amoco Documentation

Release 3.0 rc-1

bdcht

May 26, 2020

User Documentation

Amoco is a python (>=3.7) package dedicated to the static symbolic analysis of binary programs.

It features:

- a generic framework for decoding instructions, developed to reduce the time needed to implement support for new architectures. For example the decoder for most IA32 instructions (general purpose) fits in less than 800 lines of Python. The full SPARCv8 RISC decoder (or the ARM THUMB-1 set as well) fits in less than 350 lines. The ARMv8 instruction set decoder is less than 650 lines.
- a **symbolic** algebra module which allows to describe the semantics of every instructions and compute a functional representation of instruction blocks.
- a generic execution model wich provides an abstract memory model to deal with concrete or symbolic values transparently, and other system-dependent features.
- various classes implementing usual disassembly techniques like linear sweep, recursive traversal, or more elaborated techniques like path-predicate which relies on SAT/SMT solvers to proceed with discovering the control flow graph or even to implement techniques like DARE (Directed Automated Random Exploration).
- various generic *helpers* and arch-dependent pretty printers to allow custom look-and-feel configurations (think AT&T vs. Intel syntax, absolute vs. relative offsets, decimal or hex immediates, etc).
- a persistent database facility that allows to compare discovered graphs with other previously analysed piece of codes.
- a graphical user interface that can either be run as a standalone client or as an IDA plugin.

CHAPTER 1

Installation

Amoco is a pure python package which depends on the following packages:

- grandalf used for building, walking and rendering Control Flow Graphs
- crysp used by the generic intruction decoder (`arch.core`)
- traitlets used for managing the configuration
- pyparsing used for parsing instruction specifications

Recommended *optional* packages are:

- z3 used to simplify expressions and solve constraints
- pygments used for pretty printing of assembly code and expressions
- ccrawl used to define and import data structures

Some optional features related to UI and persistence require:

- click used to define amoco command-line app
- blessed used for terminal based debugger frontend
- tqdm used for terminal based debugger frontend
- ply for parsing *GNU as* files
- sqlalchemy for persistence of amoco objects in a database
- pyside2 for the Qt-based graphical user interface

Installation is straightforward for most packages using pip.

The z3 SMT solver is highly recommended (do `pip install z3-solver`). The pygments package is also recommended for pretty printing, and sqlalchemy is needed if you want to store analysis results and objects.

If you want to use the graphical interface you will need **all** packages.

CHAPTER 2

Getting started

This part of the documentation is intended for reversers or pentesters who want to get valuable informations about a binary blob without writting complicated python scripts. We give here a quick introduction to amoco without covering any of the implementation details.

Content

2.1 Loading binary data

The recommended way to load binary data is to use the *load_program* function, providing an input filename or a bytestring. For example, from directory `amoco/tests`, do:

```
In [1]: import amoco
In [2]: p = amoco.load_program(u'samples/x86/flow.elf')
In [3]: print(p)
<Task amoco.system.linux32.x86 'samples/x86/flow.elf'>

In [4]: print(p.bin.Ehdr)
[Ehdr]
e_ident    :[IDENT]
            ELFMAG0      :127
            ELFMAG       :b'ELF'
            EI_CLASS     :ELFCLASS32
            EI_DATA      :ELFDATA2LSB
```

(continues on next page)

(continued from previous page)

```
            EI_VERSION    :1
            EI_OSABI      :ELFOSABI_NONE
            EI_ABIVERSION:0
            unused        :(0, 0, 0, 0, 0, 0, 0)
e_type      :ET_EXEC
e_machine   :EM_386
e_version   :EV_CURRENT
e_entry     :0x8048380
e_phoff     :52
e_shoff     :4416
e_flags     :0x0
e_ehsize    :52
e_phentsize:32
e_phnum     :9
e_shentsize:40
e_shnum     :30
e_shstrndx :27
```

If you have the **click_** python package installed, you can also rely on the `amoco` shell command and simply do:

```
% amoco load samples/x86/flow.elf
```

If the binary data uses a registered executable format (currently *system.pe*, *system.elf*, *system.macho* or an HEX/SREC format in *system.utils*) and targets a supported plateform (see *system* and *arch* packages), the returned object is an *abstraction* of the memory mapped program:

```
In [5]: print(p.state)
eip <- { | [0:32]->0x8048380 | }
ebp <- { | [0:32]->0x0 | }
eax <- { | [0:32]->0x0 | }
ebx <- { | [0:32]->0x0 | }
ecx <- { | [0:32]->0x0 | }
edx <- { | [0:32]->0x0 | }
esi <- { | [0:32]->0x0 | }
edi <- { | [0:32]->0x0 | }
esp <- { | [0:32]->0x7ffff000 | }

In [6]: print(p.state.mmap)
<MemoryZone rel=None :
  <mo [08048000,08049000] data:b'\x7fELF\x01\x01\x01\x00\x00\x0...'>
  <mo [08049f14,08049ff0] data:b'\xff\xff\xff\xff\x00\x00\x00\x...'>
  <mo [08049ff0,08049ff4] data:@__gmon_start__>
  <mo [08049ff4,0804a000] data:b'(\x9f\x04\x08\x00\x00\x00\x00\...'>
  <mo [0804a000,0804a004] data:@__stack_chk_fail>
  <mo [0804a004,0804a008] data:@malloc>
  <mo [0804a008,0804a00c] data:@__gmon_start__>
  <mo [0804a00c,0804a010] data:@__libc_start_main>
  <mo [0804a010,0804af14] data:b'\x00\x00\x00\x00\x00\x00\x00\x...'>
  <mo [7fffd000,7ffff000] data:b'\x00\x00\x00\x00\x00\x00\x00\x...'>>
```

(other more specific executable formats are supported but they need to be loaded manually.) Also note that it is possible to provide a *raw* bytes string as input and then manually load the architecture:

```
In [1]: import amoco
In [2]: shellcode = (b
 ↪"\xeb\x16\x5e\x31\xd2\x52\x56\x89\xe1\x89\xf3\x31\xc0\xb0\x0b\xcd"
```

(continues on next page)

(continued from previous page)

```
                    b
↪"\x80\x31\xdb\x31\xc0\x40\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69"
                    b"\x6e\x2f\x73\x68")
In [3]: p = amoco.load_program(shellcode)
[WARNING] amoco.system.core       : unknown format
[WARNING] amoco.system.raw        : a cpu module must be imported

In [4]: from amoco.arch.x86 import cpu_x86
In [5]: p.cpu = cpu_x86

In [6]: print(p)
<RawExec - '(sc-eb165e31...)'>

In [7]: print(p.state.mmap)
<MemoryZone rel=None :
      <mo [00000000,00000024] data:'\xeb\x16^1\xd2RV\x89\xe1\x89\xf...'>>
```

The *shellcode* is mapped at address 0 by default, but can be relocated:

```
In [8]: p.relocate(0x4000)
In [9]: print(p.state.mmap)
<MemoryZone rel=None :
      <mo [00004000,00004024] data:'\xeb\x16^1\xd2RV\x89\xe1\x89\xf...'>>
```

2.2 Decoding blocks of instructions

Decoding some bytes as an *arch.core.instruction* needs only to load the desired cpu module, for example:

```
In [10]: cpu_x86.disassemble(b'\xeb\x16')
Out[10]: <amoco.arch.x86.spec_ia32 JMP ( length=2 type=2 )>
In [11]: print(_)
jmp         .+22
```

If a mapped binary program has been instanciated, we can start disassembling instructions or *data* located at some virtual address:

```
In [12]: print(p.read_instruction(0x4000))
jmp         *0x4018
In [13]: p.read_data(0x4000,2)
Out[13]: ['\xeb\x16']
```

Now, rather than manually adjusting the address to fetch the next instruction, we can use any of the code analysis strategies implemented in amoco to disassemble *basic blocks* directly:

```
% amoco load samples/x86/flow.elf
[...]
In [3]: z = amoco.sa.lsweep(p)

In [4]: z.getblock(0x8048380)
Out[4]: <block object (0x8048380-0x80483a1) with 13 instructions>

In [5]: b=_
In [6]: print(b.view)
———————— block 0x8048380 ——————————————————————
```

(continues on next page)

(continued from previous page)

```
0x8048380  '31ed'          xor         ebp, ebp
0x8048382  '5e'            pop         esi
0x8048383  '89e1'          mov         ecx, esp
0x8048385  '83e4f0'        and         esp, 0xfffffff0
0x8048388  '50'            push        eax
0x8048389  '54'            push        esp
0x804838a  '52'            push        edx
0x804838b  '6810860408'    push        0x8048610
0x8048390  '68a0850408'    push        0x80485a0
0x8048395  '51'            push        ecx
0x8048396  '56'            push        esi
0x8048397  '68fd840408'    push        0x80484fd
0x804839c  'e8cfffffff'    call        *0x8048370
──────────────────────────────────────────────────────
```

Note that a *block* view will show non-transformed instructions' operands (appart from PC-relative branch offsets which are shown as absolute addresses.) Block views can be *enhanced* by several analyses that will possibly add symbols related to addresses (provided by the program's symbol table) or more semantic-related information. These views are usually available only through the higher level *task* view object and add various comment tokens to instruction lines. For example:

```
In [7]: print( p.view.codeblock(b) )
────────── codeblock 0x8048380 ──────────────────────────────────────
0x8048380.text  '31ed'          xor         ebp, ebp
0x8048382.text  '5e'            pop         esi
0x8048383.text  '89e1'          mov         ecx, esp
0x8048385.text  '83e4f0'        and         esp, 0xfffffff0
0x8048388.text  '50'            push        eax
0x8048389.text  '54'            push        esp
0x804838a.text  '52'            push        edx
0x804838b.text  '6810860408'    push        0x8048610<__libc_csu_fini>
0x8048390.text  '68a0850408'    push        0x80485a0<__libc_csu_init>
0x8048395.text  '51'            push        ecx
0x8048396.text  '56'            push        esi
0x8048397.text  '68fd840408'    push        0x80484fd<main>
0x804839c.text  'e8cfffffff'    call        0x8048370<__libc_start_main>
──────────────────────────────────────────────────────────────────────
```

2.3 Symbolic representations of blocks

A *block* object provides instructions of the program located at some address in memory. A *node* object takes a block and allows to get a symbolic functional representation of what this block sequence of instructions is doing:

```
In [8]: n = amoco.cfg.node(b)
In [8]: print(n.map.view)
eip                                                 (eip+-0x10)
eflags:
 ├ cf                                               0x0
 ├ pf                                               (0x6996>>(esp+0x4)[4:8])[0:1]
 ├ af                                               af
 ├ zf                                               ({[ 0: 4] -> 0x0, [ 4:32] ->␣
↪(esp+0x4)[4:32]}==0x0)
 ├ sf                                               ({[ 0: 4] -> 0x0, [ 4:32] ->␣
↪(esp+0x4)[4:32]}<0x0)
```

(continues on next page)

(continued from previous page)

```
  | tf                                                  tf
  | df                                                  df
  | of                                                  0x0
ebp                                                     0x0
esp                                                     ({[ 0: 4] -> 0x0, [ 4:32] ->␣
↪(esp+0x4)[4:32]}-0x24)
esi                                                     M32(esp)
ecx                                                     (esp+0x4)
({ | [0:4]->0x0 | [4:32]->(esp+0x4)[4:32] | }-4)        eax
({ | [0:4]->0x0 | [4:32]->(esp+0x4)[4:32] | }-8)        ({[ 0: 4] -> 0x0, [ 4:32] ->␣
↪(esp+0x4)[4:32]}-0x4)
({ | [0:4]->0x0 | [4:32]->(esp+0x4)[4:32] | }-12)       edx
({ | [0:4]->0x0 | [4:32]->(esp+0x4)[4:32] | }-16)       0x8048610
({ | [0:4]->0x0 | [4:32]->(esp+0x4)[4:32] | }-20)       0x80485a0
({ | [0:4]->0x0 | [4:32]->(esp+0x4)[4:32] | }-24)       (esp+0x4)
({ | [0:4]->0x0 | [4:32]->(esp+0x4)[4:32] | }-28)       M32(esp)
({ | [0:4]->0x0 | [4:32]->(esp+0x4)[4:32] | }-32)       0x80484fd
({ | [0:4]->0x0 | [4:32]->(esp+0x4)[4:32] | }-36)       (eip+0x21)
```

Here we are with the *map* of the block. Now what this *mapper* object says is for example that once the block is executed esi register will be set to the 32 bits value pointed by esp, that the carry flag will be 0, or that the top of the stack will hold value eip+0x21. Rather than extracting the entire view of the mapper we can query any *expression* out if it:

```
In [9]: print(n.map(p.cpu.ecx))
(esp+0x4)
```

There are some caveats when it comes to query memory expressions but we will leave this for later (see *cas.mapper.mapper*).

The n.map object also provides a better way to see how the memory is modified by the block:

```
In [10]: print(n.map.mmap)
<MemoryZone rel=None :>
<MemoryZone rel={ | [0:4]->0x0 | [4:32]->(esp+0x4)[4:32] | } :
            <mo [-0000024,-0000020] data:(eip+0x21)>
            <mo [-0000020,-000001c] data:b'\xfd\x84\x04\x08'>
            <mo [-000001c,-0000018] data:M32(esp)>
            <mo [-0000018,-0000014] data:(esp+0x4)>
            <mo [-0000014,-0000010] data:b'\xa0\x85\x04\x08'>
            <mo [-0000010,-000000c] data:b'\x10\x86\x04\x08'>
            <mo [-000000c,-0000008] data:edx>
            <mo [-0000008,-0000004] data:({ | [0:4]->0x0 | [4:32]->(esp+0...>
            <mo [-0000004,00000000] data:eax>>
```

The *cas.mapper.mapper* class is an essential part of amoco that captures the semantics of the block by interpreting its' instructions in a symbolic way. Note that it takes no input state or whatever but just expresses what the block would do independently of what has been done before and even where the block is actually located.

For any mapper object, we can get the lists of *input* and *output* expressions, and replace inputs by any chosen expression:

```
In [11]: for x in set(n.map.inputs()): print(x)
esp
eip
M32(esp)
```

(continues on next page)

(continued from previous page)

```
In [12]: m = n.map.use(eip=0x8048380, esp=0x7fcfffff)
In [13]: print(m.view)
eip             <- 0x8048370
eflags:
| cf            <- 0x0
| sf            <- 0x0
| tf            <- tf
| zf            <- 0x0
| pf            <- 0x0
| of            <- 0x0
| df            <- df
| af            <- af
ebp             <- 0x0
esp             <- 0x7fcfffdc
esi             <- M32(0x7fcfffff)
ecx             <- 0x7fd00003
(0x7fd00000-4)  <- eax
(0x7fd00000-8)  <- 0x7fcfffc
(0x7fd00000-12) <- edx
(0x7fd00000-16) <- 0x8048610
(0x7fd00000-20) <- 0x80485a0
(0x7fd00000-24) <- 0x7fd00003
(0x7fd00000-28) <- M32(0x7fcfffff)
(0x7fd00000-32) <- 0x80484fd
(0x7fd00000-36) <- 0x80483a1
```

Its fine to disassemble a block at some address and get some symbolic representation of it, but we are still far from getting the picture of the entire program. In order to reason later about execution paths, we need a way to *chain* block mappers. This is provided by the mapper's shifts operators:

```
In [14]: mm = amoco.cas.mapper.mapper()
In [15]: amoco.conf.Cas.noaliasing = True
In [16]: mm[p.cpu.eip] = p.cpu.mem(p.cpu.esp+4,32)
In [17]: print( (n.map>>mm)(p.cpu.eip) )
0x80484fd
```

Here, taking a new mapper as if it came either from a block or a stub, and assuming that there is no memory aliasing, the sequential execution of `n.map` followed by `mm` would branch to address `0x80484fd` (`<main>`).

2.4 Starting some analysis

Important note:

> *** **The merge with emul branch has broken the static-analysis module.** This is going to be fixed only once the merge is fully integrated ***

CHAPTER 3

Examples

CHAPTER 4

Configuration

CHAPTER 5

Advanced features

CHAPTER 6

Overview

Amoco is composed of 5 sub-packages

- *arch*, deals with CPU architecures' to provide instructions disassemblers, and instructions' semantics for several CPUs, microcontrollers or "virtual machines":
  - x86, x64
  - armv7, armv8 (aarch64)
  - sparc (v8)
  - MIPS (R3000)
  - riscv
  - msp430
  - avr
  - pic/F46K22
  - v850
  - sh2, sh4
  - z80
  - BPF/eBPF (vm)
  - Dwarf (vm)
- *cas*, implements the *computer algebra system* to provide operations and mappings with symbolic expressions. It allows to represent architectures' registers values either as *concrete* or *symbolic* values, and to describe instructions' semantics as a *map* of expressions to registers or memory addresses. If z3 is installed, boolean expressions formulas can be translated to z3 bitvectors and checked by its solver. If satisfiable, a z3 model can be translated back into a :class:‘mapper‘ instance (with amoco expressions.)
- *system*, implements all *system* features like an abstract memory suited for symbolic expressions, as well as support for executable formats (ELF,PE,Mach-O,...) and their loaders to provide an abstraction of a "task" (a memory-mapped binary exectuable.)

- *sa* implements various *static analysis* methods to recover and build the control flow graph of functions.
- *ui* deals with how instructions and expressions are displayed either in a terminal or in a graphical *user interface*.

Modules *code* and *cfg* provide high-level abstractions of basic blocks, functions, and control flow graphs. Module *config*, *logger*, and signals provide the global configuration, logging and signaling facilities to all other modules.

CHAPTER 7

The architecture package

Supported CPU architectures are implemented in this package as subpackages and all use the *arch.core* generic classes. The interface to a CPU used by *system* classes is implemented as a `cpu_XXX.py` module usually in the architecture's subpackage.

This CPU module will:

- provide the CPU *environment* (registers and other internals)
- provide an instance of *arch.core.disassembler* class, which requires to:
  - define an instruction class based on *arch.core.instruction*
  - define the *arch.core.ispec* of every instruction for the generic decoder,
  - and define the *semantics* of every instruction with *cas.expressions*.
- optionnally define the output assembly format, and the GNU *as* (or any other) assembly parser.
- optionnally define the function `PC()` that allows generic analysis to which register represents the instructions' pointer.

A simple example is provided by the arch.arm.v8 architecture which implements a model of ARM AArch64: The interface CPU module is `arch.arm.cpu_armv8`, which imports everything from the `arch.arm.v8` subpackage.

7.1 Adding support for a new cpu module

7.1.1 The cpu environment

It all starts with the definition of the cpu *environment* in a dedicated module. This module defines registers as instances of *cas.expressions.reg*, and associated register slices with *cas.expressions.slc* if necessary. For example, x86 register `eax` and its slices are defined in `arch.x86.env` as:

```
eax = reg("eax",32)
ax  = slc(eax, 0, 16, "ax")
al  = slc(eax, 0, 8 , "al")
ah  = slc(eax, 8, 8 , "ah")
```

In order to improve code analysis and views, some registers should be bound to their special *cas.expressions.regtype*, using one of the dedicated callable or context manager. For example, the stack pointer should be bound to regtype `'STACK'` using:

```
esp = is_reg_stack(reg('esp',32))
```

or alternatively using a context manager:

```
with is_reg_stack:
    esp = reg('esp',32)
```

Defined regtypes are:

- `cas.expressions.is_reg_pc`
- `cas.expressions.is_reg_flags`
- `cas.expressions.is_reg_stack`
- `cas.expressions.is_reg_other`

Once all needed registers are defined, it is recommended to define also an ordered list called `registers` which will be used by emulator instances for registers views.

Finally, the cpu *environment* sometimes also needs to define some internal parameters that change the way instructions are decoded or the memory endianness. For example, the *arch.arm.v7.env* module defines internals for `isetstate` to change the instruction set from ARM to Thumb, and `endianstate` to change endianness. These *internal* parameters differ from regular registers by the fact that they are not defined as expressions and thus cannot be symbolic.

7.1.2 Instructions specifications

The instructions' specifications are then defined in a module as well. An instruction's *specification* is an instance of *arch.core.ispec* that decorates a function which performs setup of an instruction's instance. The *specification* describes how the instruction is decoded out of bytes in a way that allows the decorated function to setup instruction's operands and any other characteristics from the decoded values. This description allows to follow CPU datasheet's instructions manual very closely. Moreover, thanks to how decorator work, several specs can share the same setup function. For example, we have in the MIPS R3000 instructions' spec module:

```
@ispec("32<[ 001100 rs(5) rt(5) imm(16) ]", mnemonic="ANDI")
@ispec("32<[ 001101 rs(5) rt(5) imm(16) ]", mnemonic="ORI")
@ispec("32<[ 001110 rs(5) rt(5) imm(16) ]", mnemonic="XORI")
def mips1_dri(obj, rs, rt, imm):
    src1 = env.R[rs]
    imm = env.cst(imm, 32)
    dst = env.R[rt]
    obj.operands = [dst, src1, imm]
    obj.type = type_data_processing
```

Here, `obj` is an instruction instanciated by the disassembler, if decoded bytes matches one of these spec definitions. In such case, the setup function is called with arguments `rs`, `rt` and `imm` being ints decoded from the corresponding bits (see *arch.core.ispec* below.) Any instruction setup should define at least an `obj.operands` list and should indicate one of the following `obj.type`:

- `type_data_processing`, which are well-defined instructions,
- `type_control_flow`, which mark default ending of assembly blocks,
- `type_cpu_state`, which may change the cpu internal registers,
- `type_system`, which have usually no impact on code semantics,
- `type_other`

7.1.3 The cpu disassembler

When the specification module is done, the cpu disassembler can be instanciated. First a new local instruction class should be derived from the generic *arch.core.instruction* with:

```
from amoco.arch.core import instruction
instruction_X = type("instruction_X", (instruction,), {})
```

Then, a disassembler instance is obtained with:

```
from amoco.arch.core import disassembler
from amoco.arch.X import spec_X, spec_thumb
disassemble = disassembler([spec_X], iclass=instruction_X)
```

The first argument is the list of available specifications. Most architectures have only one mode but some like ARM can switch from a default mode (ARM) to an alternate mode like Thumb (see class definition `mode` argument.) The second is our new instruction class. By default, disassemblers will fetch instructions in little-endian, but the `endian` parameter allows to fetch in big-endian. For example the ARMv7 architecture's disassembler is:

```
mode = lambda: internals["isetstate"]
endian = lambda: 1 if internals["ibigend"] == 0 else -1
disassemble = disassembler([spec_armv7, spec_thumb],
                           instruction_armv7,
                           mode,
                           endian)
```

which allows the semantics to possibly change both the mode and the instructions' endianness dynamically.

7.1.4 Instructions semantics

An instruction's semantics is a function associated to the instruction's mnemonic which operates on a *cas.mapper.mapper* object. The function's name should be "i_XXX" for mnemonic "XXX". The mapper argument enables transitions from a state to another state. For example, the semantics of all MIPS R3000 `AND` instructions is:

```
@__npc
def i_AND(ins, fmap):
    dst, src1, src2 = ins.operands
    if dst is not zero:
        fmap[dst] = fmap(src1&src2)
```

The first argument is the disassembled instruction object and the second argument is the mapper (i.e. the state). We simply create local variables from the operands list and then update the state according to these operands: Thus, the mapper is modified by setting the first operand expression to the mapper's evaluation of the *cas.expressions.op* formed by `src1 & src2`.

Of course, since we want *symbolic* semantics these functions might end-up being quite complex especially for conditional stuff. For example, like in the case of this weird *unaligned load word* MIPS R3000 instruction:

```
@__npc
def i_LWL(ins, fmap):
    dst, base, src = ins.operands
    addr = base+src
    if dst is not zero:
        fmap[dst[24:32]] = fmap(mem(addr,8))
        cond1 = (addr%4)!=0
        fmap[dst[16:24]]  = fmap(tst(cond1,mem(addr-1,8),dst[16:24]))
        addr = addr - 1
        cond2 = cond1 & ((addr%4)!=0)
        fmap[dst[8:16]] = fmap(tst(cond2,mem(addr-1,8),dst[8:16]))
        fmap[dst] = fmap[dst].simplify()
```

Here, the number of bytes read from memory depends on the word-alignement of the address value. This instruction is thus normally coupled with a LWR which performs the read from memory of the rest of bytes accross the word-alignment. In concrete semantics, this is quite simple to write since address alignment is always computable and thus 3 cases are possible. In *symbolic* semantics, things are more tricky since address is symbolic and thus the resulting writeback to dst register is a symbolic expression that must take into account 3 cases at once.

Updating the cpu instruction pointer

Now, instruction's semantics must also update the cpu `PC()`. In the MIPS case, this is performed by using the `__npc` decorator role which updates `pc` and `npc` as well to handle delay slot cases. Architectures without delay slots can just advance their program's counter by the length of the instruction. Architectures with delay slots can always handle delayed branches by relying on intermediate (hidden) program counters. This is the case for `arch.sparc` and `arch.MIPS` where `__npc` does:

```
pc  <- npc
npc <- npc+4
```

and since branch instructions have an effect on `npc` once they have been processed, the next instruction to execute (the one located at `pc`,) is still just after the branch instruction.

However, special care must be taken to avoid pitfalls... A common mistake is to believe that the delay slot instruction is executed *before* the branch instruction as if the two instructions were simply swapped. This is not true. The branch effectively occurs after, but its operands are still evaluated before the delay slot has had time to execute! For example the MIPS R3000 sequence:

```
liu   t7, 0x5
liu   t6, 0x2
bne   t7, t6, *somewhere
addiu t7, t7, -0x3
```

will lead to a branch *not taken*. See pipelining discussion below for details...

A Note on cpu pipelining and cycle-accurate emulation

For most architectures, the instruction parallelism introduced by the underlying pipeline does not interfer with the semantics. What this means is that for example, assuming `R1=0, R2=1, R3=1` the generic case of:

```
OR   R1, R2, R3
ADD  R4, R1, 1
```

should obviously lead to `R4=2` anyways, because pipelining is implemented to improve performance but shouldn't have any impact on semantics. Hence, **we can always emulate instructions as if no parallelism existed**. Right ? Well, not exactly...

All pipelines have *pipeline hazard*, ie. situations that could lead to undefined behaviors if not handled correctly. In our example above, the `R1` register is really updated after the ALU has performed its operation on `R2` and `R3` values. Meanwhile, the `ADD` instruction wants to read `R1` value as soon as the instruction is decoded (after it was fetched,) and would consequently read its value *before* it is updated. Thus, pipelines have internal mechanism to detect these kind of situations and either stall the pipeline (wait for `R1` to be written back before being used) or forward things back to other stages as soon as possible. In this case, the ALU forwards its result immediately to back to the ALU entry multiplexer before being updated in `R1` later.

Unfortunately, some old architectures like MIPS[#]_ R3000 handled only a limited set of these *pipeline hazard* and heavily relied on the compiler to avoid some instructions' flows (usually by inserting nops.) In MIPS R3000 architecture, the above case is handled correctly unless a load/store is involved like in:

```
lbu v0, 0x1(a1)
nop
sll v0, v0, 0x8
```

Here, the compiler has inserted a `nop` to ensure that the loaded byte has been fetched and can be forwarded to the ALU for `sll`. Hence, as long as we emulate code produced by compliant compilers, we still can ignore the underlying pipeline operations. But this is not true anymore in the general cases. Since most of the time we can't make this assumption, instructions can't formally be emulated as if no parallelism existed. If we ever have MIPS R3000 code with:

```
lbu v0, 0x1(a1)
sll v0, v0, 0x8
```

then the resulting mapper is not `v0 <- mem(a1+0x1,8)<<8` but rather something that highly depends on the involved pipeline interlocking mechanism, most likely `v0 <- v0<<8`.

Like for delay slots of branch instructions that can be handled with an additional `npc` register, we can always simulate the pipeline delay by introducing a kind of hidden "register". In amoco the mapper has an internal `delayed` attribute that allows explict delayed updates. (these updates are triggered by explicit calls to `mapper.update_delayed()`, usually right in the middle of every instructions, as if the result of the delayed load was forwarded to the current ALU stage.)

7.1.5 Instructions format

Now that instructions specifications and semantics are defined, it is recommended to define at least one formatter to print instructions according to the CPU's Instruction Set Assembly manual. Available formatters for a CPU ISA are instances of the *`arch.core.Formatter`* class. These formatters are initiated from a dict object that maps instructions' mnemonic or setup function name to iterable formatting functions operating on the instruction object. For example:

```
format_default = (mnemo, opers)

MIPS_full_formats = {
    "mips1_loadstore": (mnemo, opers_mem),
    "mips1_jump_abs": (mnemo, opers),
    "mips1_jump_rel": (mnemo, opers_rel),
    "mips1_branch": (mnemo, opers_adr),
}
```

(continues on next page)

(continued from previous page)

```
MIPS_full = Formatter(MIPS_full_formats)
MIPS_full.default = format_default
```

Here, the available format is `MIPS_full`, instanciated from the `MIPS_full_formats` dict which maps spec setup functions to their corresponding formatting tuples. Functions `mnemo`, and `opers` take the instruction and return a Pygments-compatible list of tokens if support for pretty-printing is implemented, or simply a string. When an instruction is printed, the formatter starts by matching its mnemonic or its setup function, or takes the default formatting iterable, and then joins all outputs from the iterables.

7.1.6 The cpu module

Finally, the *cpu* module can be fully created. This module should import all from the architecture's *environment* and define its disassembler as shown above.

The semantics is associated to the instruction class with the `arch.core.instruction.set_uarch(dict)()` which takes a mapping from mnemonics to the corresponding instruction semantics function. Thus, in most cpu modules this binding is done with:

```
from .asm import *
uarch = dict(filter(lambda kv: kv[0].startswith("i_"), locals().items()))
instruction_X.set_uarch(uarch)
```

The chosen formatter is bound to the instruction class with:

```
from .formats import X_full
instruction_X.set_formatter(X_full)
```

(Eventually, if not already defined in the *environment*, the `PC()` function is defined to return the instruction's pointer.)

Note that whenever a disassembler is available, the entire architecture ISA decision tree can be displayed with:

```
>>> from amoco.ui.views import archView
>>> from amoco.arch.mips.cpu_r3000LE import disassemble
>>> print(archView(disassemble))
─[& f0000000 == 0]
  │─[& fc000000 == 0]
  │  │─[& fc00003f == 8]
  │  │  │─JR              : 32<[ 000000 rs(5) 00000 00000 00000 001000]
  │  │─[& fc00003f == 12]
  │  │  │─MFLO            : 32<[ 000000 00000 00000 rd(5) 00000 010010 ]
  │  │─[& fc00003f == 10]
  │  │  │─MFHI            : 32<[ 000000 00000 00000 rd(5) 00000 010000 ]
  │  │─[& fc00003f == 13]
  │  │  │─MTLO            : 32<[ 000000 rs(5) 00000 00000 00000 010011 ]
  │  │─[& fc00003f == 11]
  │  │  │─MTHI            : 32<[ 000000 rs(5) 00000 00000 00000 010001 ]
  │  │─[& fc00003f == 19]
  │  │  │─MULTU           : 32<[ 000000 rs(5) rt(5) 00000 00000 011001]
  │  │─[& fc00003f == 18]
  │  │  │─MULT            : 32<[ 000000 rs(5) rt(5) 00000 00000 011000]
  │  │─[& fc00003f == 1b]
  │  │  │─DIVU            : 32<[ 000000 rs(5) rt(5) 00000 00000 011011]
  │  │─[& fc00003f == 1a]
  │  │  │─DIV             : 32<[ 000000 rs(5) rt(5) 00000 00000 011010]
  │  │─[& fc00003f == 9]
```

(continues on next page)

(continued from previous page)

```
   |  |  |  |—JALR            : 32<[ 000000 rs(5) 00000 rd(5) 00000 001001]
   |  |  |—[& fc00003f == 2b]
   |  |  |  |—SLTU            : 32<[ 000000 rs(5) rt(5) rd(5) 00000 101011]
   |  |  |—[& fc00003f == 2a]
   |  |  |  |—SLT             : 32<[ 000000 rs(5) rt(5) rd(5) 00000 101010]
   |  |  |—[& fc00003f == 6]
   |  |  |  |—SRLV            : 32<[ 000000 rs(5) rt(5) rd(5) 00000 000110]
   |  |  |—[& fc00003f == 7]
   |  |  |  |—SRAV            : 32<[ 000000 rs(5) rt(5) rd(5) 00000 000111]
   |  |  |—[& fc00003f == 4]
   |  |  |  |—SLLV            : 32<[ 000000 rs(5) rt(5) rd(5) 00000 000100]
   |  |  |—[& fc00003f == 26]
   |  |  |  |—XOR             : 32<[ 000000 rs(5) rt(5) rd(5) 00000 100110]
   |  |  |—[& fc00003f == 25]
   |  |  |  |—OR              : 32<[ 000000 rs(5) rt(5) rd(5) 00000 100101]
   |  |  |—[& fc00003f == 27]
   |  |  |  |—NOR             : 32<[ 000000 rs(5) rt(5) rd(5) 00000 100111]
   |  |  |—[& fc00003f == 24]
   |  |  |  |—AND             : 32<[ 000000 rs(5) rt(5) rd(5) 00000 100100]
   |  |  |—[& fc00003f == 23]
   |  |  |  |—SUBU            : 32<[ 000000 rs(5) rt(5) rd(5) 00000 100011]
   |  |  |—[& fc00003f == 21]
   |  |  |  |—ADDU            : 32<[ 000000 rs(5) rt(5) rd(5) 00000 100001]
   |  |  |—[& fc00003f == 22]
   |  |  |  |—SUB             : 32<[ 000000 rs(5) rt(5) rd(5) 00000 100010]
   |  |  |—[& fc00003f == 20]
   |  |  |  |—ADD             : 32<[ 000000 rs(5) rt(5) rd(5) 00000 100000]
   |  |  |—[& fc00003f == 2]
   |  |  |  |—SRL             : 32<[ 000000 00000 rt(5) rd(5) sa(5) 000010 ]
   |  |  |—[& fc00003f == 3]
   |  |  |  |—SRA             : 32<[ 000000 00000 rt(5) rd(5) sa(5) 000011 ]
   |  |  |—[& fc00003f == 0]
   |  |  |  |—SLL             : 32<[ 000000 00000 rt(5) rd(5) sa(5) 000000 ]
   |  |  |—[& fc00003f == c]
   |  |  |  |—SYSCALL         : 32<[ 000000 .code(20) 001100]
   |  |  |—[& fc00003f == d]
   |  |  |  |—BREAK           : 32<[ 000000 .code(20) 001101]
   |  |—[& fc000000 == 4000000]
   |  |  |—BLTZAL          : 32<[ 000001 rs(5) 10000 ~imm(16) ]
   |  |  |—BLTZ            : 32<[ 000001 rs(5) 00000 ~imm(16) ]
   |  |  |—BGEZAL          : 32<[ 000001 rs(5) 10001 ~imm(16) ]
   |  |  |—BGEZ            : 32<[ 000001 rs(5) 00001 ~imm(16) ]
   |  |—[& fc000000 == c000000]
   |  |  |—JAL             : 32<[ 000011 t(26)]
   |  |—[& fc000000 == 8000000]
   |  |  |—J               : 32<[ 000010 t(26)]
—[& f0000000 == 40000000]
   |—[& f2000000 == 40000000]
   |  |—MTC             : 32<[ 0100 .z(2) 00100 rt(5) rd(5) 00000000000 ]
   |  |—CTC             : 32<[ 0100 .z(2) 00110 rt(5) rd(5) 00000000000 ]
   |  |—MFC             : 32<[ 0100 .z(2) 00000 rt(5) rd(5) 00000000000 ]
   |  |—CFC             : 32<[ 0100 .z(2) 00010 rt(5) rd(5) 00000000000 ]
   |—[& f2000000 == 42000000]
   |  |—COP             : 32<[ 0100 .z(2) 1 .cofun(25) ]
—[& f0000000 == 30000000]
   |—LUI             : 32<[ 001111 00000 rt(5) imm(16) ]
   |—XORI            : 32<[ 001110 rs(5) rt(5) imm(16) ]
```

(continues on next page)

(continued from previous page)

```
   |—ORI             : 32<[ 001101 rs(5) rt(5) imm(16) ]
   |—ANDI            : 32<[ 001100 rs(5) rt(5) imm(16) ]
—[& f0000000 == 10000000]
   |—BLEZ            : 32<[ 000110 rs(5) 00000 ~imm(16) ]
   |—BGTZ            : 32<[ 000111 rs(5) 00000 ~imm(16) ]
   |—BNE             : 32<[ 000101 rs(5) rt(5) ~imm(16) ]
   |—BEQ             : 32<[ 000100 rs(5) rt(5) ~imm(16) ]
—[& f0000000 == 20000000]
   |—SLTIU           : 32<[ 001011 rs(5) rt(5) ~imm(16) ]
   |—SLTI            : 32<[ 001010 rs(5) rt(5) ~imm(16) ]
   |—ADDIU           : 32<[ 001001 rs(5) rt(5) ~imm(16) ]
   |—ADDI            : 32<[ 001000 rs(5) rt(5) ~imm(16) ]
—[& f0000000 == b0000000]
   |—SWR             : 32<[ 101110 base(5) rt(5) offset(16) ]
—[& f0000000 == 90000000]
   |—LWR             : 32<[ 100110 base(5) rt(5) offset(16) ]
   |—LHU             : 32<[ 100101 base(5) rt(5) offset(16) ]
   |—LBU             : 32<[ 100100 base(5) rt(5) offset(16) ]
—[& f0000000 == a0000000]
   |—SWL             : 32<[ 101010 base(5) rt(5) offset(16) ]
   |—SW              : 32<[ 101011 base(5) rt(5) offset(16) ]
   |—SH              : 32<[ 101001 base(5) rt(5) offset(16) ]
   |—SB              : 32<[ 101000 base(5) rt(5) offset(16) ]
—[& f0000000 == 80000000]
   |—LWL             : 32<[ 100010 base(5) rt(5) offset(16) ]
   |—LW              : 32<[ 100011 base(5) rt(5) offset(16) ]
   |—LH              : 32<[ 100001 base(5) rt(5) offset(16) ]
   |—LB              : 32<[ 100000 base(5) rt(5) offset(16) ]
—[& f0000000 == e0000000]
   |—SWC             : 32<[ 1110 .z(2) base(5) rt(5) offset(16) ]
—[& f0000000 == c0000000]
   |—LWC             : 32<[ 1100 .z(2) base(5) rt(5) offset(16) ]
```

If several specification modes are provided, they are listed one after the other.

arch/core.py

The architecture's core module implements essential classes for the definition of new cpu architectures:

- the *instruction* class models cpu instructions decoded by the disassembler.
- the *disassembler* class implements the instruction decoding logic based on provided specifications.
- the *ispec* class is a function decorator that allows to define the specification of an instruction.
- the *Formatter* class is used for instruction pretty printing

class arch.core.**icore**(*istr=b''*)

This is the core class for the generic parent instruction class below. It defines the mandatory API for all instructions.

bytes

instruction's bytes

Type bytes

type

one of (type_data_processing, type_control_flow, type_cpu_state, type_system, type_other) or type_undefined (default) or type_unpredictable.

Type int

spec
the specification that was decoded by the disassembler to instanciate this instruction.

Type *ispec*

mnemonic
the mnemonic string as defined by the specification.

Type str

operands
the list of operands' expressions.

Type list

misc
a defaultdict for passing various arch-dependent infos (which returns None for undefined keys.)

Type dict

classmethod set_uarch(*uarch*)
class method to define the instructions' semantics uarch dict

typename()
returns the instruction's type as a string

length
length of the instruction in bytes

class arch.core.**instruction**(*istr*)
The generic instruction class allows to define instruction for any cpu instructions set and provides a common API for all arch-independent methods. It extends the *icore* with an *address* attribute and formatter methods.

address
the memory address where this instruction as been disassembled.

Type *cst*

classmethod set_formatter(*f*)
classmethod that defines the formatter for all instances

static formatter(*i*, *toks=False*)
default formatter if no formatter has been set, will return the highlighted list from tokens for raw mnemonic, and comma-separated operands expressions.

toks()
returns the (unjoined) list of formatted tokens.

exception arch.core.**InstructionError**(*i*)

exception arch.core.**DecodeError**

class arch.core.**disassembler**(*specmodules*, *iclass=<class 'arch.core.instruction'>*, *iset=<function disassembler.<lambda>>*, *endian=<function disassembler.<lambda>>*)
The generic disassembler class will decode a byte string based on provided sets of instructions specifications and various parameters like endianess and ways to select the appropriate instruction set.

Parameters

- **specmodules** – list of python modules containing ispec decorated funcs
- **iclass** – the specific instruction class based on *instruction*

- **iset** – lambda used to select module (ispec list)
- **endian** – instruction fetch endianess (1: little, -1: big)

maxlen
the length of the longest instruction found in provided specmodules.

iset
the lambda used to select the right specifications for decoding

endian
the lambda used to define endianess.

specs
the *tree* of *ispec* objects that defines the cpu architecture.

setup(*ispecs*)
setup will (recursively) organize the provided ispecs list into an optimal tree so that __call__ can efficiently find the matching ispec format for a given bytestring (we don't want to search all specs until a match, so we need to separate formats as much as possible). The output tree is (f,l) where f is the submask to check at this level and l is a defaultdict such that l[x] is the subtree of formats for which submask is x.

class arch.core.**ispec**(*format*, ***kargs*)
ispec (customizable) decorator

@ispec allows to easily define instruction decoders based on architecture specifications.

Parameters

- **spec** (*str*) – a human-friendly *format* string that describes how the ispec object will (on request) decode a given bytestring and how it will expose various decoded entities to the decorated function in order to define an instruction.
- ****kargs** – additional arguments to ispec decorator **must** be provided with `name=value` form and are declared as attributes/values within the instruction instance *before* calling the decorated function. See below for conventions about names.

format
the spec format passed as argument (see Note below).

Type str

hook
the decorated python function to be called during decoding. The hook function name is relevant only for instructions' formatter. See *arch.core.Formatter*.

Type callable

iattr
the dictionary of instruction attributes to add before decoding. Attributes and their values are passed from the spec's kargs when the name does not start with an underscore.

Type dict

fargs
the dictionary of keywords arguments to pass to the hook. These keywords are decoded from the format or given by the spec's kargs when name starts with an underscore.

Type dict

precond
an optional function that takes the instruction object as argument and returns a boolean to indicate wether the hook can be called or not. (This allows to avoid decoding when a prefix is missing for example.)

Type *func*

size
the bit length of the format (LEN value)

Type int

fix
the values of fixed bits within the format

Type Bits

mask
the mask of fixed bits within the format

Type Bits

Examples

This statement creates an ispec object with hook `f`, and registers this object automatically in a SPECS list object within the module where the statement is found:

```
@ispec("32[ .cond(4) 101 1 imm24(24) ]", mnemonic="BL", _flag=True)
def f(obj,imm24,_flag):
    [...]
```

When provided with a bytestring, the `decode()` method of this ispec object will:

- proceed with decoding ONLY if bits 27,26,25,24 are 1,0,1,1 or raise an exception
- instanciate an instruction object (obj)
- decode 4 bits at position [28,29,30,31] and provide this value as an integer in 'obj.cond' instruction instance attribute.
- decode 24 bits at positions 23..0 and provide this value as an integer as argument 'imm24' of the decorated function f.
- set obj.mnemonic to 'BL' and pass argument _flag=True to f.
- call f(obj,...)
- return obj

Note: The `spec` string format is `LEN ('<' or '>') '[' FORMAT ']' ('+' or '&' NUMBER)`

- **`LEN` is either an integer that represents the bit length of the instruction or '*'.** Length must be a multiple of 8, '*' is used for a variable length instruction.
- **`FORMAT` is a series of *directives* (see below.)** Each directive represents a sequence of bits ordered according to the spec direction : '<' (default) means that directives are ordered from MSB (bit index LEN-1) to LSB (bit index 0) whereas '>' means LSB to MSB.

The spec string is optionally terminated with '+' to indicate that it represents an instruction *prefix*, or by '&' NUMBER to indicate that the instruction has a *suffix* of NUMBER more bytes to decode some of its operands. In the *prefix* case, the bytestring matching the ispec format is stacked temporarily until the rest of the bytestring matches a non prefix ispec. In the *suffix* case, only the spec bytestring is used to define the instruction but the `read_instruction()` fetcher will provide NUMBER more bytes to the `xdata()` method of the instruction.

The directives defining the `FORMAT` string are used to associate symbols to bits located at dedicated offsets within the bitstring to be decoded. A directive has the following syntax:

- - (indicates that current bit position is not decoded)
- `0` (indicates that current bit position must be 0)
- `1` (indicates that current bit position must be 1)

or

- `type SYMBOL location` where:
  - `type` is an *optional* modifier char with possible values:
    - **.** indicates that the `SYMBOL` will be an *attribute* of the *instruction*.
    - ~ indicates that the decoded value will be returned as a Bits instance.
    - # indicates that the decoded value will be returned as a string of [01] chars.
    - = indicates that decoding should *end* at current position (overlapping)

    if not present, the `SYMBOL` will be passed as a keyword argument to the function with value decoded as an integer.
  - `SYMBOL`: is a mandatory string matching regex `[A-Za-z_][0-9A-Za-z_]*`
  - `location`: is an optional string matching the following expressions:
    - **(len)** [indicates that the value is decoded from the next len bits starting] from the current position of the directive within the `FORMAT` string.
    - **(*)** [indicates a *variable length directive* for which the value is decoded] from the current position with all remaining bits in the `FORMAT`. If the `LEN` is also variable then all remaining bits from the instruction buffer input string are used.

    default location value is `(1)`.

The special directive `{byte}` is a shortcut for 8 fixed bits. For example `8>[{2f}]` is equivalent to `8>[ 1111 0100 ]`, or `8<[ 0010 1111 ]`.

---

class `arch.core.`**Formatter**(*formats*)

Formatter is used for instruction pretty printing

Basically, a `Formatter` object is created from a dict associating a key with a list of functions or format string. The key is either one of the mnemonics or possibly the name of a @ispec-decorated function (this allows to group formatting styles rather than having to declare formats for every possible mnemonic.) When the instruction is printed, the formatting list elements are "called" and concatenated to produce the output string.

CHAPTER 8

The computer algebra system package

Contents

Symbolic expressions are provided by several classes found in module `cas/expressions`:

- Constant `cst`, which represents immediate (signed or unsigned) value of fixed size (bitvector),
- Symbol `sym`, a Constant equipped with a reference string (non-external symbol),
- Register `reg`, a fixed size CPU register *location*,
- External `ext`, a reference to an external location (external symbol),
- Floats `cfp`, constant (fixed size) floating-point values,
- Composite `comp`, a bitvector composed of several elements,
- Pointer `ptr`, a memory *location* in a segment, with possible displacement,
- Memory `mem`, a Pointer to represent a value of fixed size in memory,
- Slice `slc`, a bitvector slice of any element,
- Test `tst`, a conditional expression, (see below.)
- Operator `uop`, an unary operator expression,
- Operator `op`, a binary operator expression. The list of supported operations is not fixed although several predefined operators allow to build expressions directly from Python expressions: say, you don't need to write `op('+',x,y)`, but can write `x+y`. Supported operators are:
  - `+`, `-`, `*` (multiply low), `**` (multiply extended), `/`

- `&`, `|`, `^`, `~`
- `==`, `!=`, `<=`, `>=`, `<`, `>`
- `>>`, `<<`, `//` (arithmetic shift right), `>>>` and `<<<` (rotations).

See *cas.expressions._operator* for more details.

All elements inherit from the `exp` class which defines all default methods/properties. Common attributes and methods for all elements are:

- `size`, a Python integer representing the size in bits,
- `sf`, the True/False *sign-flag*.
- `length` (size/8)
- `mask` (1<<size)-1
- extend methods (`signextend(newsize)`, `zeroextend(newsize)`)
- `bytes(sta,sto,endian=1)` method to retreive the expression of extracted bytes from sta to sto indices.

All manipulation of an expression object usually result in a new expression object except for `simplify()` which performs a few in-place elementary simplifications.

8.1 cas/expressions.py

The expressions module implements all above *exp* classes. All symbolic representation of data in amoco rely on these expressions.

class cas.expressions.**exp**(*size=0*, *sf=False*)

the core class for all expressions. It defines mandatory attributes, shared methods like dumps/loads etc.

size

the bit size of the expression (default is 0.)

Type int

sf

the sign flag of the expression (default is False: unsigned.)

Type Bool

length

the byte size of the expression.

Type int

mask

the bit mask of the expression.

Type int

Note: len(exp) returns the byte size, assuming that size is a multiple of 8.

---

signed()

consider expression as signed

unsigned()

consider expression as unsigned

eval(*env*)
evalute expression in given `mapper` env

simplify(***kargs*)
simplify expression based on predefined heuristics

depth()
depth size of the expression tree

dumps()
pickle expression

loads(*s*)
unpickle expression

toks(***kargs*)
returns list of pretty printing tokens of the expression

pp(***kargs*)
pretty-printed string of the expression

bit(*i*)
extract i-th bit expression of the expression

bytes(*sta=0*, *sto=None*, *endian=1*)
returns the expression slice located at bytes [sta,sto] taking into account given endianess 1 (little) or -1 (big). Defaults to little endian.

extend(*sign*, *size*)
extend expression to given size, taking sign into account

signextend(*size*)
sign extend expression to given size

zeroextend(*size*)
zero extend expression to given size

to_smtlib(*solver=None*)
translate expression to its smt form

class cas.expressions.**top**(*size=0*, *sf=False*)
top expression represents symbolic values that have reached a high complexity threshold.

Note: This expression is an absorbing element of the algebra. Any expression that involves a top expression results in a top expression.

depth()
depth size of the expression tree

class cas.expressions.**cst**(*v*, *size=32*)
cst expression represents concrete values (constants).

value
get the integer of the expression, taking into account the sign flag.

Type int

toks(***kargs*)
returns list of pretty printing tokens of the expression

to_sym(*ref*)
cast into a symbol expression associated to name ref

eval(*env*)
evalute expression in given `mapper` env

zeroextend(*size*)
zero extend expression to given size

signextend(*size*)
sign extend expression to given size

class cas.expressions.**sym**(*ref*, *v*, *size=32*)
symbol expression extends cst with a reference name for pretty printing

class cas.expressions.**cfp**(*v*, *size=32*)
floating point concrete value expression

toks(***kargs*)
returns list of pretty printing tokens of the expression

eval(*env*)
evalute expression in given `mapper` env

class cas.expressions.**reg**(*refname*, *size=32*)
symbolic register expression

etype
int([x]) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

toks(***kargs*)
returns list of pretty printing tokens of the expression

eval(*env*)
evalute expression in given `mapper` env

class cas.expressions.**regtype**(*t*)
decorator and context manager (with...) for associating a register to a specific category among STD (standard), PC (program counter), FLAGS, STACK, OTHER.

class cas.expressions.**ext**(*refname*, ***kargs*)
external reference to a dynamic (lazy or non-lazy) symbol

toks(***kargs*)
returns list of pretty printing tokens of the expression

call(*env*, ***kargs*)
explicit call to the ext's stub

class cas.expressions.**lab**(*refname*, ***kargs*)
label expression used by the assembler

cas.expressions.**composer**(*parts*)
composer returns a comp object (see below) constructed with parts from low significant bits parts to most significant bits parts. The last part sf flag propagates to the resulting comp.

class cas.expressions.**comp**(*s*)
composite expression, represents an expression made of several parts.

parts
expressions parts dictionary. Each key is a tuple (pos,sz) and value is the exp part. pos is the bit position for this part, and sz is its size.

Type dict

smask
mapping of bit index to the part's key that defines this bit.

Type list

Note: Each part can be accessed by 'slicing' the comp to obtain another comp or the part if the given slice indices match the part position.

---

toks(***kargs*)
returns list of pretty printing tokens of the expression

eval(*env*)
evalute expression in given `mapper` env

simplify(***kargs*)
simplify expression based on predefined heuristics

cut(*start*, *stop*)
cut will scan the parts dict to find those spanning **over** start and/or stop bounds then it will split them and remove their inner parts.

Note: cut is in in-place method (affects self).

---

restruct()
restruct will aggregate consecutive cst expressions in order to minimize the number of parts.

depth()
depth size of the expression tree

class cas.expressions.**mem**(*a*, *size=32*, *seg=None*, *disp=0*, *mods=None*, *endian=1*)
memory expression represents a symbolic value of length size, in segment seg, at given address expression.

a
a pointer expression that represents the address.

Type *ptr*

endian
1 means little, -1 means big.

Type int

mods
list of possibly aliasing operations affecting this exp.

Type list

Note: The mods list allows to handle aliasing issues detected at fetching time and adjust the eval result accordingly.

---

toks(***kargs*)
returns list of pretty printing tokens of the expression

eval(*env*)
evalute expression in given `mapper` env

simplify(***kargs*)
simplify expression based on predefined heuristics

bytes(*sta=0*, *sto=None*, *endian=0*)
returns the expression slice located at bytes [sta,sto] taking into account given endianess 1 (little) or -1 (big). Defaults to little endian.

class cas.expressions.**ptr**(*base*, *seg=None*, *disp=0*)
ptr holds memory addresses with segment, base expressions and displacement integer (offset relative to base).

base
symbolic expression for the base of pointer address.

Type *exp*

disp
offset relative to base for the pointer address.

Type int

seg
segment register (or None if unused.)

Type *reg*

toks(***kargs*)
returns list of pretty printing tokens of the expression

simplify(***kargs*)
simplify expression based on predefined heuristics

eval(*env*)
evalute expression in given `mapper` env

cas.expressions.**slicer**(*x*, *pos*, *size*)
wrapper of slc class that returns a simplified version of x[pos:pos+size].

class cas.expressions.**slc**(*x*, *pos*, *size*, *ref=None*)
slice expression, represents an expression part.

x
reference to the symbolic expression

Type *exp*

pos
start bit for the part.

Type int

ref
an alternative symbolic name for this part.

Type str

etype
int([x]) -> integer int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments are given. If x is a number, return x.__int__(). For floating point numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string, bytes, or bytearray instance representing an integer literal in the given base. The literal can be preceded by '+' or '-' and be surrounded by whitespace. The base defaults to 10. Valid bases are 0 and 2-36. Base 0 means to interpret the base from the string as an integer literal. >>> int('0b100', base=0) 4

raw()
returns the raw symbolic name (ignore the ref attribute.)

toks(***kargs*)
returns list of pretty printing tokens of the expression

depth()
depth size of the expression tree

eval(*env*)
evalute expression in given `mapper` env

simplify(***kargs*)
simplify expression based on predefined heuristics

class cas.expressions.**tst**(*t*, *l*, *r*)
Conditional expression.

tst
the boolean expression that represents the condition.

Type *exp*

l
the resulting expression if test == bit1.

Type *exp*

r
the resulting expression if test == bit0.

Type *exp*

toks(***kargs*)
returns list of pretty printing tokens of the expression

eval(*env*)
evalute expression in given `mapper` env

simplify(***kargs*)
simplify expression based on predefined heuristics

depth()
depth size of the expression tree

cas.expressions.**oper**(*opsym*, *l*, *r=None*)
wrapper of the operator expression that detects unary operations

class cas.expressions.**op**(*op*, *l*, *r*)
op holds binary integer arithmetic and bitwise logic expressions

op
binary operator

Type _operator

prop
type of operator (ARITH, LOGIC, CONDT, SHIFT)

Type int

l
left-hand expression of the operator

Type *exp*

r
right-hand expression of the operator

Type *exp*

eval(*env*)
evalute expression in given `mapper` env

toks(***kargs*)
returns list of pretty printing tokens of the expression

simplify(***kargs*)
simplify expression based on predefined heuristics

depth()
depth size of the expression tree

class cas.expressions.**uop**(*op*, *r*)
uop holds unary integer arithmetic and bitwise logic expressions

op
unary operator

Type _operator

prop
type of operator (ARITH, LOGIC, CONDT, SHIFT)

Type int

l
returns None in case uop is treated as an op instance.

Type None

r
right-hand expression of the operator

Type *exp*

eval(*env*)
evalute expression in given `mapper` env

toks(***kargs*)
returns list of pretty printing tokens of the expression

simplify(***kargs*)
simplify expression based on predefined heuristics

depth()
depth size of the expression tree

cas.expressions.**ror**(*x*, *n*)
high-level rotate right n bits

cas.expressions.**rol**(*x*, *n*)
high-level rotate left n bits

cas.expressions.**ltu**(*x*, *y*)
high-level less-than-unsigned operation

cas.expressions.**geu**(*x*, *y*)
high level greater-or-equal-unsigned operation

cas.expressions.**symbols_of**(*e*)
returns all symbols contained in expression e

cas.expressions.**locations_of**(*e*)
returns all locations contained in expression e

cas.expressions.**complexity**(*e*)
evaluate the complexity of expression e

cas.expressions.**eqn1_helpers**(*e*, ***kargs*)
helpers for simplifying unary expressions

cas.expressions.**eqn2_helpers**(*e*, *bitslice=False*, *widening=False*)
helpers for simplifying binary expressions

cas.expressions.**extract_offset**(*e*)
separate expression e into (e' + C) with C cst offset.

class cas.expressions.**vec**(*l=None*)
vec holds a list of expressions each being a possible representation of the current expression. A vec object is obtained by merging several execution paths using the merge function in the mapper module. The simplify method uses the complexity measure to eventually "reduce" the expression to top with a hard-limit currently set to op.threshold.

toks(***kargs*)
returns list of pretty printing tokens of the expression

simplify(***kargs*)
simplify expression based on predefined heuristics

eval(*env*)
evalute expression in given `mapper` env

depth()
depth size of the expression tree

class cas.expressions.**vecw**(*v*)
vecw is a *widened* vec expression: it allows to limit the list of possible values to a fixed range and acts as a top (absorbing) expression.

toks(***kargs*)
returns list of pretty printing tokens of the expression

eval(*env*)
evalute expression in given `mapper` env

8.2 cas/smt.py

The smt module defines the amoco interface to the SMT solver. Currently, only z3 is supported. This module allows to translate any amoco expression into its z3 equivalent formula, as well as getting the z3 solver results back as *cas.mapper.mapper* instances.

cas.smt.**newvar**(*pfx*, *e*, *slv*)
return a new z3 BitVec of size e.size, with name prefixed by slv argument

cas.smt.**top_to_z3**(*e*, *slv=None*)
translate top expression into a new _topN BitVec variable

cas.smt.**cst_to_z3**(*e*, *slv=None*)
translate cst expression into its z3 BitVecVal form

cas.smt.**cfp_to_z3**(*e*, *slv=None*)
translate cfp expression into its z3 RealVal form

cas.smt.**reg_to_z3**(*e*, *slv=None*)
translate reg expression into its z3 BitVec form

cas.smt.**comp_to_z3**(*e*, *slv=None*)
translate comp expression into its z3 Concat form

cas.smt.**slc_to_z3**(*e*, *slv=None*)
translate slc expression into its z3 Extract form

cas.smt.**ptr_to_z3**(*e*, *slv=None*)
translate ptr expression into its z3 form

cas.smt.**mem_to_z3**(*e*, *slv=None*)
translate mem expression into z3 a Concat of BitVec bytes

cas.smt.**cast_z3_bool**(*x*, *slv=None*)
translate boolean expression into its z3 bool form

cas.smt.**cast_z3_bv**(*x*, *slv=None*)
translate expression x to its z3 form, if x.size==1 the returned formula is (If x ? 1 : 0).

cas.smt.**tst_to_z3**(*e*, *slv=None*)
translate tst expression into a z3 If form

cas.smt.**op_to_z3**(*e*, *slv=None*)
translate op expression into its z3 form

cas.smt.**uop_to_z3**(*e*, *slv=None*)
translate uop expression into its z3 form

cas.smt.**vec_to_z3**(*e*, *slv=None*)
translate vec expression into z3 Or form

cas.smt.**to_smtlib**(*e*, *slv=None*)
return the z3 smt form of expression e

cas.smt.**model_to_mapper**(*r*, *locs*)
return an amoco mapper based on given locs for the z3 model r

8.3 cas/mapper.py

The mapper module essentially implements the *mapper* class and the associated *merge()* function which allows to get a symbolic representation of the *union* of two mappers.

class cas.mapper.**mapper**(*instrlist=None*, *csi=None*)
A mapper is a symbolic functional representation of the execution of a set of instructions.

Parameters

- **instrlist** (*list[instruction]*) – a list of instructions that are symbolically executed within the mapper.
- **csi** (*Optional[object]*) – the optional csi attribute that provide a *concrete* initial state

__map
is an ordered list of mappings of expressions associated with a location (register or memory pointer). The order is relevant only to reflect the order of write-to-memory instructions in case of pointer aliasing.

__Mem
is a memory model where symbolic memory pointers are addressing separated memory zones. See MemoryMap and MemoryZone classes.

conds
is the list of conditions that must be True for the mapper

csi
is the optional interface to a *concrete* state

conds

csi

view

inputs()
list antecedent locations (used in the mapping)

outputs()
list image locations (modified in the mapping)

has(*loc*)
check if the given location expression is touched by the mapper

history(*loc*)

delayed(*k*, *v*)

update_delayed()

rw()
get the read sizes and written sizes tuple

clear()
clear the current mapper, reducing it to the identity transform

getmemory()
get the local `MemoryMap` associated to the mapper

setmemory(*mmap*)
set the local `MemoryMap` associated to the mapper

mmap
get the local `MemoryMap` associated to the mapper

generation()

R(*x*)
get the expression of register x

M(*k*)
get the expression of a memory location expression k

aliasing(*k*)
check if location k is possibly aliased in the mapper: i.e. the mapper writes to some other symbolic location expression after writing to k which might overlap with k.

update(*instr*)
opportunistic update of the self mapper with instruction

safe_update(*instr*)
update of the self mapper with instruction *only* if no exception occurs

restruct()

eval(*m*)
return a new mapper instance where all input locations have been replaced by there corresponding values in m.

rcompose(*m*)
composition operator returns a new mapper corresponding to function x -> self(m(x))

interact()

use(**args*, ***kargs*)
return a new mapper corresponding to the evaluation of the current mapper where all key symbols found in kargs are replaced by their values in all expressions. The kargs "size=value" allows for adjusting symbols/values sizes for all arguments. if kargs is empty, a copy of the result is just a copy of current mapper.

usemmap(*mmap*)
return a new mapper corresponding to the evaluation of the current mapper where all memory locations of the provided mmap are used by the current mapper.

assume(*conds*)

cas.mapper.**merge**(*m1*, *m2*, ***kargs*)
union of two mappers

CHAPTER 9

The system package

Modules of this package implement all classes that relate to operating system specific operations as well as userland stubs or high-level language structures.

Contents

9.1 system/core.py

This module defines all task/process core classes related to binary format and execution inherited by all system specific execution classes of the `amoco.system` package.

class `system.core.`**CoreExec**(*p*, *cpu=None*)

This class implements the base class for Task(s). CoreExec or Tasks are used to represent a memory mapped binary executable program, providing the generic instruction or data fetchers and the mandatory API for `amoco.emu` or `amoco.sa` analysis classes. Most of the `amoco.system` modules use this base class to implement a OS-specific Task class (see Linux/x86, Win32/x86, etc).

bin
the program executable format object. Currently supported formats are provided in *system.elf* (Elf32/64), *system.pe* (PE) and *system.utils* (HEX/SREC).

cpu
reference to the architecture cpu module, which provides a generic access to the PC() program counter and obviously the CPU registers and disassembler.

OS
optional reference to the OS associated to the child Task.

state
the `mapper` instance that represents the current state of the executable program, including mapping of registers as well as the `MemoryMap` instance that represents the virtual memory of the program.

read_data(*vaddr*, *size*)
fetch size data bytes at virtual address vaddr, returned as a list of items being either raw bytes or symbolic expressions.

read_instruction(*vaddr*, ***kargs*)
fetch instruction at virtual address vaddr, returned as an cpu.instruction instance or cpu.ext in case an external expression is found at vaddr or vaddr is an external symbol.

Raises MemoryError in case vaddr is not mapped, and returns None if disassembler fails to decode bytes at vaddr.

Note: Returning a cpu.ext expression means that this instruction starts an external stub function. It is the responsibility of the fetcher (emulator or analyzer) to eventually call the stub to modify the state mapper.

getx(*loc*, *size=8*, *sign=False*)
high level method to get the expressions value associated to left-value loc (register or address). The returned value is an integer if the expression is constant or a symbolic expression instance. The input loc is either a register string, an integer address, or associated expressions' instances. Optionally, the returned expression sign flag can be adjusted by the sign argument.

setx(*loc*, *val*, *size=0*)
high level method to set the expressions value associated to left-value loc (register or address). The value is possibly an integer or a symbolic expression instance. The input loc is either a register string, an integer address, or associated expressions' instances. Optionally, the size of the loc expression can be adjusted by the size argument.

get_int64(*loc*)
get 64-bit int expression of current state(loc)

get_uint64(*loc*)
get 64-bit unsigned int expression of current state(loc)

get_int32(*loc*)
get 32-bit int expression of current state(loc)

get_uint32(*loc*)
get 32-bit unsigned int expression of current state(loc)

get_int16(*loc*)
get 16-bit int expression of current state(loc)

get_uint16(*loc*)
get 16-bit unsigned int expression of current state(loc)

get_int8(*loc*)
get 8-bit int expression of current state(loc)

get_uint8(*loc*)
get 8-bit unsigned int expression of current state(loc)

class system.core.**DefineStub**(*obj*, *refname*, *default=False*)
decorator to define a stub for the given 'refname' library function.

class system.core.**BinFormat**
Base class for binary format API, just to define default attributes and recommended properties. See elf.py, pe.py and macho.py for example of child classes.

class system.core.**DataIO**(*f*)
This class simply wraps a binary file or a bytes string and implements both the file and bytes interface. It allows an input to be provided as files of bytes and manipulated as either a file or a bytes object.

system.core.**read_program**(*filename*)
Identifies the program header and returns an ELF, PE, Mach-O or DataIO.

Parameters **filename** (*str*) – the program to read.

Returns an instance of currently supported program format (ELF, PE, Mach-O, HEX, SREC)

class system.core.**DefineLoader**(*fmt*, *name=''*)
A decorator that allows to register a system-specific loader while it is implemented. All loaders are stored in the class global LOADERS dict.

Example

@DefineLoader('elf',elf.EM_386) def loader_x86(p):

...

Here, a reference to function loader_x86 is stored in LOADERS['elf'][elf.EM_386].

system.core.**load_program**(*f*, *cpu=None*)
Detects program format header (ELF/PE/Mach-O/HEX/SREC), and *maps* the program in abstract memory, loading the associated "system" (linux/win) and "arch" (x86/arm), based header informations.

Parameters **f** (*str*) – the program filename or string of bytes.

Returns a Task, ELF/PE (old CoreExec interfaces) or RawExec instance.

9.2 system/memory.py

This module defines all Memory related classes.

The main class of amoco's Memory model is *MemoryMap*. It provides a way to represent both concrete and abstract symbolic values located in the virtual memory space of a process. In order to allow addresses to be symbolic as well, the MemoryMap is organised as a collection of *MemoryZone*. A zone holds values located at addresses that are integer offsets related to a symbolic expression. A default zone with related address set to None holds values at concrete (virtual) addresses in every MemoryMap.

class system.memory.**MemoryMap**
Provides a way to represent concrete and abstract symbolic values located in the virtual memory space of a process. A MemoryMap is organised as a collection of *MemoryZone*.

_zones
dictionary of zones, keys are the related address expressions.

newzone(*label*)
creates a new memory zone with the given label related expression.

locate(*address*)
returns the memory object that maps the provided address expression.

reference(*address*)
returns a couple (rel,offset) based on the given address, an integer, a string or an expression allowing to find a candidate zone within memory.

read(*address*, *l*)
reads l bytes at address. returns a list of datadiv values.

write(*address*, *expr*, *endian=1*)
writes given expression at given (possibly symbolic) address. Default endianness is 'little'. Use endian=-1 to indicate big endian convention.

restruct()
optimize all zones to merge contiguous raw bytes into single mo objects.

grep(*pattern*)
find all occurences of the given regular expression in the raw bytes objects of all memory zones.

merge(*other*)
update this MemoryMap with a new MemoryMap, merging overlapping zones with values from the new map.

class system.memory.**MemoryZone**(*rel=None*)
A MemoryZone contains mo objects at addresses that are integer offsets related to a symbolic expression. A default zone with related address set to None holds values at concrete addresses in every *MemoryMap*.

Parameters **rel** (exp) – the relative symbolic expression, defaults to None.

rel
the relative symbolic expression, or None.

_map
the ordered list of mo objects of this zone.

range()
returns the lowest and highest addresses currently used by mo objects of this zone.

locate(*vaddr*)
if the given address is within range, return the index of the corresponding mo object in _map, otherwise return None.

read(*vaddr*, *l*)
reads l bytes starting at vaddr. returns a list of datadiv values, unmapped areas are returned as *bottom* exp.

write(*vaddr*, *data*)
writes data expression or bytes at given (offset) address.

addtomap(*z*)
add (possibly overlapping) *mo* object z to the _map, eventually adjusting other objects.

restruct()
optimize the zone to merge contiguous raw bytes into single mo objects.

shift(*offset*)
shift all mo objects by a given offset.

grep(*pattern*)
find all occurences of the given regular expression in the raw bytes objects of the zone.

class system.memory.**mo**(*vaddr*, *data*, *endian=1*)
A mo object essentially associates a datadiv with a memory offset, and provides methods to detect if an address

is located within this object, to read or write bytes at a given address. The offset is relative to the start of the *MemoryZone* in which the mo object is stored.

vaddr
a python integer that represents the offset within the memory zone that contains this memory object (mo).

data
the datadiv object located at this offset.

trim(*vaddr*)
if this mo contains data at given offset, cut out this data and points current object to this offset. Note that a trim is generally the result of data being overwritten by another mo.

read(*vaddr*, *l*)
returns the list of datadiv objects at given offset so that the total length is at most l, and the number of bytes missing if the total length is less than l.

write(*vaddr*, *data*)
updates current mo to reflect the writing of data at given offset and returns the list of possibly new mo objects to be inserted in the zone.

class system.memory.**datadiv**(*data*, *endian*)
A datadiv represents any data within memory, including symbolic expressions.

Parameters

- **data** – either a string of bytes or an amoco expression.
- **endian** – either [-1,1], used when data is any symbolic expression. 1 is for little-endian, -1 for big-endian.

val
the reference to the data object.

_is_raw
a flag indicating that the data object is a string of bytes.

cut(*l*)
cut out the first l bytes of the current data, keeping only the remaining part of the data.

setlen(*l*)
cut out trailing bytes of the current data, keeping only the first l bytes.

getpart(*o*, *l*)
returns a pair (result, counter) where result is a part of data of length at most l located at offset o (relative to the beginning of the data bytes), and counter is the number of bytes missing (l-len(result)) if the current data length is less than l.

setpart(*o*, *data*)
returns a list of contiguous datadiv objects that correspond to overwriting self with data at offset o (possibly extending the current datadiv length).

system.memory.**mergeparts**(*P*)
This function will detect every contiguous raw datadiv objects in the input list P, and will return a new list where these objects have been merged into a single raw datadiv object.

Parameters **P** (*list*) – input list of datadiv objects.

Returns the list after raw datadiv objects have been merged.

Return type list

9.3 system/structs.py

The system structs module implements classes that allow to easily define, encode and decode C structures (or unions) as well as formatters to print various fields according to given types like hex numbers, dates, defined constants, etc. This module extends capabilities of `struct` by allowing formats to include more than just the basic types and add *named* fields. It extends `ctypes` as well by allowing formatted printing and "non-static" decoding where the way a field is decoded depends on previously decoded fields.

Module `system.imx6` uses these classes to decode HAB structures and thus allow for precise verifications on how the boot stages are verified. For example, the HAB Header class is defined with:

```
@StructDefine("""
B :  tag
H :> length
B :  version
""")
class HAB_Header(StructFormatter):
    def __init__(self,data="",offset=0):
        self.name_formatter('tag')
        self.func_formatter(version=self.token_ver_format)
        if data:
            self.unpack(data,offset)
    @staticmethod
    def token_ver_format(k,x,cls=None):
        return highlight([(Token.Literal,"%d.%d"%(x>>4,x&0xf))])
```

Here, the *StructDefine* decorator is used to provide the definition of fields of the HAB Header structure to the HAB_Header class.

The *tag* *Field* is an unsigned byte and the *StructFormatter* utilities inherited by the class set it as a `name_formatter()` allow the decoded byte value from data to be represented by its constant name. This name is obtained from constants defined with:

```
with Consts('tag'):
    HAB_TAG_IVT = 0xd1
    HAB_TAG_DCD = 0xd2
    HAB_TAG_CSF = 0xd4
    HAB_TAG_CRT = 0xd7
    HAB_TAG_SIG = 0xd8
    HAB_TAG_EVT = 0xdb
    HAB_TAG_RVT = 0xdd
    HAB_TAG_WRP = 0x81
    HAB_TAG_MAC = 0xac
```

The *length* field is a bigendian short integer with default formatter, and the *version* field is an unsigned byte with a dedicated formatter function that extracts major/minor versions from the byte nibbles.

This allows to decode and print the structure from provided data:

```
In [3]: h = HAB_Header('\xd1\x00\x0a\x40')
In [4]: print(h)
[HAB_Header]
tag                 :HAB_TAG_IVT
length              :10
version             :4.0
```

class system.structs.**Consts**(*name*)

Provides a contextmanager to map constant values with their names in order to build the associated reverse-dictionary.

All revers-dict are stored inside the Consts class definition. For example if you declare variables in a Consts('example') with-scope, the reverse-dict will be stored in Consts.All['example']. When StructFormatter will lookup a variable name matching a given value for the attribute 'example', it will get Consts.All['example'][value].

Note: To avoid attribute name conflicts, the lookup is always prepended the stucture class name (or the 'alt' field of the structure class). Hence, the above 'tag' constants could have been defined as:

```
with Consts('HAB_header.tag'):
    HAB_TAG_IVT = 0xd1
    HAB_TAG_DCD = 0xd2
    HAB_TAG_CSF = 0xd4
    HAB_TAG_CRT = 0xd7
    HAB_TAG_SIG = 0xd8
    HAB_TAG_EVT = 0xdb
    HAB_TAG_RVT = 0xdd
    HAB_TAG_WRP = 0x81
    HAB_TAG_MAC = 0xac
```

Or the structure definition could have define an 'alt' attribute:

```
@StructDefine("""
B :  tag
H :> length
B :  version
""")
class HAB_Header(StructFormatter):
    alt = 'hab'
    [...]
```

in which case the variables could have been defined with:

```
with Consts('hab.tag'):
[...]
```

system.structs.**token_default_fmt**(*k*, *x*, *cls=None*)
The default formatter just prints value 'x' of attribute 'k' as a literal token python string

system.structs.**token_address_fmt**(*k*, *x*, *cls=None*)
The address formatter prints value 'x' of attribute 'k' as a address token hexadecimal value

system.structs.**token_constant_fmt**(*k*, *x*, *cls=None*)
The constant formatter prints value 'x' of attribute 'k' as a constant token decimal value

system.structs.**token_mask_fmt**(*k*, *x*, *cls=None*)
The mask formatter prints value 'x' of attribute 'k' as a constant token hexadecimal value

system.structs.**token_name_fmt**(*k*, *x*, *cls=None*)
The name formatter prints value 'x' of attribute 'k' as a name token variable symbol matching the value

system.structs.**token_flag_fmt**(*k*, *x*, *cls*)
The flag formatter prints value 'x' of attribute 'k' as a name token variable series of symbols matching the flag value

system.structs.**token_datetime_fmt**(*k*, *x*, *cls=None*)
The date formatter prints value 'x' of attribute 'k' as a date token UTC datetime string from timestamp value

class system.structs.**Field**(*ftype*, *fcount=0*, *fname=None*, *forder=None*, *falign=0*, *fcomment=''*)

A Field object defines an element of a structure, associating a name to a structure typename and a count. A count of 0 means that the element is an object of type typename, a count>0 means that the element is a list of objects of type typename of length count.

typename
name of a Structure type for this field.

Type str

count
A count of 0 means that the element is an object of type typename, a count>0 means that the element is a list of length count of objects of type typename

Type int=0

name
the name associated to this field.

Type str

type
getter for the type associated with the field's typename.

Type *StructFormatter*

comment
comment, useful for pretty printing field usage

Type str

order
forces the endianness of this field.

Type str

size()
number of bytes eaten by this field.

format()
format string that allows to struct.(un)pack the field as a string of bytes.

unpack(*data*, *offset=0*)
unpacks a data from given offset using the field internal byte ordering. Returns the object (if count is 0) or the list of objects of type typename.

get(*data*, *offset=0*)
returns the field name and the unpacked value for this field.

pack(*value*)
packs the value with the internal order and returns the byte string according to type typename.

format()
a (non-Raw)Field format is always returned as matching a finite-length string.

unpack(*data*, *offset=0*)
returns a (sequence of count) element(s) of its self.type

class system.structs.**RawField**(*ftype*, *fcount=0*, *fname=None*, *forder=None*, *falign=0*, *fcomment=''*)

A RawField is a Field associated to a *raw* type, i.e. an internal type matching a standard C type (u)int8/16/32/64, floats/double, (u)char. Contrarily to a generic Field which essentially forward the unpack call to its subtype, a RawField relies on the struct package to return the raw unpacked value.

format()
a (non-Raw)Field format is always returned as matching a finite-length string.

unpack(*data*, *offset=0*)
returns a (sequence of count) element(s) of its self.type

class system.structs.**VarField**(*ftype*, *fcount=0*, *fname=None*, *forder=None*, *falign=0*, *fcomment=''*)
A VarField is a RawField with variable length, associated with a termination condition that will end the unpack method. An instance of VarField has an infinite size() unless it has been unpacked with data.

format()
a (non-Raw)Field format is always returned as matching a finite-length string.

unpack(*data*, *offset=0*)
returns a (sequence of count) element(s) of its self.type

class system.structs.**CntField**(*ftype*, *fcount=0*, *fname=None*, *forder=None*, *falign=0*, *fcomment=''*)
A CntField is a RawField where the amount of elements to unpack is provided as first bytes, encoded as either a byte/word/dword.

format()
a (non-Raw)Field format is always returned as matching a finite-length string.

unpack(*data*, *offset=0*)
returns a (sequence of count) element(s) of its self.type

class system.structs.**StructDefine**(*fmt*, ***kargs*)
StructDefine is a decorator class used for defining structures by parsing a simple intermediate language input decorating a StructFormatter class.

class system.structs.**UnionDefine**(*fmt*, ***kargs*)
UnionDefine is a decorator class based on StructDefine, used for defining unions.

class system.structs.**StructCore**
StructCore is a ParentClass for all user-defined structures based on a StructDefine format. This class contains essentially the packing and unpacking logic of the structure.

Note: It is mandatory that any class that inherits from StructCore can be instanciated with no arguments.

class system.structs.**StructFormatter**
StructFormatter is the Parent Class for all user-defined structures based on a StructDefine format. It inherits the core logic from StructCore Parent and provides all formatting facilities to pretty print the structures based on wether the field is declared as a named constant, an integer of hex value, a pointer address, a string or a date.

Note: Since it inherits from StructCore, it is mandatory that any child class can be instanciated with no arguments.

class system.structs.**StructMaker**
The StructMaker class is a StructFormatter equipped with methods that allow to interactively define and adjust fields at some given offsets or when some given sample bytes match a given value.

system.structs.**StructFactory**(*name*, *fmt*, ***kargs*)
Returns a StructFormatter class build with name and format

system.structs.**UnionFactory**(*name*, *fmt*, ***kargs*)
Returns a StructFormatter (union) class build with name and format

exception system.structs.**StructureError**(*message*)

9.4 system/elf.py

The system elf module implements Elf classes for both 32/64bits executable format.

exception system.elf.**ElfError**(*message*)
ElfError is raised whenever Elf object instance fails to decode required structures.

class system.elf.**Elf**(*f*)
This class takes a DataIO object (ie an opened file of BytesIO instance) and decodes all ELF structures found in it.

entrypoints
list of entrypoint addresses.

Type list of int

filename
binary file name.

Type str

Ehdr
the ELF header structure.

Type *Ehdr*

Phdr
the list of ELF Program header structures.

Type list of Phdr

Shdr
the list of ELF Section header structures.

Type list of Shdr

dynamic
True if the binary wants to load dynamic libs.

Type Bool

basemap
base address for this ELF image.

Type int

functions
a list of function names gathered from internal definitions (if not stripped) and import names.

Type list

variables
a list of global variables' names (if found.)

Type list

getsize()
total file size of all the Program headers

getinfo(*target*)
target is either an address provided as str or int, or a symbol str searched in the functions dictionary.

Returns a triplet with:

- section index (0 is error, -1 is a dynamic call)

- offset into section (idem)
- base virtual address (0 for dynamic calls)

data(*target*, *size*)
returns 'size' bytes located at target virtual address

getfileoffset(*target*)
converts given target virtual address back to offset in file

readsegment(*S*)
returns segment S data padded to S.p_memsz

loadsegment(*S*, *pagesize=None*)
If S is of type PT_LOAD, returns a dict {base: bytes} indicating that segment data bytes (extended to pagesize boundary) need to be mapped at virtual base address. (Returns None if not a PT_LOAD segment.)

readsection(*sect*)
returns the given section data bytes from file.

checksec()
check for usual security features.

class system.elf.**IDENT**(*data=None*)

class system.elf.**Ehdr**(*data=None*)

class system.elf.**Shdr**(*data=None*, *offset=0*, *order=None*, *x64=False*)

class system.elf.**Sym**(*data=None*, *offset=0*, *order=None*, *x64=False*)

class system.elf.**Rel**(*data=None*, *offset=0*, *order=None*, *x64=False*)

class system.elf.**Rela**(*data=None*, *offset=0*, *order=None*, *x64=False*)

class system.elf.**Phdr**(*data=None*, *offset=0*, *order=None*, *x64=False*)

class system.elf.**Note**(*data=None*, *offset=0*, *order=None*, *x64=False*)

class system.elf.**Dyn**(*data=None*, *offset=0*, *order=None*, *x64=False*)

class system.elf.**Lib**(*data=None*, *offset=0*, *order=None*, *x64=False*)

9.5 system/pe.py

The system pe module implements the PE class which support both 32 and 64 bits executable formats.

exception system.pe.**PEError**(*message*)
PEError is raised whenever PE object instance fails to decode required structures.

class system.pe.**PE**(*data*)
This class takes a DataIO object (ie an opened file of BytesIO instance) and decodes all PE structures found in it.

data
a reference to the input data file/bytes object.

Type *DataIO*

entrypoints
list of entrypoint addresses.

Type list of int

filename
binary file name.

Type str

DOS
the DOS Header (only if present.)

Type *DOSHdr*,optional

NT
the PE header.

Type *COFFHdr*

Opt
the Optional Header

Type *OptionalHdr*

basemap
base address for this ELF image.

Type int

sections
list of PE sections.

Type list of SectionHdr

functions
a list of function names gathered from internal definitions (if not stripped) and import names.

Type list

variables
a list of global variables' names (if found.)

Type list

tls
the Thead local Storage table (or None.)

Type TlsTable

locate(*addr*, *absolute=False*)

returns a tuple with:

- the section that holds addr (rva or absolute), or 0 or None.
- the offset within the section (or addr or 0).

Note: If returned section is 0, then addr is within SizeOfImage, but is not found within any sections. Then offset is addr. If returned section is None, then addr is not mapped at all, and offset is set to 0.

---

getdata(*addr*, *absolute=False*)
get section bytes from given virtual address to end of mapped section.

loadsegment(*S*, *pagesize=0*, *raw=False*)
returns a dict {base: bytes} (or only bytes if optional arg raw is True,) indicating that section S data bytes (padded and extended to pagesize bounds) need to be mapped at virtual base address.

Note: If S is 0, returns base=0 and the first Opt.SizeOfHeaders bytes.

getfileoffset(*addr*)
converts given address back to offset in file

class system.pe.**DOSHdr**(*data=None*)

class system.pe.**COFFHdr**(*data=None*, *offset=0*)

class system.pe.**OptionalHdr**(*data=None*, *offset=0*)

class system.pe.**DataDirectory**(*data=None*, *offset=0*)

class system.pe.**SectionHdr**(*data=None*, *offset=0*)

class system.pe.**COFFRelocation**(*data=None*, *offset=0*)

class system.pe.**COFFLineNumber**(*data=None*, *offset=0*)

class system.pe.**StdSymbolRecord**(*data=None*, *offset=0*)

class system.pe.**AuxSymbolRecord**(*data=None*, *offset=0*)

class system.pe.**AuxFunctionDefinition**(*data=None*, *offset=0*)

class system.pe.**Aux_bf_ef**(*data=None*, *offset=0*)

class system.pe.**AuxWeakExternal**(*data=None*, *offset=0*)

class system.pe.**AuxFile**(*data=None*, *offset=0*)

class system.pe.**AuxSectionDefinition**(*data=None*, *offset=0*)

class system.pe.**AttributeCertificate**

class system.pe.**DelayLoadDirectoryTable**(*data=None*, *offset=0*)

class system.pe.**ExportTable**(*data=None*, *offset=0*)

class system.pe.**ImportTableEntry**(*data=None*, *offset=0*)

class system.pe.**TLSTable**(*data*, *magic*)

class system.pe.**LoadConfigTable**(*data*, *magic*)

9.6 system/macho.py

The system macho module implements the Mach-O executable format parser.

exception system.macho.**MachOError**(*message*)
MachOError is raised whenever MachO object instance fails to decode required structures.

class system.macho.**MachO**(*f*)
This class takes a DataIO object (ie an opened file of BytesIO instance) and decodes all Mach-O structures found in it.

entrypoints
list of entrypoint addresses.

Type list of int

filename
binary file name.

Type str

header
the Mach header structure.

Type *struct_mach_header*

archs
the list of MachO instances in case the provided binary file is a "fat" format.

Type list of MachO

cmds
the list of all "command" structures.

Type list

dynamic
True if the binary wants to load dynamic libs.

Type Bool

basemap
Base address of the binary (or None.)

Type int

symtab
the symbol table.

Type list

dysymtab
the dynamic symbol table.

Type list

dyld_info
a container with dyld_info attributes rebase, bind, weak_bind, lazy_bind and export.

Type container

function_starts
list of function start addresses.

Type list,optional

la_symbol_ptr
address to lazy symbol bindings

Type dict

nl_symbol_ptr
address to non-lazy symbol bindings

Type dict

read_fat_arch(*a*)
takes a struct_fat_arch instance and sets its 'bin' attribute to the corresponding MachO instance.

read_commands(*offset*)
returns the list of struct_load_command starting from given offset

getsize()
total size of LC_SEGMENT/64 commands

getinfo(*target*)
getinfo return a triplet (s,off,vaddr) with segment, offset into segment, and segment virtual base address that contains the target argument.

checksec()
check for usual OSX security features.

data(*target*, *size*)
returns 'size' bytes located at target virtual address

getfileoffset(*target*)
converts given target virtual address back to offset in file

readsegment(*S*)
returns data of segment/section S

loadsegment(*S*, *pagesize=None*)
returns padded & aligned data of segment/section S

readsection(*sect*)
returns the segment/section data bytes matching given sect name

getsection(*sect*)
returns the segment/section matching given sect name

class system.macho.**struct_fat_header**(*data=None*)

class system.macho.**struct_fat_arch**(*data=None*, *offset=0*)

class system.macho.**struct_mach_header**(*data=None*)

class system.macho.**struct_mach_header_64**(*data=None*)

class system.macho.**struct_load_command**(*data=None*, *offset=0*)

class system.macho.**MachoFormatter**

class system.macho.**struct_segment_command**(*data=None*, *offset=0*)

class system.macho.**struct_segment_command_64**(*data=None*, *offset=0*)

class system.macho.**SFLG**(*data=None*, *offset=0*)

class system.macho.**struct_section**(*data=None*, *offset=0*)

class system.macho.**struct_section_64**(*data=None*, *offset=0*)

class system.macho.**lc_str**(*data=None*, *offset=0*)

class system.macho.**struct_fvmlib**(*data=None*, *offset=0*)

class system.macho.**struct_fvmlib_command**(*data=''*, *offset=0*)

class system.macho.**struct_dylib**(*data=''*, *offset=0*)

class system.macho.**struct_dylib_command**(*data=''*, *offset=0*)

class system.macho.**struct_sub_framework_command**(*data=''*, *offset=0*)

class system.macho.**struct_sub_client_command**(*data=''*, *offset=0*)

class system.macho.**struct_sub_umbrella_command**(*data=''*, *offset=0*)

class system.macho.**struct_sub_library_command**(*data=''*, *offset=0*)

class system.macho.**struct_prebound_dylib_command**(*data=''*, *offset=0*)

class system.macho.**struct_dylinker_command**(*data=''*, *offset=0*)

class system.macho.**struct_thread_command**(*data=''*, *offset=0*)

class system.macho.**struct_x86_thread_state32**(*data=''*, *offset=0*)

class system.macho.**struct_x86_thread_state64**(*data=''*, *offset=0*)

class system.macho.**struct_arm_thread_state32**(*data=''*, *offset=0*)

class system.macho.**struct_arm_thread_state64**(*data=''*, *offset=0*)

class system.macho.**struct_routines_command**(*data=''*, *offset=0*)

class system.macho.**struct_routines_command_64**(*data=''*, *offset=0*)

class system.macho.**struct_symtab_command**(*data=''*, *offset=0*)

class system.macho.**struct_nlist**(*data=''*, *offset=0*)

class system.macho.**struct_nlist64**(*data=''*, *offset=0*)

class system.macho.**struct_dysymtab_command**(*data=''*, *offset=0*)

class system.macho.**struct_dylib_table_of_contents**(*data=''*, *offset=0*)

class system.macho.**struct_dylib_module**(*data=''*, *offset=0*)

class system.macho.**struct_dylib_module_64**(*data=''*, *offset=0*)

class system.macho.**struct_dylib_reference**(*data=''*, *offset=0*)

class system.macho.**struct_twolevel_hints_command**(*data=''*, *offset=0*)

class system.macho.**twolevel_hint**(*data=''*, *offset=0*)

class system.macho.**struct_prebind_cksum_command**(*data=''*, *offset=0*)

class system.macho.**struct_uuid_command**(*data=''*, *offset=0*)

class system.macho.**struct_rpath_command**(*data=''*, *offset=0*)

class system.macho.**struct_linkedit_data_command**(*data=''*, *offset=0*)

class system.macho.**struct_encryption_info_command**(*data=''*, *offset=0*)

class system.macho.**struct_dyld_info_command**(*data=''*, *offset=0*)

class system.macho.**struct_symseg_command**(*data=''*, *offset=0*)

class system.macho.**struct_ident_command**(*data=''*, *offset=0*)

class system.macho.**struct_fvmfile_command**(*data=''*, *offset=0*)

class system.macho.**struct_entry_point_command**(*data=''*, *offset=0*)

class system.macho.**struct_data_in_code_entry**(*data=''*, *offset=0*)

class system.macho.**struct_note_command**(*data=''*, *offset=0*)

class system.macho.**struct_source_version_command**(*data=''*, *offset=0*)

class system.macho.**struct_version_min_command**(*data=''*, *offset=0*)

class system.macho.**struct_build_version_command**(*data=''*, *offset=0*)

class system.macho.**struct_build_tool_version**(*data=''*, *offset=0*)

class system.macho.**struct_relocation_info**(*data=''*, *offset=0*)

class system.macho.**struct_indirect_entry**(*data=''*, *offset=0*)

class system.raw.**RawExec**(*p*, *cpu=None*)

9.7 system/utils.py

The system utils module implements various binary file format like Intel HEX or Motorola SREC, commonly used for programming MCU, EEPROMs, etc.

exception system.utils.**FormatError**(*message*)

class system.utils.**HEX**(*f*, *offset=0*)

class system.utils.**SREC**(*f*, *offset=0*)

CHAPTER 10

The static analysis package

CHAPTER 11

The user interface package

CHAPTER 12

code.py

This module defines classes that represent assembly instructions blocks, functions, and calls to *external* functions. In amoco, such objects are found as `node.data` in nodes of a *cfg.graph*. As such,they all provide a common API with:

- `address` to identify and locate the object in memory
- `support` to get the address range of the object
- `view` to display the object

class code.**block**(*instrlist*)

A block instance holds a sequence of instructions.

Parameters **instr** (*list[instruction]*) – the sequence of continuous (ordered) instructions

instr

the list of instructions of the block.

Type list

view

holds the `ui.views` object used to display the block.

Type `blockView`

length

the byte length of the block instructions sequence.

Type int

support

the memory footprint of the block

Type tuple

address

the address of the first instruction in the block.

Type address (`cst`)

cut(*address*)
cutting the block at given address will remove instructions after this address, (which needs to be aligned with instructions boundaries.) The effect is thus to reduce the block size.

Parameters **address** (cst) – the address where the cut occurs.

Returns the number of instructions removed from the block.

Return type int

raw()
returns the *raw* bytestring of the block instructions.

class code.**func**(*g=None*)
A graph of blocks that represents a function's Control-Flow-Graph (CFG).

Parameters **g** (*graph_core*) – the connected graph component of nodes.

cfg
the graph_core CFG of the function (see *cfg*.)

Type graph_core

blocks
the list of blocks in the CFG

Type list[*block*]

support
the memory footprint of the function

Type tuple

blocks
the list of blocks within the function.

Type blocks (list)

class code.**tag**
defines keys as class attributes, used in misc attributes to indicate various relevant properties of blocks within functions.

classmethod list()
get the list of all defined keys

classmethod sig(*name*)
symbols for tag keys used to compute the block's signature

CHAPTER 13

cfg.py

This module provides elements to define *control flow graphs* (CFG). It is based essentially on classes provided by the grandalf package.

class cfg.**node**(*acode*)

A node is a graph vertex that embeds a *code* object. It extends the Vertex class in order to compare nodes by their data blocks rather than their id.

Parameters **acode** – an instance of block, func or xfunc.

data

the reference to the acode argument above.

e

inherited from grandalf, the list of edges with this node. In amoco, edges and vertices are called links and nodes.

Type list[*link*]

c

reference to the connected component that contains this node.

Type graph_core

view

the block or func view object associated with our data.

map

the map object associated with out data.

Type *mapper*

cut(*address*)

reduce the block size up to given address if data is block.

deg()

returns the *degree* of this node (number of its links).

N(*dir=0*)

provides a list of *neighbor* nodes, all if *dir* parameter is 0, parent nodes if *dir<0*, children nodes if *dir>0*.

e_dir(*dir=0*)
provides a list of *links*, all if *dir* parameter is 0, incoming links if *dir<0*, outgoing links if *dir>0*.

e_in()
a shortcut for `e_dir(-1)`.

e_out()
a shortcut for `e_dir(+1)`.

e_with(*v*)
provides a *link* to or from v. Should be used with caution: if there is several links between current node and v this method gives the first one listed only, independently of the direction.

e_to(*v*)
provides the *link* from current node to node v.

e_from(*v*)
provides the *link* to current node from node v.

view
view property of the node's code object.

Type view

class cfg.**link**(*x*, *y*, *w=1*, *data=None*, *connect=False*)
A directed edge between two nodes. It extends Edge class in order to compare edges based on their data rather than id.

Parameters

- **x** (node) – the source node.
- **y** (node) – the destination node.
- **w** (*int*) – an optional weight value, default 1.
- **data** – a list of conditional expressions associated with the link.
- **connect** – a flag to indicate that a new node should be automatically added to the connected component of its parent/child if it is defined (default False).

name
the name property returns the string composed of source and destination node's *addresses*.

deg
1 if source and destination are the same node, 0 otherwise.

Type int

v
inherited from grandalf, the 2-tuple (source,dest) nodes of the link.

Type tuple[*node*]

feedback
a flag indicating that this link is involved in a loop, used internally by grandalf layout algorithm.

attach()
add current link to its *node.e* attribute list.

detach()
remove current link from its *node.e* attribute list.

class cfg.**graph**(**args*, ***kargs*)
a <grandalf:Graph> that represents a set of functions as its individual connected components.

Parameters

- **V** (*iterable[node]*) – the set of (possibly detached) nodes.
- **E** (*iterable[link]*) – the set of links of this graph.

C
the list of `graph_core` connected components of the graph.

support
the abstract memory zone holding all nodes contained in this graph.

Type `MemoryZone`

overlay
defaults to None, another instance of MemoryZone with nodes of the graph that overlap other nodes already mapped in *support*.

get_by_name(*name*)
get the node with the given name (as string).

get_with_address(*vaddr*)
get the node that contains the given *vaddr* *cst* expression.

add_vertex(*v*[, *support=None*])
add node v to the graph and declare node support in the default MemoryZone or the overlay zone if provided as support argument. This method deals with a node v that cuts or swallows a previously added node.

remove_vertex(*v*)
remove node v from the graph.

add_edge(*e*)
add link to the graph as well as possible new nodes.

remove_edge(*e*)
remove the provided link.

get_vertices_count()
a synonym for *order()*.

V()
generator of all nodes of the graph.

E()
generator of all links of the graph.

N(*v*, *f_io=0*)
returns the neighbors of node v in direction f_io.

path(*x*, *y*, *f_io=0*, *hook=None*)

order()
number of nodes in the graph.

norm()
number of links in the graph.

deg_min()
minimum degree of nodes.

deg_max()
maximum degree of nodes.

deg_avg()
average degree of nodes.

eps()
ratio of links over nodes (norm/order).

connected()
boolean flag indicating that the graph as only one connected component.

components()
synonym for attribute *C*.

CHAPTER 14

db.py

This module implements all amoco's database facilities using the sqlalchemy package, allowing to store many analysis results and pickled objects.

db.**createdb**(*url=None*)

creates the database engine and bind it to the scoped Session class. The database URL (see `config.py`) is opened and the schema is created if necessary. The default URL uses *sqlite* dialect and opens a temporary file for storage.

CHAPTER 15

config.py

This module defines the default amoco configuration and loads any user-defined configuration file. It is based on the traitlets package.

`config.`**`conf`**
holds in a Config object based on Configurable traitlets, various parameters mostly related to how outputs should be formatted.

The defined configurable sections are:

- 'Code' which deals with how basic blocks are printed, with options:
  - 'helper' will use codeblock helper functions to pretty print code if True (default)
  - 'header' will show a dashed header line including the address of the block if True (default)
  - 'footer' will show a dashed footer line if True
  - 'segment' will show memory section/segment name in codeblock view if True (default)
  - 'bytecode' will show the hex encoded bytecode string of every instruction if True (default)
  - 'padding' will add the specified amount of blank chars to between address/bytecode/instruction (default 4).
  - 'hist' number of instruction's history shown in emulator view (default 3).
- 'Cas' which deals with parameters of the algebra system:
  - 'noaliasing' will assume that mapper's memory pointers are not aliased if True (default)
  - 'complexity' threshold for expressions (default 100). See *cas.expressions* for details.
  - 'memtrace' store memory writes as mapper items if True (default).
  - 'unicode' will use math unicode symbols for expressions operators if True (default False).
- 'DB' which deals with database backend options:
  - 'url' allows to define the dialect and/or location of the database (default to sqlite)
  - 'log' indicates that database logging should be redirected to the amoco logging handlers

- 'Log' which deals with logging options:
  - 'level' one of 'ERROR' (default), 'VERBOSE', 'INFO', 'WARNING' or 'DEBUG' from less to more verbose,
  - 'tempfile' to also save DEBUG logs in a temporary file if True (default is False),
  - 'filename' to also save DEBUG logs using this filename.
- 'UI' which deals with some user-interface pretty-printing options:
  - 'formatter' one of 'Null' (default), 'Terminal', "Terminal256', 'TerminalDark', 'TerminalLight', 'Html'
  - 'graphics' one of 'term' (default), 'qt' or 'gtk'
  - 'console' one of 'python' (default), or 'ipython'
  - 'unicode' will use unicode symbols for drawing lines and icons if True
- 'Server' which deals with amoco's server parameters:
  - 'wbsz' sets the size of the server's internal shared memory buffer with spawned commands
  - 'timeout' sets the servers's internal timeout for the connection with spawned commands
- 'Emu' which deals with amoco's emulator parameters:
  - 'hist' defines the size of the emulator's instructions' history list (defaults to 100.)
- 'Arch' which allows to configure assembly format parameters:
  - 'assemble' (unused)
  - 'format_x86' one of 'Intel' (default), 'ATT'
  - 'format_x64' one of 'Intel' (default), 'ATT'

Type *Config*

class config.**DB**(**kwargs*)

Configurable parameters related to the database.

url

defaults to sqlite:// (in-memory database).

Type str

log

If True, merges database's logs into amoco loggers.

Type Bool

class config.**Code**(**kwargs*)

Configurable parameters related to assembly blocks (code.block).

helper

use block helpers if True.

Type Bool

header

display block header dash-line with its name if True.

Type Bool

footer
display block footer dash-line if True.

Type Bool

segment
display memory section/segment name if True.

Type Bool

bytecode
display instructions' bytes.

Type Bool

padding
add space-padding bytes to bytecode (default=4).

Type int

hist
number of history instructions to show in emulator's code frame view.

Type int

class config.**Cas**(***kwargs*)
Configurable parameters related to the Computer Algebra System (expressions).

complexity
limit expressions complexity to given value. Defaults to 10000, a relatively high value that keeps precision but can lead to very large expressions.

Type int

unicode
use unicode character for expressions' operators if True.

Type Bool

noaliasing
If True (default), then assume that symbolic memory expressions (pointers) are **never** aliased.

Type Bool

memtrace
keep memory writes in mapper in addition to MemoryMap (default).

Type Bool

class config.**Log**(***kwargs*)
Configurable parameters related to logging.

level
terminal logging level (defaults to 'INFO'.)

Type str

filename
if not "" (default), a filename receiving VERBOSE logs.

Type str

tempfile
log at VERBOSE level to a temporary tmp/ file if True.

Type Bool

Note: observers for Log traits are defined in the amoco.logger module (to avoid module cyclic imports.)

class config.**UI**(***kwargs*)
Configurable parameters related to User Interface(s).

formatter
pygments formatter for pretty printing. Defaults to Null, but recommended to be set to 'Terminal256' if pygments package is installed.

Type str

graphics
rendering backend. Currently only 'term' is supported.

Type str

console
default python console, either 'python' (default) or 'ipython'.

Type str

completekey
client key for command completion (Tab).

Type str

cli
client frontend. Currently only 'cmdcli' is supported.

Type str

class config.**Server**(***kwargs*)
Configurable parameters related to the Server mode.

wbsz
size of the shared buffer between server and its command threads.

Type int

timeout
timeout for the servers' command threads.

Type int

class config.**Arch**(***kwargs*)
Configurable parameters related to CPU architectures.

assemble
unused yet.

Type Bool

format_x86
select disassembly flavor: Intel (default) vs. AT&T (att).

Type str

format_x64
select disassembly flavor: Intel (default) vs. AT&T (att).

Type str

class config.**Emu**(***kwargs*)
Configurable parameters related to the amoco.emu module.

hist
size of the emulated instruction history list (defaults to 100.)

Type int

class config.**System**(**kwargs*)
Configurable parameters related to the system sub-package.

pagesize
provides the default memory page size in bytes.

Type int

aslr
simulates ASLR if True. (not supported yet.)

Type Bool

nx
unused.

Type Bool

class config.**Config**(*f=None*)
A Config instance takes an optional filename argument or looks for .amoco/config or .amocorc files to load a traitlets.config.PyFileConfigLoader used to adjust UI, DB, Code, Arch, Log, Cas, System, and Server parameters.

Note: The Config object supports a print() method to display the entire configuration.

CHAPTER 16

logger.py

This module defines amoco logging facilities. The `Log` class inherits from a standard `logging.Logger`, with minor additional features like a `'VERBOSE'` level introduced between `'INFO'` and `'DEBUG'` levels, and a progress method that can be useful for time consuming activities. See below for details.

Most amoco modules start by creating their local `logger` object used to provide various feedback. Users can thus focus on messages from selected amoco modules by adjusting their level independently, or use the `set_quiet()`, `set_debug()` or `set_log_all(level)` functions to adjust all loggers at once.

Examples

Setting the mapper module to `'VERBOSE'` level:

```
In [1]: import amoco
In [2]: amoco.cas.mapper.logger.setlevel('VERBOSE')
```

Setting all modules loggers to `'ERROR'` level:

```
In [2]: amoco.logger.set_quiet()
```

Note: All loggers can be configured to log both to *stderr* with selected level and to a unique temporary file with `'DEBUG'` level. See configuration.

class logger.**Log**(*name*, *handler=<StreamHandler <stderr> (NOTSET)>*)

This class is intended to allow amoco activities to be logged simultaneously to the *stderr* output with an adjusted level and to a temporary file with full verbosity.

All instanciated Log objects are tracked by the Log class attribute `Log.loggers` which maps their names with associated instances.

The recommended way to create a Log object is to add, near the begining of amoco modules:

```
from amoco.logger import Log
logger = Log(__name__)
```

setLevel(*lvl*)
Set the logging level of this logger. level must be an int or a str.

logger.**set_quiet**()
set all loggers to 'ERROR' level

logger.**set_debug**()
set all loggers to 'DEBUG' level

logger.**set_log_all**(*level*)
set all loggers to specified level

Parameters **level** (*int*) – level value as an integer.

logger.**reset_log_file**(*filename*, *level=10*)
set DEBUG log file for all loggers.

Parameters **filename** (*str*) – filename for the FileHandler added to all amoco loggers

CHAPTER 17

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

c

d

l

s

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Z