Use the XTA from the command line

IN THIS DOCUMENT

The XTA tool can be used interactively on the command-line or the console in xTIMEcomposer Studio.

1 Frequently used commands

This section summarizes a number of frequently used commands that can be run from the command line.

1.1 Loading a binary

To load a binary type:

```
load <FILE NAME>
```

1.2 Routes

A *route* is a timing-critical section of code. It consists of the set of all paths through which control can flow between two points in a program (*endpoints*). A route can be created by timing a function, timing a loop or by timing between endpoints.

1.3 Endpoints

An *endpoint* is any source line that, during the compilation process, must be preserved, and its order with respect to other endpoints must be maintained.

To show a list of all endpoints type:

```
list allendpoints
```

If specifying a route with respect to assembly code then any valid label/program counter (PC) can be used as an endpoint. However, program counters are classed

as non-portable endpoints as they are likely to change between compilations and their use in scripts is therefore discouraged.

1.4 Adding endpoints to source

Source lines can be labeled with endpoint pragmas to ensure that the endpoints are portable. For example, Figure 1 shows a function that has been annotated with endpoint pragmas called `start` and `stop`.

```
int g(in port p) {
    int x, y;

    # pragma xta endpoint " start "
    p :> x;

    # pragma xta endpoint " stop "
    p :> y;

    return (y - x);
}
```

Figure 1: Putting an endpoint pragma into the souce

To show a list of endpoints type:

```
list endpoints
```

1.5 Timing between endpoints

To time between endpoints type:

```
analyze endpoints <from ENDPOINT> <to ENDPOINT>
```

The XTA does not time code across multiple xCORE tiles so both endpoints must be on the same tile.

One analysis can result in multiple routes being generated.

1.6 Timing functions

Type the function name on the console:

```
analyze function <FUNCTION>
```

This will create a route which describes the set of all possible paths from the function entry point to all the function return points.

1.7 Timing loops

To time a loop type:

```
analyze loop <ANY>
```

This creates a route that describes all possible paths through the loop. It it effectively a shortcut for timing between endpoints where the start and stop endpoint is the same, the point is within a loop and an exclusion has been placed such that everything outside the loop is excluded.

One analysis can result in multiple routes being generated.

1.8 Setting timing requirements

To define the timing requirements for a route type:

```
set required <route id> <value> <MODE>
```

The supported timing modes are defined in X.

The route IDs can be found by typing:

```
print summary
```

Alternatively, the - character can be used on the command-line or in a script to refer to the last route analyzed.

2 Viewing results

2.1 Route IDs

All analyzed routes are given a unique route ID. However, when referring to routes in a script, using the route ID may not always result in portable or robust scripts. In many cases, the only route that needs to be referenced is the one that was last analyzed. This can be achieved by using the '-' character as the route ID. If the last command created multiple routes then the '-' character refers to all of the routes created.

2.2 Node IDs

Within a single route, all nodes are assigned a unique ID number. This is required as input for some of the console commands. The '-' character can be used in this context to refer to the top level node of the route.

2.3 Summary

To show a list of all routes type:

```
print summary
```

Details for a specific route are shown using the command:

```
print routeinfo <route id>
```

2.3.1 Violation

When a timing requirement has been set for a route and the route takes more time to execute than required, the time difference is called a *violation*. This value specifies how much faster the route needs to be executed in order to meet the timing requirement.

2.3.2 Slack

When a timing requirement has been set for a route and the route takes less time to execute than required, the time difference is called *slack*. This value specifies how much slower the route could be executed and still meet the timing requirement.

2.4 Structure

To display the the structure of a route in xTIMEcomposer Studio type:

```
print structure <route id>
```

The structure used by the XTA is described in §4.2.

2.5 Source code annotation

To display the source code which is executed by a route type:

```
print src <route id>
```

If only a part of a route should be used then the node ID can be specified:

```
print src <route id> <node id>
```

2.6 Instruction traces

To help developers understand the execution flow of a route, the XTA can create representative instruction traces. Type:

```
print trace <route id>
```

As a result of loops being unrolled when tracing, it is possible for the traces to get very large. The trace operation can be cancelled at any time by pressing *CTRL+C* in the command-line tool.

A trace can be redirected to a file by typing:

```
print trace <route id> > <file>
```

By default, the trace for worst-case path is printed. This can be changed to print the best-case path instead by typing:

```
config case best
```

2.7 Fetch no-ops

The xCORE device may need to pause at certain times while more instructions are fetched from memory. This results in the issue of fetch no-op instructions. These are shown in the traces as `FNOP` at the points they will happen on the hardware.

In xTIMEcomposer Studio they are inserted into the disassembly at the points they occur.

2.8 Scaling Results

By default, the XTA scales all timing results. This means that the appropriate unit (ms, us, ns) will be used to print time values. This can be changed so that all times are printed in ns by typing:

```
config scale false
```

2.9 Unknowns

The XTA may not always be able to determine the exact timing of a section of code if it is unable to determine loop iteration counts or the execution time of instructions. These unknown conditions can be displayed on the console by typing:

```
list unknowns <route id>
```

§3 describes how to address these warnings.

3 Refining timing results

There are cases where the XTA is unable to fully determine the timing of a section of code, due to, for example, not being able to determine a loop count. This can be addressed by adding *defines*. Defines can be added in two ways, to a *global list*, or to a *route-specific list*. Those added to the *global list* get applied to every route when upon creation.

The use of the global list can result in more concise scripts. However, It is important to be careful with defines added to the *global list* since they are ignored if they fail to get applied to a route. This allows a full set of defines to be created before any routes, but does mean that errors in these defines might be missed. Route specific defines (added post route creation) will always flag an error if there is one.

3.1 Exclusions

Not all paths of execution in a route may be timing-critical. The route may contain cases to handle errors where the timing of the code is not important. These paths can be ignored in the timing script by adding exclusions. *Exclusions* tell the XTA to ignore all paths which pass through that code point. Exclusions can be added to the global list or applied to a specific route.

To set an exclusion on an existing route type:

```
set exclusion <route id> <ANY>
```

To add an exclusion to the list of exclusions to be taken into account during route creation type:

```
add exclusion <ANY>
```

To list the global list of exclusions type:

```
list exclusions
```

To remove an exclusion from the global list type:

```
remove exclusion <ANY|*>
```

For example, consider the code in Figure 2.

```
int calculate ( int a, int b) {
    if ( willOverflow (a, b) {
        # pragma xta label " overflow "
        return processOverflow ();
    }
    return a + b;
}
```

Figure 2: Excluding an invalid path

To time the `calculate` function ignoring the error case:

- Using route-specific defines:
  - `analyze function calculate`
  - `set exclusion - overflow`

- Using global defines:
  - `add exclusion overflow`
  - `analyze function calculate`

Although functionally equivalent, exclusion via the global defines mechanism can result in faster, and more memory efficient, route creation. This is because the global exclusions can be taken into account during route creation, so the search space can be reduced. For post route creation exclusions, the complete route is created before any pruning occurs.

3.2 Loop Iterations

Loop iteration counts can be unknown. Whenever possible, the compiler tells the XTA about loop iteration counts. However, some loop counts are not known statically. In these cases developers must specify worst-case values.

The compiler does not emit any loop iteration counts unless optimizations have been enabled (-O1 or greater).

Some loops are self loops (loops whose body is the same as the header) and therefore have a minimum iteration count of 1.

To set loop iterations on an existing route type:

```
set loop <route id> <ANY> <iterations>
```

To add an iteration count to the list of iteration counts to be used during route creation type:

```
add loop <ANY> <iterations>
```

To list the current global loop iteration counts type:

```
list loops
```

To remove a loop iteration count from the global list type:

```
remove loop <ANY|*>
```

For example, consider the code in Figure 3.

To time the `test` function:

- Using route-specific defines:
  - `analyze function test`
  - `set loop - delay_loop 10`
- Using global defines:

```
void delay ( int j) {
    for ( unsigned int i = 0; i < j; ++i) {
        # pragma xta label " delay_loop "
        delay_us (1);
    }
}

int test () {
    delay (10);
}
```

Figure 3: Setting loop iterations.

- `add loop delay_loop 10`
- `analyze function test`

3.3 Loop path iterations

A loop may contain multiple paths through it. When a loop iteration count has been set the tools assumes that all iterations will take the worst-case path of execution through the loop. This is not always the case, and a more realistic worst-case can be established by specifying the number of iterations on individual paths through the loop.

To set loop path iterations on an existing route type:

```
set looppath <route id> <ANY> <iterations>
```

To add a loop path count to the list of loop path counts to be used during route creation type:

```
add looppath <ANY> <iterations>
```

To display the current list of global loop path counts type:

```
list looppaths
```

To remove a loop path count from the global list type:

```
remove looppath <ANY|*>
```

There are some rules that need to be followed when setting loop path iterations:

- In a nested loop, the outer loop iterations need to be set first.
- The loop path iterations set must be less than or equal to the loop iterations set on the enclosing loop.

- If the loop path iterations set are less than that of the enclosing loop, then there must exist another path within the loop without its iterations set to which the remaining iterations can be allocated.

For example, consider the code in Figure 4:

```
void f( int j) {
    for ( unsigned int i = 0; i < j; ++i) {
        # pragma xta label " f_loop "
        if ((i & 1) == 0) {
            # pragma xta label " f_if "
            g ();
        }
    }
}

int test () {
    f (10);
}
```

Figure 4: Setting loop path iterations.

To time the `test` function:

- Using route-specific defines:
  - `analyze function test`
  - `set loop - f_loop 10`
  - `set looppath - f_if 5`
- Using global defines:
  - `add loop f_loop 10`
  - `add looppath f_if 5`
  - `analyze function test`

3.4 Loop scope

By default, the XTA assumes that the iterations for loops are *relative*—the iterations for an inner loop will be multiplied by the iterations of enclosing loops. However this is not sufficient to describe all loop structures. If this assumption is not correct a loop count can be set to *absolute*. The iteration count set on an absolute loop is not multiplied up by the iterations set on enclosing loops.

To set loop scope on an existing route type:

```
set loopscope <route id> <ANY> <absolute|relative>
```

To add a loop scope to the list of loop scopes to be used during route creation type:

```
add loopscope <ANY> <absolute|relative>
```

To display the current list of global loop scopes, type:

```
list loopscopes
```

To remove a loop scope from the global list, type:

```
remove loopscope <ANY|*>
```

For example, consider the code in Figure 5

```
void f( int l) {
    for ( unsigned int i = 0; i < l; ++i) {
        # pragma xta label " outer_loop "
        for ( unsigned int j = 0; j < i; ++j) {
            # pragma xta label " inner_loop "
            g ();
        }
    }
}

void test () {
    f (10);
}
```

Figure 5: Setting loop scope.

To time the `test` function:

- Using route-specific defines:
  - `analyze function test`
  - `set loop - outer_loop 10`
  - `set loop - inner_loop 45`
  - `set loopscope - inner_loop absolute`
- Using global defines:
  - `add loop outer_loop 10`
  - `add loop inner_loop 45`
  - `add loopscope inner_loop absolute`
  - `analyze function test`

3.5 Instruction times

Some instructions can pause the processor. By default, the XTA reports timing assuming that no instructions pause, but flags them as warnings. Developers must specify what the worst-case execution time of instructions are.

To set an instruction time in an existing route, type:

```
set instructiontime <route id> <ENDPOINT> <value> <MODE>
```

To add an instruction time to the list of instruction times to be used during route creation, type:

```
add instructiontime <ENDPOINT> <value> <MODE>
```

To display the current list of global instruction times, type:

```
list instructiontimes
```

To remove an instruction time from the global list, type:

```
remove instructiontime <ANY|*>
```

For example, consider the code in Figure 6.

Figure 6: Setting an instruction time.

```
void f( port p) {
    # pragma endpoint " instr "
    p :> value ;
}
```

To time the `f` function:

- Using route-specific defines:
  - `analyze function f`
  - `set instructiontime - instr 100.0 ns`
- Using global defines:
  - `add instructiontime instr 100.0 ns`
  - `analyze function f`

3.6 Function times

In some cases it is necessary to define the time it takes to execute an entire function. The XTA supports defining a function time. Once a function time is

defined, all the unknowns within it are ignored and any routes which span this function will use the defined time instead of calculating it.

To set a function time on an existing route, type:

```
set functiontime <route id> <FUNCTION> <value> <MODE>
```

To add a function time to the list of function times to be used during route creation, type:

```
add functiontime <FUNCTION> <value> <MODE>
```

To display the current list of global function times, type:

```
list functiontimes
```

To remove an function time from the global list, type:

```
remove functiontime <FUNCTION|*>
```

For example, consider the code in Figure 7.

```
void delayOneSecond () {
    g ();
    }

void test () {
    delayOneSecond ();
}
```

Figure 7: Setting a function time.

To time the `test` function:

- Using route-specific defines:
  - `analyze function test`
  - `set functiontime - delayOneSecond 1000.0 ms`
- Using global defines:
  - `add functiontime delayOneSecond 1000.0 ms`
  - `analyze function test`

3.7 Path times

In some cases it is necessary to define the time it takes to execute a particular section of code. The XTA supports defining a path time for this case. Once a path

time is defined all the unknowns within it are ignored, and any routes which span this section of code will use the defined time instead of calculating it.

To set a path time on an existing route, type:

```
set pathtime <route id> <from ENDPOINT> <to ENDPOINT> <value> <MODE>
```

To add a path time to the list of path times to be used during route creation, type:

```
add pathtime <from ENDPOINT> <to ENDPOINT> <value> <MODE>
```

To display the current list of global path times, type:

```
list pathtimes
```

To remove an path time from the global list, type:

```
remove pathtime <from ENDPOINT|*> <to ENDPOINT|*>
```

For example, consider the code in Figure 8.

```
int f() {
    int time ;
    timer t;
    # pragma xta endpoint " start "
    t :> time ;
    # pragma xta endpoint " stop "
    t when timerafter ( time + 100) :> time ;
}

void test () {
    f ();
}
```

Figure 8: Setting a path time.

To time the `test` function:

- Using route-specific defines:
  - `analyze function test`
  - `set pathtime - start stop 1000.0 ns`
- Using global defines:
  - `add pathtime start stop 1000.0 ns`
  - `analyze function test`

3.8 Active tiles

By default the XTA finds routes on all tiles within a program. However, it is possible to restrict the XTA to work only on a subset of the tiles in the program. The set of tiles all commands apply to is called the *active* tiles.

To select which tiles are active, type:

- `add tile <tile id>`
- `remove tile <tile id|*>`
- `list tiles`

3.9 Node frequency

An xCORE device consists of a number of nodes, each one composed of a number of xCORE tiles. The frequency at which a node runs is defined in the binary and the XTA reads this and configures the node frequencies when it loads the binary. It is possible to experiment to determine what will happen at different frequencies if desired.

To change the frequency for the node, type:

```
config freq <node id> <tile frequency>
```

3.10 Number Of logical cores

The maximum number of logical cores run on a tile is known at compile time and the XTA extracts this information from the binary for each tile. It is possible to experiment to determine what will happen if running with a different number of cores if desired.

To change the number of cores for the node/tile, type:

```
config cores <tile id> <num cores>
```

4 Program structure

Programs are written in multiple source files, each containing functions. Each function will contain sequences of statements, loops (e.g. `for` / `while` / `do`), conditionals (e.g. `if` / `switch`) and function calls.

4.1 Compiling for the XTA

The compiler outputs information which allows the XTA to make associations between source and instructions. This information is on by default but can be disabled by adding the following flag to the compiler options:

```
-fno-xta-info
```

The compiler also supports adding debug information without affecting optimizations. Debug information is not required for the XTA to analyze code, but the mapping between instructions and source code is not available without the debug information. In order to add debug information compile with:

```
-g
```

4.2 Structural nodes

The compiler tools create a binary file with one program per xCORE tile. The XTA uses the binary file to produce accurate timing results.

When a route is created, the XTA analyzes the binary to create a structure which closely represents the high-level program structure. It decomposes the program into structural nodes which can be displayed as a tree.

The worst and best case time is then calculated for each of the structural nodes. The way this is calculated depends on the type of structural node. The worst and best case times for the overall route is built up from the worst and best case times of the sub nodes.

The structural nodes can be of the following types:

- Instruction: the most basic building block of the program is the instruction.
- Block: a list of instruction nodes with no conditional branching which is therefore executed in sequence. The worst/best case time for a block is the sum of its component instructions.
- Sequence: a list of structural nodes which are executed in order. The worst/best case time for a sequence is the sum of the worst/best case times of its sub nodes.
- Conditional: a set of structural nodes out of which at most one node is executed. If this is within a loop then on each iteration a different node might be chosen. In some cases the entire conditional is optional. In those cases the best case time is for none of the options to be taken. The worst/best case time for a conditional is determined by the worst/best case time of each of its sub nodes.
- Loop: consists of a header and a body (both of which are structural nodes). The header corresponds to the conditional test part of the loop, and the body corresponds to the code that is executed if the loop is taken. This roughly corresponds to high level code structures such as `while` or `for` loops.

  The body is executed once per iteration. The header always executes once more than the number of iterations. The worst/best case times for a loop is the worst/best case time of its header multiplied by (number of iterations + 1) plus the worst/best case time of the body multiplied by the number of iterations.

- Self-loop: a loop where the header and body are the same. It is therefore considered to have a minimum loop count of 1. This roughly corresponds to high level code structures such as `do` loops. The worst/best case time for a self-loop is determined by the worst/best case time of its body multiplied up buy the number of iterations.

- Function: is the high-level construct of the function and consists of a list of other structural nodes. The worst/best case time for a function is calculated in the same way as that of a sequence.

4.3 Identifying nodes: code references

A `code reference` is the way to specify a particular location in an application. A code reference is made up of a base and an optional backtrail. The base consists of a `reference type` and the backtrail consists of a comma separated list of `reference types`.

There are a number of different reference types, all of which map to one or more instruction program counters (PCs). This will usually be one PC, but can be more than one due to compiler optimizations or because the user has explicitly named multiple instructions with the same reference. Compiler optimizations such as inlining or unrolling will result in the same reference mapping to multiple PCs.

The different reference types are detailed below. The commands to list the instances of them for the currently loaded executable in the console are detailed with each type.

- **Source file-line** references are valid for source lines which the compiler has defined as belonging to a source-level basic block. The valid lines can be listed in the console using:

  ```
  list allsrclabels
  ```

- **Source labels** are added to source code using the `#pragma xta label`. To list the source labels in the console, type:

  ```
  list srclabels
  ```

- **Call file-line** references are valid for source lines which map to function calls. To list the valid source lines in the console, type:

  ```
  list allcalls
  ```

- **Call labels** are added to source code using the `#pragma xta call`. To list the source labels in the console, type:

  ```
  list calls
  ```

- **Endpoint file-line** references are available for source lines which map to a valid *endpoint*. To list the endpoints in the console type:

  ```
  list allendpoints
  ```

- **Endpoint labels** are added to the source using `#pragma xta endpoint`. They must be on the line before an input/output operation. To list the labeled endpoints in the console, type:

  ```
  list endpoints
  ```

- **Labels** are arbitrary text strings referring to any source or assembly label. To list the labels in the console, type:

  ```
  list labels
  ```

  Labels in assembly must be within an executable section.

- **Functions** are the functions contained within the binary. To list the labels in the console, type:

  ```
  list functions
  ```

  Functions in assembler must be labeled as functions with the `.type` directive to be correctly detected by the XTA (see xTIMEcomposer Studio User Guide). They must also be within an executable section.

- **Program counters** (PC) are the lowest-level reference, giving a hexadecimal program counter value starting with `0x`. They must map to the PC of an instruction within the executable section of the program.

4.4 Reference Classes

Particular console commands only work on particular types of references. The sets of reference types that are defined for a particular command are know as reference classes.

- **ENDPOINT**: A reference that can be used for timing. This means any reference in assembler (PC/label) and only source references which map to lines which can be reliably used for timing. Compiler optimizations cannot remove them or re-order them with respect to each other. In XC code these correspond to source lines with I/O operations. The following console command lists the types available in the class:

  ```
  help ENDPOINT
  ```

- **CALL**: References that map to function calls. These are used in *back trails* to identify unique instances of a code reference. The following console command lists the types available in the class:

  ```
  help CALL
  ```

- **FUNCTION**: References that map to functions. The following console command lists the types available in the class:

  ```
  help FUNCTION
  ```

- **LABEL**: The following console command lists the types available in the class:

  ```
  help LABEL
  ```

- **PC**: The following console command lists the types available in the class:

```
help PC
```

It is possible to have a code reference which could map to multiple types. For example there could be an endpoint which has been given the same name as a function in the program. The way a reference in a backtrail is matched can depend upon the type of the reference. To resolve this potential ambiguity, it is possible to force the code reference to a certain type by prefixing with its type.

4.5 Back trails

A code reference's base may occur multiple times within a program. For example, a function can be called from multiple places. The *back trail* for a reference is a way of restricting a reference to specific instances. Consider the example file shown in Figure 9.

```
1  void delay_n_seconds ( int j) {
2      for ( unsigned int i = 0; i < j; ++i) {
3          # pragma xta label " delay_loop "
4          delay_1_second ();
5      }
6  }
7
8  int test () {
9      # pragma xta call " delay_1 "
10     delay_n_seconds (10);
11     # pragma xta call " delay_2 "
12     delay_n_seconds (20);
13     return 0;
14 }
```

Figure 9: Using backtrails.

The following commands could be used to time the `test` function:

- `analyze function test`
- `set loop - delay_loop 10`

That would have the effect of setting the number of loop iterations for the loop in both instances of the `delay_n_seconds` to 10. However, as the number of iterations are passed as a parameter to delay_n_seconds, the value is different for each call.

To time test correctly the loop iterations for each instance needs to be specified differently. This can be achieved by the use of the call references and backtrails. For example:

- `analyze function test`
- `set loop - delay_1,delay_loop 10`
- `set loop - delay_2,delay_loop 20`

This tells the tool to set `delay_loop` to 10 iterations when called from `delay_1`, and to 20 iterations when called from `delay_2`. The references used in the above case are composed of a base reference of type source label, and a backtrial or size one, of type call label. The above can also be achieved using the file-line equivalents. For example:

- `analyze function test`
- `set loop - source.xc:10,source.xc:3 10`
- `set loop - source.xc:12,source.xc:3 20`

However, this would not result in a portable and robust script implementation, so using file-line references in this way from a script is not encouraged.

When the compiler inlines some code (for example the `delay_n_seconds` function above) then some references will no longer be valid. In this case the following reference would not exist because the call no longer exists:

```
source.xc:10,source.xc:3
```

However, if the call has been labeled with a call label, the compiler ensures that the reference is still valid even if the code is inlined. So, in the above case, the following reference will still be valid;

```
delay_1,delay_loop
```

4.6 Scope of references

References can have either *global* or *local* scope. Globally scoped references are those which apply to (or get resolved on) the global tree. The global tree is the notional structural representation of the whole program, prior to any route analysis taking place. Locally scoped references are those which apply to (or get resolved on) a user created route tree. Whether a particular reference is globally of locally scoped depends on the command being executed. The following commands used globally scoped references:

- `analyze path`
- `analyze function`
- `analyze loop`
- `add exclusion`
- `add branch`

The following commands used locally scoped references:

- `set/add loop`
- `set/add looppath`

- `set/add loopscope`
- `set/add instructiontime`
- `set/add pathtime`
- `set/add functiontime`

In general, globally scoped references can lead to multiple route creation.

5 Automating the process

The XTA can be automated to ensure that new versions of an application meet timing requirements using a script.

5.1 Writing a script

The script file is a sequence of XTA console commands. Each one on a separate line. Any line starting with the # symbol is considered a comment.

Developers must insert `pragmas` into the source code where required to make the script portable. If the script creation process modifies the source (e.g. by inserting pragmas) the relevant binary must be rebuilt before the script can be successfully executed.

It is recommended not to put a `load` or `exit` command in the script. These commands should be done at the time of calling the script.

XTA scripts must use the `.xta` extension in order to be used by the compiler and understood correctly by xTIMEcomposer Studio.

5.2 Running a script

Scripts can be run in different ways, either in xTIMEcomposer Studio or on the command-line.

- **During compilation**: On the command line the `.xta` scripts must be passed to the compiler manually. By default, timing failures are treated as warnings and syntax errors in the script as errors.

  To treat timing failures as errors, add the following to the compiler arguments:

  ```
  -Werror=timing
  ```

  To treat script syntax errors as warnings, add the following to the compiler arguments:

  ```
  -Wno-error=timing-syntax
  ```

- **Batch mode**: In batch mode the XTA takes command-line arguments and interprets them as XTA commands. For example, to run an XTA script (`script.xta`) on a binary (`test.xe`) use:

  ```
  xta -load test.xe -source script.xta -exit
  ```

Note: the '-' character is used as a separator between commands.

5.3 Embedding commands into source

The XTA can embed commands into source code using a `command` pragma. For example,

```
#pragma xta command "print summary"
```

All commands embedded into the source are run every time the binary is loaded into the XTA. Commands are executed in the order they occur in the file, but the order between commands in different source files is not defined.

Pragmas are only supported in XC code.

6 Scripting XTA via the Jython interface

The XTA supports scripts written using the Jython language (an implementation of Python running on the Java virtual machine). XTA Jython scripts must have the extension `.py`. They can be executed in the same way as command based XTA scripts. From within Jython, XTA features are made available though the globally accessible *xta* object. See Figure 10 for an example script. This scripts loads the binary `test.xe` into the XTA and analyzes the function `functionName`. It then sets a loop count on each of the resulting routes and finally, prints the best and worst case times for each.

```
import sys
import java

try :
    xta . load (" test .xe");

except java . lang . Exception , e:
    print e. getMessage ()

try :
    ids = xta . analyzeFunction (" functionName ");

    for id in ids :
        xta . setLoop (id , " loopReference ", 10)

    for id in ids :
        print xta . getRouteDescription (id),
        print xta . getWorstCase (id , "ns"),
        print xta . getBestCase (id , "ns")

except java . lang . Exception , e:
    print e. getMessage ()
```

Figure 10: Example of an XTA Jython script.