

Kratos 2: an all-in-one C development companion

A.A. 2024/25

Andrea Caravano Alberto Cantele Biagio Cancelliere

May 9, 2025



POLITECNICO
MILANO 1863

Agenda

- 1 Kratos 2: a brief introduction
- 2 Model checking and SATisfiability: a theoretical overview
- 3 The Assembly-inspired intermediate language K2
- 4 An introductory complete example: from C to Kratos
- 5 The Competition on Software Verification
- 6 An efficient modulo operation
- 7 Symbolic execution: a theoretical overview
- 8 Commutativity of Reducers
- 9 Additional functionalities and intermediate representations
- 10 Applications and practical real-world implementations
- 11 Demo
- 12 Conclusions and considerations

KRATOS²

Kratos 2 is a formal verification tool for **imperative programs**, mostly aimed at C projects. It translates code into an intermediate verification language called **K2**, specifically designed for Kratos, which has a **formal semantics, mimicking the Assembly language**. It embeds precise checking of both **safety** (e.g., no assertion failure) and **liveness** properties (e.g., eventual termination), through **assertions**.

Kratos 2 integrates state-of-the-art **SAT/SMT solvers** and supports **symbolic execution**, **counterexample generation**, and **interactive simulation**.

It provides a **customizable C front-end**, a flexible **Python API** and can translate verification tasks into multiple low-level formalisms, for easy integration into development pipelines.

C programs can be **directly translated into the intermediate language K2** via a complete translator, implemented in Python.

Kratos' main engines have been developed with verification of **low-level software** in mind, like system modules or bit-manipulation computations.

Running configurations

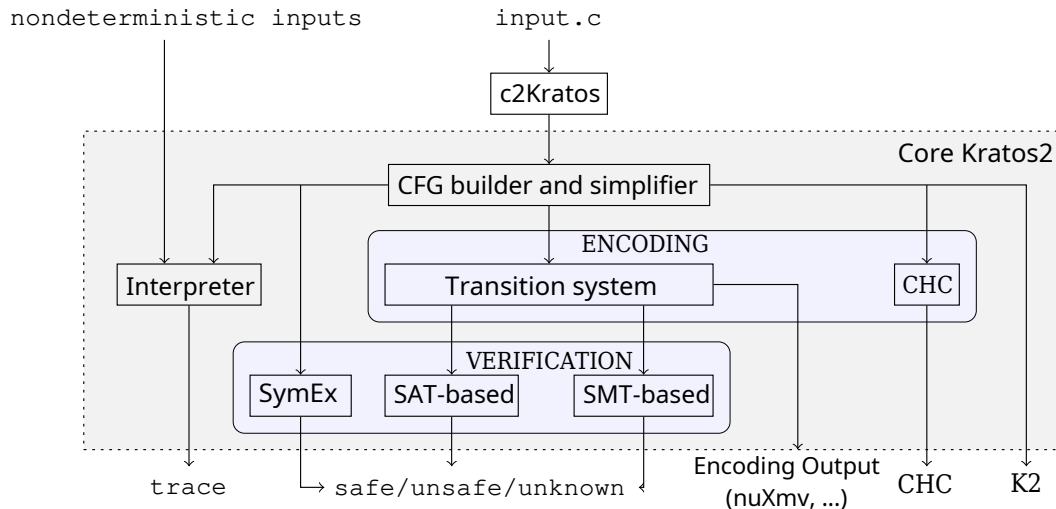
A Kratos **running configuration** is composed of a set of **inter-dependent** stages:

Stages

- cex: cex trace reconstruction.
- cfg: CFG building.
- check_inv: check given inductive invariant.
- flatten: flattening of expressions.
- inline: function inlining.
- mc: model checking.
- parse: input file parsing.
- sim: interactive simulation.
- smt: use Kratos as an SMT solver (MathSAT).
- symexec: perform symbolic execution.
- trans: transition system encoding.

Intermediate computations can be dumped, but they are mostly in a **proprietary format**.

Overall architecture



It is distributed by FONDAZIONE BRUNO KESSLER (Trento, Italy) as a **closed-source, pre-compiled** binary for both **Linux** and **Windows**, only on x64 architecture. Currently, the latest version publicly available is version 2.2.

Download

- Version 2.2: [Linux x64](#),
- Version 2.1: [Linux x64](#), [Windows x64](#).

Kratos 2 **does not require installation** on the system, being distributed as a **complete binary**, carrying most of the dependencies needed to run on modern operating systems. The tool is **not distributed for neither macOS nor ARM-based chips**: a DOCKER container (as part of a DEVELOPMENT CONTAINER) or a Virtual Machine are the most reasonable choices in which running Kratos, in those cases.

A sketched Docker container providing Kratos

DOCKERFILE

```
# Python's Debian-based image
FROM --platform=linux/amd64 python:3.11-bookworm
# Python 3.11 is the maximum supported version for C2Kratos

#### PYTHON DEPENDENCIES (for the C to Kratos conversion tool) ####
RUN pip install pycparser pcpp

#### STANDARD C DEVELOPMENT TOOLCHAIN ####
RUN apt update \ # upgrade and environment setup omitted...
    && apt install gdb hotspot valgrind build-essential cmake nano curl tar gzip

#### DOWNLOAD KRATOS2 ####
ARG KRATOS_RELEASE=https://kratos.fbk.eu/releases/kratos-2.2-linux64.tar.gz
ARG DESTINATION_FOLDER=kratos-2.2-linux64

RUN curl --output ./release ${KRATOS_RELEASE}

RUN tar -zxvf ./release
RUN mv ./${DESTINATION_FOLDER}/* /kratos

# Permissions, path, users and environment cleanup omitted...
```


Model checking is an automated technique to verify whether a system (often modeled as a transition system) satisfies a given specification, typically expressed in temporal logic.

There are two main types of properties:

- **Safety properties:** assert that *nothing bad ever happens* (e.g., a bad prefix).
- **Liveness properties:** assert that *something good eventually happens* (e.g., every request eventually gets a response).

In Kratos, they are expressed as **program assertions** through *labels*.

- **SAT (Boolean Satisfiability)** is the problem of deciding if a propositional logic formula can be made true by assigning truth values to its variables.

SAT solvers, like Kratos, are fast and widely used to verify purely boolean conditions.

- **SMT (Satisfiability Modulo Theories)** extends SAT by handling richer data types and theories:
 - **Integers and real numbers:** Handling arithmetic operations and inequalities.
 - **Arrays:** Reasoning about array properties, access, and updates.
 - **Bit-vectors:** Working with fixed-width bit strings and bitwise operations.
 - **Uninterpreted Functions:** Dealing with functions whose specific implementation is not known, but whose properties (like consistency) can still be reasoned about.

SMT solvers (e.g., MathSAT, on which Kratos is based) are particularly well-suited for **software verification**. Programs often involve a mix of boolean logic and operation on various data types.

SMT allows verification tool to reason about the complex interplay between these elements.

Tools like **Kratos 2** embed state-of-the-art SMT solvers, which enables them to:

- **Verify properties of a program:** check for properties expressed as assertions by examining whether the negation of the property is satisfiable by any reachable state.
- **Perform symbolic execution:** explore all possible program paths without the need to run the program with concrete fixed inputs. Instead, they replace variables with symbolic values and maintain constraints on these values as the execution proceeds.
- **Generate counterexamples:** if a property is found to be violated, the SMT solver can often provide a concrete trace and assignment map of its variables, leading to the violation and showing a useful starting point for debugging purposes.

Kratos implements three main model checking **algorithms**, with an eye on low-level software verification:

- **BMC**: standard Bounded Model Checking.
- **IC3**: the main algorithm used by Kratos, based on barriers.
- Symbolic model checking-based algorithms, using **mathematical induction**.

Bounded Model Checking

BMC stems from the idea that a cycle occurs within a path of bounded length (number of states + 1): this helps with the detection of **back loops**, underlining infinite paths.

Given $k > 0$, we can encode the **unfolding** up to k steps: suitable constraints are used to build a formula exhibiting a counterexample to the interested property of length at most k .

The IC3 model checking algorithm: a sketch

Incremental Construction of Inductive Clauses (IC3) is a SAT-based model checking algorithm that aims at optimizing the number of inspected groups of states through **barriers**.

At each step, the algorithm's evolution is described by **frames**: an inductive invariant.

Implementation sketch

- Frame 0 (F_0) is built: the system's initial state collection.
- Frame 1 (F_1) is built: a transition step is performed (it must contain F_0).
- A set of constraints is formed and checked for: if, from F_1 , a violating state in the group is reachable, we try to identify a descriptive clause (c).
- For violating constraints, the generalized explicit exclusion clause (c , **barrier**) is added to F_1 .
- If the resulting frame is stable (no violations derive out of the generalization \Rightarrow an invariant has been found), the new constraints are propagated and F_2 is built.
- Otherwise, a counterexample results from the contradiction of the initial states!
- The algorithm ends when $F_i = F_{i+1}$ (induction step: each transition stays in F_i).

K2 uses S-expression for its concrete syntax, representing structured data in textual format.

Basic Types

`bool`, `int`, `real`

Bit-vectors

`(sbv SIZE)`, `(ubv SIZE)`

Floating-point numbers

`(fp EXPONENT MANTISSA)`

Maps (arrays)

`(map INDEX ELEM)`

Enumerations

`(enum val1 val2 ...)`

Expressions and declarations

Functions

`(fun NAME (ARG_1 ...) (RET_1 ...))`

Variables

`(var NAME TYPE)`

Constants

`(const VALUE TYPE)`

Type conversions

`(cast ...), (bitcast ...)`

Logical operations

`and, or, not`

Arithmetic

`add, sub, mul, div, rem`

Statements

Bit-level operations

`bitand, bitor, bitxor, bitnot, lshift, rshift`

Comparisons

`eq, lt, le, gt, ge`

Maps

`mapget, mapset`

Floating-point checking

`isnan, isinf, etc.`

Assignments

`(assign VAR EXPR)`

Assumptions

`(assume EXPR)`

Statements

Labels

(label NAME)

Labels are heavily used for **model checking**, their usage will be further expanded later.

Jumps (deterministic)

(jump TARGET)

Jumps (conditionals)

(condjump COND TARGET)

Sequential code block

(seq STMT1 STMT2 ...)

Function call

(call FUNC ARGS RETS)

Function definition

```
(function NAME (PARAMS) (return RETS)  
(locals VARS)  
BODY)
```

- **NAME**: function identifier
- **PARAMS, RETS**: input/return parameters
- **VARs**: local variables
- **BODY**: sequence of statements

Program structure

```
(type TYPENAME TYPE)
(entry MAIN_FUNC)
(init CONSTRAINT)
(globals VARS)
FUNCTION_1
...
FUNCTION_N
```

- **TYPES**: aliases like type definitions
- **ENTRY POINT**: main function
- **INIT**: global state constraints
- **GLOBALS**: declared global variables

Verification properties

(! SEXP :KEY VALUE)

- (! **LABEL :error NAME**): this point is unreachable
- (! **LABEL :live NAME**): must be reachable infinitely often
- (! **LABEL :notlive NAME**): must eventually not be reached

A first problem: an infinite loop

C implementation

```
#include <assert.h>

int glbl = 0;

int f(int x) {
    if (glbl > 0) {
        return x - 1;
    } else {
        // our case
        glbl = 0;
        return x;
    }
}
```

K2 translation

```
(type cint (sbv 32))
(entry init_and_main)
(globals (var glbl cint))

(function f ((var x cint))
  (return (var ret cint)) (locals)
  (seq
    (jump (label then) (label else))
    (label then)
    (assume (op gt glbl (const 0 cint)))
    (assign ret (sub x (const 1 cint)))
    (jump (label end))
    (label else)
    (assume (op not (op gt glbl (const 0 cint))))
    (assign glbl (const 0 cint))
    (assign ret x)
    (label end)))
```

A first problem: an infinite loop

C implementation

```
void main(void) {  
    int y;  
    while (y > 0) {  
        y = f(y);  
    }  
  
    // failure of the assertion  
    // is an unreachable point!  
    assert(glbl == 0);  
}
```

K2 translation

```
(function main () (return) (locals (var y cint))  
  (seq  
    (label while)  
    (jump (label inwhile) (label endwhile))  
    (label inwhile)  
    (assume (op gt y (const 0 cint)))  
    (call f y y)  
    (jump (label while))  
    (label endwhile)  
    (assume (op not (op gt y (const 0 cint))))  
    (jump (label then) (label else))  
    (label then)  
    (assume (op not (op eq glbl (const 0 cint))))  
    (! (label err) :error assert-fail)  
    (label else)))  
  
(function init_and_main () (return) (locals)  
  (seq  
    (assign glbl (const 0 cint))  
    (call main)))
```

A first problem: an infinite loop

Translation of C source code to the K2 intermediate language is possible through the embedded C2Kratos translator, implemented as a PYTHON project.

Let's **translate** our example:

Linux shell command

```
python /kratos/tools/c2kratos.py ./example.c -o ./example.k2
```

A first problem: an infinite loop

Translation of C source code to the K2 intermediate language is possible through the embedded C2Kratos translator, implemented as a PYTHON project.

Let's **translate** our example:

Linux shell command

```
python /kratos/tools/c2kratos.py ./example.c -o ./example.k2
```

Let's now apply **model checking** and confirm our intuition about the unreachability of the assertion failure condition, meaning the assertion cannot fail.

Linux shell command

```
kratos -stage=mc ./example.k2
```


A first problem: an infinite loop

Kratos output

```
parsing... done (0.024)
flattening... done (0.004)
inlining functions... done (0.004)
flattening after inlining... done (0.000)
building control-flow graph... done (0.004)
encoding into transition system... done (0.026)
Trans encoder statistics:
...
model checking...
checking property 0
...
model checking... done (0.081)

Model checker statistics:
...

Global statistics:
  total_time = 0.183
  memory_used_mb = 267

safe
```



SV-COMP (Competition on Software Verification) aims at **standardizing comparison** among software-verification tools, thanks to an established set of verification tasks.

It provides a **snapshot** of the state-of-the-art in the software verification community, providing benefit to students and researchers, **crediting** their development work.

In its current state, Kratos' implementation is **heavily focused** on tackling and optimizing the resolution to the **verification tasks provided by SV-COMP**.

It provides specific declarations for SV-COMP specifications and a wrapper for automated benchmarking.

C2Kratos manual configuration

```
python /kratos/tools/c2kratos.py  
    --svcomp --svcomp-spec properties/property.prp  
    code.c | kratos -stage=mc [eventual options...]
```

Kratos 2 wrappers for SV-COMP

In its current state, Kratos' implementation is **heavily focused** on tackling and optimizing the resolution to the **verification tasks provided by SV-COMP**.

It provides specific declarations for SV-COMP specifications and a wrapper for automated benchmarking.

C2Kratos manual configuration

```
python /kratos/tools/c2kratos.py  
    --svcomp --svcomp-spec properties/property.prp  
    code.c | kratos -stage=mc [eventual options...]
```

Automated wrapper

```
python kratos-svcomp.py code.c
```

Results of a comparison

Family	CPAchecker			Kratos2			VeriAbs		
	U	S	W	U	S	W	U	S	W
arrays	70	5	0	75	7	0	106	261	0
bitvectors	13	31	0	13	33	0	14	31	0
combinations	295	36	0	282	47	0	277	77	0
controlflow	39	36	0	40	37	0	40	47	0
eca	223	481	0	210	365	0	467	600	0
oats	41	356	0	43	350	0	43	393	0
heap	71	118	1	67	102	0	70	120	0
loops	152	334	2	159	307	0	192	427	0
productlines	265	332	0	262	315	0	260	322	0
recursive	40	36	1	43	28	0	46	41	0
sequentialized	347	108	0	361	68	0	361	123	0
xcsp	50	52	0	51	51	0	52	52	0
Total	1606	1925	4	1606	1710	0	1928	2494	0

Let's consider **numbers in the form** $2^s - 1$, with $s \in \mathbb{Z}^+$

They represent, in binary notation, all combinations that are represented **only by ones**:

$$s = 1 \Rightarrow 2^s - 1 = 1_{10} = 1_2$$

$$s = 2 \Rightarrow 2^s - 1 = 3_{10} = 11_2$$

$$s = 3 \Rightarrow 2^s - 1 = 7_{10} = 111_2$$

$$s = 4 \Rightarrow 2^s - 1 = 15_{10} = 1111_2$$

$$s = 5 \Rightarrow 2^s - 1 = 31_{10} = 11111_2$$

And so on...

An efficient modulo operation

Now, we would like to use these to **speed up the standard modulo operation** implementation.

$$2^s = (2^s - 1) + 1 \implies 2^s \bmod (2^s - 1) = 1$$

Modular arithmetic rules show that:

If $a \bmod n = b \implies a \equiv b \pmod{n}$, then $a^k \equiv b^k \pmod{n}$ for $k \in \mathbb{Z}$

So, the same rule applies also for higher orders!

$$(2^s)^2 \bmod (2^s - 1) = 1^2 = 1$$

$$(2^s)^3 \bmod (2^s - 1) = 1^3 = 1$$

$$(2^s)^4 \bmod (2^s - 1) = 1^4 = 1$$

$$(2^s)^5 \bmod (2^s - 1) = 1^5 = 1$$

And so on...

An efficient modulo operation

Let's remember how a number n in base 10 is decomposed in powers of 2 (the same rule applied for conversions between the two basis):

$$n = a_0 + a_1 \cdot (2^s) + a_2 \cdot (2^s)^2 + a_3 \cdot (2^s)^3 + a_4 \cdot (2^s)^4 + \cdots + a_k \cdot (2^s)^k$$

Where a_i is a number from 0 to $2^s - 1$ clearly.

Therefore,

$$n \bmod (2^s - 1) = (a_0 + a_1 + a_2 + \cdots + a_k) \bmod (2^s - 1)$$

Since, as we've shown earlier,

$$(2^s)^k \bmod (2^s - 1) = 1^k = 1$$

An efficient modulo operation

C implementation

```
int main() {  
    ... // (s is less than 32 of course)  
    unsigned int d = (1 << s) - 1; /* so d is either 1, 3, 7, 15, 31, ... */  
  
    if (d > 0) {  
        m = n; // n is the numerator, m will contain the result  
        while (n > d) { // iteratively reduces the operation span  
            m = 0;  
            while (n > 0) { // incrementally builds up coefficients a_i  
                m += n & d; // extracts a_0  
                n = n >> s; // then, will form a_1...  
            }  
            n = m;  
        }  
        if (m == d) { // m could be d, so we make it circulating back to 0  
            m = 0;  
        }  
  
        __VERIFIER_assert(m == n % d);  
    }  
    ...  
}
```

An efficient modulo operation

Time complexity

Let n be the number of bits of the numerator,

$T(n) = \Theta(n \cdot \log(n))$, while the basic one is normally $T(n) = \Theta(n^2)$

An efficient modulo operation

Time complexity

Let n be the number of bits of the numerator,

$T(n) = \Theta(n \cdot \log(n))$, while the basic one is normally $T(n) = \Theta(n^2)$

Let's apply **model checking** to our example:

Linux shell command

```
python /kratos/tools/c2kratos.py  
    --svcomp --svcomp-spec properties/unreach-call.prp  
    modulus.c | kratos -stage=mc
```

An efficient modulo operation

Kratos output

```
parsing... done (0.016)
flattening... done (0.005)
inlining functions... done (0.004)
flattening after inlining... done (0.000)
building control-flow graph... done (0.003)
encoding into transition system... done (0.027)
Trans encoder statistics:
  ...
model checking...
checking property 0
  ...
model checking... done (5.278)

Model checker statistics:
  ...

Global statistics:
  total_time = 5.708
  memory_used_mb = 1291

safe
```

Symbolic execution is a program analysis technique that explores the **execution paths** of a program using symbolic values for input parameters instead of concrete data: instead of running the program with specific numbers or strings, symbolic execution assigns **symbolic variables** (like X , Y instead of $x = 15$ and $y = 16$) to inputs.

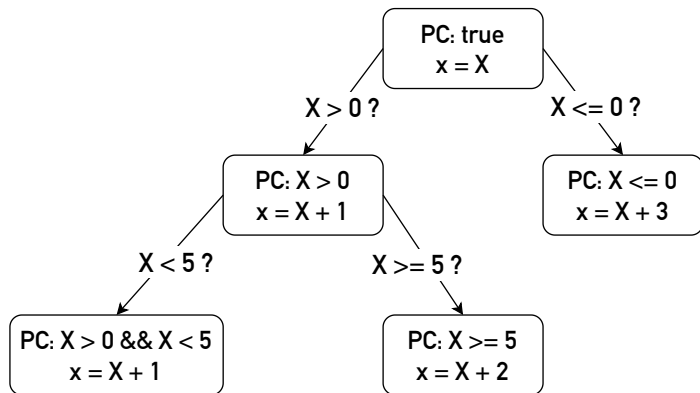
As the program runs, operations are **performed on these symbolic values**.

Conditional branches (like `if` statements or loops) create different execution paths that are identified by a **path constraint** being generated, a logical formula associated to the specific path trace.

Conditional branches in symbolic execution

C implementation sketch

```
int f(int x) {  
    if (x > 0) {  
        if (x < 5) {  
            x = x + 1;  
        } else x = x + 2;  
    } else x = x + 3;  
  
    return x;  
}
```



Kratos can, therefore:

- **Check path feasibility:** Cut out exploration on unreachable path constraints.
- **Generate inputs:** If a path constraint is satisfiable, the solver provides test assignments to symbolic inputs that satisfies the constraints.
- **Check properties:** In the same way, it can describe a violation in the validity of the assertion, providing a counterexample.

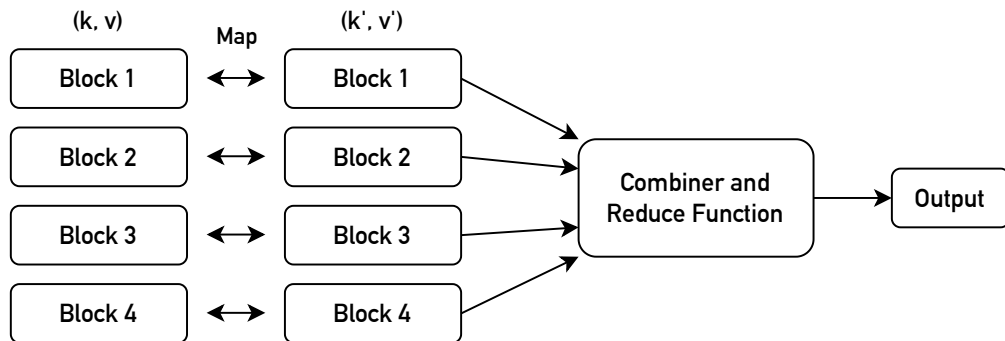
Resulting in a systematic exploration of the program's behavior and related properties.

The Map-Reduce programming model

The **Map-Reduce** programming paradigm is a high-level abstraction used in the design of functional programming algorithms and distributed systems.

The computation is split in two phases:

- **Map**: individual elements in the domain are reorganized in a new output format.
- **Reduce**: a functional transformation is applied to matching entries.



Commutativity of Reducers

C implementation

```
int rangesum(int x[N]) {  
    int i, cnt = 0;  
    long long ret = 0;  
  
    // sum up array values  
    for (i = 0; i < N; i++) {  
        // a bug! only the second half is summed up...  
        if (i > N / 2) {  
            ret = ret + x[i];  
            cnt = cnt + 1;  
        }  
    }  
  
    // standard average  
    if (cnt != 0)  
        return ret / cnt;  
    else  
        return 0;  
}
```

Commutativity of Reducers

A simple testing strategy

```
int main() {  
    // array size  
    N = __VERIFIER_nondet_int();  
    if (N > 1) {  
        int x[N], temp, ret1, ret2, ret3;  
        // initialize the array  
        init_nondet(x);  
  
        // base sum  
        ret1 = rangesum(x);  
  
        // swap x[0] and x[1]  
        temp = x[0];  
        x[0] = x[1];  
        x[1] = temp;  
        ret2 = rangesum(x);
```

10	20	30	40	50
----	----	----	----	----

$$\text{ret1} = (40 + 50) / 2 = 45$$

20	10	30	40	50
----	----	----	----	----

$$\text{ret2} = (40 + 50) / 2 = 45$$

Commutativity of Reducers

A simple testing strategy

```
// shift left (circularly)
temp = x[0];
for (int i = 0; i < N - 1; i++) {
    x[i] = x[i + 1];
}
x[N - 1] = temp;
ret3 = rangesum(x);

// compare results
if (ret1 != ret2 || ret1 != ret3) {
    reach_error();
}
return 1;
}
```

20	10	30	40	50
----	----	----	----	----

$$\text{ret2} = (40 + 50) / 2 = 45$$

10	30	40	50	20
----	----	----	----	----

$$\text{ret3} = (50 + 20) / 2 = 35$$

Let's apply **symbolic execution** to our example:

Linux shell command

```
python /kratos/tools/c2kratos.py  
    --svcomp --svcomp-spec properties/unreach-call.prp  
    rangesum.c | kratos -stage=symexec -error_id=reach_error
```

Let's apply **symbolic execution** to our example:

Linux shell command

```
python /kratos/tools/c2kratos.py  
    --svcomp --svcomp-spec properties/unreach-call.prp  
    rangesum.c | kratos -stage=symexec -error_id=reach_error
```

The wrapper makes the generation of **counterexamples** easier:

Linux shell command

```
python kratos-svcomp.py rangesum.c --cex
```

Commutativity of Reducers

Kratos output

```
-----  
translating input C program to k2  
-----
```

```
translation_time = 1.609  
-----
```

```
running kratos with configuration: symexec  
-----
```

```
parsing... done (0.019)
```

```
flattening... done (0.006)
```

```
building control-flow graph... done (0.004)
```

```
performing symbolic execution...
```

```
Counterexample trace
```

```
...
```

```
done (3.717)
```

```
Symbolic execution statistics:
```

```
...
```

```
Global statistics:
```

```
...
```

```
unsafe
```

Intermediate stages

The intermediate stage of Kratos processing can be dumped: some of them are **proprietary**. Others, like the CFG, can be analyzed with standard diagram drawing software.

Linux shell command

```
kratos -stage=mc  
  -intermediate_output_dir=.  
  -dump_intermediate_stages=true  
example.k2
```

Intermediate stages

The intermediate stage of Kratos processing can be dumped: some of them are **proprietary**. Others, like the CFG, can be analyzed with standard diagram drawing software.

Linux shell command

```
kratos -stage=mc
      -intermediate_output_dir=.
      -dump_intermediate_stages=true
example.k2
```

Kratos output

```
parsing... done (0.028)
writing to ./parsed.k2
flattening... done (0.007)
writing to ./flattened.k2
inlining functions... done (0.007)
flattening after inlining... done (0.000)
writing to ./inlined.k2
building control-flow graph... done (0.007)
writing to ./cfg.dot
writing to ./bbcfg.dot
writing to ./cfg.k2
encoding into transition system... done (0.055)
writing to ./ts.vmt
```


Intermediate stages

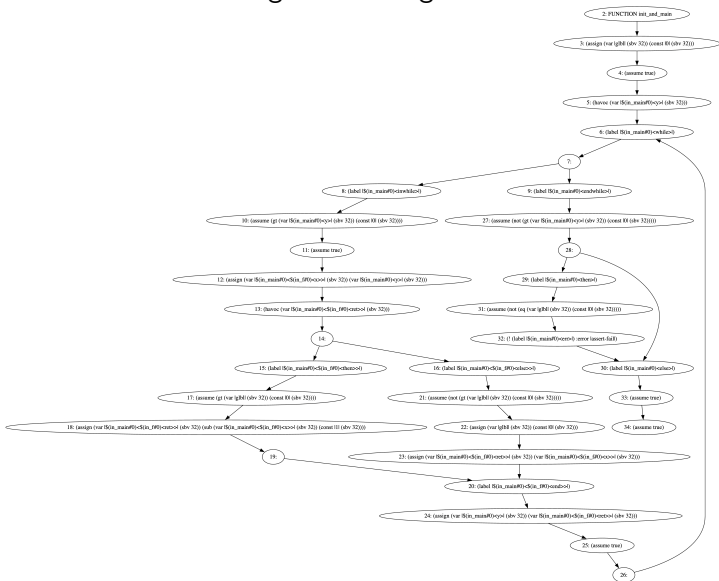
The intermediate stage of Kratos processing can be dumped: some of them are **proprietary**. Others, like the CFG, can be analyzed with standard diagram drawing software.

Linux shell command

```
kratos -stage=mc  
-intermediate_output_dir=  
-dump_intermediate_stages=true  
example.k2
```

Kratos output

```
parsing... done (0.028)  
writing to ./parsed.k2  
flattening... done (0.007)  
writing to ./flattened.k2  
inlining functions... done (0.007)  
flattening after inlining... done (0.000)  
writing to ./inlined.k2  
building control-flow graph... done (0.007)  
writing to ./cfg.dot  
writing to ./bbcfig.dot  
writing to ./cfg.k2  
encoding into transition system... done (0.055)  
writing to ./ts.vmt
```



Interactive simulation: a first debugging technique

In Kratos, **interactive simulation** is a technique that aims at providing a first debugging end-point, by imposing **variable values, function returns and branches outcomes**.

Execution ends when an assumption is violated.

Linux shell command

```
kratos -stage=sim ./example.k2
```

Interactive simulation: a first debugging technique

Kratos output

```
parsing... done (0.022)
flattening... done (0.006)
inlining functions... done (0.005)
flattening after inlining... done (0.000)
building control-flow graph... done (0.011)
Enter input value for (var |glbl| (sbv 32)) (use ? for a random value)
```

0

```
executing statement: (assign (var |glbl| (sbv 32)) (const |0| (sbv 32)))
executing statement: (assume true)
executing statement: (havoc (var |$(in_main#0)<y>| (sbv 32)))
Enter input value for (var |$(in_main#0)<y>| (sbv 32)) (use ? for a random value)
```

1

```
executing statement: (label |$(in_main#0)<while>|)
Enter target value for (jump (label |$(in_main#0)<inwhile>|) (label |$(in_main#0)<endwhile>|)) (use ? for a random value)
$(in_main#0)<inwhile>
executing statement: (label |$(in_main#0)<inwhile>|)
executing statement: (assume (gt (var |$(in_main#0)<y>| (sbv 32)) (const |0| (sbv 32))))
executing statement: (assume true)
executing statement: (assign (var |$(in_main#0)<$(in_f#0)<x>>| (sbv 32)) (var |$(in_main#0)<y>| (sbv 32)))
executing statement: (havoc (var |$(in_main#0)<$(in_f#0)<ret>>| (sbv 32)))
Enter input value for (var |$(in_main#0)<$(in_f#0)<ret>>| (sbv 32)) (use ? for a random value)
```

1

```
Enter target value for (jump (label |$(in_main#0)<$(in_f#0)<then>>|) (label |$(in_main#0)<$(in_f#0)<else>>|)) (use ? for random value)
$(in_main#0)<$(in_f#0)<else>>
```

Interactive simulation: a first debugging technique

Kratos output

```
executing statement: (label |$(in_main#0)<$(in_f#0)<else>>|)
executing statement: (assume (not (gt (var |glbl| (sbv 32)) (const |0| (sbv 32)))))
executing statement: (assign (var |glbl| (sbv 32)) (const |0| (sbv 32)))
executing statement: (assign (var |$(in_main#0)<$(in_f#0)<ret>>| (sbv 32)) (var |$(in_main#0)<$(in_f#0)<x>>| (sbv 32)))
executing statement: (label |$(in_main#0)<$(in_f#0)<end>>|)
executing statement: (assign (var |$(in_main#0)<y>| (sbv 32)) (var |$(in_main#0)<$(in_f#0)<ret>>| (sbv 32)))
executing statement: (assume true)
executing statement: (label |$(in_main#0)<while>|)
Enter target value for (jump (label |$(in_main#0)<inwhile>|) (label |$(in_main#0)<endwhile>|)) (use ? for a random value)
```

\$(in_main#0)<inwhile>

```
executing statement: (label |$(in_main#0)<inwhile>|)
executing statement: (assume (gt (var |$(in_main#0)<y>| (sbv 32)) (const |0| (sbv 32))))
executing statement: (assume true)
executing statement: (assign (var |$(in_main#0)<$(in_f#0)<x>>| (sbv 32)) (var |$(in_main#0)<y>| (sbv 32)))
executing statement: (havoc (var |$(in_main#0)<$(in_f#0)<ret>>| (sbv 32)))
Enter input value for (var |$(in_main#0)<$(in_f#0)<ret>>| (sbv 32)) (use ? for a random value)
```

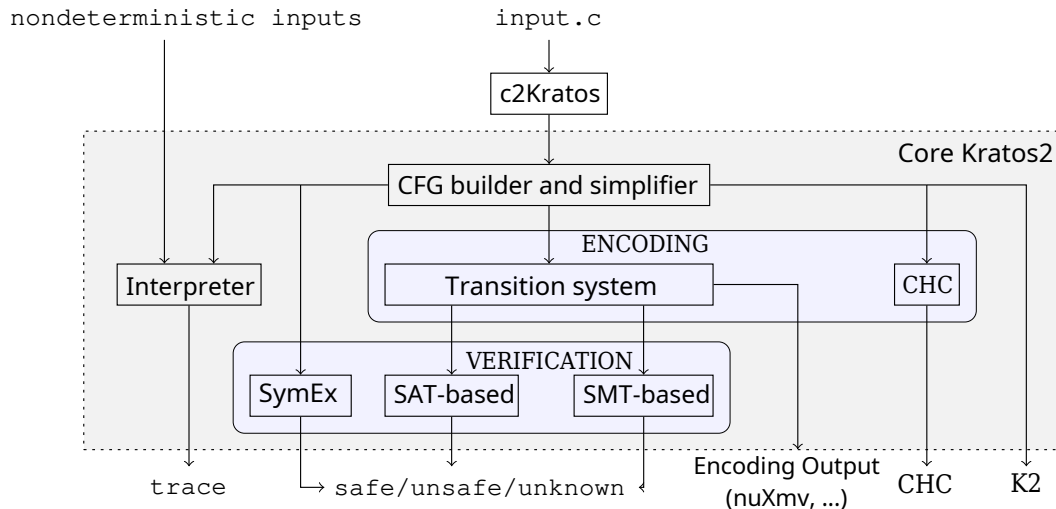
1

Enter target value for (jump (label |\$(in_main#0)<\$(in_f#0)<then>>|) (label |\$(in_main#0)<\$(in_f#0)<else>>|)) (use ? for random value)

\$(in_main#0)<\$(in_f#0)<else>>

```
executing statement: (label |$(in_main#0)<$(in_f#0)<else>>|)
executing statement: (assume (not (gt (var |glbl| (sbv 32)) (const |0| (sbv 32)))))
executing statement: (assign (var |glbl| (sbv 32)) (const |0| (sbv 32)))
executing statement: (assign (var |$(in_main#0)<$(in_f#0)<ret>>| (sbv 32)) (var |$(in_main#0)<$(in_f#0)<x>>| (sbv 32)))
executing statement: (label |$(in_main#0)<$(in_f#0)<end>>|)
executing statement: (assign (var |$(in_main#0)<y>| (sbv 32)) (var |$(in_main#0)<$(in_f#0)<ret>>| (sbv 32)))
executing statement: (assume true)
executing statement: (label |$(in_main#0)<while>|)
```

Overall architecture



AUTOSAR (AUTomotive Open System ARchitecture) is a worldwide consortium of car manufacturers and component or service providers in the **automotive domain**, with the main goal of providing a standardized software architecture for the development and deployment of software components.

AUTOSAR wants to develop a platform for software development which ensures **safety**.

This platform is complemented by traditional V&V (**Verification and Validation**) techniques.

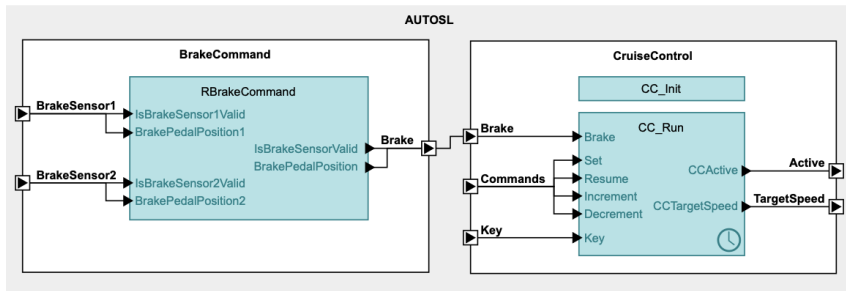
AUTOSAR found a Solution in **EVA**, a framework for the **integration** of modern verification tools.

Main characteristics:

- Automatic end-to-end verification of **system-level properties**.
- Combines software model checking techniques with a **contract-based analysis** for verifying their correct composition.
- Implements all the features required for **usability** in a typical industrial context.
- Includes a **frontend** for the AUTOSAR's development environment.

AUTOSAR and the framework EVA

Arrows represent the flow of data: the input ones are the **Require Ports** and output ones are **Provide Ports**.



The set of properties in AUTOSAR's environment is defined in simple Linear Temporal Logic (**LTL**) through predicates called **Contracts**.

in the future within [2,2]
it shall always_be that
 (CCActive and Brake is_greater_than 0) implies
 in the future within [0,2] (not next(CCActive))
holds_true

Automated driving functions are among the **most critical software components** to develop. Before deployment in real vehicles, they must be proven **safe** for both regulatory and traffic compliance.

Despite the coverage that can be reached with huge numbers of test drives, **corner cases** are possible and potential logic errors must be found as early as possible.

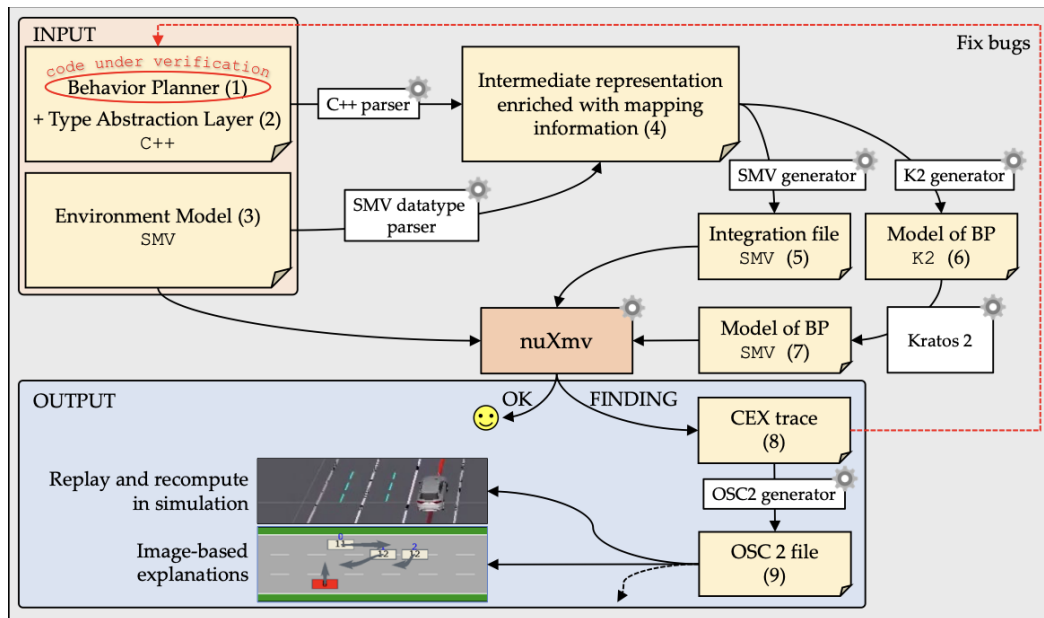
Due to time-to-delivery constraints, the V&V activities which mainly involve model checking of infinite-state transition systems are supposed to proceed **in parallel** with the development.

The main challenge is to **automate** V&V process to increase efficiency and effectiveness in **edge case** discovery and **logical violation** during the development phase of an autonomous agent.

Kratos implements effective model checking engines that are being incrementally integrated in development process.

For example, it has been deployed in the **Behaviour Planner**, the portion of the agent's code that decides the rules to adopt given a set of inputs from the sensors (whether to brake, accelerate, turn left/right, ...).

Model checking in the automotive sector

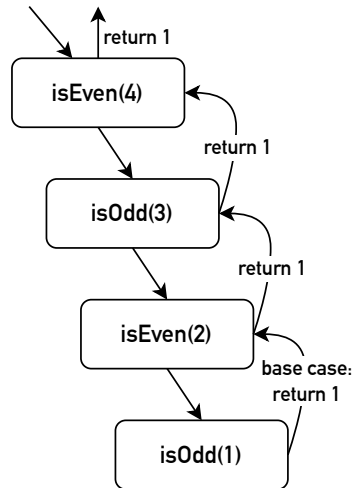


Introduction to the demo: Even or Odd? A recursive case

C implementation

```
// returns 1 if the input n is odd
int isOdd(int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return isEven(n - 1);
    }
}

// returns 1 if the input n is even
int isEven(int n) {
    if (n == 0) {
        return 1;
    } else if (n == 1) {
        return 0;
    } else {
        return isOdd(n - 1);
    }
}
```



Time for the demo!








Strengths

- Very precise translation engine.
- Efficient model checking and symbolic execution models and algorithms.
- Light on system resources.

Weaknesses

- Poor documentation and distribution management.
- Explicitly scoped only for research and evaluation.
- Optimized on specific use-cases.

References

-  [1] Fondazione Bruno Kessler (2023), Kratos 2: documentation.
-  [2] Alberto Griggio and Martin Jonás (2023), Kratos2: an SMT-Based Model Checker for Imperative Programs.
-  [3] Alberto Griggio and Martin Jonás (2023), Artifact for Kratos2: an SMT-Based Model Checker for Imperative Programs (experimental evaluation).
-  [4] AA. VV. (2023), EVA: a Tool for the Compositional Verification of AUTOSAR Models.
-  [5] AA. VV. (2024), Towards Safe Autonomous Driving: Model Checking a Behavior Planner during Development.
-  [6] Software and Computational Systems Lab and AA. VV. at LMU Munich, SV-COMP: Collection of Verification Tasks.
-  [7] Alberto Griggio and Marco Roveri (2015), Comparing Different Variants of the IC3 Algorithm for Hardware Model Checking.
-  [8] Sean Eron Anderson (2005), Bit manipulation hacks: modulus division without a division operator.