"PyNuSMV: NuSMV as a Python Library"

Busard, Simon ; Pecheur, Charles

Abstract

NuSMV is a state-of-the-art model checker providing BDD-based and SAT-based techniques and a rich modeling language. While the tool is powerful, it is hard to customize it because of the size and complexity of its code base (more than 200K LOC). This paper presents PyNuSMV, a Python framework for prototyping and experimenting with BDD-based model-checking algorithms based on NuSMV. PyNuSMV provides a rich and flexible programmable platform to implement new logics and experiment with custom model-checking algorithms. Thanks to PyNuSMV, it is possible to use NuSMV functionalities without understanding its whole code base or struggling with implementation details such as memory management. PyNuSMV has already been used to implement model-checking algorithms for rich logics such as ARCTL and CTLK. This paper describes the structure and usage of PyNuSMV, illustrates its use by re-implementing CTL model checking, and reports initial performance results showing negligible impact compared to native NuSMV.

Busard, Simon ; Pecheur, Charles. *PyNuSMV: NuSMV as a Python Library*. 5th NASA Formal Methods Symposium (NFM 2013) (NASA Ames Research Center, Moffett Field, CA, USA, du 14/05/2013 au 16/05/2013). In: Guillaume Brat, Neha Rungta, Arnaud Venet, *NASA Formal Methods*, (2013), p.453-458

PyNuSMV: NuSMV as a Python Library

Simon Busard* and Charles Pecheur

ICTEAM Institute, Université catholique de Louvain, Louvain-la-Neuve, Belgium
{simon.busard,charles.pecheur}@uclouvain.be

Abstract. NuSMV is a state-of-the-art model checker providing BDD-based and SAT-based techniques and a rich modeling language. While the tool is powerful, it is hard to customize it because of the size and complexity of its code base (more than 200K LOC). This paper presents PyNuSMV, a Python framework for prototyping and experimenting with BDD-based model-checking algorithms based on NuSMV.

PyNuSMV provides a rich and flexible programmable platform to implement new logics and experiment with custom model-checking algorithms. Thanks to PyNuSMV, it is possible to use NuSMV functionalities without understanding its whole code base or struggling with implementation details such as memory management. PyNuSMV has already been used to implement model-checking algorithms for rich logics such as ARCTL and CTLK.

This paper describes the structure and usage of PyNuSMV, illustrates its use by re-implementing CTL model checking, and reports initial performance results showing negligible impact compared to native NuSMV.

Keywords: Symbolic Model Checking, NuSMV, Python Interface, Binary Decision Diagrams.

1 Introduction

NuSMV is a state-of-the-art BDD-based and SAT-based model checker for temporal logics providing additional features such as model simulation [4]. While it is a very powerful tool, its (open-source) code base adds up to more than 200K lines of C code, making it difficult to extend or customize to implement new logics or new model-checking algorithms.

PyNuSMV is a Python framework for prototyping and experimenting with BDD-based model-checking algorithms based on NuSMV. It gives access to some of NuSMV's main functionalities, such as source model parsing and BDD manipulation, while hiding NuSMV implementation details by providing wrappers to NuSMV functions and data structures. In particular, NuSMV models can be read, parsed and compiled, giving full access to SMV's rich modeling language and vast collection of existing models. It makes it easy to implement new BDD-based model-checking algorithms and has already been

* This work is supported by the European Fund for Regional Development and by the Walloon Region.

G. Brat, N. Rungta, and A. Venet (Eds.): NFM 2013, LNCS 7871, pp. 453–458, 2013.

used to implement (1) rich counter-examples for CTL, (2) ARCTL model checking, an extension of CTL reasoning about the actions of a model, and (3) CTLK model checking, an extension of CTL reasoning about knowledge of the agents of a system [3,6,7]. The tool, including implementations for rich counter-examples, ARCTL and CTLK model checking, is available at `http://lvl.info.ucl.ac.be/Tools/PyNuSMV`.

Python has been retained to implement PyNuSMV because it comes with a full standard library and a full-fledged programming language supporting high-level programming (garbage collection, functional closures). PyNuSMV uses SWIG [1], a wrapper generator for C code, to wrap all NuSMV functions. On top of this wrapper, PyNuSMV provides a library of classes and modules reflecting NuSMV's main data structures (BDDs, expressions) at the Python level. Thanks to these classes and modules, it is easy to use NuSMV functionalities in Python, without struggling with implementation details such as memory management.

Note that SWIG has already been used in the RATSY tool to provide a wrapper of NuSMV functions at Python level [2]. But the goal of the tool was to support RATSY features by implementing them in NuSMV, not to provide a library of NuSMV functionalities.

The remainder of this paper is structured as follows: Section 2 presents the structure of PyNuSMV, Section 3 demonstrates its uses and reports initial evaluation results, and Section 4 describes future work.

2 PyNuSMV

The architecture of PyNuSMV, depicted in Figure 1, consists of three layers. The first one consists in the original **code of NuSMV** written in C. On this layer is the **lower interface**, composed of all modules generated by SWIG. Finally, the **upper interface**, built upon the lower one, consists of additional classes and modules providing access to some NuSMV main functionalities with Python capabilities such as garbage collection.

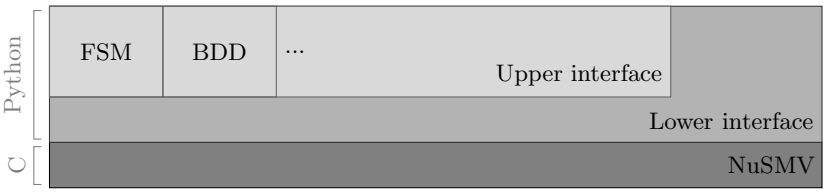

Fig. 1. PyNuSMV three-layer architecture

The lower interface is composed of a set of Python modules generated by SWIG. For every NuSMV package there is a SWIG interface generating a Python module that provides wrappers for functions and data structures of the package.

The upper interface is composed of classes wrapping data structures of NuSMV, and modules giving access to main functionalities such as CTL model checking and model parsing. The classes of the upper interface give access to:

- BDDs, states and inputs (i.e. actions) of the model and standard operations on BDDs provided as built-in operators: `&` (conjunction), `|` (disjunction), `~` (negation);
- the model itself, encoded as BDDs, and basic functionalities like computing the pre- or post-image of a set of states through the transition relation of the model;
- CTL formulae expressing properties of the model;
- functions acting on the global environment of NuSMV: intializing and finalizing NuSMV, reading the model and encoding it into BDDs;
- the parser of NuSMV to get, for example, the AST of a given simple expression;
- the CTL model-checking algorithms implemented in NuSMV.

Both interfaces have their advantages. The lower interface, fully generated by SWIG, allows the user to directly access NuSMV functions and data structures from the Python level, but the user has to manage all the implementation details he would manage at C level, memory in particular. On the other hand, the upper interface abstracts implementation details such as memory management and allows the user to focus on design and algorithmic concerns, at the cost of an additional level of indirection. While PyNuSMV gives access to both interfaces, most PyNuSMV applications are expected to rely only on the upper interface.

3 Evaluation

This section provides an initial evaluation of the tool. Section 3.1 shows how to re-implement CTL model checking with PyNuSMV and Section 3.2 presents some performance measures on small models.

3.1 Re-implementing CTL Model Checking

In order to illustrate how PyNuSMV allows to quickly and concisely experiment with new logics and custom model-checking algorithms while abstracting away from implementation details, this section presents an implementation of CTL model-checking algorithms [5]. The full code (about 175 LOC), of which only some pieces are presented here, is provided with the PyNuSMV distribution.

Figure 2 presents the main function of the program. It encodes the system into BDDs (line 3) and, for each CTL formula identified in the model file, computes the set of states of `fsm` satisfying the formula (line 8, `eval_ctl(fsm, spec)`) and the set of initial states violating the specification. Finally, the specification is reported as true if and only if this set is empty (lines 9 and 10).

Figure 3 shows parts of the `eval_ctl` function. Dedicated functions are implemented to evaluate basic Boolean operators and `EX`, `EU` and `EG` operators while the other operators are computed by standard reduction to these operators.[1]

[1] $f \rightarrow g \equiv \neg f \vee g$ and $A[f\ U\ g] \equiv \neg(E[\neg g\ U\ \neg g \wedge \neg f] \vee EG\neg g)$ are shown.

```
def main(modelPath):
  init_nusmv()
  fsm = BddFsm.from_filename(modelPath)
  propDb = glob.prop_database()
  for prop in propDb:
    if prop.type == propTypes['CTL']:
      spec = prop.exprcore
      violating = fsm.init & ~eval_ctl(fsm, spec)
      print('Specification',str(spec),
            'is',str(violating.is_false()))
      # We could generate counter-examples here
  deinit_nusmv()
```

Fig. 2. The main function of the CTL model checking algorithm

```
def eval_ctl(fsm, spec):
  ...
  elif spec.type == parser.IMPLIES:
    left = eval_ctl(fsm, spec.car, context)
    right = eval_ctl(fsm, spec.cdr, context)
    return ~left | right
  elif spec.type == parser.AU:
    left = eval_ctl(fsm, spec.car, context)
    right = eval_ctl(fsm, spec.cdr, context)
    return ~(eu(fsm,~right,~left & ~right) | eg(fsm,~right))
  ...
```

Fig. 3. CTL evaluation: implication and `AU` operator cases

Figure 4 presents the implementation dedicated to the `EU` operator. Note how Python's lambda-abstractions allow to express this in an abstract, declarative style reflecting the mathematical definition, $\mathtt{E}[\phi\mathtt{U}\psi] = \mu Z.\psi \vee (\phi \wedge \mathtt{EX}Z)$, using a generic higher-level `fixpoint` function.

3.2 Performance Comparisons

As an initial performance assessment, we verified a sample of NuSMV models both with native NuSMV and with CTL model checking implemented in PyNuSMV. Note that the PyNuSMV version used here, provided with the tool, has been adjusted to reproduce exactly the algorithms implemented in NuSMV and is not the version presented in the previous section. The main difference is that it takes reachable states into account.

The results are summarized in Table 1. Eight models, taken from the NuSMV distribution, have been processed by the two tools; the first four are very small (up to 6000 states), the last four are a bit larger (from 10^6 to 10^{16} states). Each model features up to three CTL formulae. The measured time is the time needed to check the specifications only; the time needed to initialize NuSMV and to build the model is not taken into account and is very similar in both cases.

```
1  def eu(fsm, phi, psi):
2      return fixpoint(lambda Z: psi | (phi & fsm.pre(Z)),
3                      BDD.false(fsm.bddEnc.DDmanager))
4  def fixpoint(funct, start):
5      old = start
6      new = funct(start)
7      while old != new:
8          old = new
9          new = funct(old)
10     return old
```

Fig. 4. CTL evaluation: `EU` operator implementation

Table 1. NuSMV and PyNuSMV times to evaluate model specifications (in seconds)

Model	NuSMV	PyNuSMV
counter	0	0
mutex	0.001	0.009
dme1	0.275	0.288
gas-nq7	8.913	11.027

Model	NuSMV	PyNuSMV
msi_wtrans	23.285	23.652
dme1-16	61.246	64.733
ftp3	75.607	78.771
key10	100.614	103.606

These results are very promising: the overhead caused by the two Python layers of PyNuSMV remains very low. This was expected, as most of the time needed to evaluate the CTL formulae is spent in computing BDDs operations, and these operations take place within NuSMV in both cases.

4 Future Work

While PyNuSMV's lower interface is automatically generated by SWIG, the upper interface is hand-crafted and needs more work to be developed. A number of NuSMV features remain to be supported: only CTL- and BDD-related functionalities are provided. For now, SAT-based and LTL model checking, and simulation-related features, are not exposed at the Python level. Note that there should be no additional difficulties to expose them in the upper interface, but a significant amount of engineering work.

Second, NuSMV can react in various ways when an error occurs. It can output a message on `stderr`, or in other cases return an error value. It also integrates a try/fail mechanism using `longjmp` functionalities. Some additional work should be provided to hide these different behaviors and provide a homogeneous error management in the upper interface, based on Python exceptions.

5 Conclusion

This paper presents PyNuSMV, a framework for experimenting BDD-based model-checking algorithms for new logics based on NuSMV. It allows the user to

use some NuSMV main functionalities such as model building, BDD manipulation and model-checking algorithms without having to understand the NuSMV code base and to struggle with implementation details such as memory management. Model checkers for rich logics such as CTLK have been re-implemented in a matter of days and a few thousands lines of Python code. Models can be written using the rich NuSMV modeling language and existing NuSMV models can be directly processed. Initial evaluation shows very little loss of efficiency compared to native NuSMV. While no performance tests have been performed yet on ARCTL and CTLK implementations, the overhead of PyNuSMV Python layers should remain low in these cases, too.

On the other hand, because NuSMV is primarily a standalone program, its developers made some implementation choices that make it not ideal to use as a library. For example, a lot of data structures are global, such as the parsing abstract syntax tree of the model, the main flat hierarchy or the proposition database. This imposes some limitations to PyNuSMV users that could be avoided if the platform was developed from scratch.

References

1. Beazley, D.M.: SWIG: an easy to use tool for integrating scripting languages with C and C++. In: Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, TCLTK 1996, vol. 4, p. 15. USENIX Association, Berkeley (1996)
2. Bloem, R., Cimatti, A., Greimel, K., Hofferek, G., Könighofer, R., Roveri, M., Schuppan, V., Seeber, R.: RATSY – A new requirements analysis tool with synthesis. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 425–429. Springer, Heidelberg (2010)
3. Busard, S., Pecheur, C.: Rich counter-examples for temporal-epistemic logic model checking. In: Reich, J., Finkbeiner, B. (eds.) Proceedings Second International Workshop on Interactions, Games and Protocols, Tallinn, Estonia, March 25. Electronic Proceedings in Theoretical Computer Science, vol. 78, pp. 39–53. Open Publishing Association (2012)
4. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An opensource tool for symbolic model checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
5. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
6. Pecheur, C., Raimondi, F.: Symbolic model checking of logics with actions. In: Edelkamp, S., Lomuscio, A. (eds.) MoChArt IV. LNCS (LNAI), vol. 4428, pp. 113–128. Springer, Heidelberg (2007)
7. Penczek, W., Lomuscio, A.: Verifying epistemic properties of multi-agent systems via bounded model checking. Fundamenta Informaticae 55(2), 167–185 (2003)