x86-64 Instruction Decoder

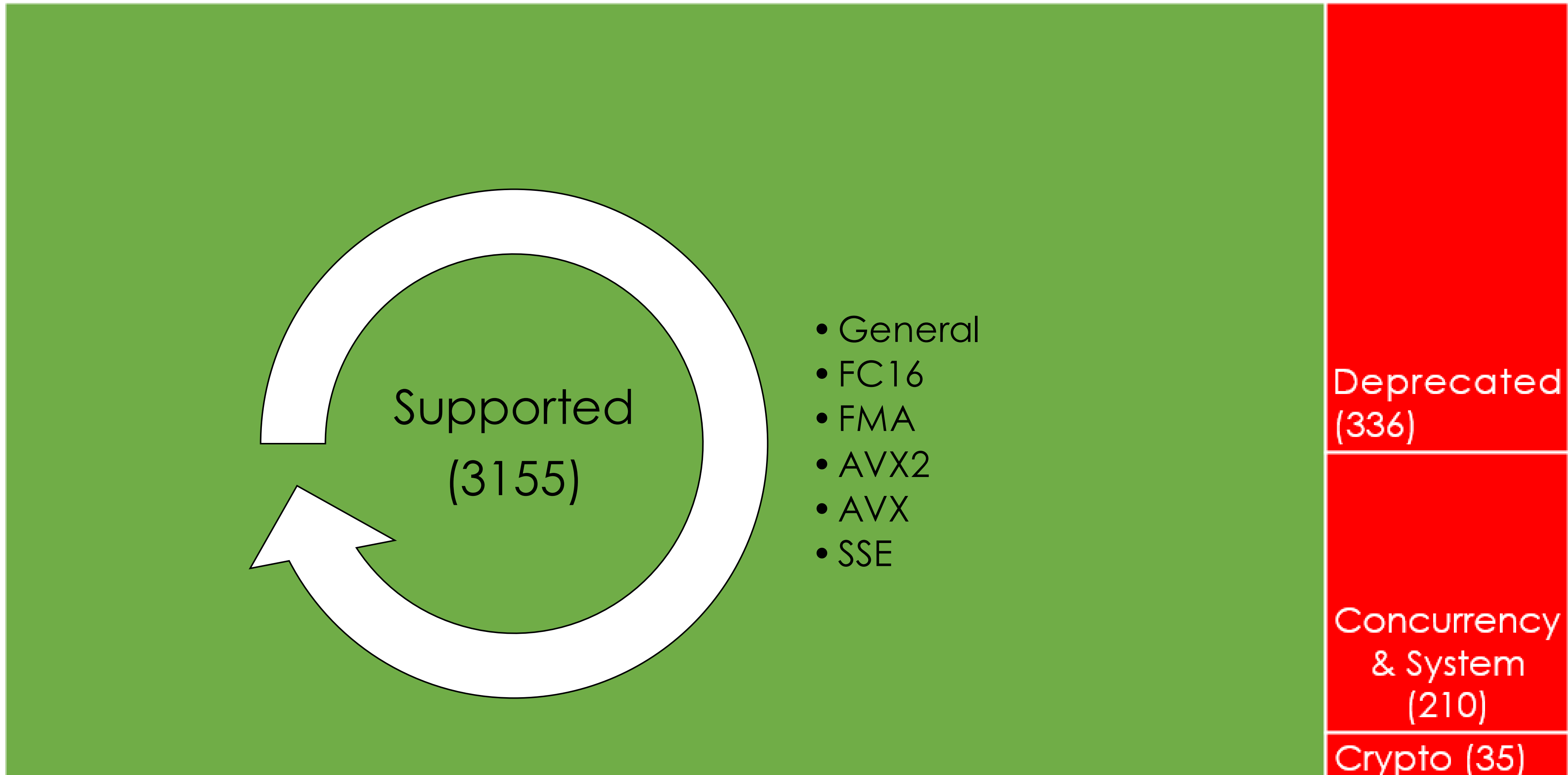
Andrew Miranti ● Sandeep Dasgupta ● Grigore Rosu
University of Illinois at Urbana Champaign
13 September 2019 @ SpISA'19

Prior Work: x86-64 Semantics [PLDI'19]

We defined the **most complete** and **thoroughly tested** formal semantics of **user-level** x86-64 ISA

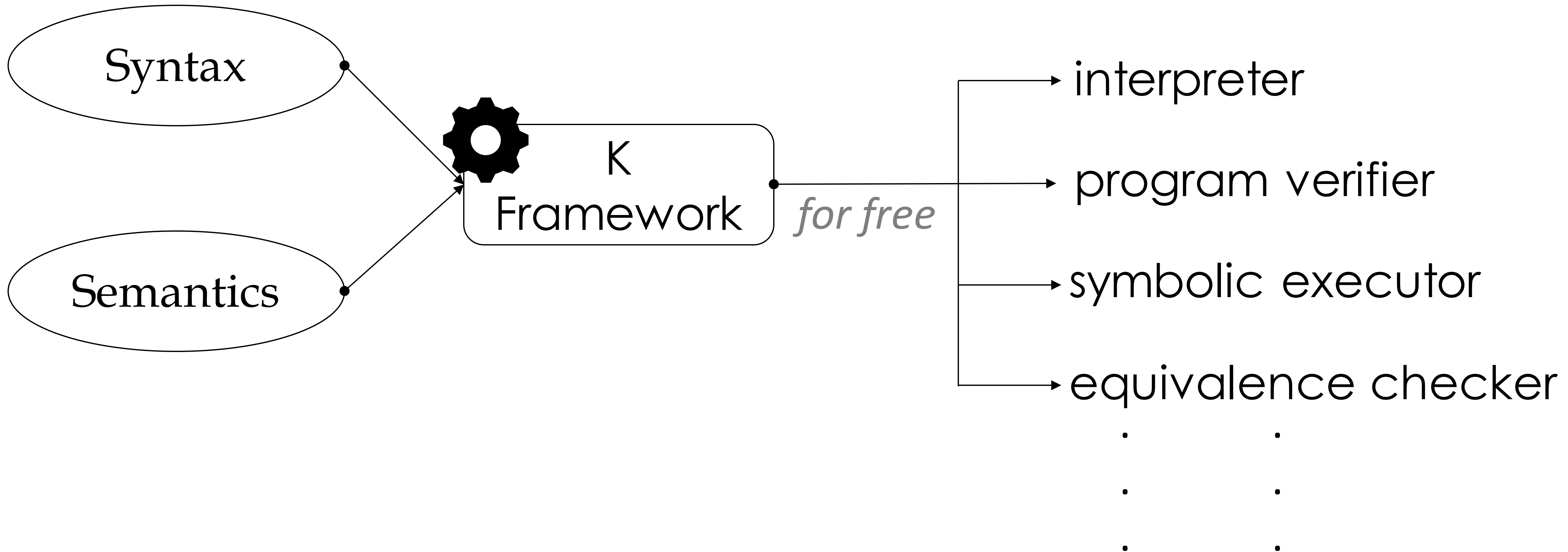
github.com/kframework/X86-64-semantics

Scope of Work (3155 / 3736)



Based on: K-Framework [Rosu et al. 2010]

Language semantics engineering framework (kframework.org)



Approach Overview

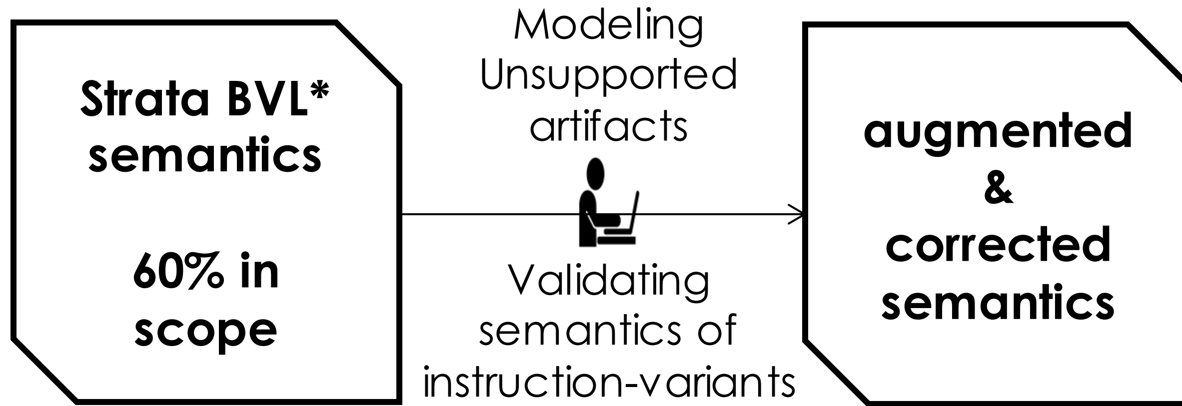
Approach Overview

Strata BVL*
semantics

60% in
scope

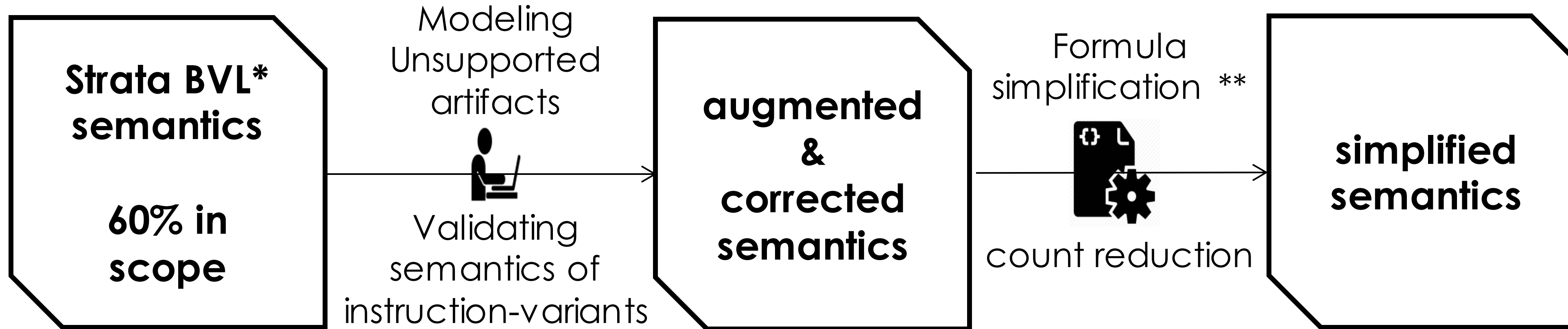
** BVL: Bit-vector logic*

Approach Overview



** BVL: Bit-vector logic*

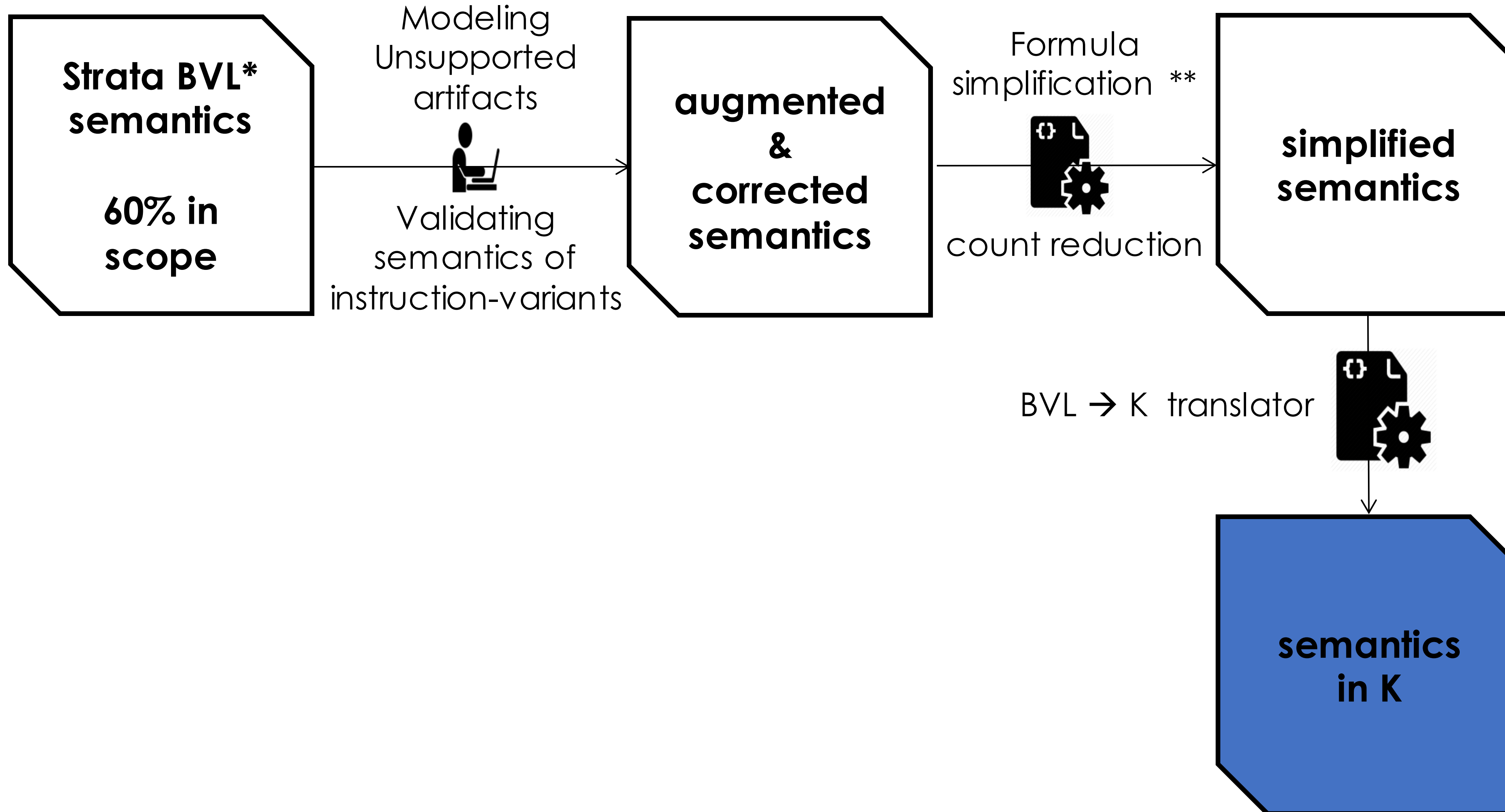
Approach Overview



* *BVL: Bit-vector logic*

** *30+ simplification rules. BVL formula of shrxl with 8971 terms simplified to 7 terms*

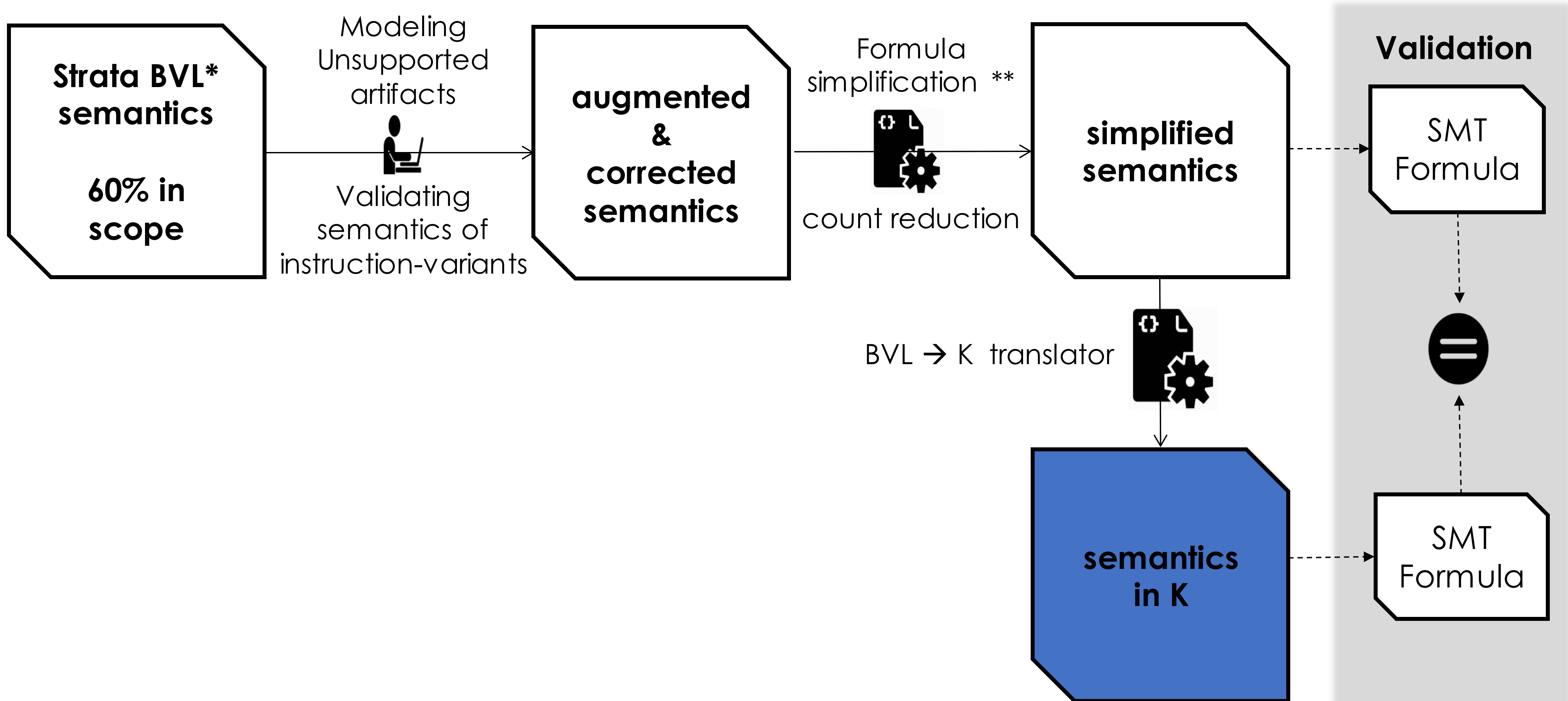
Approach Overview



BVL: Bit-vector logic

*** 30+ simplification rules. BVL formula of shrxl with 8971 terms simplified to 7 terms*

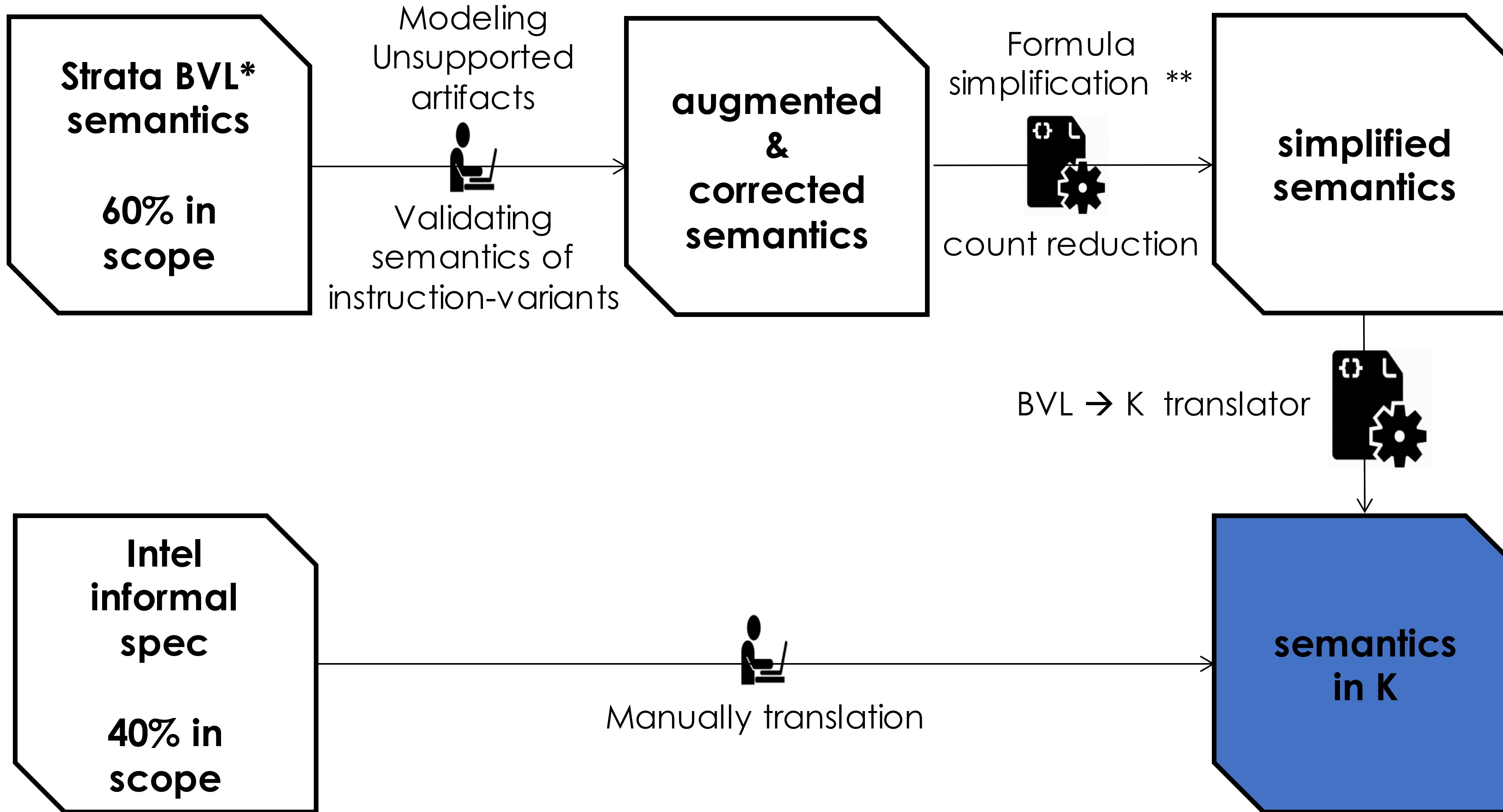
Approach Overview



* BVL: Bit-vector logic

** 30+ simplification rules. BVL formula of shrxl with 8971 terms simplified to 7 terms

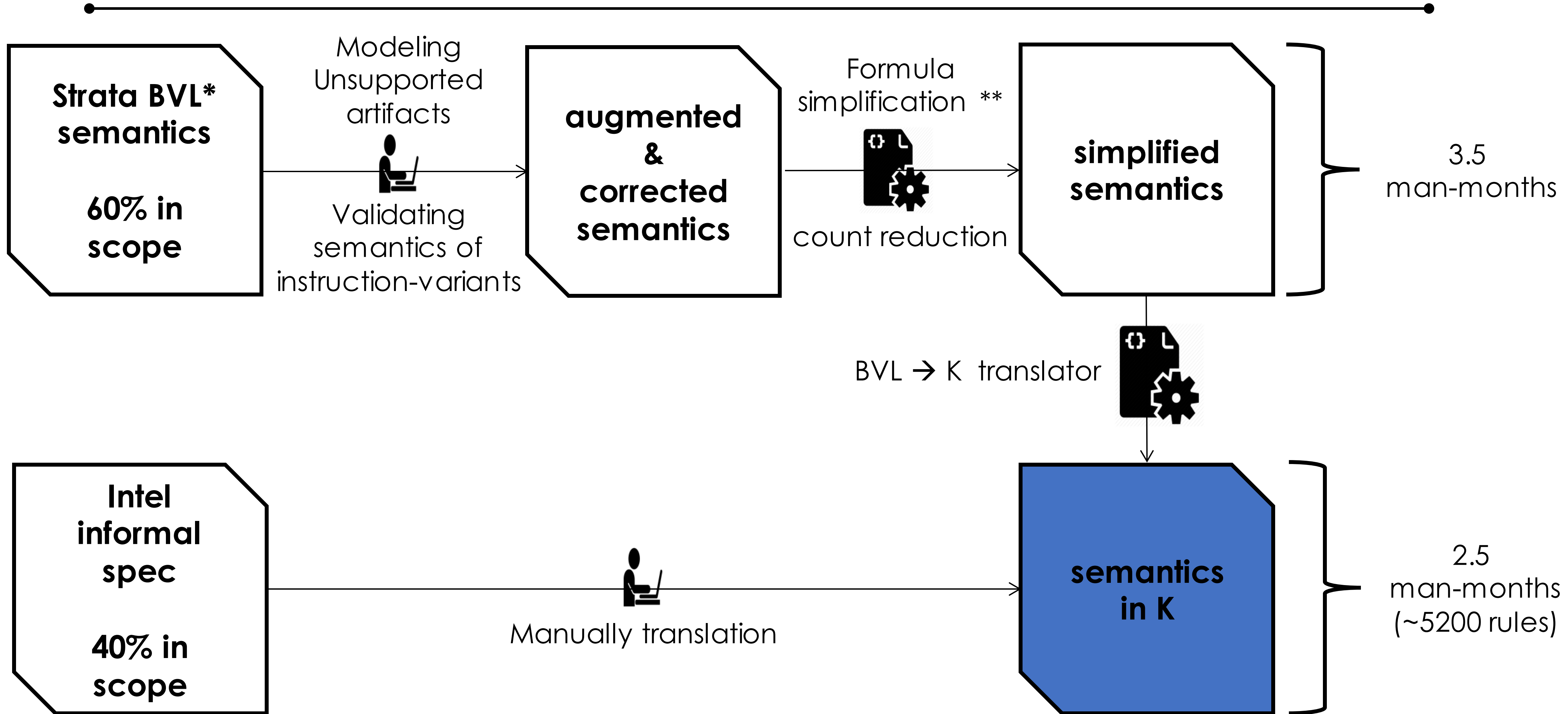
Approach Overview



BVL: Bit-vector logic

** *30+ simplification rules. BVL formula of shrxl with 8971 terms simplified to 7 terms*

Approach Overview



** BVL: Bit-vector logic*

*** 30+ simplification rules. BVL formula of shrxl with 8971 terms simplified to 7 terms*

Validation of Semantics

Comparing with hardware

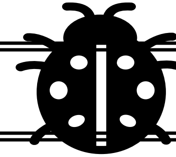
- Instruction Level Testing (7000+ inputs states)
- Program Level Testing (GCC-c torture tests)

Comparing with Stoke

- Comparing SMT formula

12+ Bugs reported

- *Intel Manual*
- Strata formulas



40+ Bugs reported In Stoke

A Few Reported Bugs

Intel Manual Vol. 2: March 2018

VPSRAVD (VEX.128 version)
COUNT_0 ← SRC2[31 : 0]
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[100 : 96];
DEST[31:0] ← SignExtend(SRC1[31:0] >> COUNT_0);
(* Repeat shift operation for 2nd through 4th dwords *)
DEST[127:96] ← SignExtend(SRC1[127:96] >> COUNT_3);
DEST[MAXVL-1:128] ← 0;

Intel Manual Vol. 2: May 2019

VPSRAVD (VEX.128 version)
COUNT_0 ← SRC2[31 : 0]
(* Repeat Each COUNT_i for the 2nd through 4th dwords of SRC2*)
COUNT_3 ← SRC2[127 : 96];
DEST[31:0] ← SignExtend(SRC1[31:0] >> COUNT_0);
(* Repeat shift operation for 2nd through 4th dwords *)
DEST[127:96] ← SignExtend(SRC1[127:96] >> COUNT_3);
DEST[MAXVL-1:128] ← 0;

A Few Reported Bugs

Stoke Implementation May 2018

VCVTSI2SD (VEX.128 encoded version)
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] ←Convert_Integer_To_Double_Precision_Floating_Point(SRC2[63:0]);
ELSE
    DEST[63:0] ←Convert_Integer_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] ←(Unmodified)

Intel Manual Vol. 2: May 2019

VCVTSI2SD (VEX.128 encoded version)
IF 64-Bit Mode And OperandSize = 64
THEN
    DEST[63:0] ←Convert_Integer_To_Double_Precision_Floating_Point(SRC2[63:0]);
ELSE
    DEST[63:0] ←Convert_Integer_To_Double_Precision_Floating_Point(SRC2[31:0]);
FI;
DEST[127:64] ←SRC1[127:64]

A Few Reported Bugs

Stoke Implementation May 2018

```
PSLLD (with 64-bit operand)
    IF (COUNT > 31)
    THEN
        DEST[64:0] ← 0000000000000000H;
    ELSE
        DEST[31:0] ← ZeroExtend(DEST[31:0] << COUNT [31:0]);
        DEST[63:32] ← ZeroExtend(DEST[63:32] << COUNT [63:32]);
    FI;
```

Intel Manual Vol. 2: May 2019

```
PSLLD (with 64-bit operand)
    IF (COUNT > 31)
    THEN
        DEST[64:0] ← 0000000000000000H;
    ELSE
        DEST[31:0] ← ZeroExtend(DEST[31:0] << COUNT);
        DEST[63:32] ← ZeroExtend(DEST[63:32] << COUNT);
    FI;
```

A Few Potential Applications

- Program verification
- Translation validation of compiler optimization
- Security vulnerability tracking

Formalizing a decoder: Motivation

- The original x86 semantics accepted assembly code
  - Practical concerns - unlikely to be available unless source is
- In which case, why not use source?
  - Assuming, of course, we trust the compiler
  - What if source code is unavailable?
- Could use an off-the-shelf disassembler as a pre-processor
  - Requires trusting the correctness of this disassembler
  - Potential compatibility issues
  - Loses potential for tighter integration with semantics in the future
  - On the other hand…

Enter XED

- In principle, the definitive source of x86 instruction encodings is the x86 manual
  - **But the sheer size of the instruction set eliminates the possibility of encoding these by hand**
- Instead, we chose to port Intel's XED™'s disassembler to the K Framework
- Gives us to XED's datafiles, a source for the decoding algorithm, and a standard to test against
- Disadvantage: Trusting the correctness of XED

Intro to x86 Instruction Encoding

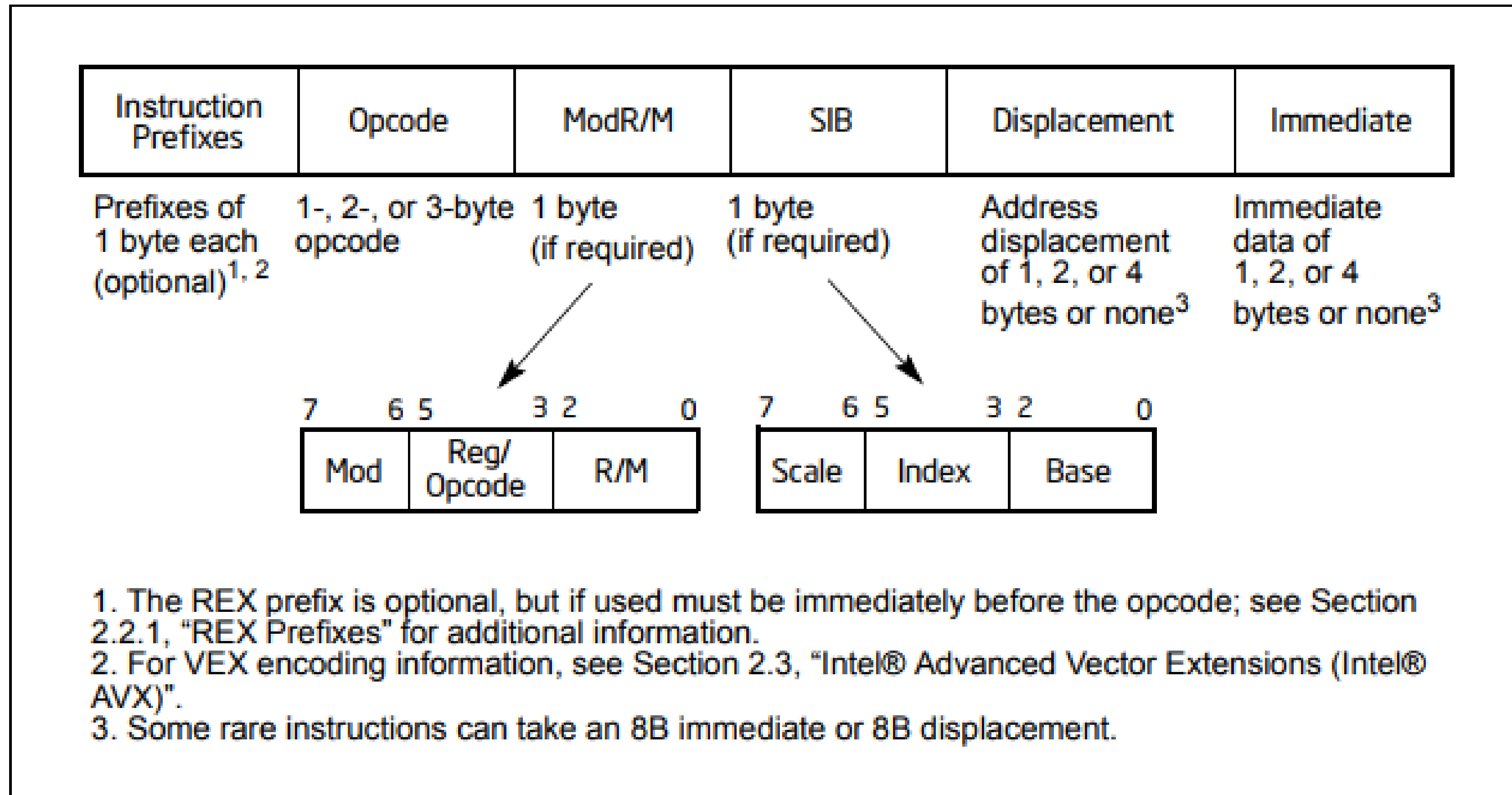


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

* Picture is borrowed from Vol.2 "Intel 64 and IA-32 Architectures, Software Developer's Manual"

A Decoding Algorithm

Scan for prefixes. Legacy prefixes set flag variables. Repeat until you read a byte that is not a legacy prefix.

If this byte was 0x62, 0x8F, 0xC4, or 0xC5 then branch to handle these special cases (vector extension instructions) - We'll skip these.

Read opcode bytes and determine if there is a MODRM byte.

If there is, read it and extract the MOD, REG, RM bits to determine if a SIB byte exists. Else skip

If there is a SIB byte, read it and extract the Index, Base, Scale bits.

Determine the effective address and operand sizes

A Decoding Algorithm: Cont. …

- At this point, we have enough information extracted to determine the precise instruction variant (operation and operands) - K makes this easy! Just match on the relevant properties
- We now know if we have a displacement and immediate(s) from the chosen variant - read and extract these.
- Decoding complete! Output in an appropriate form.

An example (generated) K rule

```
rule <IMM0> _ => 1  </IMM0>
<k> DynamicDecodeInstruction => SIMM8 ~> ScanForDisp ~>
ScanForImmediate ~> GPR8_B ~> OUTREGToREG0 ... </k>
<ICLASS> _ => SUB  </ICLASS>
<INUM> _ => 166  </INUM>
<CATEGORY> _ => "CATEGORY_BINARY"  </CATEGORY>
<INAME> _ => "sub"  </INAME>
<ATTRIBUTES> _ => ListItem(A_BYTEOP)  </ATTRIBUTES>
<OPERANDS> _ => /* Removed for length */ </OPERANDS>
<dynamicDecoderBuffer> 128 _:Ints  </dynamicDecoderBuffer>
<MOD> 3 </MOD>
<REG> 5 </REG>
<VEXVALID> 0 </VEXVALID>
```

Evaluation in isolation

The decoder has been tested in isolation by comparison with XED.

- It successfully disassembled elf_reader, our trusted C loader program
- It successfully disassembled all instructions supported by the execution semantics

After validation, we started integration with the semantics.

Integration with Semantics: Challenges

- Rewrite of program loading, addition of instruction fetch & decode steps
- Designed for different backend implementations of K – needed to port Semantics
- Semantics memory model did not support an efficient way to load a large text segment into memory.
    - Improves performance in K framework compared to long list structures
- XED's implementation of AT&T syntax was inconsistent with semantics (and gas' implementation)
- Immediate length inconsistencies due to a bug in the semantics.

Evaluation in combination w/ semantics

The decoder has been tested in isolation and in combination with the semantics.

- It successfully executed 468 of 482 selected gcc-torture tests
    - Compiled with miniature versions of stdlib for performance and compatibility reasons

Limitations

- K does not support 'parsing' binaries – thus still requires a trusted external binary reader to transform an elf executable into a format K can read
- The semantics offers little system call support; Although the decoder can decode them
- Dynamic linking not supported – detected at load time

Future Work

- ❑ Leverage decoder to simplify semantics
- ❑ Add modelling of OS interaction (e.g. syscall instruction)
  - Already support decoding these instructions, but can’t execute them
- ❑ Performance optimizations
  - **Reduction in memory usage of particular importance**
  - **Work on porting semantics to future K backends**
- ❑ Reducing trusted code base - in particular, moving the implementation of reading elf file segments to K Framework as well
- ❑ Support for symbolic execution

Thank You