# KRATOS – A Software Model Checker for SystemC

A. Cimatti, A. Griggio*, A. Micheli, I. Narasamdya, and M. Roveri

Fondazione Bruno Kessler — Irst
{cimatti,griggio,amicheli,narasamdya,roveri}@fbk.eu

**Abstract.** The growing popularity of SystemC has attracted research aimed at the formal verification of SystemC designs. In this paper we present KRATOS, a software model checker for SystemC. KRATOS verifies safety properties, in the form of program assertions, by allowing users to explore two directions in the verification. First, by relying on the translation from SystemC designs to sequential C programs, KRATOS is capable of model checking the resulting C programs using the symbolic lazy predicate abstraction technique. Second, KRATOS implements a novel algorithm, called ESST, that combines *Explicit* state techniques to deal with the SystemC *Scheduler*, with *Symbolic* techniques to deal with the *Threads*. KRATOS is built on top of NUSMV and MATHSAT, and uses state-of-the-art SMT-based techniques for program abstractions and refinements.

## 1 introduction

Formal verification of SystemC has recently gained significant interests [20, 14, 19, 24, 17, 23, 10]. Despite its importance, verification of SystemC designs is hard and challenging. A SystemC design is a complex entity comprising a multi-threaded program where scheduling is cooperative, according to a specific set of rules [22], and the execution of threads is mutually exclusive.

In this paper we present KRATOS, a new software model checker for SystemC. KRATOS provides two different analyses for verifying safety properties (in the form of program assertions) of SystemC designs. First, KRATOS implements a sequential analysis based on lazy predicate abstraction [16] 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 [10], that combines *Explicit* state techniques to deal with the SystemC *Scheduler*, with *Symbolic* techniques, based on lazy predicate abstraction, to deal with the *Threads*.

In this paper we describe the verification flow of KRATOS, its architecture, and the novel functionalities that it features. Due to space limit, the results of an experimental evaluation that compares KRATOS with other model checkers on various benchmarks can be found in [9]. KRATOS is available for download at `https://es.fbk.eu/tools/kratos`.

* Supported by Provincia Autonoma di Trento and the European Community's FP7/2007-2013 under grant agreement Marie Curie FP7 - PCOFUND-GA-2008-226070 "progetto Trentino", project ADAPTATION.

## 2 Verification Flow

The flow of SystemC verification using KRATOS consists of two directions, as shown in Figure 1. The first direction relies on the translation of a SystemC design into a sequential C program, such that the C program contains a function modelling each of the SystemC threads and the encoding of the SystemC scheduler. KRATOS implements a sequential analysis that model checks sequential C programs. This sequential analysis is essentially lazy predicate abstraction [16], which is based on the construction of an abstract reachability tree (ART) by unwinding the control-flow automaton (CFA) of the C program. The ART itself represents the reachable abstract state space. The sequential analysis that KRATOS implements is not restricted to the results of SystemC translations, but it can also handle general sequential C programs.

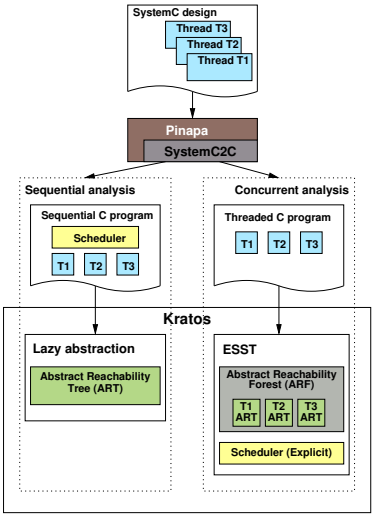


**Fig. 1.** The SystemC verification flow.

The second direction uses the concurrent analysis, which is the ESST algorithm, to model check SystemC designs. Similar to the first direction, in the second direction the SystemC design is translated into a so-called threaded C program that contains a function for modelling each SystemC thread. But, unlike the sequential C program above, the encoding of the SystemC scheduler is no longer part of the threaded C program. The SystemC scheduler itself is now part of the ESST algorithm and its states are tracked explicitly. ESST is based on the construction of an abstract reachability forest (ARF) where each tree in the forest is an ART of the running thread. The ESST algorithm is described in detail in [10].

Translations from SystemC designs into sequential and threaded C programs are performed by SYSTEMC2C, a new back-end of PINAPA [21].

## 3 Architecture

The architecture of KRATOS is shown in Figure 2. It consists of a front-end that includes a parser for C programs, a type checker, a CFA encoder, and static data-flow analyses and optimization phases.

The parser translates a textual C program into its abstract syntax tree (AST) representation. The AST is then traversed by the type checker to build a symbol table. The CFA encoder builds a CFA or a set of CFAs from the AST. Currently, the CFA encoder provides three encodings: small-block encoding (SBE), basic-block encoding (BBE), and large-block encoding (LBE). In SBE each block consists only of at most one statement. In BBE each block is a sequence of statements that is always entered at the beginning and exited at the end. In LBE, as described in [1], each block is the largest directed acyclic fragment of the CFA. LBE improves performances by reducing the number of abstract post image computation [1].

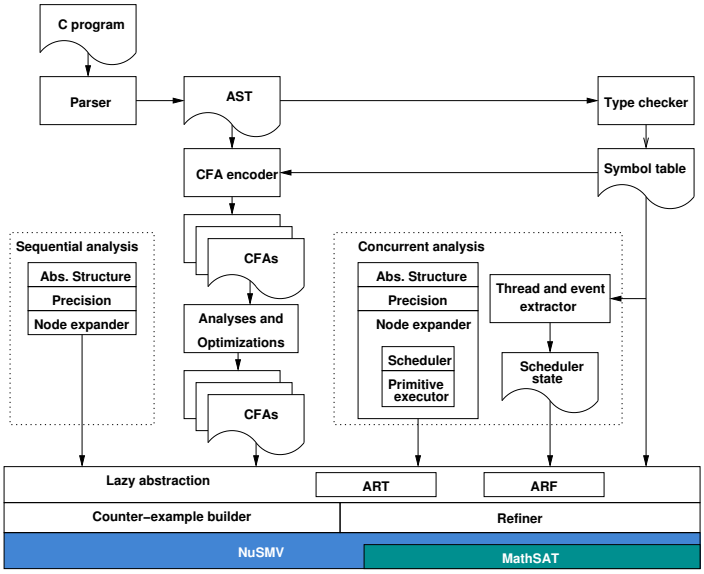


**Fig. 2.** The architecture of KRATOS.

Static analyses and optimizations implemented by KRATOS include a simple cone-of-influence reduction that removes nodes of CFAs that do not lead to the error nodes, dead-code elimination, and constant propagation.

The sequential analysis is a reimplementation of the same analysis performed by existing software model checkers based on the lazy predicate abstraction, like BLAST [2] and CPACHECKER [3]. The analysis consists of an abstraction structure, a precision, and a node expander. The abstraction structure contains the representation of abstract states that label ART nodes. A state typically consists of a location (or node) in CFA, a formula denoting the abstract data state, and a stack that keeps track of the trace of function calls. The structure also implements the coverage criteria that stop the expansion of ART nodes. The precision encodes the mapping from locations in CFA to sets of predicates that have been discovered so far. These predicates are relevant predicates used to compute the abstract post images. The node expander expands an ART node by (1) unwinding each of the outgoing edges of the CFA node in the state labelling the ART node, and (2) computing the abstract post image of the state with respect to the statement labelling the outgoing edge. The node expander currently implements depth-first search (DFS), breadth-first search (BFS), and topological ordering strategies for expanding nodes.

For the concurrent analysis, we extract the threads and events from the input threaded C program to create the initial state of scheduler. In this analysis, the node expander is also equipped with a scheduler and a primitive executor. The scheduler explores all possible schedules given a scheduler state as an input. The primitive executor executes calls to functions that modify the state of scheduler. The executor only assumes that the actual arguments of the calls are known statically.

We remark that, the architecture of the concurrent analysis does not assume that the scheduler is a SystemC scheduler. In fact any implementation of a cooperative scheduler with one exclusively running thread in each schedule can be plugged into the analysis.

The sequential and concurrent analyses construct, respectively, the ART and the ARF by following the standard counterexample-guided abstraction refinement (CEGAR) loop [13]. When the analyses cannot reach any error location, then the analyzed program is safe (no assertion violation can occur). When the analyses reach an error location, then the counterexample builder builds a counterexample by constructing the path from the node labelled with the error location to the root of the ART, or to the root of the first ART in the ARF. If the counterexample is non-spurious, in the sense that the formula representing it is satsifiable, then the analyzed program is unsafe. If the counterexample is spurious, then it is passed to the refiner. The refiner tries to refine the precision by discovering new predicates that need to be kept track of by using the unsatisfiable core or interpolation based techniques as described in [15].

KRATOS is built on top of an extended version of NUSMV [7], which is tightly integrated with the MATHSAT SMT solver [5]. KRATOS relies on NUSMV and MATHSAT for abstraction computation, for representing the abstract state within each ART, for the coverage check, for checking the satisfiability of expressions representing counter examples, and for extracting the unsatisfiable core and for generating sequence of interpolants from counterexample paths.

## 4 Novel Functionalities

KRATOS offers the following novel functionalities.

ESST *algorithm.* The translation from SystemC designs to sequential C programs enables the verification of SystemC using the "off-the-shelf" software model checking techniques. However, such a verification is inefficient because the abstraction of SystemC scheduler is often too aggresive, and thus requires many refinements to reintroduce the abstracted details. The ESST algorithm attacks such an inefficiency by modelling the scheduler precisely, and, as shown in [10], outperforms the SystemC verification through sequentialization.

*Partial-order reduction.* Despite its relative effectiveness, ESST still has to explore a large number of thread interleavings, many of which are redundant. Such an exploration degrades the run time performance and yields high memory consumptions. Partial-order reduction (POR) is a well-known technique for tackling the state explosion problem by exploring only a representative subset of all possible interleavings. Recently a POR technique has been incorporated in the ESST algorithm [11]. KRATOS currently implements a POR technique based on persistent set, sleep set, and a combination of both.

*Advanced abstraction techniques.* KRATOS implements cartesian and boolean abstraction techniques that are implemented in BLAST and CPACHECKER. In addition, KRATOS also implements hybrid predicate abstraction that integrates BDDs and SMT solvers, as described in [12], and structural abstraction, as described in [8].

*Translators.* KRATOS is capable of translating the sequential and threaded C to the input languages of other verification engines. For example, KRATOS can translate se-

quential a C program into an SMV model. By such a translation, one can then use the model checking algorithms implemented by, for example, NUSMV [7] to verify the C program. In particular one can experiment with the bounded model checking (BMC) [4] technique of NUSMV that does not exist in KRATOS.

*Under-approximation.* KRATOS is also able to generate under-approximations for quick bug hunting. To this extent, KRATOS has recently featured a translation from threaded C programs into PROMELA models [6]. Such a translation enables the verification by under-approximations using the SPIN model checker [18].

*Transition encoding.* Each block in the CFA is translated into a transition expressed by a NUSMV expression. We have observed that different encodings for transitions can affect the performance of KRATOS. In particular, the encoding for the transitions affect the performance of MATHSAT in terms of abstraction computations and also lead MATHSAT to yielding different interpolants, and thus different discovered predicates. KRATOS provides several different encodings for transitions. They differs in the number of variables needed to encode the transition of each block of the CFA, from the fact that intermediate expressions are folded or not, or whether NUSMV if-then-else expressions are used to compactly represent intermediate expressions. Details about these encodings can be found in the user manual downloadable from the KRATOS' website. Depending on the nature of the problem, the availability of several encodings allows users to choose the most effective one for tackling the problem.

## 5 Conclusion and Future Work

We have presented KRATOS, a software model checker for SystemC. KRATOS provides two different analyses for verifying SystemC designs: sequential and concurrent analyses. The sequential analysis, based on the lazy predicate abstraction, verifies the C program resulting from the sequentialization of the SystemC design. The concurrent analysis, based on the novel ESST algorithm, combines explicit state techniques with lazy predicate abstraction to verify threaded C program that models a SystemC design. The results of an experimental evaluation, reported in [9], shows that ESST algorithm, for the verification of the considered SystemC benchmarks, outperforms all the other approaches based on sequential analysis. On the considered pure sequential benchmarks, the sequential analysis shows better performance than other state-of-the-art approaches for the majority of the benchmarks.

For future work, we will extend KRATOS to handle a larger subset of C constructs like data structures, arrays and pointers (which are currently treated as uninterpreted functions) and to be able to take into account the bit-precise semantics of operations. We will investigate how to extend the ESST approach to deal with symbolic primitive functions to generalize the scheduler exploration. We would also like to combine the over-approximation analysis, based on the lazy abstraction, with an under-approximation analysis, based on PROMELA translation or BMC. Finally, we will consider to extend the ESST techniques to the verification of concurrent C programs from other application domains (e.g. robotics, railways), where different scheduling policies have to be taken into account

## References


1. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: FMCAD. pp. 25–32. IEEE (2009)
2. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. STTT 9(5-6), 505–525 (2007)
3. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate Abstraction with Adjustable-Block Encoding. In: FMCAD. pp. 189–197 (2010)
4. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without bdds. In: TACAS. LNCS, vol. 1579, pp. 193–207. Springer (1999)
5. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MathSAT 4 SMT Solver. In: CAV. LNCS, vol. 5123, pp. 299–303. Springer (2008)
6. Campana, D., Cimatti, A., Narasamdya, I., Roveri, M.: An Analytic Evaluation of SystemC Encodings in Promela, submitted for publication
7. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NuSMV: A New Symbolic Model Checker. STTT 2(4), 410–425 (2000)
8. Cimatti, A., Dubrovin, J., Junttila, T., Roveri, M.: Structure-aware computation of predicate abstraction. In: FMCAD. pp. 9–16. IEEE (2009)
9. Cimatti, A., Griggio, A., Micheli, A., Narasamdya, I., Roveri, M.: KRATOS – A Software Model Checker for SystemC. Tech. rep., FBK-irst (2011), `https://es.fbk.eu/tools/kratos`
10. Cimatti, A., Micheli, A., Narasamdya, I., Roveri, M.: Verifying SystemC: a Software Model Checking Approach. In: FMCAD. pp. 51–59 (2010)
11. Cimatti, A., Narasamdya, I., Roveri, M.: Boosting Lazy Abstraction for SystemC with Partial Order Reduction. In: TACAS. pp. 341–356. LNCS (2011)
12. Cimatti, A., Franzén, A., Griggio, A., Kalyanasundaram, K., Roveri, M.: Tighter integration of BDDs and SMT for Predicate Abstraction. In: Proc. of DATE. pp. 1707–1712. IEEE (2010)
13. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
14. Große, D., Drechsler, R.: CheckSyC: an efficient property checker for RTL SystemC designs. In: ISCAS (4). pp. 4167–4170. IEEE (2005)
15. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL. pp. 232–244. ACM (2004)
16. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL. pp. 58–70 (2002)
17. Herber, P., Fellmuth, J., Glesner, S.: Model checking SystemC designs using timed automata. In: CODES+ISSS. pp. 131–136. ACM (2008)
18. Holzmann, G.J.: Software model checking with SPIN. Advances in Computers 65, 78–109 (2005)
19. Kroening, D., Sharygina, N.: Formal verification of SystemC by automatic hardware/software partitioning. In: MEMOCODE. pp. 101–110. IEEE (2005)
20. Moy, M., Maraninchi, F., Maillet-Contoz, L.: Lussy: A toolbox for the analysis of systems-on-a-chip at the transactional level. In: ACSD. pp. 26–35. IEEE (2005)
21. Moy, M., Maraninchi, F., Maillet-Contoz, L.: Pinapa: an extraction tool for SystemC descriptions of systems-on-a-chip. In: EMSOFT. pp. 317–324. ACM (2005)
22. Tabakov, D., Kamhi, G., Vardi, M.Y., Singerman, E.: A Temporal Language for SystemC. In: FMCAD. pp. 1–9. IEEE (2008)
23. Tabakov, D., Vardi, M.Y.: Monitoring Temporal SystemC Properties. In: MEMOCODE. pp. 123–132 (2010)
24. Traulsen, C., Cornet, J., Moy, M., Maraninchi, F.: A SystemC/TLM Semantics in Promela and Its Possible Applications. In: SPIN. LNCS, vol. 4595, pp. 204–222. Springer (2007)