall structure of the library in terms of the classes that are used to build it.
The section has broken the design down into several sections, otherwise the diagrams would be impossible to include on a page.

3.1 Control Classes

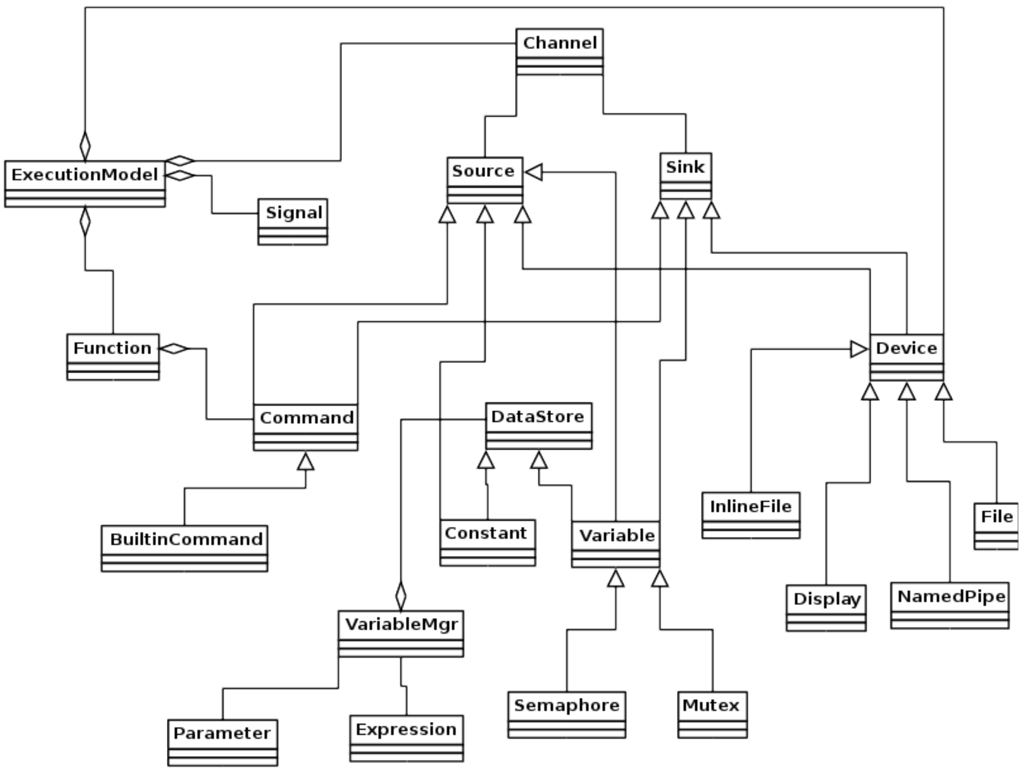
The following diagram shows the relationships between the classes that implement the library's API. These objects control the activity of the library and report back to its clients the activities of the library.



The Interpreter, InterpreterFactory and InterpreterClient classes are declared in vici.h as they define the interface to the library.

3.2 Static Classes

The InterpreterImpl uses the ScriptXml class to construct the ExecutionModel, and the static representation of the script as shown below.

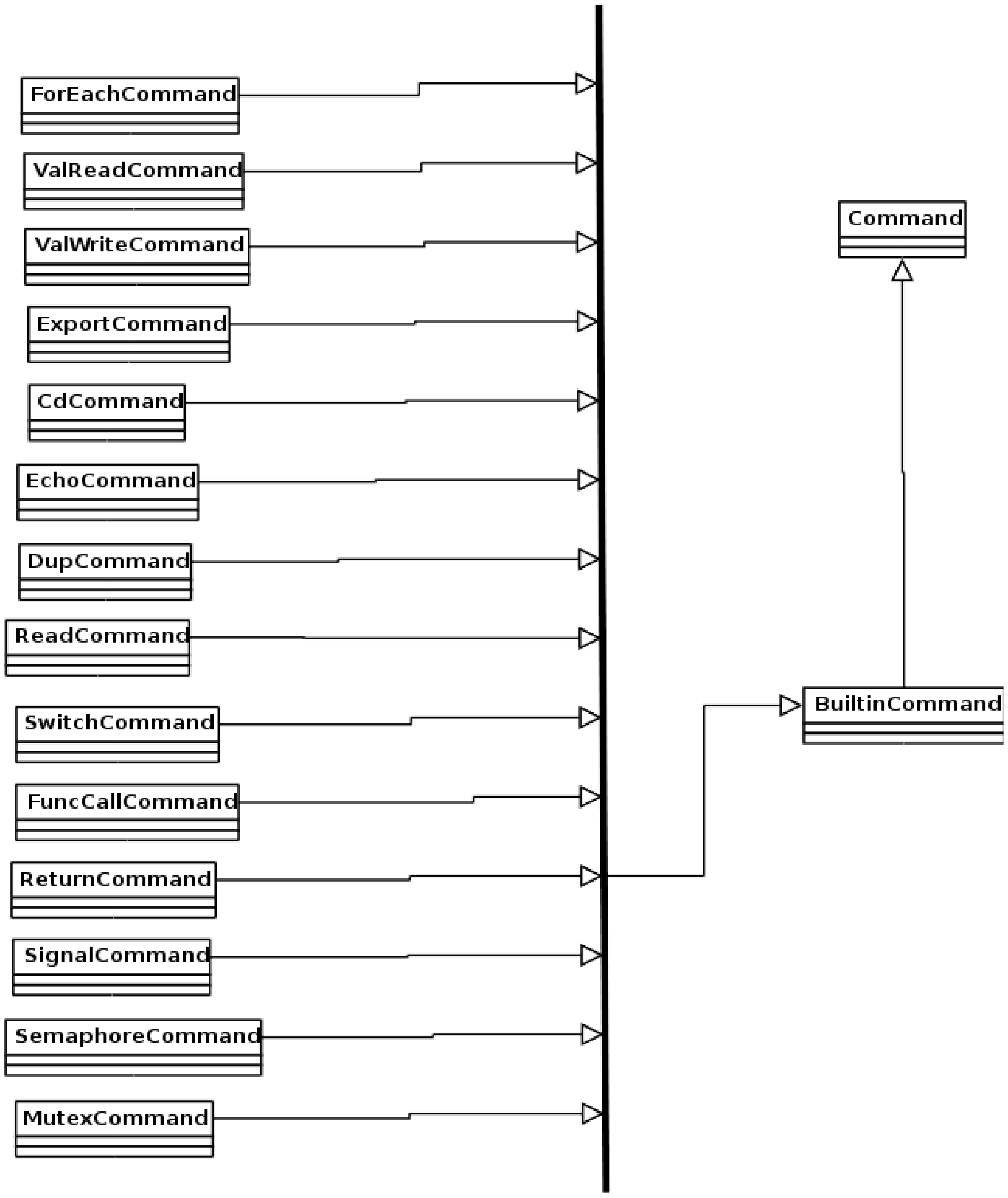


These objects mirror the objects placed on the flow chart but have behaviours according to the sort of object they represent. For example the File object includes the ability to read and write to the file.

A Channel represents a pipe that connects a source to a sink. I have made variables sources and sinks so that they can be placed at the end of pipelines. This turns out to be quite an important feature since it is not really possible to use the output of a pipeline as the parameter to a command – like the back-quote or $(command) of bash – since the flowchart has an explicit flow of control and the $(command) parameter would require an implicit flow of control, or allow some really bizarre constructs that would be difficult for the user to understand and for us to implement.

3.3 Command Classes

The built-in commands each have a class that encapsulates the implementation.



For the most part the script interpreter will start other programs to perform operations, however, there are few things that cannot be done that way, or are more efficient to do from within the interpreter. The following have been identified so far as being necessary or desirable built-in commands:

cd – this changes the working directory of subsequent commands. This must be a built in command as it would be impossible for an ordinary command to alter the working environment of commands that it is not the parent of.

dup – this copies its input stream to both its output and error streams. This is similar to the tee command but it seems like it would be more useful for a flow chart to use two output streams.

echo – writes the values of its parameters to its output stream. Not strictly necessary but makes a good test for the built in command processing.

export – this adds a new variable to the environment that is passed to subsequent commands. As with cd, this one is necessary.

for-each – this breaks up its input stream and runs the following commands for each element.

for-each-line – as above but for lines of the input stream. This, and for-each, are implemented with a separator character set to either white-space or new-line. The commands executed for each element are given their own context so that, for example, a cd will only apply to those commands.

let – performs simple integer arithmetic.

read – this reads a line of input and assigns the values to variables (similar to the bash read command). This must be a built in command since it modifies the interpreter's variables.

return – this just exits with an exit status equal to its only option. Since there is no “next command” it has the effect of terminating the thread of control within the interpreter (either the main thread, or a function running in the background).

signal – this sends a signal to the process that it is connected to. That sounds fine until you realise that the connection on the flow chart diagram is to a command, not a process, and that a command might be executing many times simultaneously if it is in a function running in the background. To keep some organisation I decided that a signal can only be sent to the commands within the same function scope (which also simplifies the diagrams) and the signal is only delivered to those processes with the same context as the signal.

switch – this has a return code that indicates which option matches the first option. This seems like a better idea for the flow chart. It relies on the flow of control lines on the chart being able to be given a specific exit code.

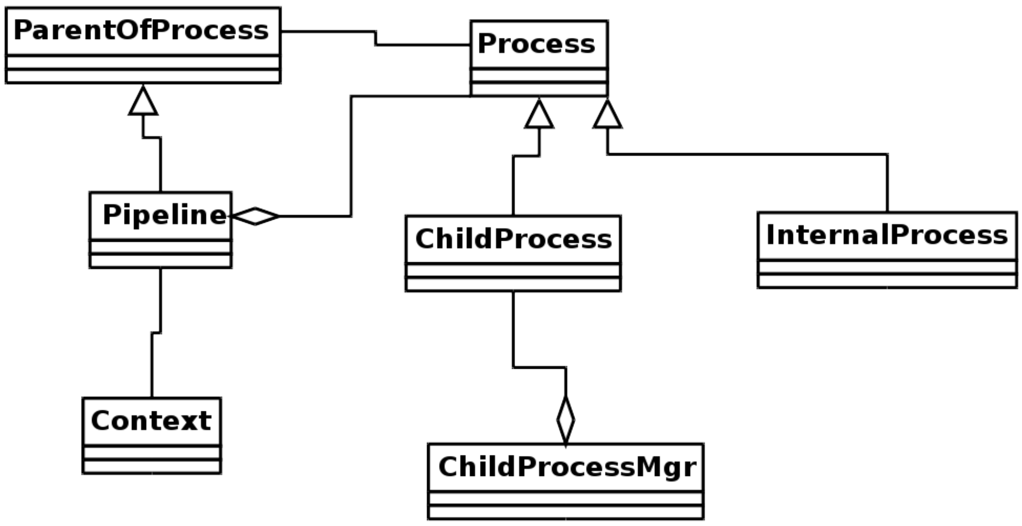
The function call is also implemented as a built in command. Like the for-each command it creates a new context and passes control to the first command of the named function. The options are passed as numbered variables (similar to bash).

There are two commands used for synchronisation between threads of control. The mutex command has lock and unlock which is used to ensure only one thread is accessing a section of the chart at a time. The semaphore command can force the script to wait until another thread posts to the semaphore.

Finally there are two anonymous built in commands used to read and write from/to variables so that a variable can be placed at the end of a pipeline.

3.4 Process Classes

The classes that represent the executing script are shown in following diagram.



The main purpose of vici is to start other programs. There are three classes that are used to manage these child processes:

Process – this is a parent class that does the processing that is common to both internal commands and child processes. This mostly involves processing the parameters to the command to expand variables and expanding glob expressions into file lists.

ChildProcess – this handles the creation of the new process. As is normal for Linux this is done by first forking the vici process and then replacing the new process with the command to be run.

ChildProcessMgr – this is used to detect when a child process has terminated. Once this has happened the corresponding Process object is updated with the exit status.

The usual approach is to have the parent instance of ChildProcess record its pid in the ChildProcessMgr so that the latter can find the correct ChildProcess object. When testing this I found that on rare occasions the ChildProcessMgr would report that it had got a signal for a pid that it did not know about. It appears that sometimes the child process gets enough run time to complete before the parent process gets a turn. To prevent this I added a pipe between the parent and child and forced the child to wait until it got a byte of data from the parent. The parent records the pid of the child before sending the data thus preventing the race.

The code to “exec” the child process might appear a bit odd as it takes some short cuts that would be inadvisable in normal circumstances. It should be remembered that this code is running in the forked child (not the main Vici process) and is about to be overwritten by the new process, hence there is little need to worry about things like memory leaks.

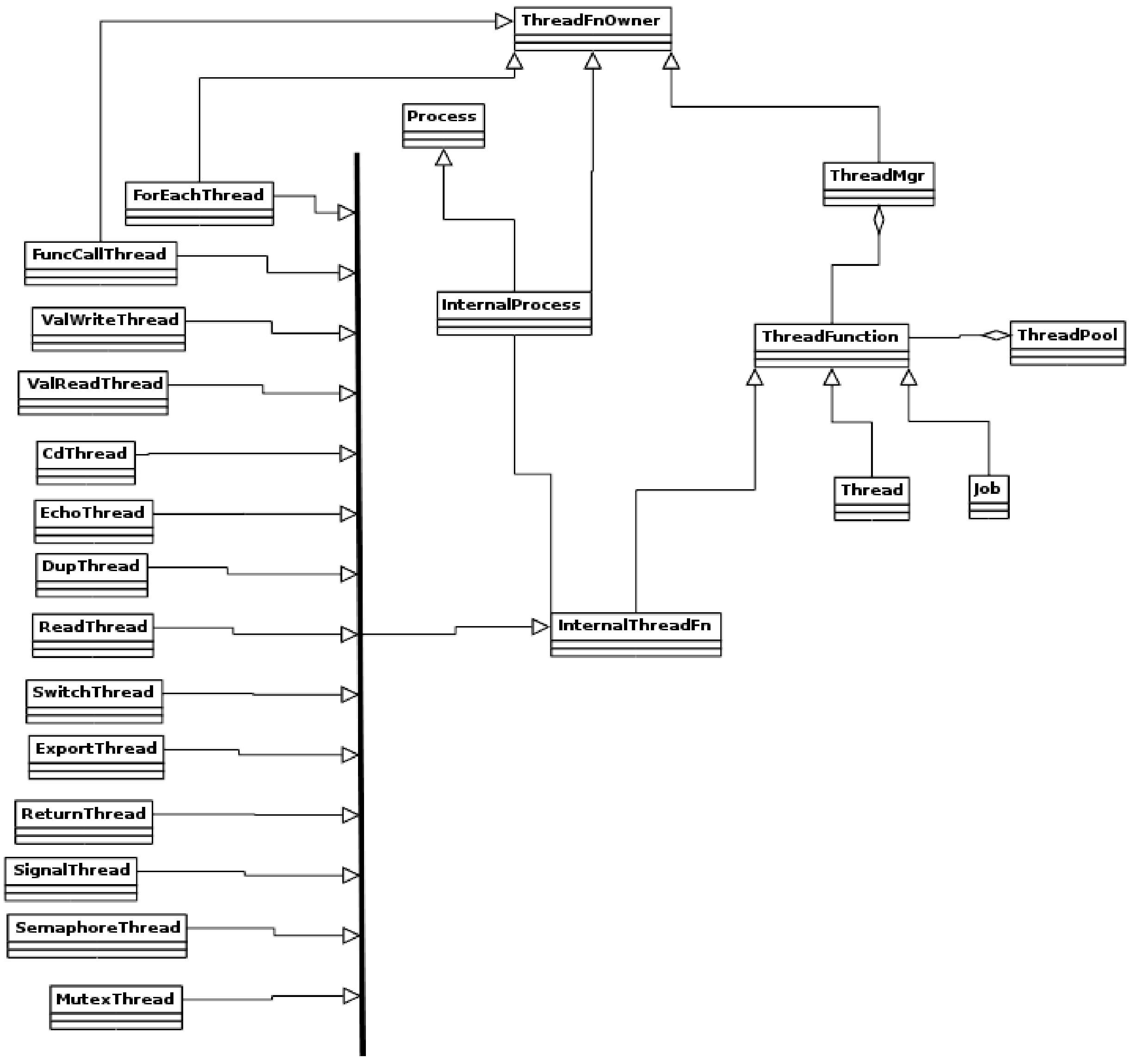
The ChildProcessMgr uses a “singleton” pattern that has been modified for use in a multi-threaded environment where there is a non-zero chance that two threads might each try to create an instance. The main point of the class is to handle the SIGCHLD signals from the terminating child processes. When a signal is received it is important to loop over the waitpid call as it is possible for one SIGCHLD interrupt to mask another which would lead to missed calls.

The unit of execution in Vici is not the process but the pipeline. All the processes that are connected via pipes need to be running simultaneously. The Pipeline class manages the creation of the Process objects and the connecting pipes. Since each process can have connections from both stdout and stderr the pipeline is actually more like a binary tree. The Pipeline starts at the first process and recursively adds the processes at its stdout and stderr to build the set of processes that must be run simultaneously.

In order for the processes to share the interpreter's state a Context object holds the shared data. A reference to this shared data is passed to each process. It includes the current working directory, the environment, the parameters to functions and file descriptors for the standard input, output and error streams.

3.5 Thread Classes

Vici uses a thread based design for the interpreter. The classes that manage these threads are shown in the following diagram.



Within the interpreter there is a lot of communication using pipes and other asynchronous communication channels. Linux provides two ways to process these – you can either use blocking read and write calls and threads, or non-blocking calls and the “select” system call. The former is easier to code since with event driven processing can become quite difficult to understand the flow of the logic of the program (at least that has been my experience). The result is that Vici's interpreter makes extensive use of threads.

There is a cost associated with using a thread based design in that it is all too easy to introduce race conditions that can cause faults that can be very difficult to resolve.

There are now some choices for using threads in C++. I decided to use the new STL thread support (mostly out of curiosity). The interpreter uses std::thread, std::mutex, std::condition_variable and std::unique_lock to implement the threads and synchronise their actions.

I tried several thread design patterns, but eventually decided that a thread pool would be the best approach. The following classes implement the design:

ThreadPool – this class is responsible for starting and running the threads. It creates as many threads as are necessary at any instant. It does not queue up jobs waiting for a thread, but will always start a new thread if none are available. If we queued up jobs there is a danger that a pipeline would be stalled waiting for data from a queued up job.

ThreadFunction – this is the base class for the thread function objects that are run by the thread pool. These objects are passed to the thread pool which then executes it within a thread. The thread pool takes ownership of the thread function object and deletes it once it has completed.

ThreadFnOwner – this is an abstract interface class that is notified when the thread function starts and stops.

One advantage of the thread based approach is that the interpreter can function independently of the GUI. The GUI can start the interpreter and immediately regain control so that the user interface remains responsive while the script is running.

Each thread of control within the interpreter runs in a Thread object (derived from ThreadFunction) which is responsible for creating the Pipeline objects according to exit status of the preceding command.

If a Pipeline is flagged to run as a background task it is wrapped in a Job object (also derived from ThreadFunction) that allows it to run independently.

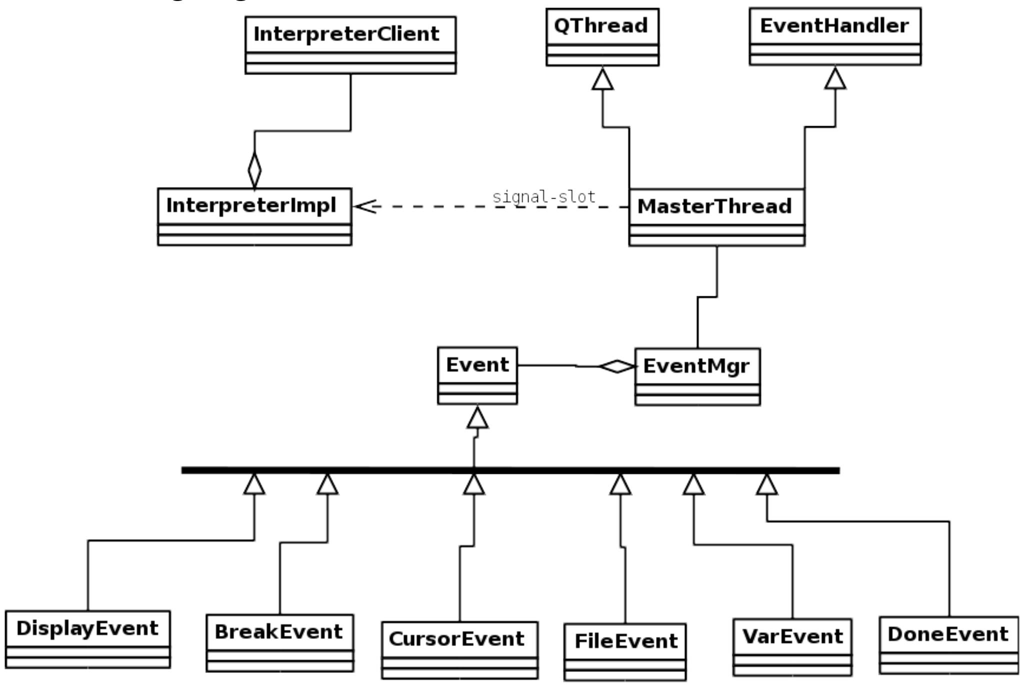
The normal commands run in separate processes so they do not need thread support, however the internal commands can be members of a pipeline and have their own read and write calls on the standard pipes. Hence each internal command is also run in its own thread.

3.6 Event Classes

When things happen in the interpreter it is necessary to notify the enclosing GUI program. This is complicated by the thread based design, and the need to interact with the GUI event loop.

When an event of interest occurs it is queued up on an Event Manager. The queue is read by a separate thread, called the MasterThread, that is derived from a QThread. This has the ability to send Qt signals to code running in the GUI event loop.

The following diagram shows the classes involved.



DisplayEvent – occurs when text is available from the output streams.

DoneEvent – occurs when the script completes.

BreakEvent – occurs when a break-point is reached.

CursorEvent – occurs for each command execution.

FileEvent – occurs when a file has been written to.

VarEvent – occurs when a variable changes its value.

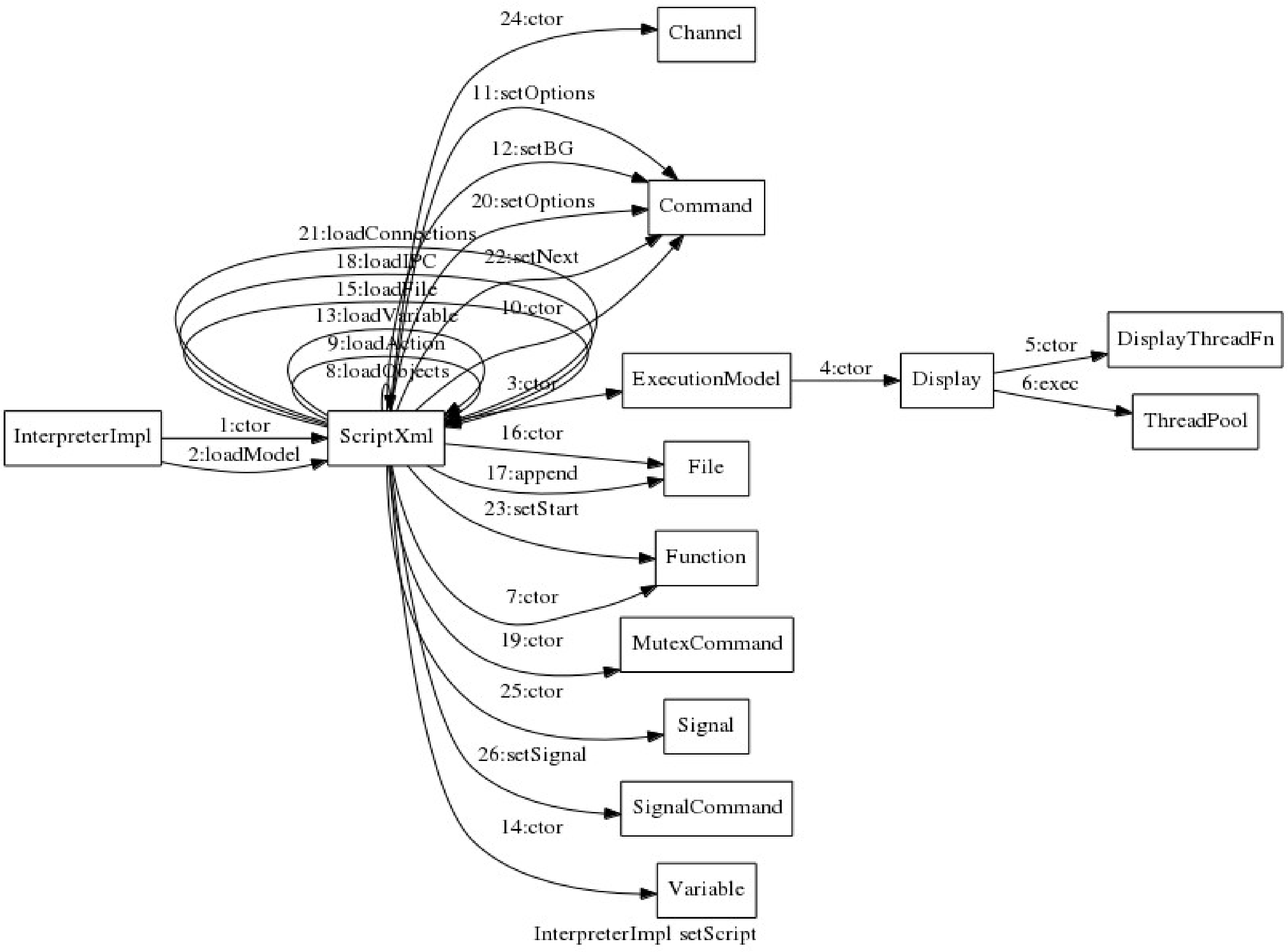
ErrorEvent – occurs when an error is detected.

4 Collaboration Diagrams

This section describes the dynamics of the library. Each responsibility is documented as a collaboration diagram showing the sequence of calls made by the objects.

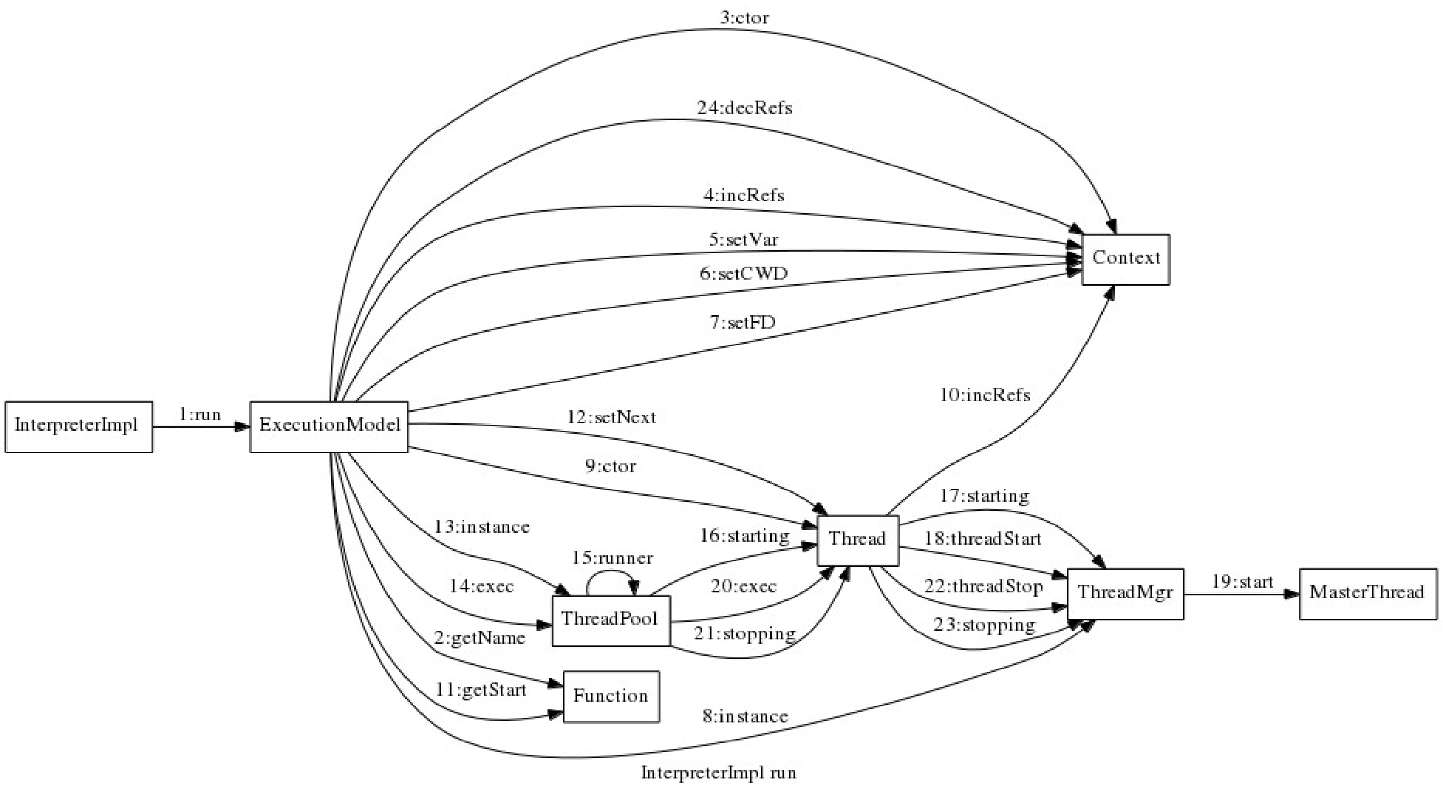
4.1 T13.1 Model Construction

This diagram describes what happens as the static model for a script is constructed. It is a simplified example with only a few representative commands and other objects portrayed.



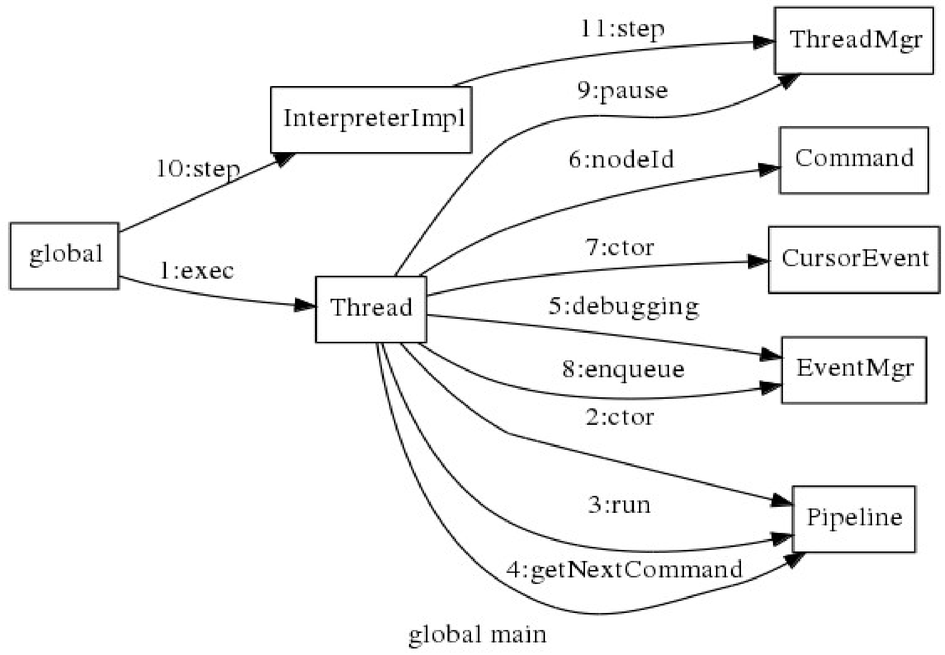
4.2 T13.2: Creating Thread Objects

This diagram displays the events surrounding the start of a function. It first constructs a Context object and then a Thread which is passed to the ThreadPool for execution while the ThreadMgr becomes responsible for its ownership.



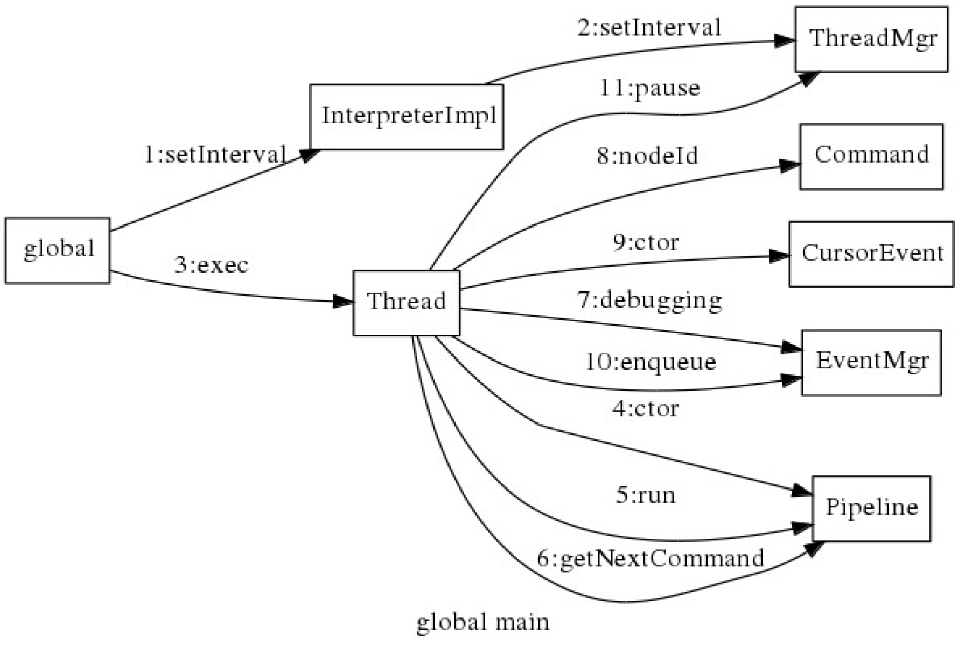
4.3 T13.3: Thread Objects Wait for User Input

If the interpreter is in debug mode and has been paused, either by a break point or by the user pausing the script, then a thread can be made to perform a single step.



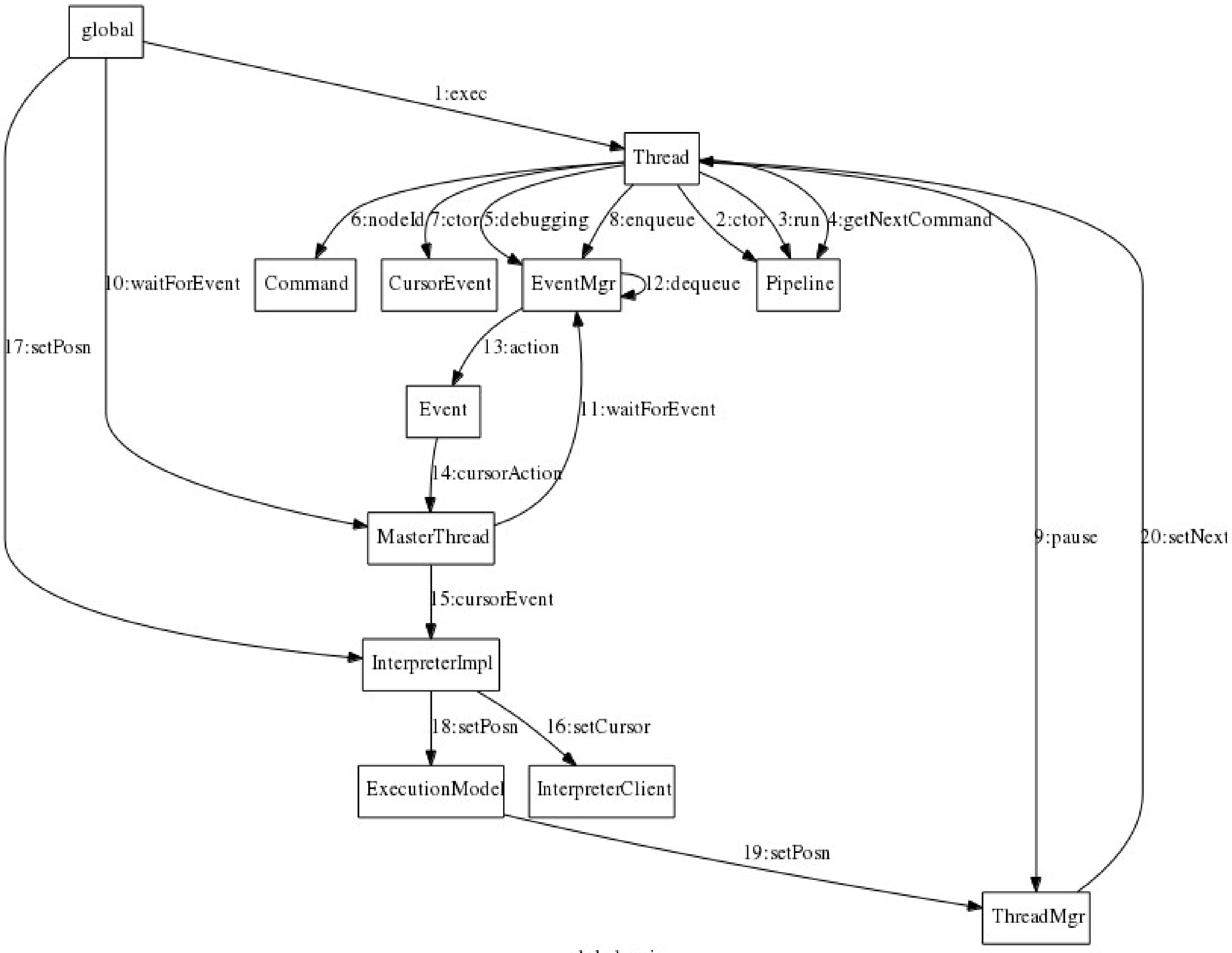
4.4 T13.4: Thread Objects Wait for Interval

The user can set an interval after which the paused script resumes for one step.



4.5 T13.5: Thread Objects Repositioned

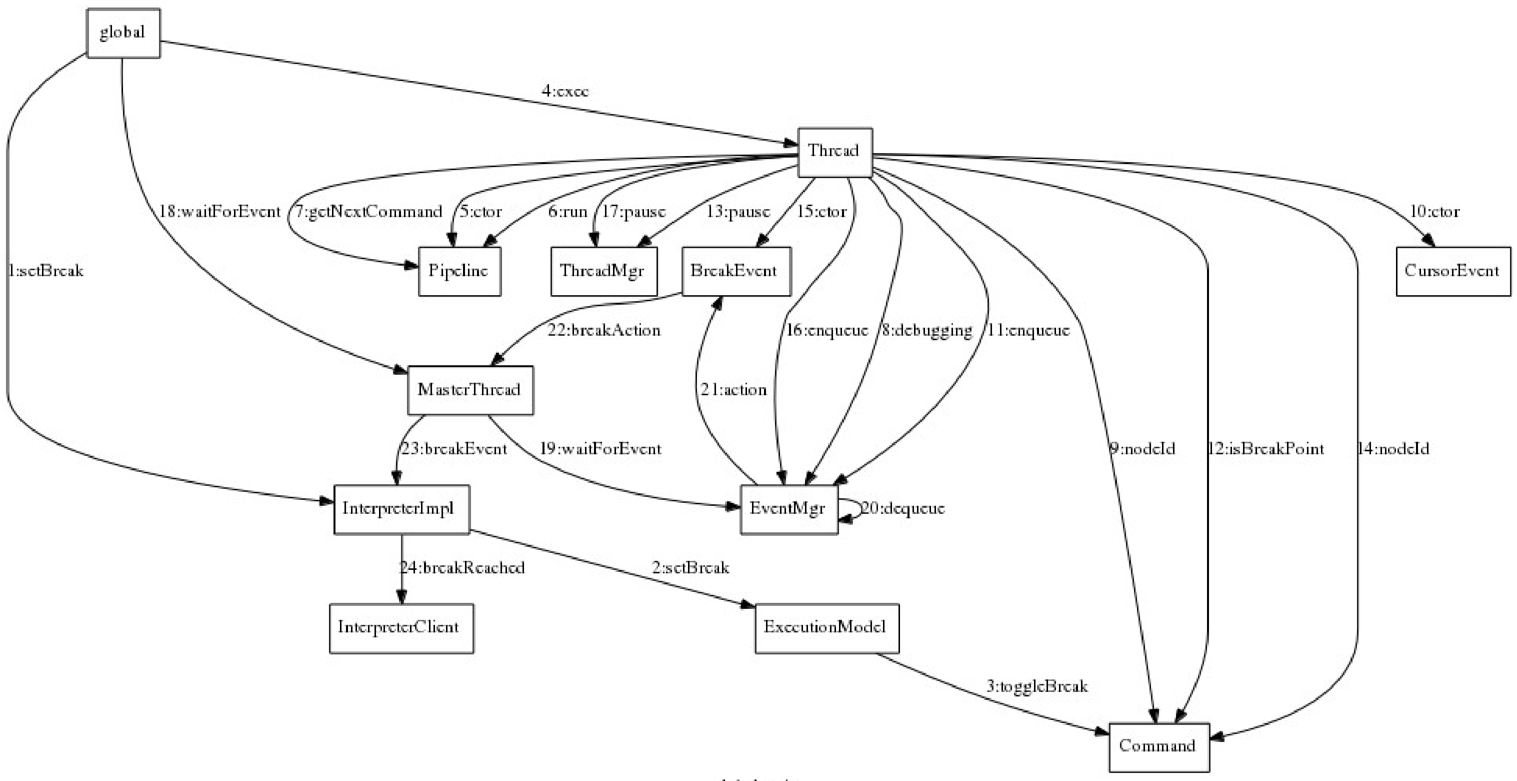
The user can change the next command to be executed.



global main

4.6 T13.6: Break Points

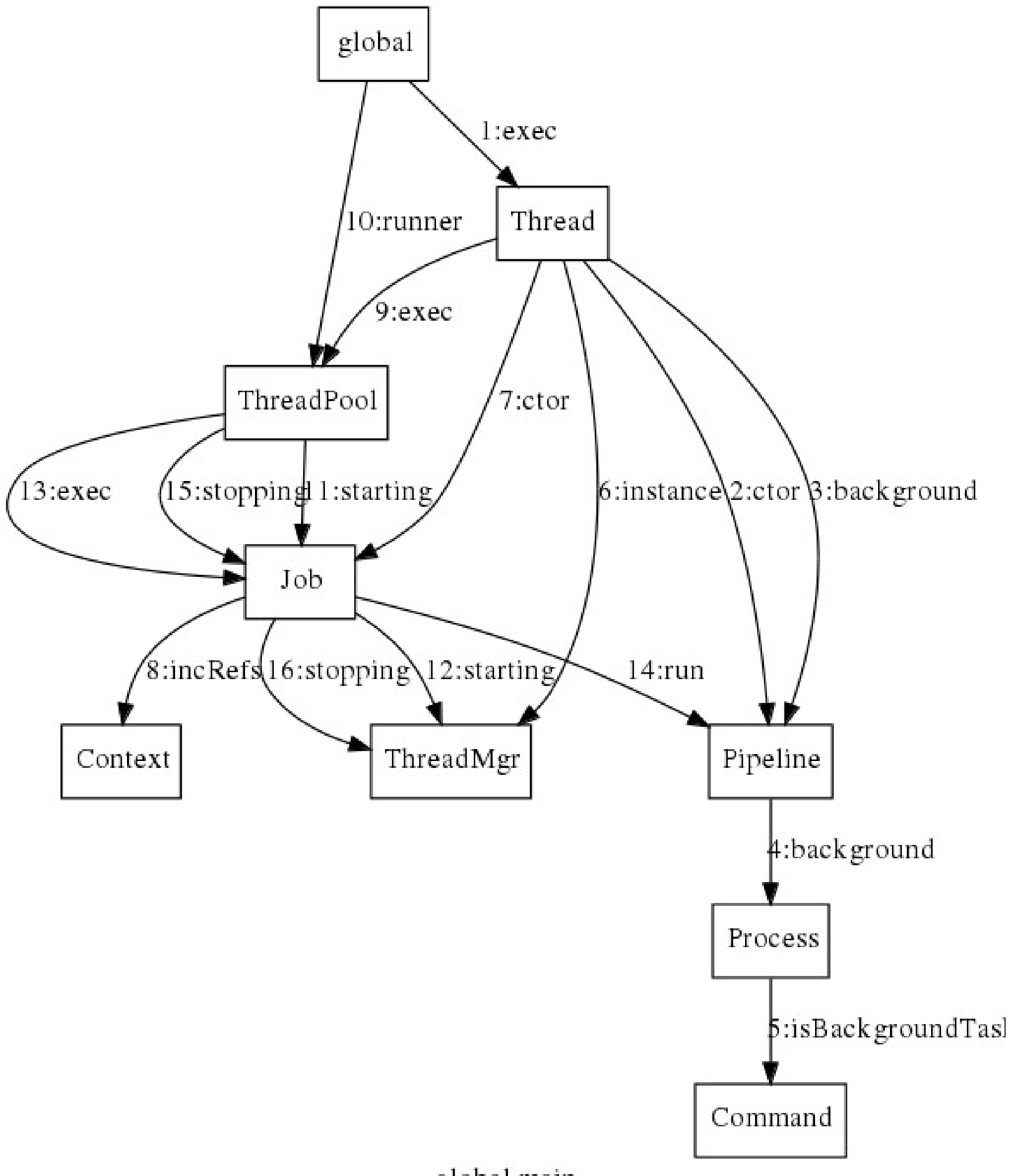
The following diagram shows the sequence of events when a user sets a break point on a command and the script reaches that command.



global main

4.7 T13.7: Background Function

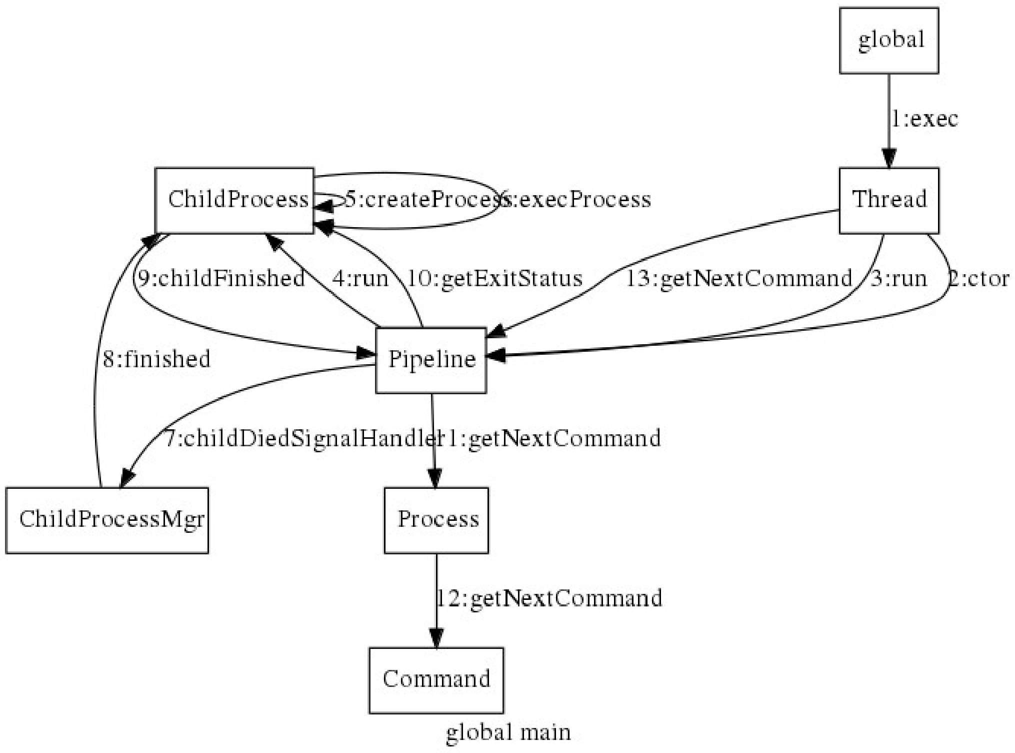
The following diagram shows the sequence of events when a command is called that has been marked for background execution.



global main

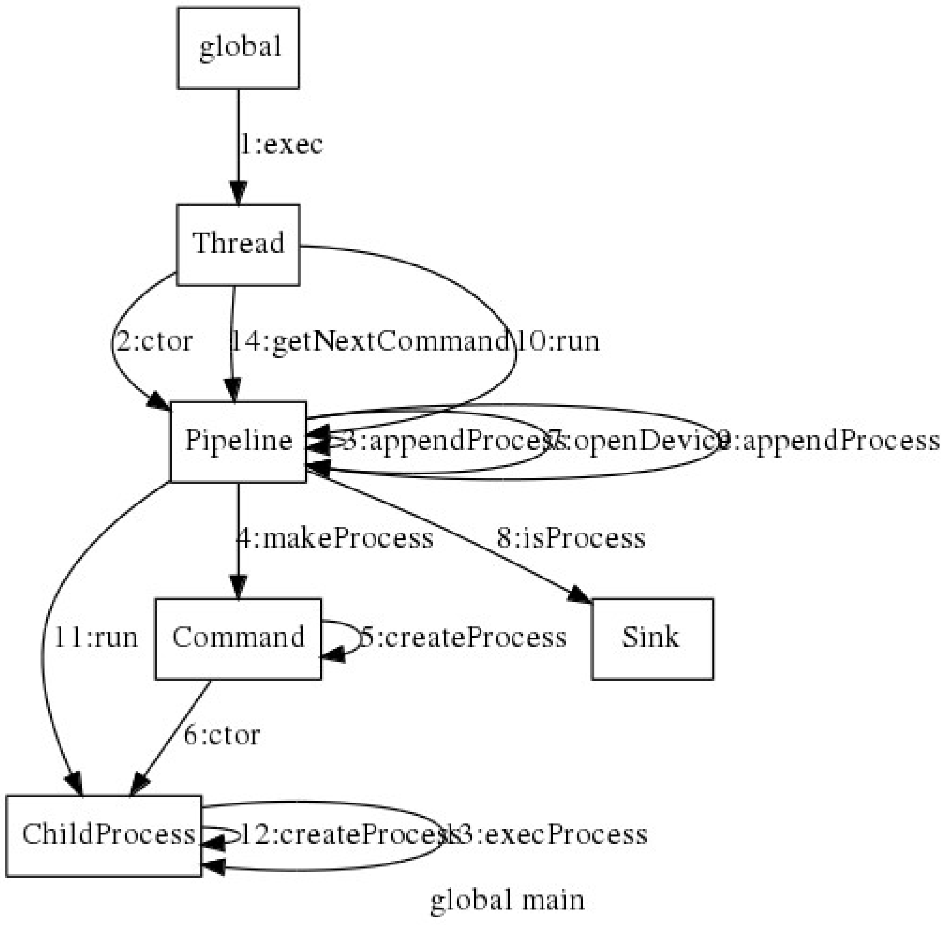
4.8 T13.8: Command to Execute Next

The following diagram shows the running of a child process and the capture of the exit status to determine which command to execute next.



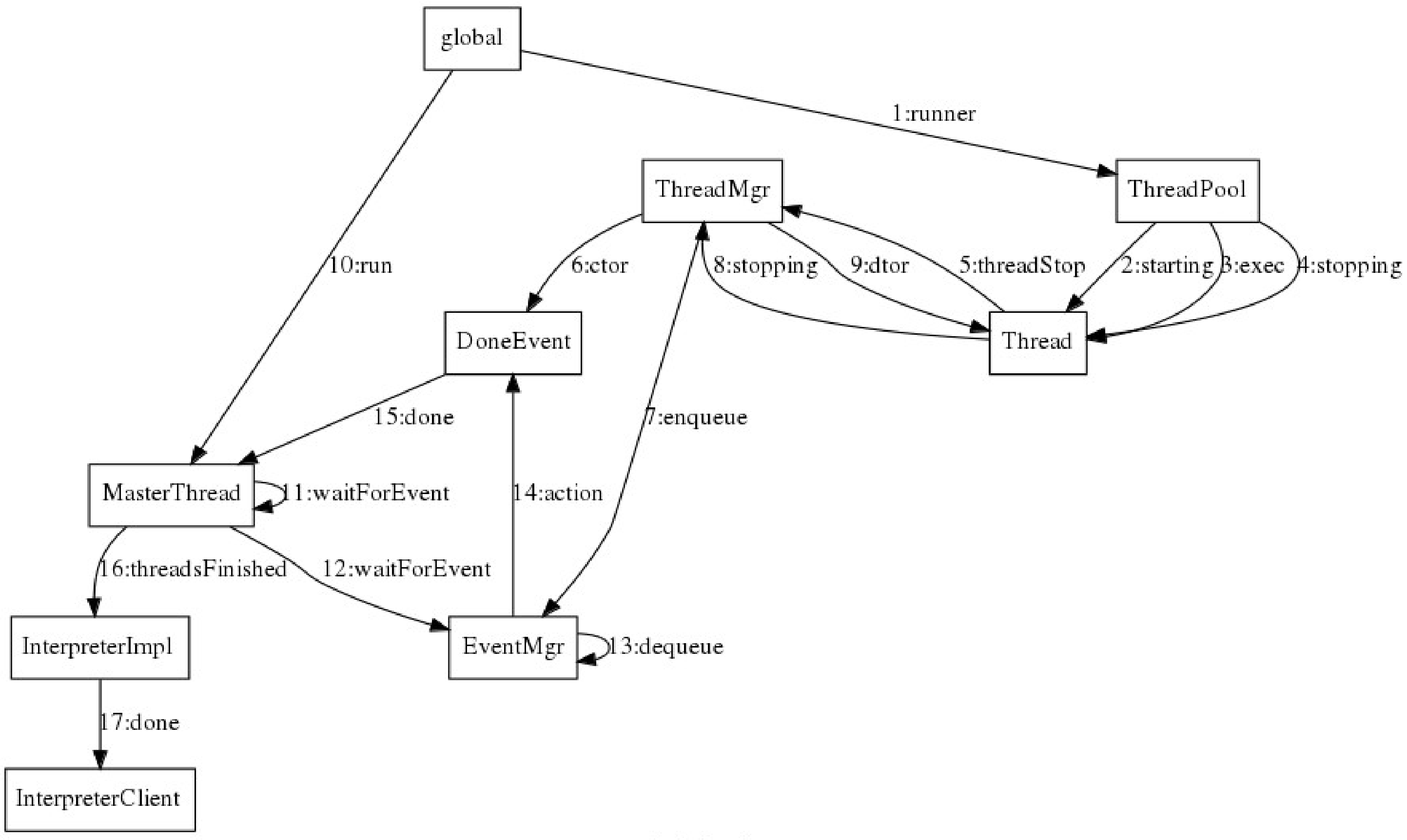
4.9 T13.9: Process Objects

The following diagram shows the construction of process objects by the Pipeline constructor.



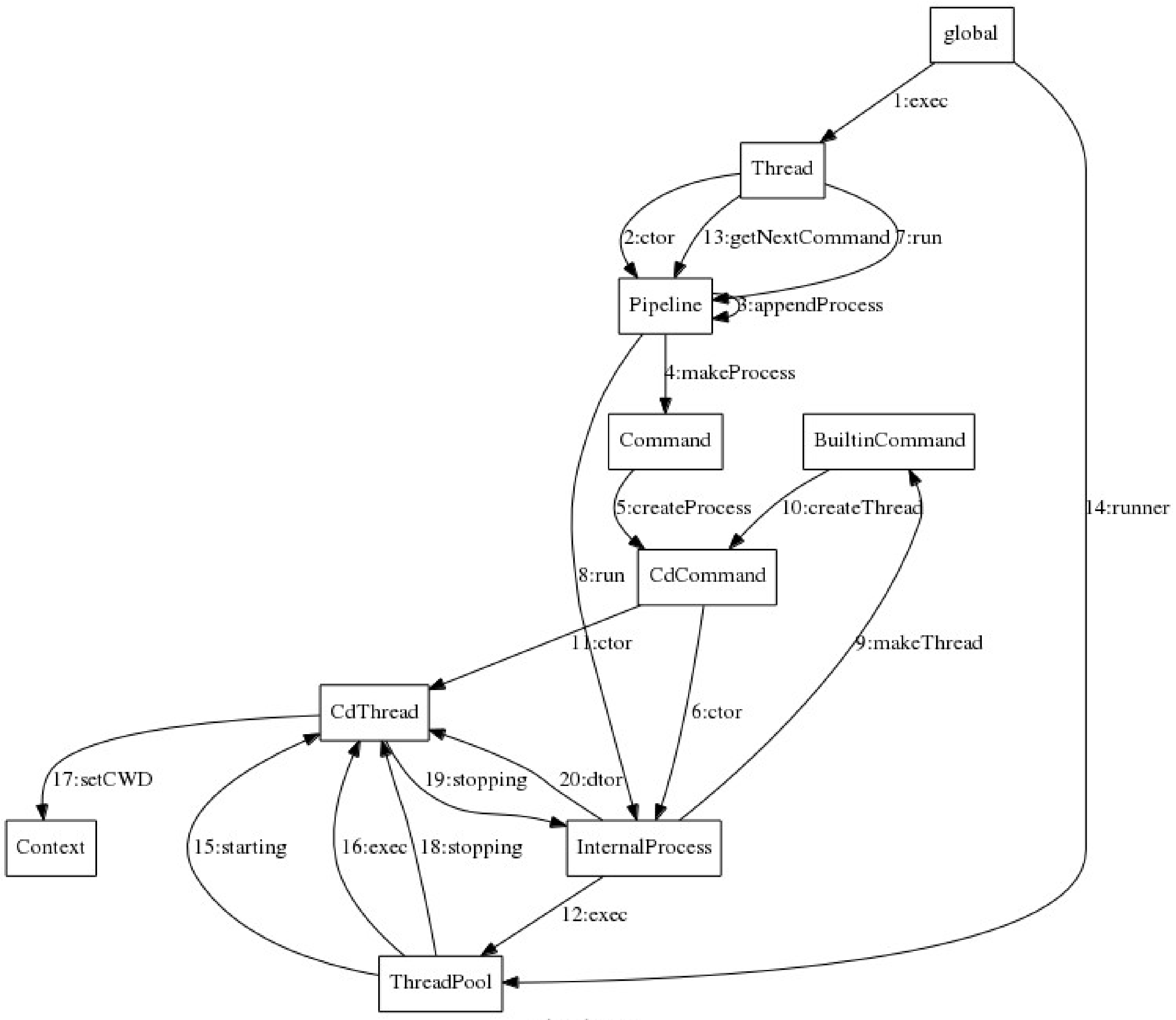
4.10 T13.10: Terminate when Processing Complete

The following diagram shows the sequence of events during the completion of a script.



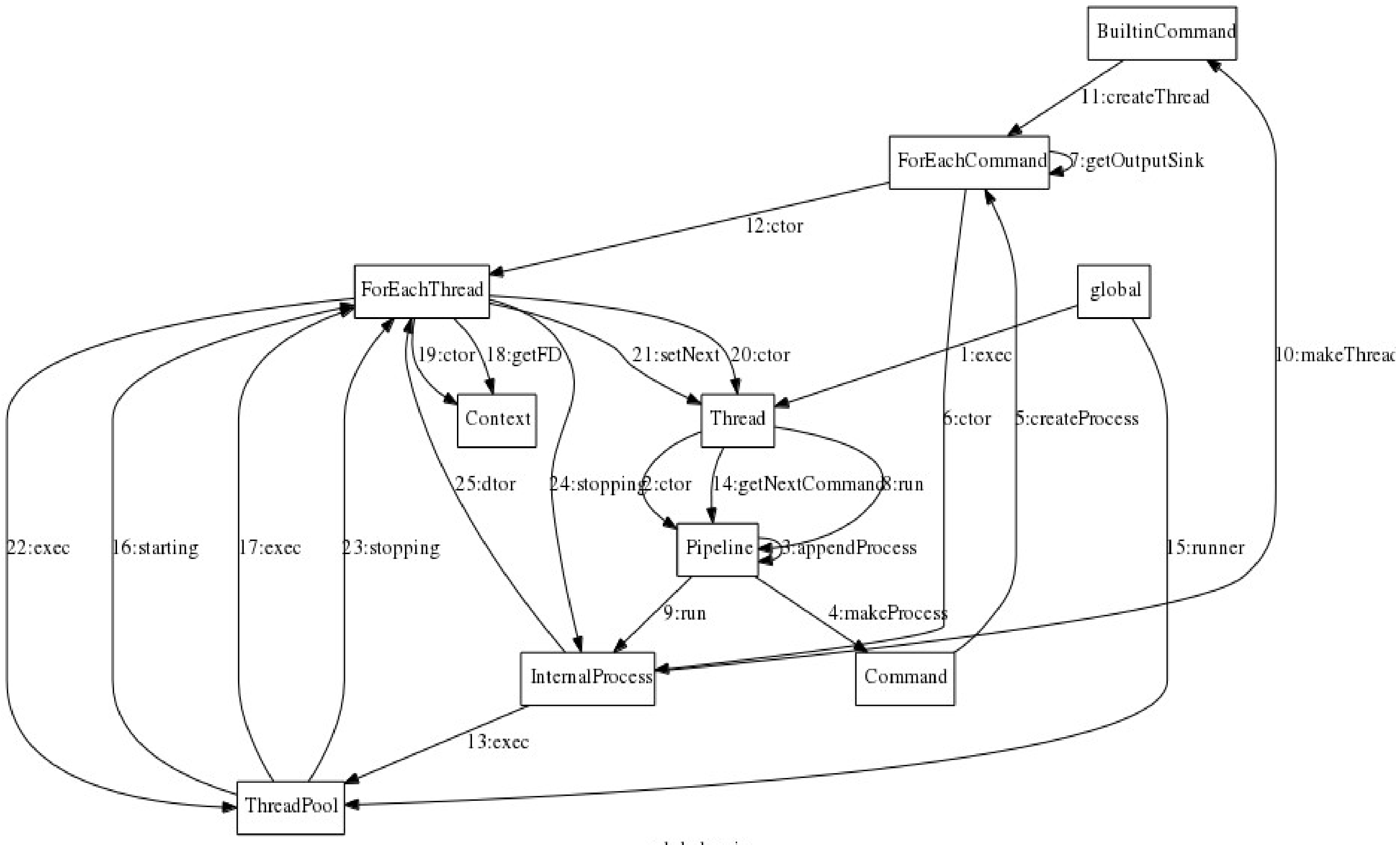
global main

4.11 T19.1: Provide a cd command



global main

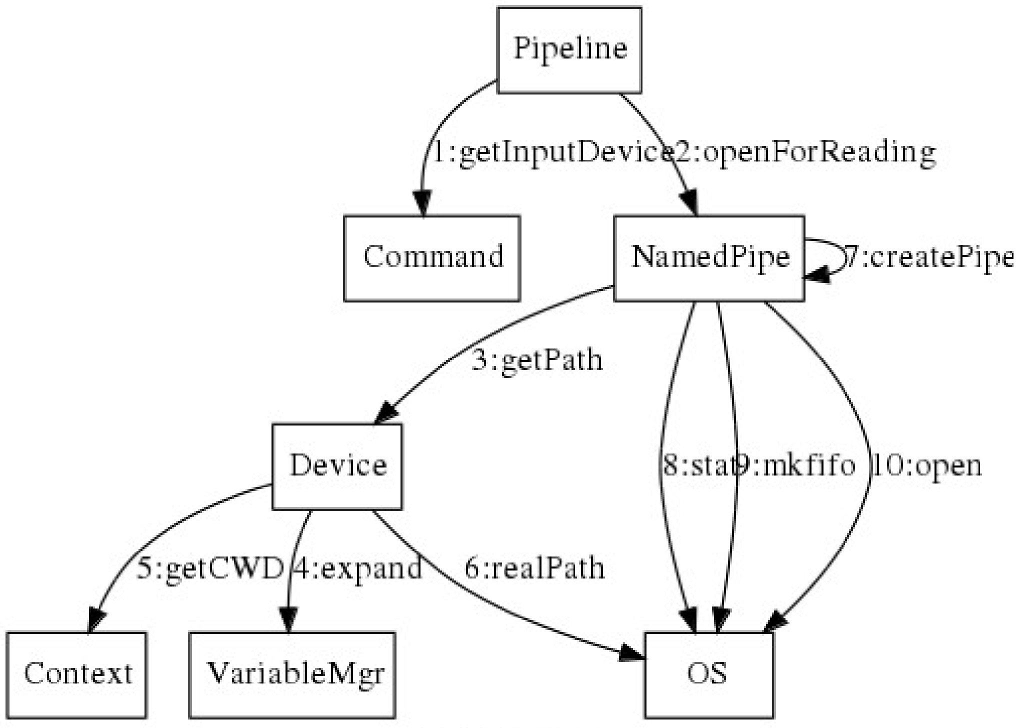
4.12 T19.2: Provide a “for each line” command.



global main

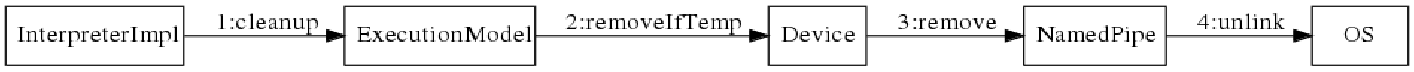
4.13 T19.4: Construct and remove named pipes.

When opening a device that is a named pipe the following sequence of events occur.



Pipeline ctor

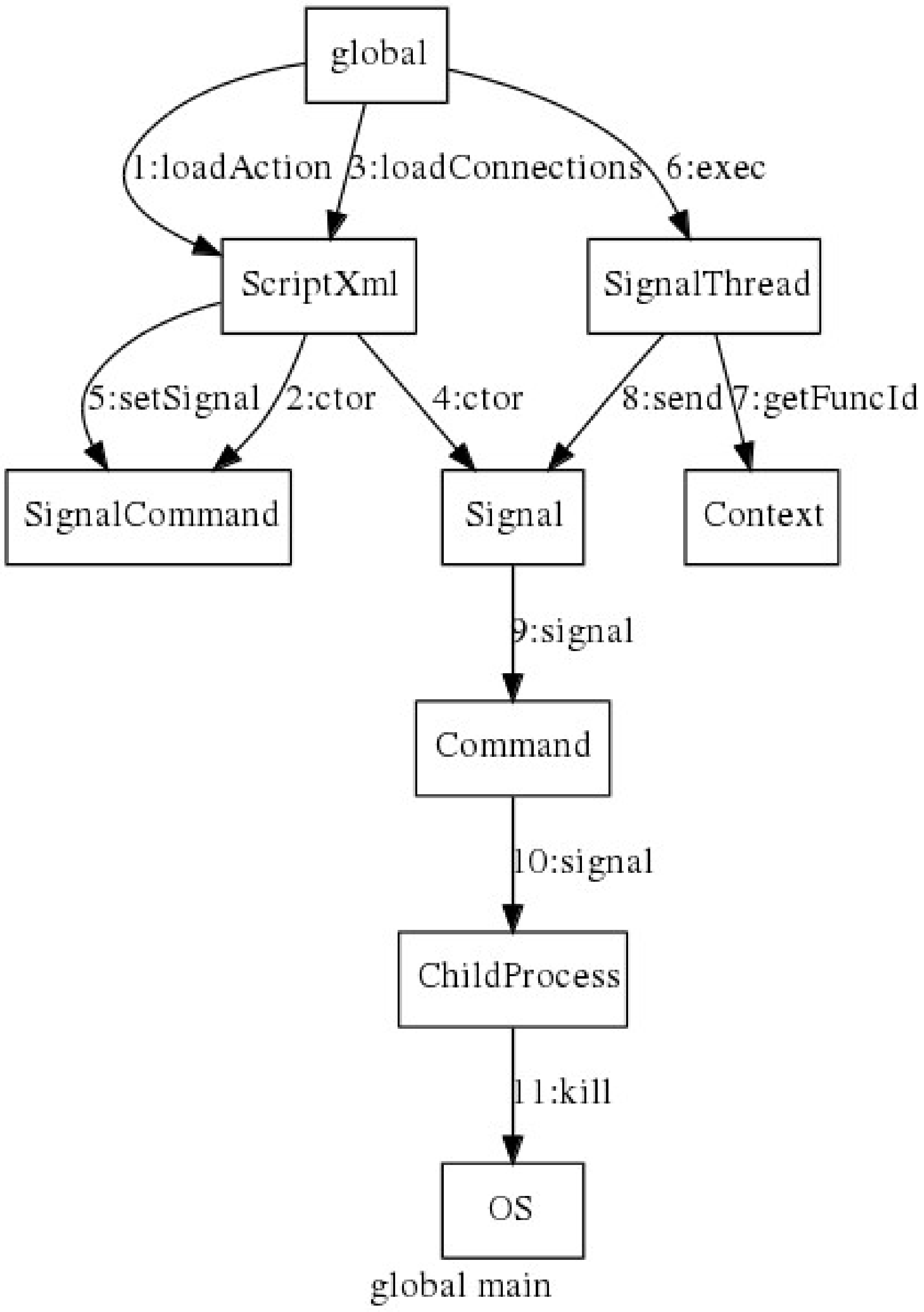
When the script terminates any temporary pipes are removed.



InterpreterImpl threadsFinished

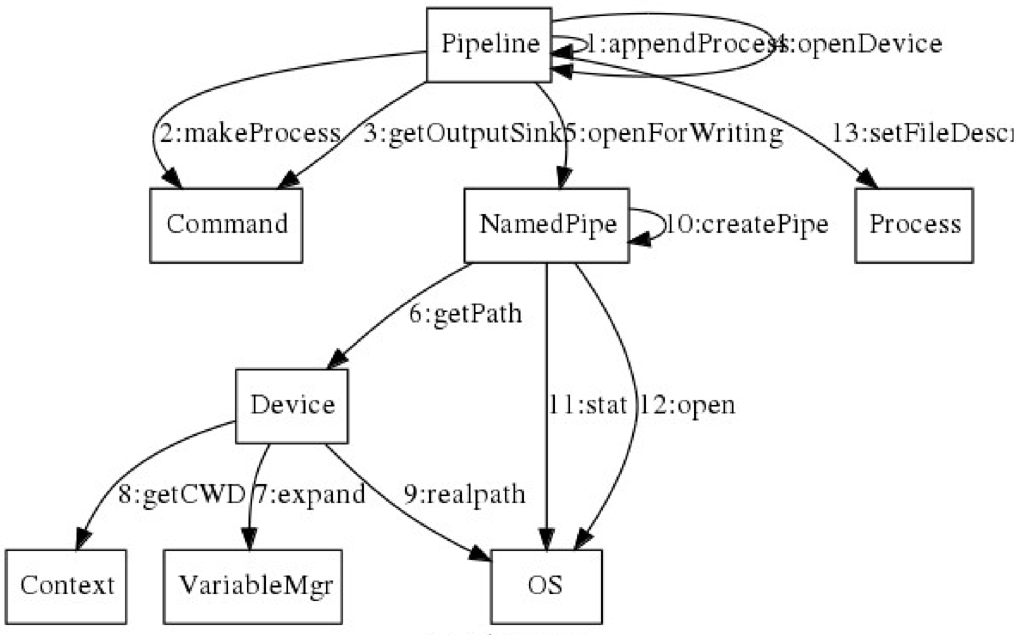
4.14 T19.5: Send a Signal to a Process.

The following sequence of events describe the steps involved in sending a signal from one command to another. The signal can only be sent to another process with the same context (i.e. the same invocation of a function).



4.15 T20.1: Connect to a Named Pipe.

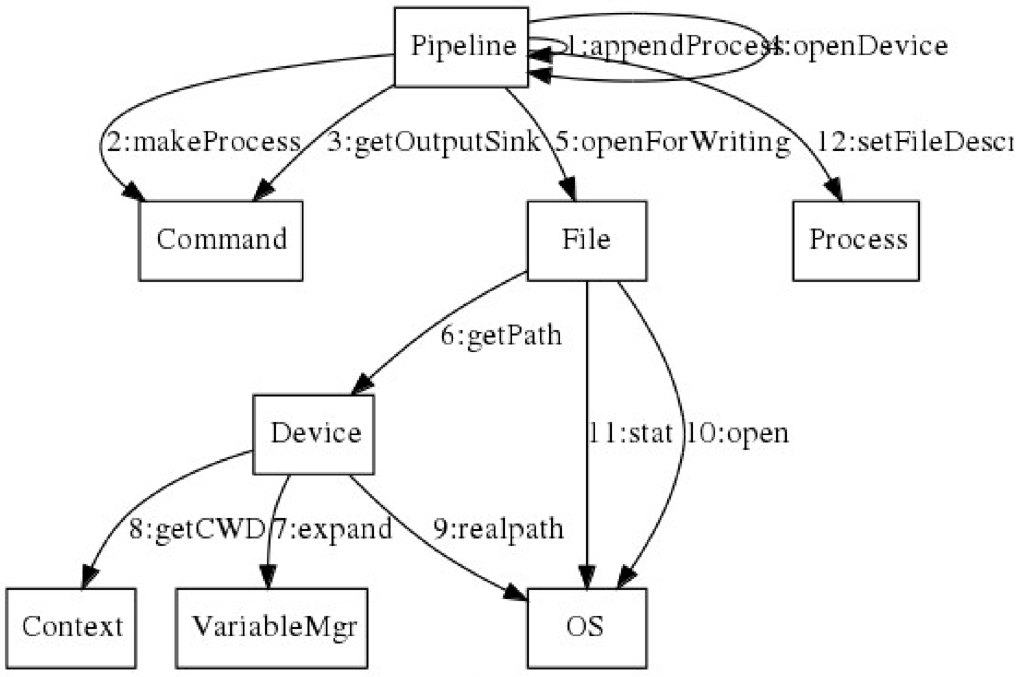
When a pipeline connects to an existing named pipe the following sequence of events occur.



Pipeline ctor

4.16 T20.2: Connect to a File

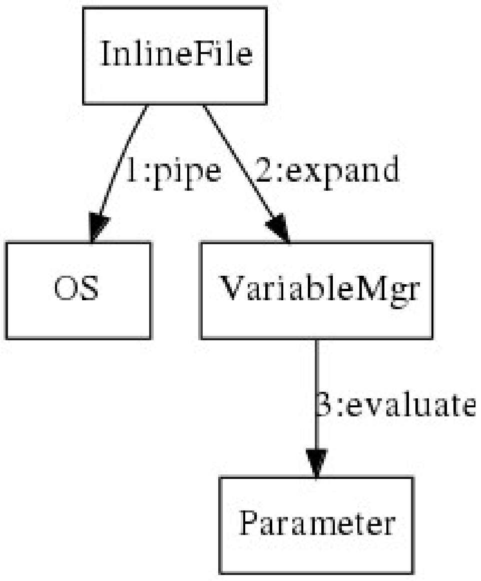
When a pipeline connects to an existing file the following sequence of events occur.



Pipeline ctor

4.17 T21.1: Expand Variables in In-Line File.

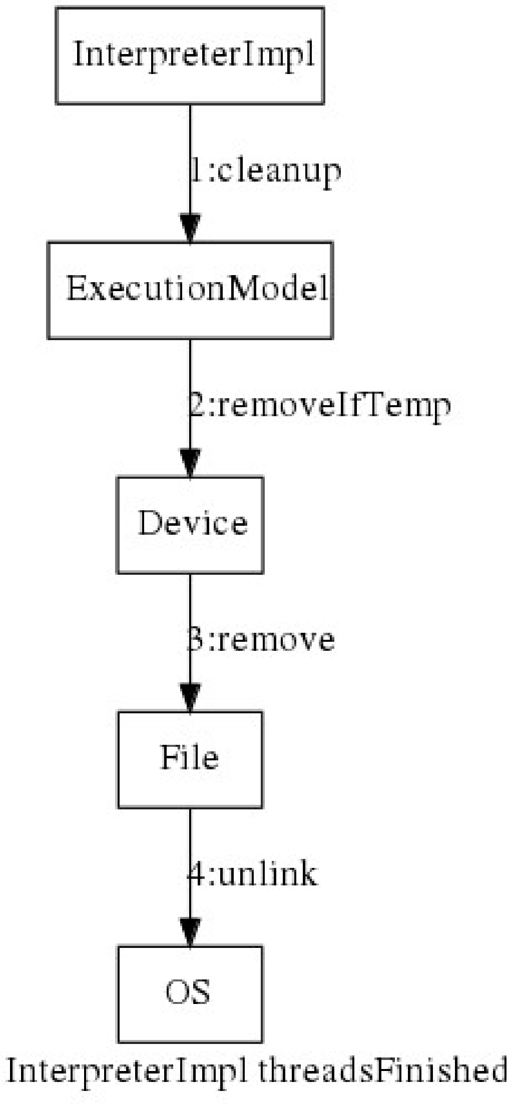
When reading from an in-line file the variables are expanded as shown in the following diagram.



InlineFile openForReading

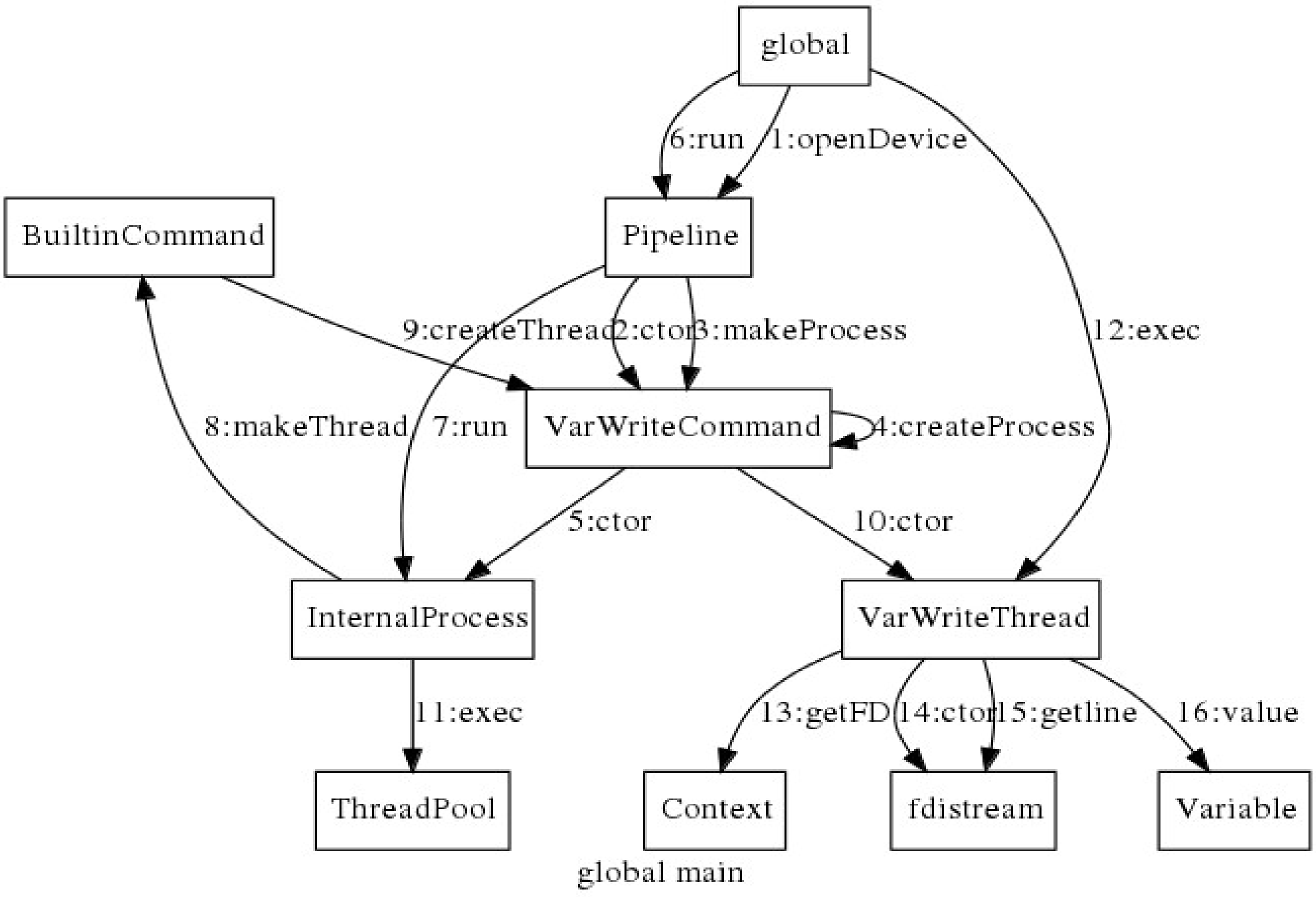
4.18 T22.1: Remove Temporary Files.

When a script has completed all files which were created and are marked as being temporary are removed. Note that existing files are not removed.



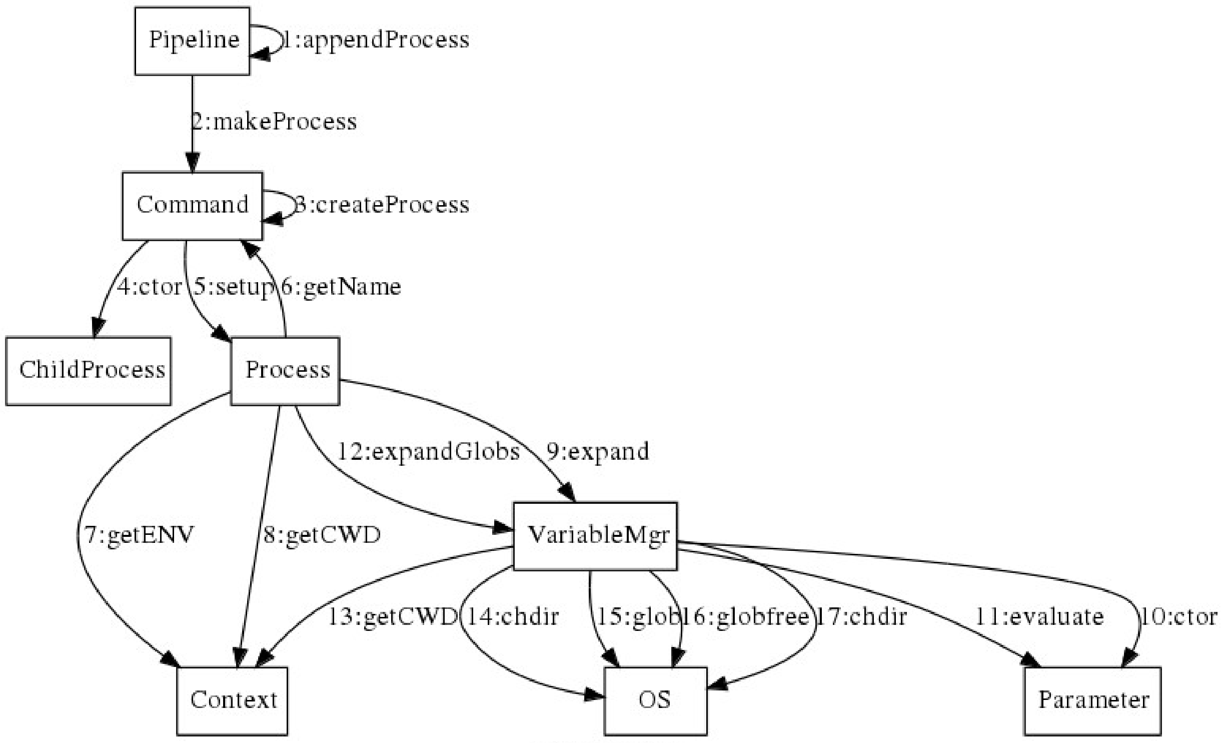
4.19 T24.1: Assign to Variables.

The output of a pipeline (either stdout or stderr) can be assigned to a variable. This uses the VarWriteCommand and its associated thread to perform the operation.



4.20 T24.2: Expand Command Lines

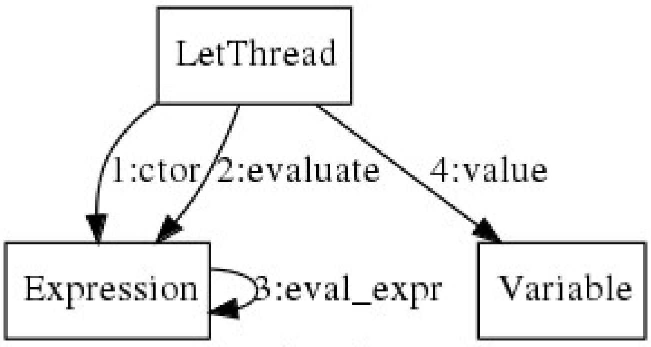
When a process is constructed it is passed the options from the associated command and the current context. The macros in the command options are expanded to their current values using values from the context and the global variables.



Pipeline ctor

4.21 T24.3: Perform Arithmetic

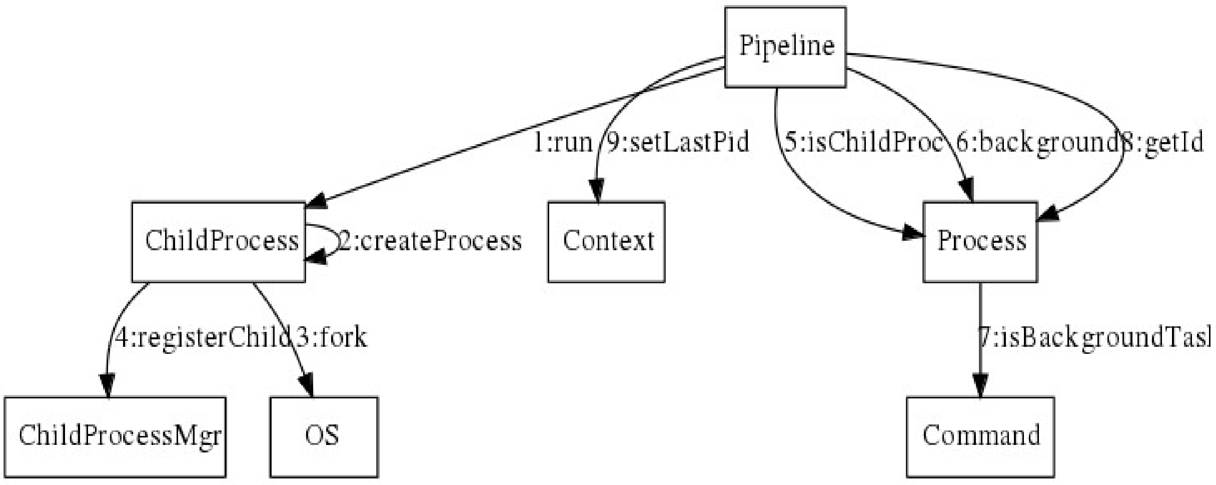
The interpreter implements a “let” command that allows simple calculations to be performed. The command creates a process thread in the same way as other internal threads. When executed it performs the following actions. The Expression object implements a recursive descent parser (not shown).



LetThread exec

4.22 T24.4: Process Ids

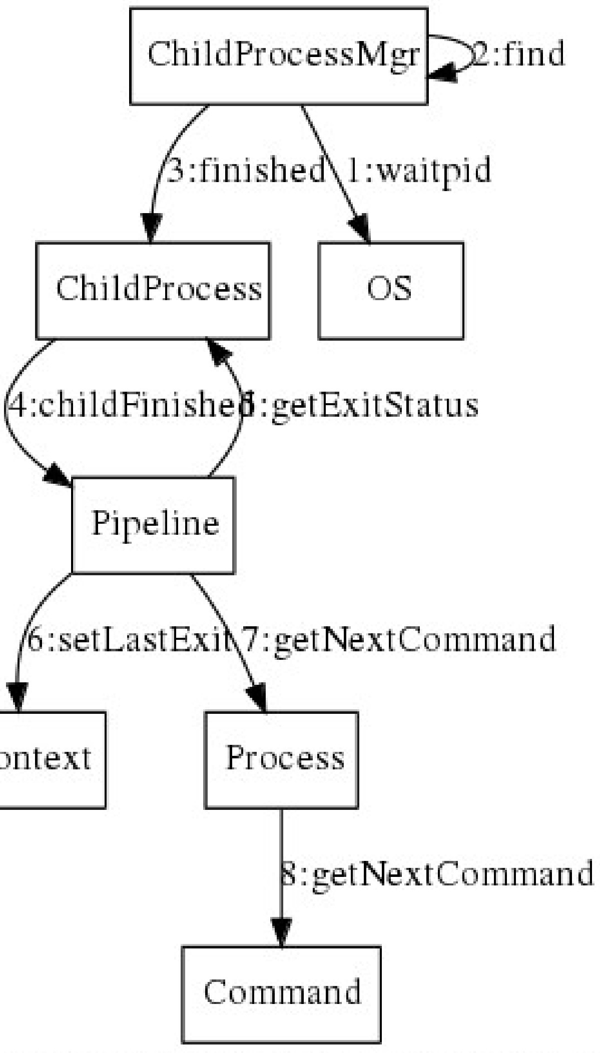
When a process is started in the background the process id is stored in the context and can be accessed using the $! variable. The following diagram show the sequence as the pid is stored.



Pipeline run

4.23 T24.5: Exit Status

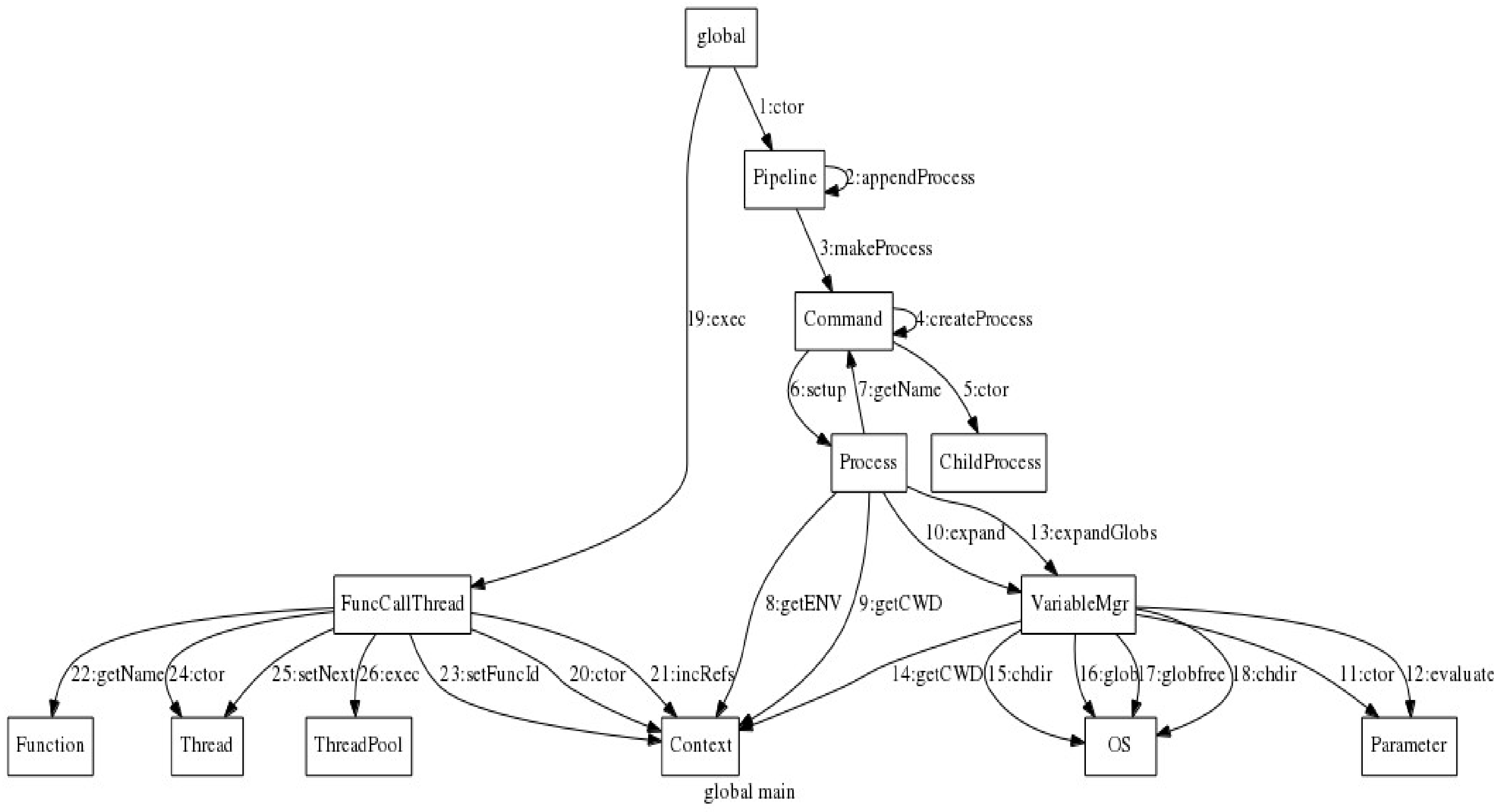
When a child process terminates its exit status is captured so as to determine which command to execute next. The following sequence of events occur on a SIGCHLD signal.



ChildProcessMgr childDiedSignalHandle

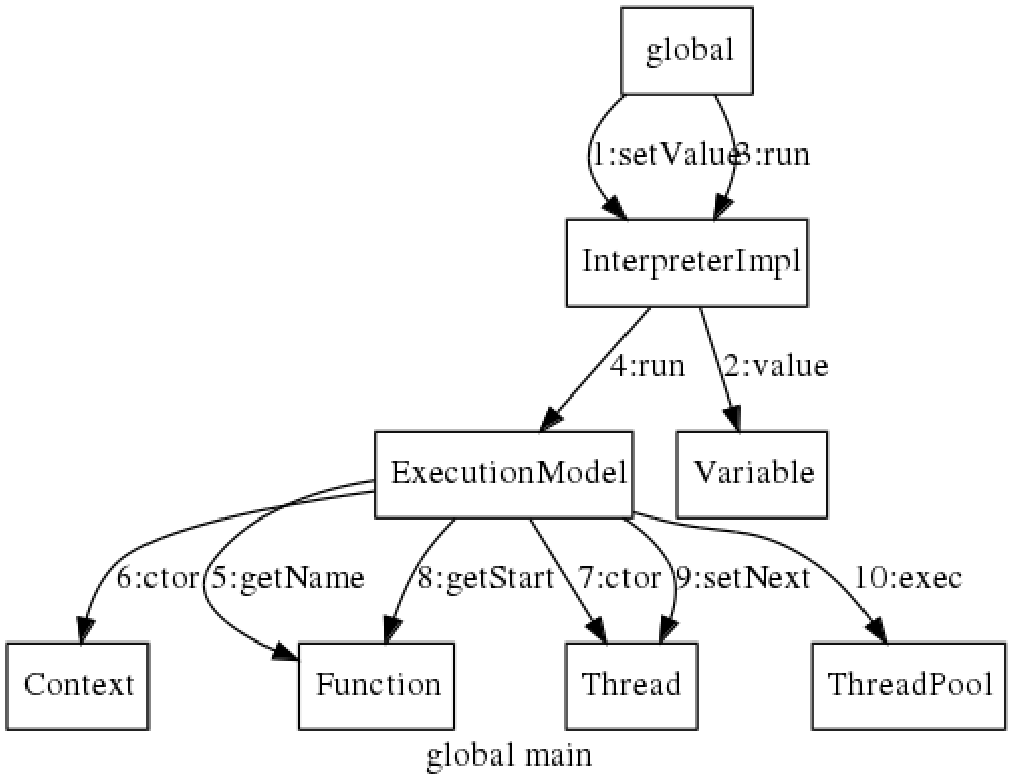
4.24 T24.6: Function Parameters.

When a function that has parameters is called these parameters are expanded and then assigned to an array within the new context.



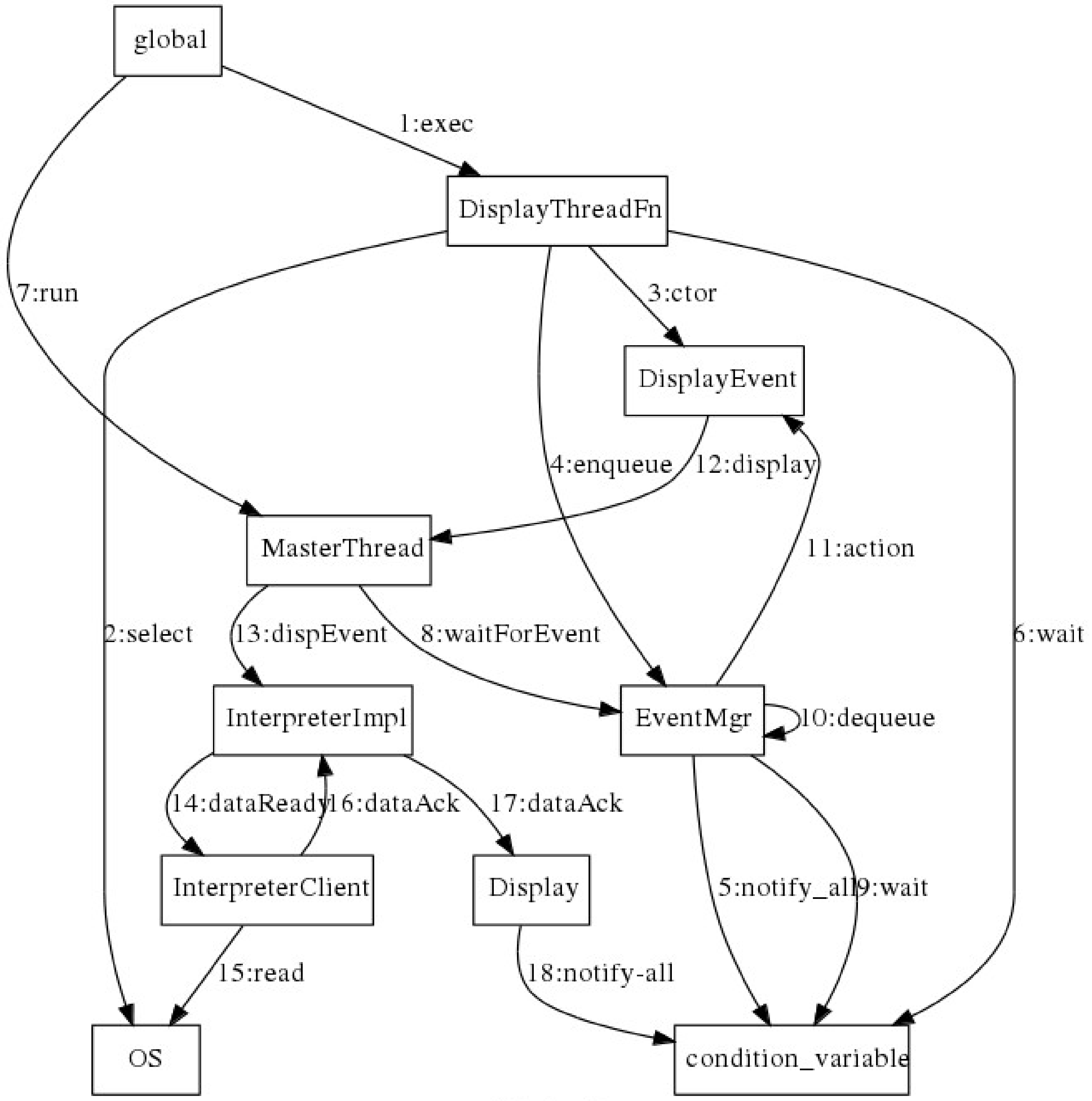
4.25 T24.7: Drag-n-Drop Variables

There is no explicit support in the interpreter for drag-n-drop variables, but its does have the means of allowing the enclosing GUI to implement this functionality. After a drop event the variable value is set and the corresponding function executed.



4.26 Display Output Streams

In order to show the output of commands to the user the GUI needs access to that data. The Display class provides the basis of the link between the pipeline and the GUI.



global main

5 Class Designs

This section describes each class, including its responsibilities, and its public and protected members.

Note: csr is a typedef for const std::string & that is defined in vici.h

5.1 BreakEvent Class

An event that notifies that processing has reached a breakpoint.

```
class BreakEvent : public Event
{
private:
      NodeId id;
      ThreadId tid;
public:
      BreakEvent(ThreadId t, NodeId i);
      virtual void action( EventHandler *eh );
      virtual void print( std::ostream &f );
};
```

5.2 BuiltinCommand Class

A specialization of Command that is implemented within the interpreter rather than as an external process.

```
class BuiltinCommand : public Command
{
public:
      // utility functions
      static unsigned int split( csr text, ArgList & result );
      static unsigned int split( csr text, int maxArgs,
                    ArgList & result );
      static bool isValidName( csr );

protected:
      virtual InternalThreadFn *
            createThread(InternalProcess *, Context &) = 0;

public:
      BuiltinCommand( NodeId, csr name );
      InternalThreadFn * makeThread( InternalProcess *, Context & );
};
```

5.3 CdCommand Class

Responsible for changing the working directory of the script commands.

```
class CdCommand : public BuiltinCommand
{
      virtual Process * createProcess( Context &, Pipeline * );
      virtual InternalThreadFn *
            createThread(InternalProcess *, Context &);
public:
      CdCommand( NodeId );
};
```

5.4 CdThread Class

The implementation of the script cd command.

```
class CdThread : public InternalThreadFn
{
private:
      std::string path;

public:
      CdThread(InternalProcess *ip, Context & ctx )
             : InternalThreadFn(ip, ctx) {}
      virtual void exec();
};
```

5.5 Channel Class

A class representing the link between a source and a sink. This corresponds to a data path on the flow chart.

```
class Channel
{
private:
      Source *source;           // reference
      Sink * sink;        // reference
      int  ident;  // 0 for any, 1 for stdout, 2 for stderr
public:
      Channel( int, Source *, Sink * );
      virtual ~Channel(){}
      virtual void print( std::ostream & );
      Source *getSource() { return source; }
      Sink *getSink() { return sink; }
      int getIdent() { return ident; }
};
```

5.6 ChildProcess Class

Represents the state of a child process

```
class ChildProcess : public Process
{
private:
      pid_t pid;
      int killSignal;

      void createProcess();     //does the process fork
      void execProcess();       // executes the new command

public:
      ChildProcess( ParentOfProcess *, Command * );
      virtual void run();
      virtual void kill();    // terminate the process
      virtual void signal( int sig );

      virtual int getId() { return (int)pid; }
      virtual bool isChildProc() { return true; }
      void finished( int x );
      virtual void setFinished() { status = Finished; }
};
```

5.7 ChildProcessMgr Class

Manage the child processes at a global level which is mostly about handling the SIGCHLD signals and ensuring the correct ChildProcess gets notified.

```
class ChildProcessMgr
{
private:
	static ChildProcessMgr *me;
	static std::mutex instanceMx;
	ChildProcessMgr();
	void installSignalHandler();
	static void childDiedSignalHandler( int sig );

	// the children we are managing, indexed by their pid
	std::map< pid_t, ChildProcess * > children;    // references

public:
	static ChildProcessMgr & instance();

	void registerChild( ChildProcess *cp );
	int  numberOfChildren() const { return children.size(); }
};
```

5.8 Command Class

The static representation of a command. Note that there may be multiple processes corresponding to a single command.

```
class Command : public Source, public Sink
{

protected:
	// map from exit status to next command to execute
	// -1 means "any exit code"
	std::map< int, Command * > next; // references
	Channel *in, *out, *err;
	std::string name;
	std::vector< std::string > options;
	bool backgroundTask;
	NodeId id;
	bool breakPoint;

	std::mutex procMx;
	std::list< std::pair<int, Process *> > runningProcesses;

	virtual Process * createProcess( Context &, Pipeline * );
	// allow process to modify context

public:
	Command( NodeId, csr nm );
	virtual ~Command();
	void setOptions( ArgList & );
	void setBG( bool );
	void setNext( int a, Command *x ) { next[a] = x; }

	virtual NodeId nodeId() { return id; }
	virtual bool isProcess() { return true; }
	virtual bool isPipeStart();
	bool isBackgroundTask() { return backgroundTask; }
	bool isBreakPoint() { return breakPoint; }
```

```
      // some internal commands will do their own pipeline
      // management.
      virtual bool isPipeManager() { return false; }

      virtual Device * getInputDevice();
      virtual Source *getInputSource();
      virtual Sink * getOutputSink();
      virtual Sink * getErrorSink();
      Interp::Command *getNextCommand( int x );

      Process *makeProcess(Context &, Pipeline *);
      void processHasEnded( Process * );
      virtual void signal( int funcId, int sig );

      virtual void print( std::ostream & );
      csr getName() const { return name; }

      // turn on or off breaking at this command
      void toggleBreak();
};
```

5.9 Constant Class

This is a specialisation of DataStore that holds a data value that does not change its value during the execution of the script.

```
class Constant : public DataStore, public Source
{
public:
      Constant( NodeId, csr name, csr val );
      virtual NodeId nodeId();
      virtual bool isProcess();
      virtual void value( csr x );
};
```

5.10 Context Class

Hold the context specific information for a thread of control.

```
class Context
{
public:
      // file descriptors for stdin, stdout, stderr
      static const int StdIn = 0;
      static const int StdOut = 1;
      static const int StdErr = 2;

private:
      // manage access to the context data
      std::mutex accessMx;

      // exit code of last command to terminate
      int lastExitCode;

      // process id (pid) of most recently started ChildProcess
      int lastPid;

      // environment
      std::map< std::string, std::string > env; // key value pairs

      // working directory
      std::string cwd;

      // file descriptors for stdin, stdout, stderr
      int fds[3];

      // identifies the function process
      int funcId;
      int refCount;
private:
      static int funcIdCounter;
public:

      Context();
      Context( const Context &);
      Context & operator = ( const Context & );
      void lock( bool );

      void setFuncId() { funcId = funcIdCounter++; }
      int getFuncId() const { return funcId; }

      std::string getCWD();
      void setCWD( csr );

      void setFD( int id, int fd);
      int getFD(int id);

      void eraseVar( csr name );
      void setVar( csr name, csr val );
      std::string getVar( csr name );
      void getEnv( std::map< std::string, std::string > & );

      void incRefs();
      void decRefs();
      bool hasRefs();

      void setLastExit(int);
      int getLastExit();

      void setLastPid(int);
      int getLastPid();

      // name of current function
      std::string funcName;

      // parameters accessed via $1, $2, etc.
      VICI::ArgList parameters;

      // process id of this Vici process
      int ourPid;

};
```

5.11 *CursorEvent Class*

An event that notifies that a processing step has been made. (There is a lot of these during debugging.)

```
class CursorEvent : public Event
{
```

```
private:
	NodeId id;
	ThreadId tid;
public:
	CursorEvent(ThreadId t, NodeId i);
	virtual void action( EventHandler *eh );
	virtual void print( std::ostream &f );
};
```

5.12 DataStore Class

This is the base class for anything that can store a data value.

```
class DataStore
{
protected:
	std::string name;
	std::string val;
	NodeId id;

public:
	virtual ~DataStore();
	virtual void print( std::ostream & );
	virtual csr value();
	virtual void value( csr x );
};
```

5.13 Device Class

An abstract class representing devices and files.

```
class Device : public Source, public Sink
{
protected:
	NodeId id;

	// gets normalised path, taking into account
	// current working dir
	std::string getPath( csr path, Context & );
	bool temporary;
	virtual void remove() = 0;
public:
	Device( NodeId, bool temp );
	virtual ~Device();
	virtual void print( std::ostream & );
	virtual NodeId nodeId() { return id; }
	virtual bool isProcess() { return false; }
	virtual int openForReading( Context & );
	virtual int openForWriting( Context & );
	void removeIfTemp();
	virtual bool hasChanged() { return false; }
};
```

5.14 Display Class

A device that is used to display the text on the GUI.

```
class Display : public Device, public ThreadFnOwner
{
private:
	virtual void remove(){}
```

```
      int fdw, fdr;// read and write file descriptors
      DisplayThreadFn *fn;
public:
      Display( NodeId );
      ~Display();
      virtual int openForWriting( Context & );
      virtual int openForReading( Context & );
      int openForReading();
      void dataAck();
      virtual void starting( ThreadFunction *, int job );
      virtual void stopping( ThreadFunction * );
};
```

5.15 *DisplayEvent Class*

An event that notifies that data is available to read from a Display object.

```
class DisplayEvent : public Event
{
private:
      NodeId id;
public:
      DisplayEvent(NodeId i);
      virtual void action( EventHandler *eh );
      virtual void print( std::ostream &f );
};
```

5.16 *DisplayThreadFn Class*

A thread function that waits for data on the stream it is monitorint, and also waits for acknowledgements that the data has been displayed.

```
class DisplayThreadFn : public ThreadFunction
{
private:
      bool terminate;
      int fdr;
      NodeId id;
      std::mutex ackMx;
      std::condition_variable ackCV;
      bool ack;
public:
      DisplayThreadFn(ThreadFnOwner *, int fd, NodeId );
      virtual void exec();
      virtual void kill();
      void acknowledge();
};
```

5.17 *DoneEvent Class*

An event that notifies that the script has completed.

```
class DoneEvent : public Event
{
public:
      DoneEvent();
      virtual void action( EventHandler *eh );
      virtual void print( std::ostream &f );
};
```

5.18 DupCommand Class

A command which reads its input stream and copies it to both its output and error streams, thus duplicating the data stream.

```
class DupCommand : public BuiltinCommand
{
      virtual Process * createProcess( Context &, Pipeline * );
      virtual InternalThreadFn *
             createThread(InternalProcess *, Context &);
public:
      DupCommand( NodeId );
};
```

5.19 DupThread Class

The thread which reads its input stream and writes to both output streams.

```
class DupThread : public InternalThreadFn
{
public:
      DupThread(InternalProcess *ip, Context & ctx )
             : InternalThreadFn(ip, ctx) {}
      virtual void exec();
};
```

5.20 EchoCommand Class

A simple command that copies its parameters to its output stream.

```
class EchoCommand : public BuiltinCommand
{
      virtual Process * createProcess( Context &, Pipeline * );
      virtual InternalThreadFn *
             createThread(InternalProcess *, Context &);
public:
      EchoCommand( NodeId );
};
```

5.21 EchoThread Class

The thread function that writes its parameters to its output stream.

```
class EchoThread : public InternalThreadFn
{
public:
      EchoThread(InternalProcess *ip, Context & ctx )
             : InternalThreadFn(ip, ctx) {}
      virtual void exec();
};
```

5.22 Event Class

A base class for the different events that can be passed through the event manager.

```
class Event
{
public:
      virtual ~Event() {}
      virtual void action( EventHandler * ) = 0;
      virtual void print( std::ostream & ) = 0;
};
```

5.23 EventHandler Class

An interface that is implemented to receive notification of events.

```
class EventHandler
{
protected:
      virtual void waitForEvent();
public:
      virtual ~EventHandler();

      virtual void done() = 0;
      virtual void varAction( csr name, csr val ) = 0;
      virtual void display( NodeId ) = 0;
      virtual void breakAction( ThreadId, NodeId ) = 0;
      virtual void cursorAction( ThreadId, NodeId ) = 0;
      virtual void fileAction( csr path ) = 0;
};
```

5.24 EventMgr Class

A singleton class responsible for queuing and dequeuing the events, possibly from different threads.

```
class EventMgr
{
private:
      EventMgr();
      static EventMgr *me;
      static std::mutex instanceMx;

      std::mutex queueMx;
      std::mutex eventMx;
      std::condition_variable eventCV;

      std::queue< Event * > eventQueue;      // owned
      EventHandler *handler;                 // reference

      bool isDebugMode;
public:
      static EventMgr & instance();
      void registerHandler( EventHandler *client );

      void enqueue( Event * );
      Event * dequeue();
      void waitForEvent();

      void debugging(bool x) { isDebugMode = x; }
      bool debugging() { return isDebugMode; }
};
```

5.25 ExecutionModel Class

Holds the data structure that represents the script.

```
class ExecutionModel
{
      friend class VICI::Interp::ScriptXml;
private:
      std::map< NodeId, Function * > functions;     // owned
      std::vector< Device * > devices;        // owned
      std::vector< Channel * > channels;      // owned
      std::vector< Signal * > gSignals;       // owned - beware name
clash with qt for "signals"

      std::map< NodeId, Command * > commands; // references
      VariableMgr &variables;

      Display *outDisplay, *errDisplay;       // references to devices
public:
      ExecutionModel();
      ~ExecutionModel();

      void run(csr funcname);
      void pause();
      void resume();
      void kill();
      void cleanup();

      Function *getFunction( VICI::csr funcname );
      Display *getDisplay(NodeId);

      // debugging and testing interface
      void print( std::ostream & );
      void setBreak( NodeId );
      void setPosn( ThreadId, NodeId );

      // not implemented - no copies to be made
      ExecutionModel( const ExecutionModel & ) = delete;
      ExecutionModel & operator = ( const ExecutionModel & ) = delete;
};
```

5.26 ExportCommand Class

A command that is used to set an environment variable which is then passed to subsequent command in this or a child context.

```
class ExportCommand : public BuiltinCommand
{
private:
      virtual Process * createProcess( Context &, Pipeline * );
      virtual InternalThreadFn *
             createThread(InternalProcess *, Context &);
public:
      ExportCommand( NodeId );
};
```

5.27 ExportThread Class

The InternalThreadFn that adds (or removes) a value from the environment of the script.

```
class ExportThread : public InternalThreadFn
{
private:
      std::string varName;
      std::string value;

public:
      ExportThread(InternalProcess *ip, Context & ctx )
             : InternalThreadFn(ip, ctx) {}
      virtual void exec();
};
```

5.28 Expression Class

This class encapsulates the parsing and evaluation of numeric expressions.

```
class Expression
{
private:
      int eval_expr( const char **p );
      int eval_term( const char **p );
      int eval_factor( const char **p );
      void skip_ws( const char **p);
public:
      Expression();
      std::string evaluate(csr expr);
};
```

5.29 File Class

Represents a file.

```
class File : public Device
{
protected:
      std::string filePath;
      std::vector< std::string > createdFiles;
      bool mAppend;
      virtual void remove();
      std::map<std::string, struct stat> fileStates;

public:
      File( NodeId, csr path, bool temp );
      virtual ~File();
      void append(bool);
      virtual void print( std::ostream & );
      virtual int openForReading( Context & );
      virtual int openForWriting( Context & );
      virtual bool hasChanged();
};
```

5.30 FileEvent Class

An event that notifies that a file has changed.

```
class FileEvent : public Event
{
private:
      std::string path;
public:
      FileEvent( csr p );
      virtual void action( EventHandler *eh );
      virtual void print( std::ostream &f );
};
```

5.31 ForEachCommand Class

A command that reads its input stream and for each object runs a new processing thread to execute a sub-process.

```
class ForEachCommand : public BuiltinCommand
{
private:
      char sepChar;
      Command *nextCommand;


      virtual Process * createProcess( Context &, Pipeline * );
      virtual InternalThreadFn *
             createThread(InternalProcess *, Context &);
public:
      ForEachCommand( NodeId, char separator );
      virtual void print( std::ostream & );
      virtual bool isPipeManager() { return true; }

};
```

5.32 ForEachThread Class

The thread function object that waits for the sub-process to complete before reading the next input.

```
class ForEachThread : public InternalThreadFn, public ThreadFnOwner
{
private:
      Command * nextCommand;
      char sepChar;
      std::mutex doneMx;
      std::condition_variable doneCV;
      bool done;

public:
      ForEachThread(InternalProcess *ip, Context & ctx,
             Command * nc, char sep )
             : InternalThreadFn(ip, ctx), nextCommand(nc),
                    sepChar(sep), done(false) {}
      virtual void exec();
      virtual void kill();
      virtual void starting( ThreadFunction *, int job );
      virtual void stopping( ThreadFunction * );

};
```

5.33 FuncCallCommand Class

Creates a new thread of control that starts at the specified function and waits until has completed.

```
class FuncCallCommand : public BuiltinCommand
{
private:
	ExecutionModel *model;
	std::string functionName;
	Function *fn;
	virtual Process * createProcess( Context &, Pipeline * );
	virtual InternalThreadFn *
		createThread(InternalProcess *, Context &);

public:
	FuncCallCommand( NodeId, csr funcname, ExecutionModel * );
	virtual void print( std::ostream & );
	virtual bool isPipeManager() { return true; }
};
```

5.34 FuncCallThread Class

The thread function object that waits for the called function to complete.

```
class FuncCallThread : public InternalThreadFn, public ThreadFnOwner
{
private:
	// ArgList options;
	Function *fn;

	std::mutex doneMx;
	std::condition_variable doneCV;
	bool done;

public:
	FuncCallThread(InternalProcess *ip, Context & ctx, Function *f )
		: InternalThreadFn(ip, ctx), fn(f), done(false) {}
	virtual void exec();
	virtual void kill();
	virtual void starting( ThreadFunction *, int job );
	virtual void stopping( ThreadFunction * );
};
```

5.35 Function Class

A collection of commands to be executed.

```
class Function
{
	friend class VICI::Interp::ScriptXml;
private:
	std::vector< Command * > commands;	// owned
	std::list< Pipeline * > pipelines;
```

```
      std::string name;
      Command *first;                         // reference
public:
      Function( csr funcname );
      ~Function();
      csr getName() const { return name; }
      void setStart( Command *cmnd ) { first = cmnd; }
      Command * getStart() { return first; }

      // testing interface
      void print( std::ostream & );

      // not implemented
      Function( const Function & );
      Function & operator = ( const Function & );
};
```

5.36 InlineFile Class

Text that is embedded in the script.

```
class InlineFile : public Device
{
private:
      std::string content;
      virtual void remove(){}
public:
      InlineFile(  NodeId );
      void setContent( csr );
      virtual void print( std::ostream & );
      virtual int openForReading( Context & );
};
```

5.37 InternalProcess Class

Represents a built in command running in a thread within this process

```
class InternalProcess : public Process, public ThreadFnOwner
{
friend class InternalThreadFn;
private:
      Context & ctx;
      InternalThreadFn *myThread;             // owned
protected:
      int job;
public:
      InternalProcess(  ParentOfProcess *, Command *, Context &ctx );
      ~InternalProcess();
      virtual void setFileDescr( int fd, int id );
      virtual void run();
      virtual void kill();
      virtual int getId();
      virtual bool isChildProc() { return false; }
      virtual void setFinished();
      virtual void starting( ThreadFunction *, int job );
      virtual void stopping( ThreadFunction * );
      void setExitStatus( int x) { exitStatus = x; }
};
```

5.38 InternalThreadFn Class

A function object for running a thread for internal processes.

```
class InternalThreadFn : public ThreadFunction
{
protected:
	Context &ctx;
	InternalProcess *ip;	// reference
	ArgList options;
	std::string errMsg;
	bool terminate;
public:
	InternalThreadFn( InternalProcess *, Context & );
	virtual void kill();
	void setErrMsg(csr);
};
```

5.39 InterpreterFactoryImpl Class

A factory for creating Interpreter objects.

```
class InterpreterFactoryImpl : public InterpreterFactory
{
public:
	InterpreterFactoryImpl(){}

	virtual Interpreter * makeInterpreter( Sec::Secure * );
};
```

5.40 InterpreterImpl Class

The facade of the interpreter library, responsible for its public interface.

```
class InterpreterImpl : public QObject, public Interpreter
{
	Q_OBJECT
private:
	std::vector< InterpreterClient * > clients;   // references
	ExecutionModel *model;
	std::map< NodeId, Display *> displays;
public:
	InterpreterImpl();
	~InterpreterImpl();

	virtual void addClient( InterpreterClient * );
	virtual void debugMode( bool );
	virtual int openDisplay( NodeId );
	virtual void dataAck(NodeId);
	virtual void setScript( csr filename );
	virtual void setValue( csr varName, csr value );
	virtual void setPosn( ThreadId, NodeId );
	virtual void setBreak( NodeId );
	virtual void setInterval( double );
	virtual void saveSnapshot( csr filename );
	virtual void loadSnapshot( csr filename );
	virtual void run();
	virtual void run( csr funcname );
	virtual void pause();
```

```
      virtual void resume();
      virtual void step( ThreadId );
      virtual void kill();

public slots:
      void threadsFinished();
      void varEvent( const std::string &name,
            const std::string &value );
      void dispEvent( NodeId );
      void breakEvent( ThreadId, NodeId );
      void cursorEvent( ThreadId, NodeId );
      void fileEvent(const std::string & path);
};
```

5.41 Job Class

A thread function that runs a background pipeline

```
class Job : public ThreadFunction
{
protected:
      Pipeline *pipeline;
      Context * ctx;

public:
      Job( ThreadFnOwner *, Context *, Pipeline * );
      ~Job();
      virtual void exec();
      virtual void kill();
};
```

5.42 LetCommand Class

The command responsible for performing simple numeric calculations.

```
class LetCommand : public BuiltinCommand
{
private:
      virtual Process * createProcess( Context &, Pipeline * );
      virtual InternalThreadFn *
            createThread(InternalProcess *, Context &);
public:
      LetCommand( NodeId );
};
```

5.43 LetThread Class

The InternalThreadFn that implements the “let” command.

```
class LetThread : public InternalThreadFn
{
public:
      LetThread(InternalProcess *ip, Context & ctx)
            : InternalThreadFn(ip, ctx) {}
      virtual void exec();
};
```

5.44 MasterThread Class

Wrap the runtime in a thread that can communicate with the Qt event loop so that we can notify the GUI of events.

```
class MasterThread : public QThread, public EventHandler
{
      Q_OBJECT
private:
      bool running;
public:
      MasterThread();
protected:
      virtual void run();

      // event handler interface
      virtual void done();
      virtual void varAction( csr name, csr val );
      virtual void display( NodeId );
      virtual void breakAction( ThreadId, NodeId );
      virtual void cursorAction( ThreadId, NodeId );
      virtual void fileAction( csr path );
signals:
      void varEvent( const std::string & name,
             const std::string & val );
      void dispEvent( NodeId );
      void breakEvent( ThreadId, NodeId );
      void cursorEvent( ThreadId, NodeId );
      void fileEvent( const std::string & path );
};
```

5.45 Mutex Class

A specialisation of Variable used to control access to a portion of the script so that only one thread can access it at a time.

```
class Mutex : public Variable
{
private:
      std::mutex padlock;
public:
      Mutex( NodeId, csr name );
      virtual ~Mutex();
      void lock();
      void unlock();
};
```

5.46 MutexCommand Class

Built in command for locking and unlocking a mutex specified as a parameter. It may block until the mutex is unlocked by another thread.

```
class MutexCommand : public BuiltinCommand
{
protected:
      Mutex *mutex;// reference
      virtual Process * createProcess( Context &, Pipeline * );
      virtual InternalThreadFn *
```

```
             createThread(InternalProcess *, Context &);
public:
      MutexCommand( NodeId );
};
```

5.47 MutexThread Class

The thread function object that implements the command. It may block waiting for a lock.

```
class MutexThread : public InternalThreadFn
{
public:
      MutexThread(InternalProcess *ip, Context & ctx )
             : InternalThreadFn(ip, ctx){}
      virtual void exec();
};
```

5.48 NamedPipe Class

A specialization of Pipe that persists beyond the lifetime of the script. It may be used to connect the script to other programs.

```
class NamedPipe : public Device
{
protected:
      std::string filePath;
      std::vector< std::string > createdPipes;
      void createPipe( csr path );
      virtual void remove();
public:
      NamedPipe( NodeId, csr name, bool temp );
      virtual ~NamedPipe();
      virtual void print( std::ostream & );
      virtual int openForReading( Context & );
      virtual int openForWriting( Context & );
};
```

5.49 Parameter Class

This class is responsible for parsing and expanding command options. A command option or parameter may include various symbols that need to be expanded in order to determine the context specific value of the parameter. This class is responsible for that expansion.

5.49.1 Syntax

The parameters to a command are expanded as they are passed to the corresponding process based on the context and the current values of the global variables.

Characters may be preceded by '\' escape character to remove their special meaning. The '\' is removed prior to the parameter being passed to the process.

The '\' can itself be escaped with a '\'.

Parts of a parameter may be surrounded by single or double quote marks. If single quote marks are used then the quotes are removed and the contents passed to the process unchanged.

The parts of a parameter surrounded by double quotes will have all symbols starting with $ expanded and the double quotes removed. If there are no quotes it will finally be passed to the glob expansion routine which may replace the parameter with multiple parameters according to the files in the current working directory of the process.

The $ symbols are interpreted as follows:

$$ - is replaced with the process id of the interpreter.

$! - is replaced by the process id of the last command started in the background.

$* - is replaced with a concatenation of all the parameters to the function.

$? - is replaced by the exit status of the previous command.

$# - is replaced by the number of parameters to the function.

$0 – is replaced by the name of the function.

$1 - $99 is replaced by the parameter to the function in this position, where 1 is the first parameter.

$name – is replaced by the value of the variable with the specified name.

$(name) - is replaced by the value of the variable with the specified name. The name may be the expansion of a variable: $($2) so that the name of the variable can be passed to a function.

The syntax is:

```
param_list ::= param [ param_list ]
param ::= sq | dq | uq
sq ::= ' text '
dq ::= “ opt_list “
opt_list ::= opt [ opt_list ]
opt ::= $var | text
var ::= number | special | name | ( item_list )
number ::= digit [ number ]
special ::= $ | * | # | ! | ?
name ::= letter [ name_chars ]
name_chars ::= name_char [ name_chars ]
name_char ::= letter | digit | _
item_list :: item [ item_list ]
item ::= name | $var
uq ::= opt_list   <---- this is then glob expanded.
```

The class uses a recursive descent algorithm and has the following interface:

```
class Parameter
{
private:
	std::string param_list;
	enum Quoted { Single, Double, Unquoted };
	Quoted quoted;
	bool hasGlobChar;

	std::string get_param(const char **p, Context &ctx);
	std::string get_param_list(const char **p, Context &ctx);
	std::string get_sq( const char **p, Context &ctx);
	std::string get_dq( const char **p, Context &ctx );
	std::string get_uq( const char **p, Context &ctx );
	std::string get_opt_list( const char **p, Context &ctx );
	std::string get_opt( const char **p, Context &ctx );
	std::string get_text( const char **p, Context &ctx );
	std::string get_var( const char **p, Context &ctx );
	std::string get_number( const char **p, Context &ctx );
	std::string get_special( const char **p, Context &ctx );
	std::string get_name( const char **p, Context &ctx );
	std::string get_item_list( const char **p, Context &ctx );
	std::string get_item( const char **p, Context &ctx );

public:
	Parameter() : quoted(Unquoted), hasGlobChar(false) {}
	std::string evaluate( csr param, Context &ctx );
};
```

5.50 *ParentOfProcess Class*

Abstract class defining the interface for owners of ChildProcess

```
class ParentOfProcess
{
public:
	virtual ~ParentOfProcess(){}
	virtual void childFinished( ChildProcess * ) = 0;
	virtual void childFinished( InternalProcess * ) = 0;
	virtual void closeFileDescriptorsExcept( int fds[3] ) = 0;
};
```

5.51 *Pipeline Class*

Run a set of processes that are connected by pipes.

```
class Pipeline : public ParentOfProcess
{
private:
	Context &ctx;
	std::vector< Process * > children; // owned
	int runningChildren;
	bool crashing;
	std::condition_variable waitOnChildren;
	std::vector< int > fileDescriptors;
	std::vector< int > childFileDescriptors;
	std::vector< Device * > writeDevices;
	Command *nextCommand;
	std::mutex runningMx;
	std::vector< BuiltinCommand * > tempCommands; // owned
```

```
      int openDevice( Sink *, int id );
      int openReader( DataStore *, int fdIn );
      void appendProcess( Command *, int fdIn );
      void saveFD( int fd, bool forChild = false );

public:
      Pipeline( Context &, Command * );
      Pipeline( const Pipeline & ) = delete;
      virtual ~Pipeline();
      bool background();
      void run();
      bool done() const;
      void childFinished( ChildProcess * );
      void childFinished( InternalProcess * );
      virtual void closeFileDescriptorsExcept( int fds[3] );
      void terminate();
      Command * getNextCommand() { return nextCommand; }

      void operator = ( const Pipeline & ) = delete;
};
```

5.52 Process Class

The abstract base class for internal and child processes.

```
class Process
{
public:
      enum Status { Created, Configured, Running, Completed, Finished
};

protected:
      ParentOfProcess *owner;          // reference
      Command * commandPtr;            // reference
      std::string command;
      std::vector< std::string > parameters;
      std::map< std::string, std::string > environment;
      std::string workDir;
      int fds[3];          // file descriptors
      Status status;
      int exitSignal, exitStatus;

public:
      Process( ParentOfProcess *, Command * );
      virtual ~Process();
      virtual void setup( Context &, ArgList &options );
      virtual void setFileDescr( int fd, int id );
      bool background();

      virtual void run() = 0;
      virtual void kill() = 0;    // terminate the process
      virtual void signal( int sig ) {}
      virtual int getId() = 0;
      virtual bool isChildProc() = 0;

      bool done() const { return (status == Finished); }
      void getExitStatus( int &status, int &sig );
      virtual void setFinished() { status = Finished; }
      void reportExitStatus();
      Status getStatus() { return status; }
      Command * getNextCommand();
};
```

5.53 ReadCommand Class

Reads a line of input and assigns the values to the variables listed as its arguments.

```
class ReadCommand : public BuiltinCommand
{
      virtual Process * createProcess( Context &, Pipeline * );
      virtual InternalThreadFn *
             createThread(InternalProcess *, Context &);
public:
      ReadCommand( NodeId );
};
```

5.54 ReadThread Class

The thread function object that reads its input stream and assigns to the variables listed as parameters.

```
class ReadThread : public InternalThreadFn
{
public:
      ReadThread(InternalProcess *ip, Context & ctx )
             : InternalThreadFn(ip, ctx) {}
      virtual void exec();
};
```

5.55 ReturnCommand Class

Sets its exit status according to its parameter. It normally does not have any next command defined and therefore causes the current Thread to complete.

```
class ReturnCommand  : public BuiltinCommand
{
      virtual Process * createProcess( Context &, Pipeline * );
      virtual InternalThreadFn *
             createThread(InternalProcess *, Context &);
public:
      ReturnCommand( NodeId );
};
```

5.56 ReturnThread Class

The internal thread function object that implements the return command using the parameter as expanded using the context.

```
class ReturnThread : public InternalThreadFn
{
private:
      std::string option;
public:
      ReturnThread(InternalProcess *ip, Context & ctx )
```

```
             : InternalThreadFn(ip, ctx) {}
      virtual void exec();
};
```

5.57 ScriptXml Class

Responsible for building the executionModel from the XML script.

```
class ScriptXml : public VICI::Xml
{
private:
      void loadObjects( ExecutionModel *, Function *, std::vector< xmlNodePtr > & );
      void loadAction( ExecutionModel *, Function *, NodeId, xmlNodePtr );
      void loadVariable( ExecutionModel *, NodeId, xmlNodePtr );
      void loadFile( ExecutionModel *, NodeId, xmlNodePtr );
      void loadIPC( ExecutionModel *, Function *, NodeId, xmlNodePtr );
      void loadConnections( ExecutionModel *, std::vector< xmlNodePtr > & );
      NodeId getNodeId( xmlNodePtr );
      NodeId getNodeId( csr );

      std::map< NodeId, Source * > sources;
      std::map< NodeId, Sink * > sinks;
public:
      ScriptXml();

      ExecutionModel * loadModel( csr filename );
};
```

5.58 Semaphore Class

A specialisation of Variable used to implement the semaphore responsibilities of the interpreter.

```
class Semaphore : public Variable
{
private:
      sem_t s4;
public:
      Semaphore( NodeId, csr name );
      virtual ~Semaphore();
      bool test( int secs );
      void signal();
};
```

5.59 SemaphoreCommand Class

Built in command for testing and signalling the semaphore specified as a parameter. It implements the “wait” and “post” commands. It may block until the semaphore is signaled.

```
class SemaphoreCommand : public BuiltinCommand
{
protected:
      virtual Process * createProcess( Context &, Pipeline * );
      virtual InternalThreadFn *
            createThread(InternalProcess *, Context &);
public:
      SemaphoreCommand( NodeId );
};
```

5.60 SemaphoreThread Class

The thread function object that implements the command.

```
class SemaphoreThread : public InternalThreadFn
{
private:
      enum Action{ None, Wait, Post};

public:
      SemaphoreThread(InternalProcess *ip, Context & ctx );
      virtual void exec();
};
```

5.61 Signal Class

Represents a UNIX signal.

```
class Signal
{
protected:
      Command *sender;    // reference
      Command *receiver;  // reference
public:
      Signal( Command *sender, Command * rcver );
      virtual ~Signal(){}
      virtual void send( int funcId, int sig );
      virtual void print( std::ostream & );
};
```

5.62 SignalCommand Class

Sends a signal to another process.

```
class SignalCommand : public BuiltinCommand
{
protected:
      Signal *sig; // reference

      virtual Process * createProcess( Context &, Pipeline * );
      virtual InternalThreadFn *
            createThread(InternalProcess *, Context &);
public:
      SignalCommand( NodeId );
      bool setSignal( Signal * );
};
```

5.63 SignalThread Class

The thread function object responsible for sending the signal.

```
class SignalThread : public InternalThreadFn
{
private:
      std::string option;
      Signal *sig;
public:
      SignalThread(InternalProcess *ip, Context & ctx, Signal *s )
             : InternalThreadFn(ip, ctx), sig(s) {}
      virtual void exec();
};
```

5.64 Sink Class

An interface class that represents anything that can be written to. It can be the output of multiple channels.

```
class Sink
{
private:
      std::vector< Channel * > inputs; // references
public:
      virtual ~Sink(){}
      void connect( Channel * );
      virtual NodeId nodeId() = 0;
      virtual bool isProcess() = 0;
protected:
      std::vector< Channel * > & getInputs() { return inputs; }
};
```

5.65 Source Class

An interface class that represents anything that can be read from. It can be the input to multiple channels.

```
class Source
{
private:
      std::vector< Channel * > outputs;      // references
public:
      virtual ~Source(){}
      void connect( Channel * );
      virtual NodeId nodeId() = 0;
      virtual bool isProcess() = 0;
protected:
      std::vector< Channel * > & getOutputs() { return outputs; }
};
```

5.66 SwitchCommand Class

Compares its first argument with its remaining arguments returning the index of the first match, or zero otherwise. Subsequent commands are selected according to the exit code.

```
class SwitchCommand : public BuiltinCommand
{
      virtual Process * createProcess( Context &, Pipeline * );
      virtual InternalThreadFn *
             createThread(InternalProcess *, Context &);
public:
      SwitchCommand( NodeId );
};
```

5.67 SwitchThread Class

The thread function object that compares the parameters that have been expanded according to the context.

```
class SwitchThread : public InternalThreadFn
{
public:
      SwitchThread(InternalProcess *ip, Context & ctx )
             : InternalThreadFn(ip, ctx) {}
      virtual void exec();
};
```

5.68 Thread Class

A thread function object that manages a thread of control within the script.

```
class Thread : public ThreadFunction
{
private:
      Context *ctx;              // part owner
      Command * nextCommand;     // reference

      bool terminate;
      Pipeline *pipeline;              // owned
      ThreadId id;
public:
      Thread( ThreadFnOwner *, Context * );
      Thread( const Thread & ) = delete;
      ~Thread();
      virtual void starting( int job );
      virtual void stopping();
      ThreadId tid() { return id; }
      virtual void exec();
      virtual void kill();
      void setNext( Command *cmd ) { nextCommand = cmd; }

      void operator = ( const Thread & ) = delete;
};
```

5.69 ThreadFnOwner Class

An interface that is implemented by objects that own a thread function so that they can be notified when the thread function starts and stops.

```
class ThreadFnOwner
{
public:
      virtual ~ThreadFnOwner(){}
      virtual void starting( ThreadFunction *, int job ) = 0;
      virtual void stopping( ThreadFunction * ) = 0;
};
```

5.70 ThreadFunction Class

A base class for a function object that is run in a thread.

```
class ThreadFunction
{
protected:
      ThreadFnOwner *owner;            // reference
public:
      ThreadFunction( ThreadFnOwner * );
      virtual ~ThreadFunction();
      virtual void starting( int job );
      virtual void exec() = 0;
      virtual void kill() = 0;
      virtual void stopping();
};
```

5.71 ThreadMgr Class

Manage the worker threads.

```
class ThreadMgr : public ThreadFnOwner
{
private:
      ThreadMgr();
      ThreadMgr( const ThreadMgr & ) = delete;
      static ThreadMgr *me;
      static std::mutex instanceMx;

      MasterThread master;

      std::list< ThreadFunction * > threads; // owned
      int runningThreads;
      std::mutex finMx;

      double secs; // pause interval
      bool paused;
      std::mutex pauseMx;
      std::condition_variable pauseCV;
      ThreadId steppingThread;

      void installSignalHandler();    // we want to catch ^C events
      static void signalHandler( int sig );

public:
      static ThreadMgr & instance();
      ~ThreadMgr();

      // ThreadFnOwner interface
      virtual void starting( ThreadFunction *, int job );
      virtual void stopping( ThreadFunction * );

      void threadStart( ThreadFunction *, int job );
      void threadStop( ThreadFunction *);

      void connect( InterpreterImpl * );
      void terminate();   // stop any processing being done by the
threads

      void pause(ThreadId);     // used by threads to wait
      void pause(bool x) { paused = x; }
      void setInterval( double secs );
      void step(ThreadId);
      void setPosn( ThreadId, Command *);

      void operator = ( const ThreadMgr & ) = delete;
};
```

5.72 ThreadPool Class

Keep a pool of live threads so we don't have to worry about them terminating.

```
class ThreadPool
{
private:
      ThreadPool();
      ThreadPool( const ThreadPool & ) = delete;
      static ThreadPool *me;
      static std::mutex instanceMx;

      bool terminating;

      std::list< std::thread * > threads;    // owned
      unsigned int runningThreads;
      int jobNumber;

      static void runner();
      ThreadFunction *work;
      std::mutex workMx;
      std::condition_variable workCV;
public:
      static ThreadPool & instance();
      ~ThreadPool();
      void exec( ThreadFunction * );// runs the function in a thread
};
```

5.73 VarEvent Class

An event that notifies that a variable has changed its value.

```
class VarEvent : public Event
{
private:
      std::string name;
      std::string value;
public:
      VarEvent( csr nm, csr val );
      virtual void action( EventHandler *eh );
      virtual void print( std::ostream &f );
};
```

5.74 Variable Class

A specialization of DataStore that holds a variable. Access is controlled via a mutex so that threads can reliably update and access the value. Changes of value are reported to the Variable Manager.

```
class Variable : public DataStore, public Source, public Sink
{
private:
      std::mutex accessMx;
public:
      Variable( NodeId, csr name );
      void value( csr );
      virtual csr value();
      virtual NodeId nodeId() { return id; }
      virtual bool isProcess() { return false; }
};
```

5.75 VariableMgr Class

A singleton class used to manage the collection of data stores. It allows access by name and includes a method that expands "$name" to its corresponding value. A valueEvent method notifies the EventMgr of a value change.

```
class VariableMgr
{
private:
      VariableMgr();
      static VariableMgr *me;

      std::map< std::string, DataStore * > variables;   // owned

public:
      static VariableMgr & instance();

      void valueEvent( csr name, csr val );

      DataStore *& operator [] ( csr name );
      bool exists( csr );
      void clear();
      void print( std::ostream & );

      std::string expand( csr, Context & );
      void expandGlobs( Context &, ArgList & );
};
```

5.76 VarReadCommand Class

Used to create a process for reading from a data store. Unlike other commands this one does not appear on the flow chart.

```
class VarReadCommand : public BuiltinCommand
{
private:
	DataStore * data;   // reference
	virtual Process * createProcess( Context &, Pipeline * );
	virtual InternalThreadFn *createThread(InternalProcess *,
Context &);

public:
	VarReadCommand( DataStore * );
};
```

5.77 VarReadThread Class

An InternalThreadFn that reads a value from a variable and writes that value to its stdout stream.

```
class VarReadThread : public InternalThreadFn
{
private:
	DataStore * data;   // reference
public:
	VarReadThread(InternalProcess *ip, Context & ctx, DataStore *ds )
		: InternalThreadFn(ip, ctx), data(ds) {}
	virtual void exec();
};
```

5.78 VarWriteCommand Class

Used to create a process for writing to a variable. It does not appear on the flow chart.

```
class VarWriteCommand : public BuiltinCommand
{
private:
	Variable *data;
	virtual Process * createProcess( Context &, Pipeline * );
	virtual InternalThreadFn *
		createThread(InternalProcess *, Context &);
public:
	VarWriteCommand( Variable * );
};
```

5.79 VarWriteThread Class

An InternalThreadFn that reads from stdin and puts the results in a variable.

```
class VarWriteThread : public InternalThreadFn
{
private:
	Variable *data;
public:
	VarWriteThread(InternalProcess *ip, Context & ctx, Variable *v)
		: InternalThreadFn(ip, ctx), data(v) {}
```

```
      virtual void exec();
};
```

6 Design Review

This section documents the findings of the design review for the libvici component.

The following responsibilities were identified and assigned to the design review task:

T41.1: Conduct design reviews focussing on finding unnecessary components.

T44.2: Conduct design reviews focussing on ease of understanding.

T47.2: Review design for unnecessary interactions between modules.

The review was conducted by reviewing the design patterns listed on Wikipedia and attempting to identify those patterns in the prototype.

6.1 Abstract Factory Pattern

The Command class is an abstract factory for creating objects derived from the Process class. The BuiltinCommand is an abstract factory for creating objects derived from the InternalThreadFn class.

In both cases the concrete factory function is defined in the specific command class.

The implementation is in accordance with the standard pattern.

6.2 Asynchronous Method Invocation Pattern

In this pattern a function is called but it returns immediately, rather than after the function has completed its processing. The caller is notified when the function completes. This pattern is used by the Process objects when they are called to run. The owning Pipeline gets the notification.

The implementation is probably in accordance with the standard pattern.

6.3 Builder Pattern

The ScriptXml class implements the builder pattern. It separates the construction of the execution model from its representation.

Our implementation does not separate an abstract builder from a concrete builder. This might be useful if scripts were to be stored in different formats, but is currently unnecessary.

6.4 Command Pattern

The Event class implements the command pattern. It is used to pass a command between threads.

The implementation is in accordance with the standard pattern.

6.5 Facade Pattern

The Interpreter class provides a facade for libvici.

The implementation is in accordance with the standard pattern.

6.6 Factory Method Pattern

The InterpreterFactory and InterpreterFactoryImpl implement the Factory Method pattern. The callers of libvici use the InterpreterFactory interface to get a pointer to the InterpreterImpl which implements the Interpreter interface.

The implementation is in accordance with the standard pattern.

6.7 Monitor Pattern

This is implemented using the STL std::condition_variable.

- The EventMgr uses the pattern to wait for Event object to be queued.
- The DisplayThreadFn uses the pattern to wait for acknowledgements that the waiting data has been read by the InterpreterClient.
- The Pipeline, ForEachThread and FuncCallThread use the pattern to wait for their sub-processes to complete.
- The ThreadPool uses the pattern to reliably transfer a work thread object to a running thread.
- The ThreadMgr uses the pattern to implement manual stepping and similar debugging actions.

The implementation is in accordance with the standard pattern.

6.8 Interpreter Pattern

The Expression and Parameter classes implement the interpreter pattern. However, rather than have a separate class for each symbol type they use a single class with methods for each symbol or production.

The implementation is not in accordance with the standard pattern but is acceptable for this application.

6.9 Lazy Initialisation Pattern

The ThreadPool uses the Lazy Initialisation pattern when creating threads. additional threads are only created when there is not enough free to satisfy the request for a thread.

6.10 Lock Pattern

This pattern is used extensively in libvici to control access to objects that are accessible from multiple threads.

6.11 Memento Pattern

This pattern is not currently used in libvici, however it might be a useful capability to allow the user to try a change and roll it back.

This might be a capability for a later iteration.

6.12 Object Pool Pattern

This pattern is not used in libvici, but there would seem to be some advantage if we could pool Pipeline objects so that they did not have to be recreated on each iteration of a loop.

This can be left to a subsequent iteration of the development.

6.13 Observer Pattern

The observer pattern is used by the EventMgr as it passes the Event objects to the object that implements the EventHandler interface. It is used by the ThreadFunction to notify objects that implement the ThreadFnOwner interface of changes in status of the the ThreadFunction object. It is used by Process to notify objects that implement ParentOfProcess. It is used by the Interpreter to notify objects that implement the InterpreterClient interface.

These implementations are in accordance with the standard pattern.

6.14 Prototype Pattern

The Context object needs to be cloned when being passed to a function or other sub-process as the copy needs to be initialised with current values.

6.15 Resource Acquisition Is Initialization

This pattern is used for getting mutex locks as it guarantees that the lock will be released.

The VariableMgr::expandGlobs should use this technique for handling directories so that there is no chance of ending up in the wrong directory.

6.16 Singleton Pattern

The ChildProcessMgr, EventMgr, ThreadMgr, ThreadPool and VariableMgr classes all use a singleton design pattern which is guaranteed by C++11 to be thread safe since only one thread gets to initialise a static variable.

```
Singleton & Singleton::instance()
{
	static Singleton s;
	return s;
}
```

Appendix A