KRATOS

A Software Model Checker for SystemC

Version 1.0

Authors
Alessandro Cimatti
Alberto Griggio
Andrea Micheli
Iman Narasamdya
Marco Roveri

Contents

1 Introduction **2**

2 Download and Installation **2**

3 License **2**

4 Using Kratos **3**

5 User Options **3**

5.1 Options in the KRATOS Script 4

5.2 Options in the KRATOS Binary Executables 4

5.2.1 Options in the Sequential and Concurrent Analyses 5

5.2.2 Options only in the Sequential Analysis 7

5.2.3 Options only in the Concurrent Analysis 8

6 Bug Reports **8**

7 Frequently Asked Questions (FAQ) **9**

References **10**

1 Introduction

KRATOS is a new software model checker for SystemC. At the moment KRATOS only verifies safety properties in the form of program assertion. It provides two different analyses for verifying such properties. First, KRATOS implements a sequential analysis based on the lazy predicate abstraction [15] for verifying sequential C programs. To verify SystemC designs using this analysis, we rely on the translation from SystemC to a sequential C program, such that the resulting C program contains both the mapping of SystemC threads in the form of C functions and the encoding of the SystemC scheduler. Second, KRATOS implements a novel concurrent analysis, called ESST [12], that combines *Explicit* state techniques to deal with the SystemC *Scheduler*, with *Symbolic* techniques, based on lazy predicate abstraction, to deal with the *Threads*.

This manual is about how to use KRATOS. Details of KRATOS' architecture can be found in [11].

2 Download and Installation

KRATOS can be downloaded from KRATOS' website: `http://es.fbk.eu/tools/kratos`

We distribute KRATOS as a single tarball file consisting of the following files:

```
./kratos                -- the command-line front end
./swchecker             -- sequential analysis
./cswchecker            -- ESST analysis
./LICENSE               -- the license
./doc/manual.pdf        -- the user's manual
./benchmarks            -- some sample benchmarks
./benchmarks/sequential -- for the sequential analysis (i.e. the
                        -- usual device drivers and SSH examples)
./benchmarks/esst       -- concurrent benchmarks derived from Syste
```

The file `kratos` is a script file that functions as a command-line front end. We use this script to invoke the proper analysis based on the action argument. Options provided by the script can be found in Section 5.1.

Essentially KRATOS consists of two binary executables that run under Linux machine:

- `swchecker` implements the sequential analysis.
- `cswchecker` implements the concurrent analysis.

The `kratos` script works as an interface to these two executable.

The above tarball file also contains this manual and provides benchmarks for the sequential and the ESST concurrent analyses.

3 License

4 Using Kratos

KRATOS has been developed for the formal verification of SystemC. To do the verfication one first has to translate the SystemC design into either a sequential C program or a threaded C program. Each function of the C program models a thread in the SystemC design. The sequential C program also contains the encoding of the SystemC scheduler. Details of translation can be found in [12]. The sequential C program is then verified using KRATOS' sequential analysis, while the threaded C program is verified using KRATOS' ESST concurrent analysis.

To perform the translation from SystemC designs to C programs, one has to download the tarball files of SYSTEMC2C and PINAPA that are also available on KRATOS' website. The tarball file of SYSTEMC2C contains a script `systemc2c` that performs the translation.

Let us call the SystemC designs `sc_input.cc`. To verify the SystemC design using the sequential analysis, one should invoke the following commands:

```
> systemc2c --sequential sc_input.cc > input.c
> kratos --action=sequential input.c
```

To verify the design using the concurrent analysis, one should invoke the following commands:

```
> systemc2c --threaded sc_input.cc > input.c
> kratos --action=esst input.c
```

Alternatively, having the threaded C program, one can verify the SystemC design corresponding to the C program by first translating it to Promela code. To do so, one should invoke the following commands:

```
> systemc2c --threaded sc_input.cc > input.c
> kratos --action=translate-promela input.c > input.pml
```

The promela code in `input.pml` can be verified using the SPIN model checker [16] in the following way:

```
> spin -a input.pml
> cc -DSAFETY -o pan pan.c
> ./pan
```

where `cc` is a C compiler.

The sequential analysis of KRATOS can readily be used to verify general C programs. Let the C program be in `input.c`. To verify the program, one should invoke the following command:

```
> kratos --action=sequential input.c
```

One can also experiment with verification techniques implemented in NUSMV [9] to analyze the sequential C program. To do so, one should invoke the following command to translate the sequential C program into an SMV model.

```
> kratos --action=translate-smv input.c > input.smv
```

The SMV model is contained in the `input.smv` file.

5 User Options

KRATOS consists of the `kratos` script and two binary executables, `swchecker` and `cswchecker`. The `kratos` script functions as a command-line interface to the two binary executables.

5.1 Options in the Kratos Script

The `kratos` script provides several options for the verification. Subsequently, to simplify the description of options below, we call the input file `input.c`. In its simplest use the `kratos` script is invoked in the following way:

```
> kratos input.c
```

By default the script will invoke the ESST concurrent analysis to be performed on the input file. To specify additional options, the script can be invoked in the following way:

```
> kratos [options] input.c
```

Options, along with their default values, that are available in the `kratos` script are the following:

`--action=esst`

This option determines the action that the `kratos` script has to do on the input file. The default action is performing the ESST concurrent analysis on the input file. Other possible values include `sequential`, `translate-smv`, and `translate-promela`.

The value `sequential` instructs the script to perform the sequential analysis. The value `translate-smv` instructs the script to perform a translation from a sequential C program into SMV model, and print the resulting model to the standard output. The value `translate-promela` instructs the script to perform a translation from a threaded C program into Promela and print the resulting model to the standard output.

`--version`

This option instructs the `kratos` script to print the version and the license of Kratos. The script simply quits after printing the version and license.

`--help-esst`

This option instructs the `kratos` script to show the options specific to the ESST concurrent analysis.

`--help-sequential`

This option instructs the `kratos` script to show the options specific to the sequential analysis.

Other specified options that are not among the above options will be passed as options to the Kratos binary executables.

5.2 Options in the Kratos Binary Executables

Kratos' executables provide several options for the verification. Some options are only available in one of the analyses, but some of them are available in both analyses. In its simplest use Kratos is invoked in the following way:

```
> swchecker  input.c
> cswchecker input.c
```

To specify additional options to the executables one invoke them in the following way:

```
> swchecker  [options] input.c
> cswchecker [options] input.c
```

5.2.1 Options in the Sequential and Concurrent Analyses

The sequential analysis of KRATOS employs the lazy predicate abstraction. That is, KRATOS constructs a so-called abstract reachability tree (ART) [6] by (1) unwinding the control-flow automaton (CFA) and (2) performing abstract image computation with respect to the instruction that labels the unwound CFA edge. The ART itself represents reachable abstract state space.

The concurrent analysis of KRATOS employs the ESST algorithm that is based on the construction of abstract reachability forest (ARF) [12]. An ARF consists of a sequence of connected ARTs such that each ART represent the reachable abstract state space of the running thread.

Options, along with their default values, that are available both in the sequential analysis `swchecker` and in the concurrent analysis `cswchecker` are the following:

`--abstraction_type=AllSmt`

This option sets the type of abstraction that KRATOS performs in the computation of abstract post images. Possible values for this option are `AllSmt`, `Cartesian`, `SimpleStructural`, `LowLevelStructural`, and `Hybrid`. The values of the options reflect the type of the abstraction. The AllSMT and Cartesian abstractions are discussed in [17, 5]. KRATOS also provides hybrid abstraction that integrates BDDs and SMT solver, as described in [8], and structural abstraction, as described in [10].

Both hybrid and structural abstractions have further options:

`--hybrid_backjumping=true`

This option instructs the hybrid abstraction mechanism to use backjumping in TCC.

`--hybrid_cs_opt=true`

This option enables CS optimization when using the hybrid abstraction.

`--hybrid_use_threshold`

This option instructs the hybrid abstraction mechanism to use a threshold for BDD arrays.

`--hybrid_threshold=2`

This option specifies the threshold for BDD arrays in the hybrid abstraction.

`--structural_analysis_conjunct_analysis=true`

This option enables the conjuct analysis in the structural abstraction mechanism.

`--structural_analysis_generate_bdds=true`

This options makes the structural abstraction mechanism work directly in BDDs instead of working in NUSMV expression and then converting to BDD in one shot.

`--structural_analysis_inlining=true`

This option enables inlining in the structural abstraction mechanism.

`--structural_analysis_use_hybrid_quantification=false`

This option instructs the structural abstraction mechanism to use hybrid BDD-AllSMT techniques.

`--low_level_structural_analysis_enable_dagostino_optimization=true`

This option enables the dagostino's optimization in the low-level structural abstraction.

`--low_level_structural_analysis_generate_dnf=false`

This option instructs the low-level structural abstraction mechanism to generates disjunctive normal form (DNF) formula.

`--low_level_structural_analysis_left_conjuct_before_right=false`

This option instructs the low-level structural abstraction mechanism to assert left conjunct first before the right one.

`--simple_structural_analysis_enable_incrementality=false`

This option enables incrementality in the simple structural abstraction mechanism.

`--simple_structural_analysis_preassert_conjuncts=false`

This option instructs the simple structural abstraction mechanism to pre-assert conjuncts.

`--transition_method=SSA`

The transition method is concerned with how KRATOS expresses a CFA edge as a transition relation. A CFA edge consist of a block of instructions. Such a block is created using the large-block encoding technique. Possible values for transition method include `Simple`, `Compact`, `SSA`, and `LocalSSA`. All encodings but the last one represent transitions as relations from variables to their primed version. The simple and compact encoding fold the intermediate expressions of the transitions. The simple encoding creates fresh variables only at the meeting points in the CFG block, while the compact encoding exploits the if-then-else expression of NUSMV at the meeting point. The local SSA encoding performs SSA for the intermediate expressions. The SSA encoding creates pure SSA encoding for the transition.

`--restart=false`

When set to `true`, this option instructs KRATOS to abandon the current ART or ARF, depending on the analysis, and restart the their construction from the scratch.

`--cex_build_method=Noinc`

This option is concerned with how KRATOS builds counter examples. Possible values for this options are `Forward`, `Backward`, and `Noinc`. The first two values instruct KRATOS to build shortest counter examples in an incremental way. `Forward` instructs KRATOS to build the shortest prefix of abstract counter example by starting from the root of ART or ARF and going forward until it finds that the prefix is infeasible. `Backward` instructs KRATOS to build the shortest suffix of abstract counter example by going backward from the error state until it finds that the suffix is infeasible. `Noinc` instructs KRATOS to build the abstract counter example in a non-incremental way.

`--pivot=false`

This option is concerned with the location in an abstract counter example where KRATOS has to prune the constructed ART or ARF. An abstract counter example is essentially a path in the ART or ARF. The default point for pruning the tree or forest is the first node in the path such that the CFA location of the node gets refined. That is, there are newly discovered predicates that should be associated with the location.

KRATOS could also use the so-called pivot point to prune the ART or ARF. The location of pivot point depends on how the abstract counter example was built (see the previous option). If it is built using `Forward` or `Noinc`, then the pivot point is the start node of the path. If it is built using `Backward`, then the pivot point is the first node during the path construction such that the path becomes infeasible.

`--opt_cfa=false`

When set to `true`, KRATOS perform optimizations on the CFA. The applied optimizations are currently only intra-procedural, and include dead-code elimination, constant propagation, and cone of influence reduction over CFA nodes (not over instructions that label CFA edges).

`--dump_cfa=0`

This option instructs KRATOS to print the CFA in the dotted form. When specified with argument 0, KRATOS simply ignores this option. The other possible values are 1 and 2. With value 1 KRATOS prints the CFA and quits, and with 2 KRATOS prints the CFA and continues the analysis. The resulting dot file in `input.c.dot`.

`--extract_precision_from_cfa=false`

This option instructs KRATOS to extract predicates from CFA. These predicates are extracted from CFA edges that correspond to the conditions used in if-then-else or while-do constructs.

`--extract_precision_from_file=file_name`

This option instruct KRATOS to read predicates supplied from an external precision file `file_name`. The precision file consists of lines in the following form:

$$PRED[node] : \varphi_1; \ldots; \varphi_n;$$

where *node* is the id of a CFA node, and φ_i, for $i = 1, \ldots, n$, are the predicates. When $[node]$ is omitted then the corresponding predicates are tracked globally.

`--dump_precision=false`

This option instructs KRATOS to print the predicates into the file `input.c.prec`.

`--verbosity_level=0`

This option sets the verbosity level of KRATOS' output. The higher the level the more detailed the information that KRATOS outputs.

`--version=false`

This option instructs KRATOS to print its version and license and quit.

`--help=false`

This option instructs KRATOS to print the help message and quit.

5.2.2 Options only in the Sequential Analysis

Options, along with their default values, that are available only in the sequential analysis `swchecker` are the following:

`--expand=DFS`

This option sets the ART node expansion strategy. For the sequential analysis, KRATOS implements three different strategies: `BFS` for breadth-first search, `DFS` for depth-first search, and `Tops` for topological sort order. The default is `DFS`.

`--predicate_scope=Local`

This option sets the way KRATOS keeps track of discovered predicates. Possible values for this option include `Local`, `Global`, and `Glocal`. The value `Local` keeps track of the discovered predicates locally. That is, KRATOS associates the predicates with the CFA nodes where they are found and computes the abstract image based only on the predicates associated with the target of unwound edge. Keeping track of relevant predicates locally is described in [14].

The value `Global` keeps track of the discovered predicates globally as if the predicates were associated with every CFA node. The value `Glocal` mixes the local and global approaches. That is, with this value KRATOS works in the same way as the value `Local`, but keeps track of predicates that speak only global variables in a global way.

`--inline_function=0`

When specify with an integer value greater than 0, this option instructs KRATOS to inline the main function. The integer argument specifies the recursion depth.

`--dump_smv=false`

This option instruct KRATOS to translate the input C program into an SMV model. To use this option, `--inline_function` must be enabled with a recursion depth greater than 0, and the transition method must be set to `Simple` or `Compact`. The resulting SMV model is written into the file `input.c.smv`.

5.2.3 Options only in the Concurrent Analysis

Options, along with their default values, that are available only in the sequential analysis `swchecker` are the following:

`--thread_expand=BFS`

This option sets the strategy for choosing the ART of running threads. KRATOS implements breadth-first and depth-first search strategies. The former strategy is enabled by setting the value to `BFS`, while the latter to `DFS`. The default is `BFS`.

`--node_expand=DFS`

This option sets the strategy for the ARF node expansion. KRATOS implements breadth-first and depth-first search strategies. The former strategy is enabled by setting the value to `BFS`, while the latter to `DFS`. The default is `DFS`.

`--ab_interference=false`

This option enables the atomic block interference analysis.

`--po_reduce=false`

This option enables the partial-order reduction based on the persistent-set method.

`--po_reduce_sleep=false`

This option enables the partial-order reduction based on the sleep-set method.

`--inline_threaded_function=0`

When specify with an integer value greater than 0, this option instructs KRATOS to inline each function that models a thread. The integer argument specifies the recursion depth.

`--dump_promela`

This option instructs KRATOS to translate the input file to a Promela model. To use this option, `--inline_threaded_function` must be enabled with a recursion depth greater than 0. The Promela model will be written into the file `input.c.pml`.

Details of options related to partial order-reduction can be found in [13]. Discussion on translations to Promela can be found in [7].

6 Bug Reports

To report bugs, please send an email to `kratos@list.fbk.eu` with the following information to allow to reproduce the bug:

- the information about the platform KRATOS was run.
  - the output of `uname -a`;
  - the output of `ulimit -a` (for bash) or `limit` (for {t}csh).
- the options passed to KRATOS;
- the input file given in input to KRATOS.
- the output generated by KRATOS (e.g. under bash run KRATOS as follows:

  ```
  ./swchecker [options] input.c 2>&1 | tee OUTPUT.txt
  ```

  and then send the file `OUTPUT.txt`).

7 Frequently Asked Questions (FAQ)

How to specify properties? KRATOS at the moment only verifies safety properties in the form of program assertion. To specify a property one can place `assert(<expr>)` in the input program, where `expr` is a boolean expression.

How to model inputs? Inputs can be modelled by variables whose values are non-deterministic. KRATOS treat variables starting with `__NONDET` as non-deterministic variables.

How to use Kratos' benchmarks for other software model checkers? Different software model checker can have different ways of specifying properties. We have been comparing KRATOS with some model checkers including SATABS [4], BLAST [1], and CPACHECKER [3].

SATABS supports program assertions that KRATOS uses to specify properties. However, unlike KRATOS, SATABS models non-deterministic inputs using special function calls `nondet_<t>()`, where `t` is a type. Thus, to use KRATOS benchmarks, one has to replace the occurences of `__NONDET*` variables in the benchmarks by function calls `nondet_<t>()`.

BLAST can specify safety properties by checking the reachability of program labels that start with `ERROR`. To use KRATOS benchmarks, every occurence of `assert(<expr>)` must be replaced by

```
if (expr == 0) { error_fn(); }
```

where the function `error_fn` is defined as

```
void error_fn() { ERROR:; return; }
```

The corresponding non-deterministic variables `__NONDET` in BLAST are variables starting with `__BLAST_NONDET`. One then has to replace every occurence of variables starting with `__NONDET` with variables starting with `__BLAST_NONDET`.

CPACHECKER specifies properties using automata. It has an automaton to detect whether program labels starting with `ERROR` are reachable. One can then follow the above steps for BLAST to run CPACHECKER on KRATOS' benchmarks.

What are the current limitations of Kratos? KRATOS is still at its infancy and currently only supports a subset of the C language. KRATOS does not support the following constructs and features of the C language:

- Composite data structures, like C struct, union or enumeration.
- Arrays, pointers and pointer arithmetic.
- Dynamic creations of objects.
- Bit-precise operators.

The above limitations do not prevent users from experimenting with KRATOS on benchmarks used in literature on software model checkers. To alleviate the problems related to composite data structures, one can flatten the use of C structs and replace enumerators by their corresponding values. An array of size N can be replaced by N global variables and two functions that store a value to the array and select the array at some index.

When appropriate, pointers and pointer arithmetic can be replaced by uninterpreted functions. To have an uninterpreted function, one can declare a function without any definition. In particular, KRATOS treats bit-precise operators as uninterpreted functions.

Does one need to pre-process input file with CIL? CIL [2] (C Intermediate Language) is a high-level representation of C programs that also consists of a source-to-source transformation. CIL simplifies C programs and clarifies their ambiguous constructs. Some software model checkers like BLAST and CPACHECKER requires pre-processing the input file using CIL because they do not handle some C constructs. For example, CPACHECKER does not handle general while-do construct.

KRATOS does not require pre-processing the input file using CIL, but such a pre-processing can help alleviate the above mentioned limitations. For example, CIL can replace enumerators by their corresponding values. To run CIL on the input file, we recommend the following CIL options:

```
--domakeCFG --printCilAsIs --noPrintLn --noInsertImplicitCasts
--rmUnusedInlines --dooneRet
```

References

[1] *Blast*. http://mtc.epfl.ch/software-tools/blast/index-epfl.php.

[2] *CIL*. http://sourceforge.net/projects/cil.

[3] *CPAChecker*. http://cpachecker.sosy-lab.org.

[4] *Satabs*. http://www.cprover.org/satabs.

[5] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. Software model checking via large-block encoding. In *FMCAD*, pages 25–32. IEEE, 2009.

[6] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast. *STTT*, 9(5-6):505–525, 2007.

[7] D. Campana, A. Cimatti, I. Narasamdya, and M. Roveri. Formal Bug Finding for SystemC. Submitted for publication.

[8] R. Cavada, A. Cimatti, A. Franzén, K. Kalyanasundaram, M. Roveri, and R. K. Shyamasundar. Computing Predicate Abstractions by Integrating BDDs and SMT Solvers. In *FMCAD*, pages 69–76. IEEE, 2007.

[9] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A New Symbolic Model Checker. *STTT*, 2(4):410–425, 2000.

[10] A. Cimatti, J. Dubrovin, T. Junttila, and M. Roveri. Structure-aware computation of predicate abstraction. In *FMCAD*, pages 9–16. IEEE, 2009.

[11] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri. Kratos – A Software Model Checker for SystemC. Submitted for publication, 2011.

[12] A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri. Verifying SystemC: a Software Model Checking Approach. In *FMCAD*, pages 51–59, 2010.

[13] A. Cimatti, I. Narasamdya, and M. Roveri. Boosting Lazy Abstraction for SystemC with Partial Order Reduction. In *TACAS*, 2011. To appear.

[14] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, pages 232–244. ACM, 2004.

[15] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[16] G. J. Holzmann. Software model checking with SPIN. *Advances in Computers*, 65:78–109, 2005.

[17] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. Smt techniques for fast predicate abstraction. In *CAV*, volume 4144 of *LNCS*, pages 424–437. Springer, 2006.