Kratos2: an SMT-Based Model Checker for Imperative Programs[*]

Alberto Griggio[1] and Martin Jonáš[1,2]

[1] Fondazione Bruno Kessler, Trento, Italy
griggio@fbk.eu
[2] Masaryk University, Brno, Czechia
martin.jonas@mail.muni.cz

Abstract. This paper describes Kratos2, a tool for the verification of imperative programs. Kratos2 operates on an intermediate verification language called K2, with a formally-specified semantics based on SMT, allowing the specification of both reachability and liveness properties. It integrates several state-of-the-art verification engines based on SAT and SMT. Moreover, it provides additional functionalities such as a flexible Python API, a customizable C front-end, generation of counterexamples, support for simulation and symbolic execution, and translation into multiple low-level verification formalisms. Our experimental analysis shows that Kratos2 is competitive with state-of-the-art software verifiers on a large range of programs. Thanks to its flexibility, Kratos2 has already been used in various industrial projects and academic publications, both as a verification back-end and as a benchmark generator.

1 Introduction

We present Kratos2, a tool for the verification of real-world imperative programs. Kratos2 is a complete rewrite and redesign of Kratos [17], improving and extending it in multiple directions. First, Kratos2 introduces a simple yet expressive intermediate language called K2, with a formally-specified semantics based on Satisfiability Modulo Theories (SMT), which is parametric on the underlying SMT theory. K2 is expressive enough to capture most of the features of real-world C programs, such as pointers, dynamic memory allocation, floating-point data types, and bit-precise semantics of bounded integers, which the old version of the tool could not handle (being limited to C programs without pointers and recursion, and in which C integers were interpreted as mathematical integers). Kratos2 comes with a separate C front-end c2Kratos that can translate C programs to K2. Second, Kratos2 includes a variety of state-of-the-art verification back-ends

---

based on either symbolic model checking or symbolic execution with SAT and SMT solvers. Besides reachability properties, Kratos2 also supports various forms of *liveness* properties, which can be used to encode termination and more complex linear-time temporal properties. Third, Kratos2 implements an interactive interpreter, which can simulate K2 programs using non-deterministic inputs provided either by the user or by external oracles. Kratos2 also supports counterexample reconstruction, another feature not available in the original Kratos.

The new intermediate language K2 enables modular translation of C programs into various verification languages. Namely, Kratos2 can be used for translating C programs into nuXmv [14], VMT [20], AIGER [9], BTOR2 [31], Constrained Horn Clauses (CHCs) [11], or Boogie [29] formats. Additionally, Kratos2 comes with a Python API for construction and manipulation of K2 programs, which the users can leverage to implement custom front-ends and generators of K2 programs and also additional translators from K2 to other formalisms.

Although Kratos2 has not been described in a publication until now, it has already been successfully used in several research and industrial projects. In particular, Kratos2 has been used as a back-end for the verification of automotive software in the context of the AUTOSAR platform [15,16]; of C code automatically generated from AADL specifications by the TASTE development environment [12]; and for verification of C code for railway interlocking systems automatically generated from the specifications in a controlled natural language [1]. Kratos2 has also been used as a benchmark generator to produce symbolic transition systems from C programs [30].

The rest of the paper is structured as follows. The functionalities offered by Kratos2 from the user perspective are described in §2; §3 introduces K2, describing its syntax and formal semantics. The internal architecture of Kratos2, with details about its main components, is presented in §4; implementation notes and experimental evaluation on C programs from the annual software verification competition SV-COMP are provided in §5. Finally, §6 concludes the paper and presents directions for future developments.

2   Functional View

In this section we provide a high-level overview of the functionalities available in Kratos2. More details will then be provided in the following sections.

An intermediate language for imperative programs. The core of Kratos2 is built around an idealized language for imperative programs called K2. Unlike common high-level real-world programming languages, K2 has a simple and clean semantics based on first-order logic modulo theories that is fully formally specified. The K2 language, similar in spirit to other intermediate verification languages proposed in the literature such as Boogie [29] or Why3 [26] (although less feature rich than the two), is at the same time simple enough to be easily manipulated and translated into formalisms used by SAT-based and SMT-based verification back-ends on one hand, and expressive enough to efficiently capture

a significant subset of C on the other, as demonstrated also by our experimental results on standard sv-comp benchmarks (see §5).

Verification of safety and liveness with multiple back-ends. Kratos2 implements multiple state-of-the-art verification algorithms based on sat and smt, supporting both bit-precise reasoning over machine integers and floating-point numbers as well as higher-level reasoning based on, e.g., mathematical integers, real numbers, and uninterpreted functions, depending on the combinations of theories used in the input K2 program under analysis. Moreover, Kratos2 supports not only the verification of safety properties (via a reduction to reachability of designated "error" program locations), but it also supports liveness properties such as proving that a specific program location is reached a finite number of times in all executions, or that it is always visited infinitely often in all infinite executions.

A Python API for program manipulations. Kratos2 provides a rich and flexible Python API for parsing, printing, and manipulating K2 programs and expressions, which can be used to implement converters from high-level languages to K2 or to directly generate K2 programs from user-specific applications.

A customizable C front-end. Kratos2 comes with a front-end for C programs which supports a wide range of customization options for controlling the translation from C to K2. These range from the choice of theories to use to encode C data types (e.g., bit-vectors or unbounded integers), to the use of customized program transformations or the injection of new built-in functions with special meaning (such as special `assume`, `malloc`, or `memset` built-ins). Thanks to its plug-in architecture, the front-end can be easily customized for domain-specific subsets of C, for example to implement special optimization passes that are safe only in the given context, or to automatically inject properties to the code based on specification files (as is, e.g., the case in sv-comp [3]).

Encoding into multiple formalisms. Kratos2 can be used as an encoder or benchmark generator because it can translate imperative programs written in C or in K2 into other formalisms, including symbolic transition systems in nuXmv [14], vmt [20], aiger [9] or Btor2 [31] formats, Constrained Horn Clauses (chcs) [11], or other intermediate verification languages like Boogie [29].

Simulation and symbolic execution. Finally, Kratos2 can be used as an interpreter, allowing an (interactive) simulation of K2 programs and their symbolic execution, as an alternative to the verification back-ends based on model checking.

3 The K2 Language

In this section we introduce K2, the intermediate verification language used by Kratos2. We present its abstract syntax, formally define its semantics, and discuss its support for safety and liveness properties.

$$\begin{aligned}
\langle\text{stmt}\rangle &::= \langle\text{assign-stmt}\rangle\ | \\
&\quad\ \langle\text{assume-stmt}\rangle\ | \\
&\quad\ \langle\text{call-stmt}\rangle\ | \\
&\quad\ \langle\text{havoc-stmt}\rangle\ | \\
&\quad\ \langle\text{jump-stmt}\rangle\ | \\
&\quad\ \langle\text{label-stmt}\rangle \\
\langle\text{assign-stmt}\rangle &::= \textbf{assign}\ \langle\text{symbol}\rangle\ \langle\text{expr}\rangle \\
\langle\text{assume-stmt}\rangle &::= \textbf{assume}\ \langle\text{expr}\rangle \\
\langle\text{call-stmt}\rangle &::= \textbf{call}\ \langle\text{symbol}\rangle \\
&\qquad \langle\text{expr-list}\rangle \\
&\qquad \langle\text{symbol-list}\rangle
\end{aligned}$$

$$\begin{aligned}
\langle\text{havoc-stmt}\rangle &::= \textbf{havoc}\ \langle\text{symbol}\rangle \\
\langle\text{jump-stmt}\rangle &::= \textbf{jump}\ \langle\text{symbol}\rangle\ \langle\text{symbol-list}\rangle \\
\langle\text{label-stmt}\rangle &::= \textbf{label}\ \langle\text{symbol}\rangle \\
\langle\text{symbol-list}\rangle &::= \langle\text{symbol}\rangle^* \\
\langle\text{expr}\rangle &::= \langle\text{var-expr}\rangle\ |\ \langle\text{op-expr}\rangle \\
\langle\text{var-expr}\rangle &::= \textbf{var}\ \langle\text{symbol}\rangle \\
\langle\text{op-expr}\rangle &::= \textbf{op}\ \langle\text{symbol}\rangle\ \langle\text{expr-list}\rangle \\
\langle\text{expr-list}\rangle &::= \langle\text{expr}\rangle^*
\end{aligned}$$

Fig. 1. Abstract syntax of K2 statements and expressions.

$$\begin{aligned}
\langle\text{program}\rangle &::= \langle\text{globals}\rangle\ \langle\text{init}\rangle\ \langle\text{functions-list}\rangle\ \langle\text{entrypoint}\rangle \\
\langle\text{globals}\rangle &::= \textbf{globals}\ \langle\text{var-decl-list}\rangle \\
\langle\text{init}\rangle &::= \textbf{init}\ \langle\text{expr}\rangle \\
\langle\text{functions-list}\rangle &::= \langle\text{function}\rangle^+ \\
\langle\text{function}\rangle &::= \textbf{function}\ \langle\text{symbol}\rangle\ \langle\text{var-decl-list}\rangle\ \langle\text{var-decl-list}\rangle\ \langle\text{var-decl-list}\rangle\ \langle\text{stmt-list}\rangle \\
\langle\text{entrypoint}\rangle &::= \textbf{entry}\ \langle\text{symbol}\rangle \\
\langle\text{stmt-list}\rangle &::= \langle\text{stmt}\rangle^+ \\
\langle\text{var-decl-list}\rangle &::= \langle\text{var-decl}\rangle^* \\
\langle\text{var-decl}\rangle &::= \textbf{var}\ \langle\text{symbol}\rangle\ \langle\text{sort}\rangle
\end{aligned}$$

Fig. 2. Abstract syntax of K2 programs.

Abstract syntax. We denote lists of elements with an overbar, i.e., $\bar{\cdot}$. If $\bar{a}$ is a list, $|\bar{a}|$ is its length, and if i is a natural number, $\bar{a}_i$ is the i-th element of $\bar{a}$. If e is an element, $\bar{a}\cdot e$ is the list obtained by appending e at the end of $\bar{a}$.

Definition 1 (Variables and Functions). *A* variable *is a symbol with an associated sort, as in the multi-sorted first-order logic. A* function *is a tuple $\langle f,\bar{a},\bar{r},\bar{l},\bar{\sigma}\rangle$, where:*

- *f, a symbol, is the name of the function;*
- *$\bar{a}$, a list of variables, are the formal parameters;*
- *$\bar{r}$, a list of variables, are the return variables;*
- *$\bar{l}$, a list of variables, are the local variables;*
- *$\bar{\sigma}$, a list of statements generated by the grammar of Figure 1, are the body.*

Given a list of variables $\bar{v}$, we define $\mathsf{syms}(\bar{v})$ as the corresponding set of symbols. Given a function $\langle f,\bar{a},\bar{r},\bar{l},\bar{\sigma}\rangle$, we denote with $\mathsf{syms}(f)$ the set $\mathsf{syms}(\bar{a})\cup\mathsf{syms}(\bar{r})\cup\mathsf{syms}(\bar{l})$. We extend the definition to lists of statements $\bar{\sigma}$ in the natural way. We now describe K2 programs, whose abstract syntax is shown in Figure 2.

Definition 2 (Programs). *A program P is a tuple $\langle \overline{g}, F, \iota, e \rangle$, where:*

- *$\overline{g}$, a list of variables, are the global variables;*
- *F is a partial mapping from symbols to functions;*
- *ι, a formula, is the constraint on initial states;*
- *e, a symbol in $\mathrm{dom}(F)$, is the entry point.*

Semantics. We use the standard notions of theory, interpretation, model, and satisfaction from many-sorted first-order logic and SMT [2]. In the following, we assume that we have fixed a theory T with equality that contains at least the sort Bool. Given an interpretation μ that is a model for T, we define the *evaluation* of an expression e (generated by the grammar of Figure 1) under μ, denoted $\mu[e]$, as $\mu[e] = \mu(v)$ for $e = \mathbf{var}\ v$ and $\mu[e] = \mu(o)(\mu[\overline{p}_1], \ldots, \mu[\overline{p}_n])$ for $e = \mathbf{op}\ o\ \overline{p}$ and $n = |\overline{p}|$. We denote with $\mu[v \mapsto e]$ the interpretation that maps v to e, and that agrees with μ everywhere else, and with $\mu[\backslash v]$ *any* interpretation that agrees with μ on all the symbols *except* v. Finally, if e is of sort Bool, we write $\mu \models e$ to denote that e evaluates to $\mathtt{true}$ under μ.

Definition 3 (Program states). *Pairs $\langle f, i \rangle$ where f is a function name and i is a natural number are called* program locations*. A* state *of a program P is a pair $s = \langle G, \overline{C} \rangle$ where:*

- *G is an interpretation for the global variables of P;*
- *$\overline{C}$ is the current call stack, a list of triples $\langle f, i, L \rangle$, where $\langle f, i \rangle$ is a program location and L is an interpretation of $\mathsf{syms}(f)$, i.e., of parameters, return variables, and local variables of $F(f)$.*

A state s is initial *if and only if $G \models \iota$, $|\overline{C}| = 1$ and $\overline{C}_1 = \langle e, 1, L \rangle$ for some L. Given a state s with $\overline{C}_{|\overline{C}|} = \langle f, i, L \rangle$, we define the* current interpretation μ *for s as $\mu(v) = G(v)$ for $v \in \mathsf{syms}(\overline{g})$ and as $\mu(v) = L(v)$ otherwise.*

We define the semantics for programs as a set of transition rules of the form $s \xrightarrow{\sigma} s'$, where s, s' are states and σ is a statement. We then call a *path* of a program P any sequence of transitions (possibly infinite) $s_0 \xrightarrow{\sigma_0} \ldots \xrightarrow{\sigma_i} s_{i+1} \ldots$ that complies with the transition rules and where s_0 is an initial state.

The rules are shown in Figure 3. In the definitions, we fix a program $P = \langle \overline{g}, F, \iota, e \rangle$ and use the following convenience functions, where f is a function name and i a natural number: $\mathsf{arg}(f, i)$ returns the variable $\overline{a}_i$ of the function $F(f)$; $\mathsf{ret}(f, i)$ returns the variable $\overline{r}_i$ of the function $F(f)$; $\mathsf{stmt}(f, i)$ returns the statement $\overline{\sigma}_i$ of $F(f)$; $\mathsf{stmts}(f)$ returns the list of statements $\overline{\sigma}$ of $F(f)$.

Reachability and liveness. We then say that a state s is *reachable* in P iff there exists a finite path $s_0 \xrightarrow{\sigma_0} \ldots \xrightarrow{\sigma_n} s$ that ends in s. Similarly, a *program location* $\langle f, i \rangle$ *is reachable* iff there exists a path as above in which $\sigma_n = \mathsf{stmt}(f, i)$[3]. Conversely, if no such path exists, then $\langle f, i \rangle$ is *unreachable*. The location $\langle f, i \rangle$

---

[3] Note that here we assume w.l.o.g. that all statements in a program are different, even when they are structurally equal, so the above definition is unambiguous.

assign-global: $\langle G, \overline{C} \cdot \langle f, i, L \rangle \rangle \xrightarrow{\mathsf{stmt}(f,i)} \langle G[v \mapsto \mu[e]], \overline{C} \cdot \langle f, i+1, L \rangle \rangle$ if $\mathsf{stmt}(f,i) = $ **assign** $v\ e$ and $v \in \mathsf{syms}(\overline{g})$;

assign-local: $\langle G, \overline{C} \cdot \langle f, i, L \rangle \rangle \xrightarrow{\mathsf{stmt}(f,i)} \langle G, \overline{C} \cdot \langle f, i+1, L[v \mapsto \mu[e]] \rangle \rangle$ if $\mathsf{stmt}(f,i) = $ **assign** $v\ e$ and $v \in \mathsf{syms}(f)$;

assume: $\langle G, \overline{C} \cdot \langle f, i, L \rangle \rangle \xrightarrow{\mathsf{stmt}(f,i)} \langle G, \overline{C} \cdot \langle f, i+1, L \rangle \rangle$ if $\mathsf{stmt}(f,i) = $ **assume** e and $\mu \models e$;

havoc-global: $\langle G, \overline{C} \cdot \langle f, i, L \rangle \rangle \xrightarrow{\mathsf{stmt}(f,i)} \langle G[\backslash v], \overline{C} \cdot \langle f, i+1, L \rangle \rangle$ if $\mathsf{stmt}(f,i) = $ **havoc** v and $v \in \mathsf{syms}(\overline{g})$;

havoc-local: $\langle G, \overline{C} \cdot \langle f, i, L \rangle \rangle \xrightarrow{\mathsf{stmt}(f,i)} \langle G, \overline{C} \cdot \langle f, i+1, L[\backslash v] \rangle \rangle$ if $\mathsf{stmt}(f,i) = $ **havoc** v and $v \in \mathsf{syms}(f)$;

call: $\langle G, \overline{C} \cdot \langle f, i, L \rangle \rangle \xrightarrow{\mathsf{stmt}(f,i)} \langle G, \overline{C} \cdot \langle f, i, L \rangle \cdot \langle g, 1, L' \rangle \rangle$ if $\mathsf{stmt}(f,i) = $ **call** $g\ \overline{e}\ \overline{r}$, where $L'(v) = \mu[\overline{e}_j]$ if $v = \mathsf{arg}(g, j)$.

return: $\langle G, \overline{C} \cdot \langle f, i, L \rangle \cdot \langle g, k, L'' \rangle \rangle \xrightarrow{\mathsf{stmt}(f,i)} \langle G', \overline{C} \cdot \langle f, i+1, L' \rangle \rangle$ if $\mathsf{stmt}(f,i) = $ **call** $g\ \overline{e}\ \overline{r}$ and $k > |\mathsf{stmts}(g)|$, where:
$$G'(v) = \begin{cases} \mu[\mathsf{ret}(g,j)] & \text{if } v = \overline{r}_j \\ G(v) & \text{otherwise} \end{cases} \quad \text{and} \quad L'(v) = \begin{cases} \mu[\mathsf{ret}(g,j)] & \text{if } v = \overline{r}_j \\ L(v) & \text{otherwise} \end{cases}$$

jump: $\langle G, \overline{C} \cdot \langle f, i, L \rangle \rangle \xrightarrow{\mathsf{stmt}(f,i)} \langle G, \overline{C} \cdot \langle f, k, L \rangle \rangle$ if $\mathsf{stmt}(f,i) = $ **jump** $\overline{t}$ and $\mathsf{stmt}(f,k) = $ **label** l with $l \in \overline{t}$;

label: $\langle G, \overline{C} \cdot \langle f, i, L \rangle \rangle \xrightarrow{\mathsf{stmt}(f,i)} \langle G, \overline{C} \cdot \langle f, i+1, L \rangle \rangle$ if $\mathsf{stmt}(f,i) = $ **label** l.

Fig. 3. Transition rules. In all the rules, μ denotes the current interpretation for the left-hand state of the rule.

is *infinitely-often reachable* iff there exists an infinite path $s_0 \xrightarrow{\sigma_0} \ldots \xrightarrow{\sigma_i} s_{i+1} \ldots$ in which for all indices j there exists an index $k > j$ such that $\sigma_k = \mathsf{stmt}(f, i)$. If no such path exists, then $\langle f, i \rangle$ is *eventually unreachable*. Finally, we say that $\langle f, i \rangle$ is *live* iff it is infinitely-often reachable *in all infinite paths of P*.

In K2, queries about reachability or liveness of program locations are expressed via *annotations* of **label** statements. Annotations are metadata that are attached to statements, in the form of key-value pairs, which do not affect the semantics of the program, but are meant to provide additional information that can be used by tools that manipulate the K2 program. Specifically, Kratos2 uses the following annotations to define properties:

error <id>: holds iff all labels annotated with the same <id> are unreachable;
notlive <id>: holds iff all labels annotated with the same <id> are eventually unreachable;
live <id>: holds iff all labels annotated with the same <id> are live.

These basic properties can be easily used to represent more common higher-level properties of programs, such as assertions and termination. For example, assertions can be reduced to reachability with a combination of `assume` and `jump` statements, whereas termination can be checked by adding a final self loop over a

label with an attached `live` annotation. Finally, eventual unreachability can be used to encode arbitrary LTL properties using the standard automata-theoretic approach combined with a symbolic encoding of the accepting automaton such as [22].[4]

3.1 Example

We conclude this section with a simple example of a C program and its equivalent formulation in K2. Both versions are shown in Figure 4. Most of the code is translated in a fairly direct way (with conditional statements and structured loops translated into nondeterministic jumps constrained by assumptions). However, since in K2, unlike in C, global variables are uninitialized by default, the K2 program contains an additional setup function (called `init_and_main` in the example) that sets `glbl` to zero before calling the original `main`. Another point to highlight is the use of the `:error` annotation (highlighted in bold) to model the C assertion.

4 Architectural View

This section describes the main components of Kratos2 and the flow of information among them. From the high-level point of view, Kratos2 is composed of the front-end c2Kratos, which converts the input C program to the K2 language, and of the core Kratos2, which is responsible for parsing, simplifications, transformations, and verification of K2 code. This separation helps to keep the core Kratos2 simple, as it does not have to handle the complex semantic nuances of C. Moreover, it makes it easy to add front-ends for new languages by writing a separate translator from the language in question to K2.

The front-end c2Kratos reads the input C file, builds its abstract syntax tree (AST) and then builds the corresponding K2 code in two passes. In the first pass, it converts the AST to an *extended* K2. Compared to the standard K2, the extended K2 also has primitives for pointers, records, complex loops, and compound instructions. These are removed in the second pass, by converting pointers to operations over maps, records to multiple variables, complex loops to sequences of assignments, jump instructions, and assumptions, and compound instructions to sequences of basic assignments to auxiliary variables.

The core Kratos2 consists of several components, whose relationships are visualized in Figure 5:

---

[4] In the case of LTL properties, the question arises as to what to consider as an atomic step of the program. This is both crucial and application-dependent: for example, in embedded software consisting of a "transition function" that is executed periodically, it might make sense to consider each call to such function as one step, whereas in other contexts a more fine-grained notion of step might be needed. K2 (and Kratos2) makes no commitment about this, providing only the support for eventual unreachability of label statements, which can always be defined unambiguously.

C version	K2 version

`int glbl;`
 	
`int f(int x)`
`{`
`  if (glbl > 0) {`
`    return x - 1;`
`  } else {`
`    glbl = 0;`
`    return x;`
`  }`
`}`
 	
`void main(void)`
`{`
`  int y;`
`  while (y > 0) {`
`    y = f(y);`
`  }`
`  assert(glbl == 0);`
`}`
	`(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)))`
	
	`(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)))`

Fig. 4. Example C program and its K2 translation.

CFG builder and simplifier reads the input K2 file and builds the corresponding interprocedural control flow graph (CFG). It then performs several simplifications of the CFG, such as constant propagation and lightweight slicing. The result can be used either by the interpreter, symbolic executor, or one of the encoders. The simplified CFG can also be converted back into a K2 representation.

Interpreter interprets the CFG using the externally provided inputs to guide the execution. The inputs contain new values for all havoc commands and also destination labels for all nondeterministic jump commands. The inputs can be provided by the user, a random generator, or by one of the verification engines. The last option is used for counterexample reconstruction and validation.

Transition system encoder encodes the CFG to a symbolic transition system over a suitable theory. The encoder first inlines all function calls in the program. It then encodes the resulting inlined program using *large block encoding* [4], which allows encoding larger acyclic subgraphs of the CFG by a single transition formula. The resulting transition system can be verified by one of the available verification back-ends, or converted to a textual representation in one of the available output formats (VMT [20], nuXmv [14], Btor2 [31], or AIGER [9]).[5]

---

[5] Depending on the features of the input K2 program, some of the verification backends or output formats might not be available. E.g., SAT-based engines are not available if the K2 program contains some infinite-state variables.
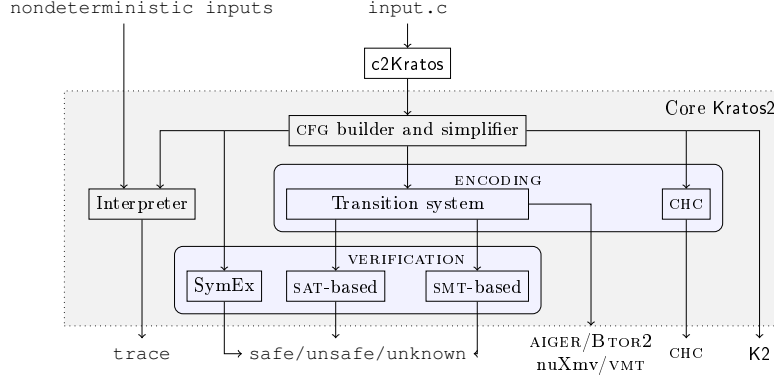
Fig. 5. Architecture of Kratos2.

CHC encoder converts the CFG to a set of Constrained Horn Clauses [11]. In contrast to the transition system encoder, the CHC encoder supports interprocedural analysis and recursive functions, encoded as a set of *non-linear* CHCs as described, e.g., in [28].

Symbolic executor implements a classical symbolic execution algorithm with iterative deepening to avoid getting stuck in long uninteresting branches. It supports (possibly recursive) K2 programs over arbitrary combinations of integers, reals, bit-vectors, floats, and arrays.

SMT-based engines encompass several SMT-based verification algorithms of symbolic transition systems. For reachability properties, Kratos2 implements standard bounded model checking (BMC) [7], k-induction [32], and IC3 with implicit predicate abstraction [18]. For liveness properties, we use a procedure combining liveness-to-safety reduction with ranking functions synthesis [23].

SAT-based engines encompass several verification algorithms of finite-state symbolic transition systems. Namely, for transition systems over the theory of bit-vectors and floats, Kratos2 offers BMC, k-induction, and different variants of IC3 [13], working over the bit-blasted Boolean transition system, for both reachability and liveness properties. Additionally, Kratos2 implements a dedicated engine for reachability properties in transition systems over the theory of bit-vectors, floats, and arrays similar to [30,10].

5 Implementation and Experimental Evaluation

Implementation. Core Kratos2 is implemented in C++ on top of the Math-SAT5 [19] SMT solver and the nuXmv [14] symbolic model checker. The SAT-based verification engine additionally makes use of the MiniSat [25] and CaDiCaL [8] SAT solvers. The front-end c2Kratos is implemented in Python and relies on pycparser for parsing of the input C program. Kratos2 is freely available for non-commercial purposes from https://kratos.fbk.eu.

Table 1. Solved benchmarks by the three compared tools. Column **U** shows the number of solved *unsafe* benchmarks, **S** of *safe* benchmarks, and **W** of *wrong* results.

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
floats	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

Experimental Setup. We performed an experimental evaluation to answer two research questions: (1) Is the K2 language expressive enough to efficiently represent realistic C programs? (2) Do the engines implemented in Kratos2 offer reasonable performance on realistic verification tasks? To this end, we considered all the C programs from the *ReachSafety* category of the 2022 edition of the annual software verification competition SV-COMP [3].The category consists of 5400 C programs divided into 12 benchmark families. We compared Kratos2 with VeriAbs 1.4.2 [24] and CPAchecker 2.2 [5], respectively the winner and runner-up of the *ReachSafety* category of SV-COMP 2022. Similarly to the approach used by CPAchecker, we executed Kratos2 in *sequential portfolio* mode, which successively runs symbolic execution, SMT-based IC3, SAT-based IC3, and SMT-based BMC with predetermined time-outs for each of the engines.

The experiments were performed on several identical PCs equipped with Intel Core i7-8700 CPU @ 3.20 GHz and 32 GiB of RAM. Each execution was limited to use a single CPU core, 15 minutes of CPU time, and 8 GiB of RAM. For reliable benchmarking, all experiments were executed using BenchExec [6]. A replication package describing the details of the setup is available at https://doi.org/10.5281/zenodo.7890411.

Results. To answer the first research question, we observe that from the total 5400 benchmarks, only 56 were not converted to K2 by c2Kratos due to unsupported floating point built-ins or features such as variable length arrays.

To answer the second research question, Table 1 shows the numbers of solved benchmarks by the individual tools and quantile plots in Figure 6 show their running times. The results show that Kratos2 is competitive with CPAchecker on all benchmark families except for eca. It is also competitive with VeriAbs on most benchmark families. There are 23 benchmarks uniquely solved by Kratos2, 48 by CPAchecker, and 1039 by VeriAbs. Moreover, both Kratos2 and VeriAbs produced no wrong results, unlike most other participants of SV-COMP.

We remark that CPAchecker is an established and optimized software verifier that regularly scores high in software verification competitions, and that VeriAbs implements algorithm selection heuristics, using both its own custom engines and
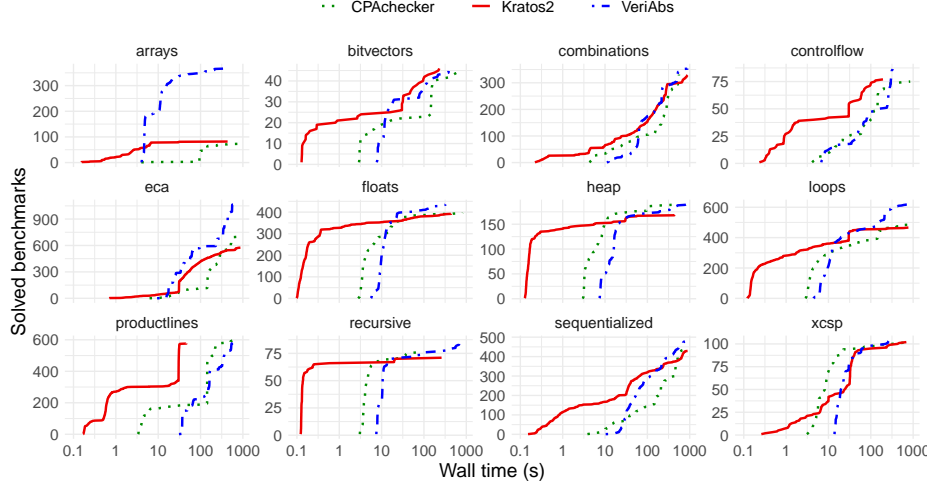
Fig. 6. Quantile plots of solved benchmarks for all three compared tools in individual benchmark families. The plot shows the number of benchmarks (y-axis) that were solved within the given number of seconds (x-axis).

external state-of-the-art verifiers. As such, it is not surprising that it performs much better than Kratos2 and CPAchecker on some of the families.

We conclude that the K2 language is expressive enough to efficiently capture a significant subset of C used in realistic programs. Furthermore, the verification engines implemented in Kratos2 mostly offer a performance comparable with state-of-the-art software verifiers.

6   Conclusions and Future Work

We have described Kratos2, a mature software verifier for imperative programs written in K2, a new intermediate verification language with a formal semantics based on SMT. Kratos2 is a complete rewrite of the original Kratos tool, offering significant extensions in functionalities and performance. The tool has already been successfully applied in various contexts, both industrial and academic.

As future work, we will consolidate the (currently alpha-quality) implementation of the ESST algorithm of the original Kratos [21] to handle multithreaded programs with cooperative scheduling. We will also investigate a tighter integration with CHC solvers to better handle recursive programs, as well as improved techniques to handle arrays and pointers such as [33,27]. On the language side, we plan to add support for contracts and pre-/post-conditions via annotations.

References

1. Arturo Amendola, Anna Becchi, Roberto Cavada, Alessandro Cimatti, Alberto Griggio, Giuseppe Scaglione, Angelo Susi, Alberto Tacchella, and Matteo Tessi. A model-based approach to the design, verification and deployment of railway interlocking system. In *ISoLA (3)*, volume 12478 of *Lecture Notes in Computer Science*, pages 240–254. Springer, 2020.
2. Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1267–1329. IOS Press, 2021.
3. Dirk Beyer. Progress on software verification: SV-COMP 2022. In *TACAS (2)*, volume 13244 of *Lecture Notes in Computer Science*, pages 375–402. Springer, 2022.
4. Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani. Software model checking via large-block encoding. In *FMCAD*, pages 25–32. IEEE, 2009.
5. Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011.
6. Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *Int. J. Softw. Tools Technol. Transf.*, 21(1):1–29, 2019.
7. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.
8. Armin Biere, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba entering the SAT Competition 2021. In *Proc. of SAT Competition 2021 – Solver and Benchmark Descriptions*, volume B-2021-1 of *Department of Computer Science Report Series B*, pages 10–13. University of Helsinki, 2021.
9. Armin Biere, Keijo Heljanko, and Siert Wieringa. AIGER 1.9 and beyond. Technical Report 11/2, Institute for Formal Models and Verification, Johannes Kepler University, 2011.
10. Per Bjesse. Word-level sequential memory abstraction for model checking. In Alessandro Cimatti and Robert B. Jones, editors, *Formal Methods in Computer-Aided Design, FMCAD 2008, Portland, Oregon, USA, 17-20 November 2008*, pages 1–9. IEEE, 2008.
11. Nikolaj S. Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II*, volume 9300 of *Lecture Notes in Computer Science*, pages 24–51. Springer, 2015.
12. Alberto Bombardelli, Marco Bozzano, Roberto Cavada, Alessandro Cimatti, Alberto Griggio, Massimo Nazaria, Edoardo Nicolodi, and Stefano Tonetta. COMPASTA: extending TASTE with formal design and verification functionality. In *IMBSA*, volume 13525 of *Lecture Notes in Computer Science*, pages 21–27. Springer, 2022.
13. Aaron R. Bradley. SAT-based model checking without unrolling. In *VMCAI*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 2011.
14. Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. In *CAV*, volume 8559 of *Lecture Notes in Computer Science*, pages 334–342. Springer, 2014.

15. Alessandro Cimatti, Sara Corfini, Luca Cristoforetti, Marco Di Natale, Alberto Griggio, Stefano Puri, and Stefano Tonetta. A comprehensive framework for the analysis of automotive systems. In *MoDELS*, pages 379–389. ACM, 2022.
16. Alessandro Cimatti, Luca Cristoforetti, Alberto Griggio, Stefano Tonetta, Sara Corfini, Marco Di Natale, and Florian Barrau. EVA: a Tool for the Compositional Verification of AUTOSAR Models. In *Proceedings of TACAS*, LNCS. Springer, 2023. To appear.
17. Alessandro Cimatti, Alberto Griggio, Andrea Micheli, Iman Narasamdya, and Marco Roveri. Kratos – A Software Model Checker for SystemC. In *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 310–316. Springer, 2011.
18. Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Infinite-state invariant checking with IC3 and predicate abstraction. *Formal Methods Syst. Des.*, 49(3):190–218, 2016.
19. Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2013.
20. Alessandro Cimatti, Alberto Griggio, and Stefano Tonetta. The VMT-LIB language and tools. In *SMT*, volume 3185 of *CEUR Workshop Proceedings*, pages 80–89. CEUR-WS.org, 2022.
21. Alessandro Cimatti, Iman Narasamdya, and Marco Roveri. Software model checking with explicit scheduler and symbolic threads. *Log. Methods Comput. Sci.*, 8(2), 2012.
22. Edmund M. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10(1):47–71, 1997.
23. Jakub Daniel, Alessandro Cimatti, Alberto Griggio, Stefano Tonetta, and Sergio Mover. Infinite-state liveness-to-safety via implicit abstraction and well-founded relations. In *CAV (1)*, volume 9779 of *Lecture Notes in Computer Science*, pages 271–291. Springer, 2016.
24. Priyanka Darke, Sakshi Agrawal, and R. Venkatesh. VeriAbs: A tool for scalable verification by abstraction (competition contribution). In *TACAS (2)*, volume 12652 of *Lecture Notes in Computer Science*, pages 458–462. Springer, 2021.
25. Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.
26. Jean-Christophe Filliâtre and Andrei Paskevich. Why3 – where programs meet provers. In *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.
27. Isabel Garcia-Contreras, Arie Gurfinkel, and Jorge A. Navas. Efficient modular SMT-based model checking of pointer programs. In *SAS*, volume 13790 of *Lecture Notes in Computer Science*, pages 227–246. Springer, 2022.
28. Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416. ACM, 2012.
29. K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 312–327. Springer, 2010.
30. Makai Mann, Ahmed Irfan, Alberto Griggio, Oded Padon, and Clark W. Barrett. Counterexample-guided prophecy for model checking modulo the theory of arrays. *Log. Methods Comput. Sci.*, 18(3), 2022.
31. Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2 , BtorMC and Boolector 3.0. In *CAV (1)*, volume 10981 of *Lecture Notes in Computer Science*, pages 587–595. Springer, 2018.

32. Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, volume 1954 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.
33. Cole Vick and Kenneth L. McMillan. Synthesizing history and prophecy variables for symbolic model checking. In Cezara Dragoi, Michael Emmi, and Jingbo Wang, editors, *VMCAI*, pages 320–340. Springer, 2023.