

# A NuSMV Extension for Graded-CTL Model Checking

Alessandro Ferrante<sup>1</sup>, Maurizio Memoli<sup>2</sup>, Margherita Napoli<sup>2</sup>,  
Mimmo Parente<sup>2</sup>, and Francesco Sorrentino<sup>3</sup>

<sup>1</sup> Embedded Systems Research Unit, Fondazione Bruno Kessler  
ferrante@fbk.eu

<sup>2</sup> Dipartimento di Informatica ed Applicazioni, Università degli Studi di Salerno  
{memoli,napoli,parente}@dia.unisa.it

<sup>3</sup> Computer Science Department, University of Illinois at Urbana Champaign  
sorrent1@uiuc.edu

**Abstract.** Graded-CTL is an extension of CTL with graded quantifiers which allow to reason about either *at least* or *all but* any number of possible futures. In this paper we show an extension of the NuSMV model-checker implementing symbolic algorithms for graded-CTL model checking. The implementation is based on the CUDD library, for BDDs and ADDs manipulation, and includes also an efficient algorithm for multiple counterexamples generation.

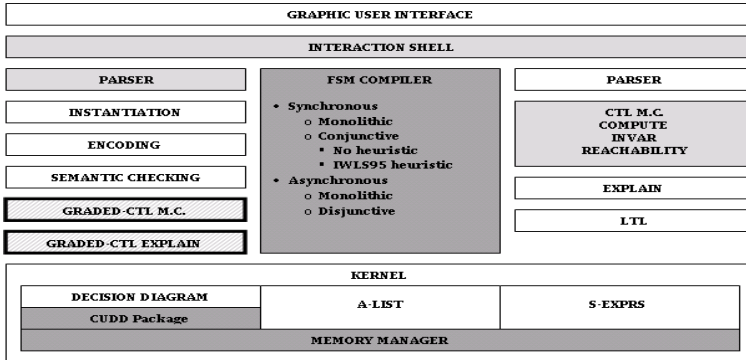
## 1 Description and Architecture

In this paper we introduce a new model-checker which is an extension of NuSMV [CCG<sup>+</sup>02], an efficient and easy to extend re-implementation and integration of the SMV model-checker [McM93, CMCH96]. Our tool implements symbolic algorithms for graded-CTL model checking<sup>1</sup>. Graded-CTL [FNP08, FNP09a] is an extension of CTL with graded quantifiers that allow to reason about either *at least* or *all but* any number of possible futures. For example, the formula  $E^{>k}\mathcal{F}(\text{critic1} \wedge \text{critic2})$  expresses that there are more than  $k$  possibilities (i.e.  $k$  different paths in the Kripke structure modeling the system) to violate the mutual exclusion property. Formulas of these types cannot be expressed in CTL and not even in  $\mu$ -calculus (though they can be easily reduced, in exponential time, to equivalent *graded*  $\mu$ -calculus formulas, [KSV02]). Graded-CTL formulas can be used to determine whether there are more than a given number of bad behaviors of a system: this, in the model-checking framework, means that one can verify the existence of a user-defined number of counterexamples for a given specification and can generate them, in a unique run of the model-checker.

Symbolic Model Checking [BCM<sup>+</sup>90] applied to CTL is known to behave efficiently, especially in hardware verification, and has been widely studied and implemented in a lot of well known model-checkers. In [FNP09b] *symbolic algorithms* to solve the graded-CTL model checking problem are shown. Graded-CTL NuSMV includes a smart implementation of these algorithms based on

---

<sup>1</sup> For better readability, we call this extension *Graded-CTL NuSMV*.



**Fig. 1.** The architecture of Graded-CTL NuSMV. Light-gray and dark-gray modules are NuSMV modules with some minor and major modifications, respectively. Modules with bold borders are new modules added to support graded-CTL model checking.

the CUDD library [Som05], which is used for an efficient manipulation of sets and multi-sets via *Binary Decision Diagrams* (BDDs) and *Algebraic Decision Diagrams* (ADDs).

NuSMV is a versatile tool, implementing BDD-based and SAT-based model checking, and processing files written in an extension of the SMV language. With this language it is possible to describe finite-state machines (declaration and instantiation of modules and processes are used to describe synchronous and asynchronous composition) and to express specifications in CTL and LTL. NuSMV works either in batch or in interactive mode, with a textual interactive shell.

Our tool preserves the structure and the modularity of NuSMV: each module implements a set of functionalities and communicates with the others via a precisely defined interface. Fig. 1 shows the architecture of Graded-CTL NuSMV pointing out the modified and the completely new modules. The *Interactive Shell*, the *Parser* and the *Compiler* modules are responsible for processing command lines and input files (including syntactic correctness check) and also for building the resulting parse tree and the BDD representation. They have been integrated with the functions and the commands to handle and represent graded-CTL formulas, as well. The *Kernel* module provides the low level functionalities to handle data structures (BDDs, etc.) and memory allocation. It has been modified with a re-implementation and an extension of the cache to store the grading values. The *Model Checking* module is the core of the NuSMV tool: it provides all the functionalities to solve the verification problem. We chose to preserve the structure of NuSMV, and thus we only modified the functions responsible for the invocation of the low level model checking routines, and implemented the graded-CTL model checking algorithms in a new module called *Graded-CTL Model Checking*. The implementation of these algorithms required also modifications to the basic operators of the CUDD package. In particular, an implementation of the *AddAnd-Abstract* operation on ADDs and a bounded leaf-value implementation of the other

operations on ADDs have been included in the package. The latter has led to a considerable improvement in terms of speed and space.

A remarkable feature of the symbolic algorithms we implemented is that they have been explicitly designed to efficiently derive multiple counterexamples for a given path formula, see [FNP09b] for other deeper arguments. In our implementation, we fully exploit this characteristic by using the partial results of the verification phase to derive the needed counterexamples. To do that, the Graded-CTL Model Checking module works interacting with the other new module, *Graded-CTL Explain*, responsible for the generation of the counterexamples (see Sect. 2).

Although no absolute criteria are available to evaluate our tool (since, at the best of our knowledge, no tools for similar computations are currently available), the experimental results are very promising. Indeed, our experiments evidenced that no substantial overhead, both in the time and in the number of BDDs, is required to process graded-CTL formulas, with respect to the classical CTL ones, even by increasing the values of the grading constants in the formulas. We are also collaborating with the NuSMV development team to include our extension in the official release. The list of our experiments and the package for graded-CTL can be found at <http://gradedctl.dia.unisa.it>.

## 2 Counterexamples

A really important feature of Graded-CTL NuSMV is the multiple counterexamples generation. To provide this functionality, the tool has been designed to store the counterexamples in a tree in which each root-to-leaf path represents a distinct counterexample. The computation of this tree is based on three different algorithms used for the evidences generation of  $E^{>k}\mathcal{X}$ ,  $E^{>k}\mathcal{G}$ ,  $E^{>k}\mathcal{U}$  respectively. The  $E^{>k}\mathcal{X}\psi_1$  case is trivial, so we focus on the others two cases. How to pick the successors and the number of paths to consider are crucial decisions in order to correctly compute the evidences. To compute an evidence-tree for a formula  $E^{>k}\theta$  (with either  $\theta = \mathcal{G}\psi_1$  or  $\theta = \psi_1\mathcal{U}\psi_2$ ) starting from a state  $s$ , our algorithm uses the sets  $[E^{>i}\theta] \setminus [E^{>i+1}\theta]^2$  ( $0 \leq i \leq k$ ) computed during the verification phase. The algorithm starts with  $i=0$  and computes the set  $POST$  of the successors of the state  $s$  that are in  $[E^{>i}\theta] \setminus [E^{>i+1}\theta]$ . Then, recursively generates  $i + 1$  evidences from each state of  $POST$  and store them in a tree rooted in  $s$ . At this point, the algorithm halts if  $k + 1$  evidences have been generated. Otherwise the procedure is repeated for  $i + 1$ . Notice that the evidences are pairwise distinct because the algorithm uses the sets  $[E^{>i}\theta] \setminus [E^{>i+1}\theta]$  that are pairwise disjoint. The decision to start with  $i = 0$  is motivated by the fact that, in this way, the algorithm generates a *wide* evidence-tree (in opposition to a *tall* evidence-tree that can be obtained by starting with  $i = k$ ) by distinguishing the evidences “as soon as possible”. Indeed, in some cases a tall evidence-tree may be constituted by paths which differ only for the number of times that they traverse a self-loop, while a wide tree includes more significative evidences (see [FNP09b] for a deeper discussion).

<sup>2</sup> For a graded-CTL formula  $\varphi$  we denote with  $[\varphi]$  the set of states where  $\varphi$  holds. For better readability, we assume  $[E^{>k+1}\theta] = \emptyset$ .

After that the evidence-tree has been generated, the occurrence of cycles in the paths should be detected and the wrong behaviors should be printed in an intelligible way for the user. In order to detect cycles, a DFS on the tree is performed. The next step is to verify whether the cycle is sink or not: we compute the evidence-tree so that *the node exposing the existence of a cycle has a successor iff the cycle is non-sink*. Finally, the printing phase starts. This phase is important because the counterexamples need to be exposed in such a way that the user can get benefits from the graded logic. In order to have a good balance between information completeness and representation compactness, the tool outputs, for each trace and for each state, a sequential number that identifies the trace and the position within it. Then, for each state, only the *differences* between the previous state are printed. The only exception is for the root of the tree and for those states starting a new branch in the tree. If a group of traces share the same prefix, then it is printed only once and for each new trace only the path from the branch to the leaf is printed. Moreover, the first state of a cycle and the last state of a non-sink cycle are marked with the labels *loop starts here* and *end loop*, respectively. Finally, when a non-sink cycle is found, the trace traversing it only once, is printed, while the traces traversing the cycle more than once are ignored.

## References

- [BCM<sup>+</sup>90] Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking:  $10^{20}$  states and beyond. In: Proc. of LICS 1990, pp. 428–439. IEEE Computer Society Press, Los Alamitos (1990)
- [CCG<sup>+</sup>02] Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV Version 2: An Open-Source Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
- [CMCH96] Clarke, E.M., McMillan, K.L., Aguiar Campos, S.V., Hartonas-Garmhausen, V.: Symbolic Model Checking. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 419–427. Springer, Heidelberg (1996)
- [FNP08] Ferrante, A., Napoli, M., Parente, M.: CTL Model-Checking with Graded Quantifiers. In: Cha, S.(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 20–32. Springer, Heidelberg (2008)
- [FNP09a] Ferrante, A., Napoli, M., Parente, M.: Model Checking for Graded-CTL. *Fundamenta Informaticae* 96(3), 323–330 (2009)
- [FNP09b] Ferrante, A., Napoli, M., Parente, M.: Graded-CTL: Satisfiability and Symbolic Model Checking. In: Cavalcanti, A. (ed.) ICFEM 2009. LNCS, vol. 5885, pp. 306–325. Springer, Heidelberg (2009)
- [KSV02] Kupferman, O., Sattler, U., Vardi, M.Y.: The Complexity of the Graded  $\mu$ -Calculus. In: Voronkov, A. (ed.) CADE 2002. LNCS (LNAI), vol. 2392, pp. 423–437. Springer, Heidelberg (2002)
- [McM93] McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers, Dordrecht (1993)
- [Som05] Somenzi, F.: CUDD: CU Decision Diagram Package, Release 2.4.1 (2005), <http://vlsi.colorado.edu/~fabio/CUDD/>