PyNuSMV Documentation

Release 1.0rc8

Simon Busard

Nov 29, 2017

Contents

1 Tutorial 3

2 Presentation of PyNuSMV 11

3 PyNuSMV Reference 17

4 Indices and tables 121

Bibliography 123

Python Module Index 125

PyNuSMV is a Python framework for prototyping and experimenting with BDD-based model checking algorithms based on NuSMV. It gives access to some main NuSMV functionalities, such as model 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 logic BDD-based model checking algorithms

CHAPTER 1

Tutorial

This page presents a short overview of PyNuSMV capabilities with a small example in the *Getting started* section. It then goes deeper into these capabilities and explains how to use them in the next sections.

Contents

- *Getting started*
- *Defining a model*
- *Loading a model*
- *Manipulating BDDs*
- *Defining properties*
- *Verifying properties*

1.1 Getting started

Let's consider the following SMV model. This model is composed of two counters, incrementing from 0 to 3, and looping. They run asynchronously and the running one is defined at each step by the `run` action.

```
MODULE counter(run, start, stop)
    -- A modulo counter
    -- Go from start (inclusive) to stop (exclusive) by 1-increments
    -- Run only when run is true
    VAR c : start..stop;
    INIT c = start
    TRANS next(c) = case    run : case  c + 1 = stop : start;
                                        TRUE : c + 1; esac;
                            !run: c;
                    esac
```

```
MODULE main
    IVAR
        run : {rc1, rc2};
    VAR
        c1 : counter(run = rc1, start, stop);
        c2 : counter(run = rc2, start, stop);
    DEFINE
        start := 0;
        stop := 3;


SPEC AF c1.c = stop - 1
```

Considering that the model is saved in the `counters.smv` file in the current directory, we can now run Python. The following Python session shows the basics of PyNuSMV. After importing `pynusmv`, the function *`init_nusmv`* **must** be called before calling any other PyNuSMV functionality. The function *`deinit_nusmv`* must also be called after using PyNuSMV to release all resources hold by NuSMV. After initializing PyNuSMV, the model is read with the function `load_from_file` and the model is computed, that is, flattened and encoded into BDDs, with the function *`compute_model`*.

```
>>> import pynusmv
>>> pynusmv.init.init_nusmv()
>>> pynusmv.glob.load_from_file("counters.smv")
>>> pynusmv.glob.compute_model()
>>> pynusmv.init.deinit_nusmv()
```

Another way to initialize and release NuSMV resources is to use the result of the *`init_nusmv`* function as a context manager with the `with` statement. The following code is equivalent to the one above:

```
>>> import pynusmv
>>> with pynusmv.init.init_nusmv():
...     pynusmv.glob.load_from_file("counters.smv")
...     pynusmv.glob.compute_model()
```

All NuSMV resources are automatically released when the context is exited.

The next Python session shows functionalities of FSMs, access to specifications of the model, calls to CTL model checking and manipulation of BDDs. First, NuSMV is initialized and the model is read. Then the model encoded with BDDs is retrieved from the main propositions database. The first (and only) proposition is then retrieved from the same database, and the specification of this proposition is isolated.

From the BDD-encoded FSM `fsm` and the specification `spec`, we call the *`eval_ctl_spec`* function to get all the states of `fsm` satisfying `spec`. Conjuncted with the set of reachables states of the model, we get `bdd`, a BDD representing all the reachable states of `fsm` satisfying `spec`. Finally, from this BDD we extract all the single states and display them, that is, we display, for each of them, the value of each state variable of the model.

```
>>> import pynusmv
>>> pynusmv.init.init_nusmv()
>>> pynusmv.glob.load_from_file("counters.smv")
>>> pynusmv.glob.compute_model()
>>> fsm = pynusmv.glob.prop_database().master.bddFsm
>>> fsm
<pynusmv.fsm.BddFsm object at 0x1016d9e90>
>>> prop = pynusmv.glob.prop_database()[0]
>>> prop
<pynusmv.prop.Prop object at 0x101770250>
>>> spec = prop.expr
>>> print(spec)
AF c1.c = stop - 1
>>> bdd = pynusmv.mc.eval_ctl_spec(fsm, spec) & fsm.reachable_states
>>> bdd
<pynusmv.dd.BDD object at 0x101765a90>
>>> satstates = fsm.pick_all_states(bdd)
>>> for state in satstates:
...     print(state.get_str_values())
...
{'c1.c': '2', 'c2.c': '2', 'stop': '3', 'start': '0'}
{'c1.c': '2', 'c2.c': '0', 'stop': '3', 'start': '0'}
{'c1.c': '2', 'c2.c': '1', 'stop': '3', 'start': '0'}
>>> pynusmv.init.deinit_nusmv()
```

This short tutorial showed the main functionalities of PyNuSMV. More of them are available, such as functionalities to parse and evaluate a simple expression, to build new CTL specifications, or to perform operations on BDDs. The rest of this page gives more details on these functionalities. The *full documentation* of the library is also given beside this tutorial.

1.2 Defining a model

As explained above, a model can be defined in SMV format and loaded into PyNuSMV through a file. PyNuSMV also provides a set of classes in the `model` module to define an SMV model directly in Python. For instance, the two-counter model above can befined with

```
from pynusmv.model import *

class counter(Module):
    COMMENT = """
        A modulo counter
        Go from start (inclusive) to stop (exclusive) by 1-increments
        Run only when run is true
    """
    run, start, stop = (Identifier(id_) for id_ in ("run", "start", "stop"))
    ARGS = [run, start, stop]
    c = Var(Range(start, stop))
    INIT = [c == start]
    TRANS = [c.next() == (Case(((run, Case((((c + 1) == stop, start),
                                            (Trueexp(), c + 1)))),
                                (~run, c))))]

class main(Module):
    start = Def(0)
    stop = Def(3)
    run = IVar(Scalar(("rc1", "rc2")))
    c1 = Var(counter(run == "rc1", start, stop))
    c2 = Var(counter(run == "rc2", start, stop))

print(counter)
print(main)
```

This prints the following

```
-- A modulo counter
-- Go from start (inclusive) to stop (exclusive) by 1-increments
-- Run only when run is true
MODULE counter(run, start, stop)
    VAR
        c: start .. stop;
    INIT
        c = start
    TRANS
        next(c) =
        case
            run:
            case
                c + 1 = stop: start;
                TRUE: c + 1;
            esac;
            ! run: c;
        esac
MODULE main
    DEFINE
        start := 0;
        stop := 3;
    IVAR
        run: {rc1, rc2};
    VAR
        c1: counter(run = rc1, start, stop);
        c2: counter(run = rc2, start, stop);
```

SMV state and input variables can be declared as members of the `Module` sub-class defining the module by instantiating *`Var`* and *`IVar`* classes. The argument to the constructor is the type of the variable, and can be either a primitive one (*`Range`*, *`Boolean`*, etc.), or instances of another module. All these instantiated objects can then be used as identifiers everywhere in the module definition. The different sections of an SMV module are declared as members with the corresponding names such as `INIT`, `TRANS`, or `ASSIGN`. Some must be iterables (such as `INIT` and `TRANS`), others must be mappings (such as `ASSIGN`). The *`model`* module supports a large variety of classes to define all concepts in SMV modules. For instance, in the code above, we can write `c1.c` for the `c` variable of the `c1` instance. Standard arithmetic operations such as additions are supported by SMV expressions, as shown with `c + 1`.

Another way to produce a Python-defined NuSMV model is to parse an existing SMV model (as a string) with the *`parser`* module functionalities. It contains the *`parseAllString`* function to parse a string according to a predefined parser. Several parsers are provided to parse identifiers (`parser.identifier`), expressions (`parser.next_expression`), modules (`parser.module`), etc.

1.3 Loading a model

The Python-defined modules can be loaded in PyNuSMV in a similar way to SMV files:

```
import pynusmv
pynusmv.init.init_nusmv()
pynusmv.glob.load(counter, main)
```

This *`load`* function accepts either sub-classes of *`Module`*, a single path to an SMV file, or a string containing the whole definition of the model. Once the model is loaded, the corresponding internal data structures such as the BDD-encoded finite-state machine are built with

```
pynusmv.glob.compute_model()
```

This *`compute_model`* function accepts the path to a file containing the BDD variable order to use for building the

BDD FSM, and whether or not single enumerations should be kept as they are, or converted into defines. Once the model is built, the BDD-encoded FSM is accessed via

```
fsm = pynusmv.glob.prop_database().master.bddFsm
```

1.4 Manipulating BDDs

The BDD-encoded finite-state machine representing the SMV model is an instance of the *BddFsm* class. It gives access to the parts of the model: the BDD representing the initial states (`fsm.init`), its reachable states (`fsm.reachable_states`), etc. It also allows us to pick one particular state from a given BDD-encoded set of states with the *pick_one_state* method, or to count the input values contained in one BDD:

```
print(fsm.count_states(fsm.init))
for state in fsm.pick_all_states(fsm.init):
    print(state.get_str_values())
```

prints

```
1
{'stop': '3', 'c1.c': '0', 'start': '0', 'c2.c': '0'}
```

The *BddFsm* class also gives access to the transition relation (`fsm.trans`) and the *pre* and *post* methods returning the pre- and post-images of given states:

```
for state in fsm.pick_all_states(fsm.post(fsm.init)):
    print(state.get_str_values())
```

prints

```
{'stop': '3', 'c1.c': '0', 'start': '0', 'c2.c': '1'}
{'stop': '3', 'c1.c': '1', 'start': '0', 'c2.c': '0'}
```

The transition relation is an instance of the *BddTrans* class and can be replaced. Several transition relations can also co-exist, separately from the FSM itself. The *BddTrans* class provides a way to define a new separated transition relation from a TRANS expression:

```
from pynusmv.fsm import BddTrans
trans = BddTrans.from_string(fsm.bddEnc.symbTable,"next(c1.c) = 0")
for state in fsm.pick_all_states(trans.post(fsm.init)):
    print(state.get_str_values())
```

prints

```
{'start': '0', 'c1.c': '0', 'stop': '3', 'c2.c': '3'}
{'start': '0', 'c1.c': '0', 'stop': '3', 'c2.c': '2'}
{'start': '0', 'c1.c': '0', 'stop': '3', 'c2.c': '1'}
{'start': '0', 'c1.c': '0', 'stop': '3', 'c2.c': '0'}
```

The BDD-encoded FSM can also return the BDD encoder *BddEnc* (through `fsm.bddEnc`) that keeps track of how the model variables are encoded into BDD variables. This encoder gives access to masks, such as *BddEnc.inputsMask*, that represents all valid values for input variables, for instance. It also gives access to cubes (*BddEnc.statesCube*, for instance) and can produce cubes for particular state or input variables via *BddEnc.cube_for_inputs_vars*. Finally, it gives access to the current order of BDD variables used for building BDDs and the set of declared variables:

```
enc = fsm.bddEnc
print(enc.stateVars)
print(enc.inputsVars)
```

prints

```
frozenset({'c1.c', 'c2.c'})
frozenset({'run'})
```

The BDD encoder also gives access to the symbols table that is used to store the symbols of the model (`bddEnc.symbTable`). This *`SymbTable`* can be used to declare new variables and to encode them into BDD variables.

Most of the parts of the FSM, such as the initial and reachable states, or the masks and cubes returned by the BDD encoder, are encoded into BDDs. These BDDs are instances of the *`BDD`* class, that provides several operations on BDDs. For instance,

```
fsm.reachable_states & fsm.fair_states
```

computes the conjunct of both BDDs, getting the fair reachable states of the model. Most common BDD operations are provided as builtin operators, such as disjunction (`|`), conjunction (`&`), and negation (`~`). These BDDs also support comparison, and the class provides a way to build the `True` and `False` canonical BDDs with `BDD.true()` and `BDD.false()`, respectively. Finally, the *`dd`* module contains some function to enable or disable BDD variable reordering:

```
pynusmv.dd.enable_dynamic_reordering()
```

1.5 Defining properties

A NuSMV property `prop` is a structure containing useful information about a given specification: its type `prop.type` (LTL or CTL specification, etc.), its name `prop.name`, its actual temoral-logic formula `prop.expr`, its status `prop.status` (unchecked, true, false), etc. These properties are represented in PyNuSMV with *`Prop`* instances. They come from a property database (*`PropDb`*) built and populated by NuSMV. The property database associated to the model built by NuSMV can be obtained through the *`glob`* module:

```
prop_db = pynusmv.glob.prop_database()
```

once the model has been built with *`compute_model`*. The property database contains all properties defined beside the loaded model, such as the specification `AF c1.c = stop - 1` defined in the `counters.smv` model at the beginning of this tutorial. It acts as a sequence of properties and particular properties can be accessed through their indices.

Property expressions `spec` are instances of the *`Spec`* class. They reflect NuSMV internal structures, so they have a type `spec.type`, a left child `spec.car` and a right child `spec.cdr` (both can be `None`, depending on the type of the expression). New specifications can be defined thanks to *`prop`* module functions such as atomic propositions with the *`atom`* function, Boolean operators (`&`, `|`, etc.), CTL operators (*`ag`*, *`ef`*, etc.), and LTL ones (`x`, `u`, etc.) For instance, the specification `AF c1.c = stop - 1` can be built with

```
from pynusmv import prop
spec = prop.af(prop.atom("c1.c = stop - 1"))
```

1.6 Verifying properties

Once a model is loaded into PyNuSMV and a specification is defined, the latter can be checked on the former. The *mc* module provides all functionalities to perform this verification. It contains high-level functions as *check_ltl_spec* and *check_ctl_spec* for directly checking formulas. For instance, the specification above can be checked with

```
from pynusmv.mc import check_ctl_spec
print(check_ctl_spec(fsm, spec)) # Prints False
```

It also gives access to lower-level functions to evaluate the BDD representing the states satisfying some CTL formula (*eval_ctl_spec*), or to evaluate particular operators (*mc.eg*, etc.) It can also explain why a given specification is not satisfied by a given model:

```
from pynusmv.mc import explain, eval_ctl_spec
explanation = explain(fsm, fsm.init & ~eval_ctl_spec(fsm, spec), spec)
```

The produced explanation is a sequence of states and inputs values representing a looping path in the model that shows why the formula is violated:

```
for state, inputs in zip(explanation[::2], explanation[1::2]):
    if state == explanation[-1]:
        print("-- Loop starts here")
    print(state.get_str_values())
    print(inputs.get_str_values())
```

prints

```
-- Loop starts here
{'start': '0', 'c1.c': '0', 'stop': '3', 'c2.c': '0'}
{'run': 'rc2'}
{'start': '0', 'c1.c': '0', 'stop': '3', 'c2.c': '1'}
{'run': 'rc2'}
{'start': '0', 'c1.c': '0', 'stop': '3', 'c2.c': '2'}
{'run': 'rc2'}
```

CHAPTER 2

Presentation of PyNuSMV

PyNuSMV is a Python interface to NuSMV, allowing to use NuSMV as a Python library. It is composed of several classes representing NuSMV data structures and providing functionalities on these data. This page describes the goals behind PyNuSMV and the architecture of the library and, covers its limitations.

2.1 Goals

The main goal of PyNuSMV is to provide a Python interface for NuSMV functionalities. This interface can be used as a library of functions on models, BDDs, SAT solvers and other data structures of NuSMV.

One subgoal is to provide all the functionalities of NuSMV at the Python level, e.g. calling the `bdd_and` function on two `bdd_ptrs`. This is achieved by using SWIG, a wrapper generator, to generate a wrapper for every function of NuSMV. Thanks to this wrapper, there are no restrictions to calling NuSMV functions and using its data structures. On the other hand, no barriers are set to forbid erroneous behaviors or to help the user.

Another subgoal is to provide a Python-like library to access the main data structures and functions of NuSMV: FSM, BDD, parser, model checking algorithms, simulation, etc. For example, providing a class `BDD` with a built-in operator `&`, such that `bdd1 & bdd2` computes `bdd_and(bdd1, bdd2)`. This library would contain the error mechanisms required to ensure the correct usage of NuSMV.

In summary, PyNuSMV has two main goals:

- providing a complete Python interface for NuSMV functions and data structures;
- providing a Python-like interface to some major data structures and functionalities.

2.2 Architecture

PyNuSMV is composed of three main layers. The first layer is NuSMV. The second layer is called the lower interface; it contains all the functions of NuSMV, at Python level, wrapped by SWIG. The third layer is called the upper interface; it contains the Python-like functionalities built upon the lower interface.

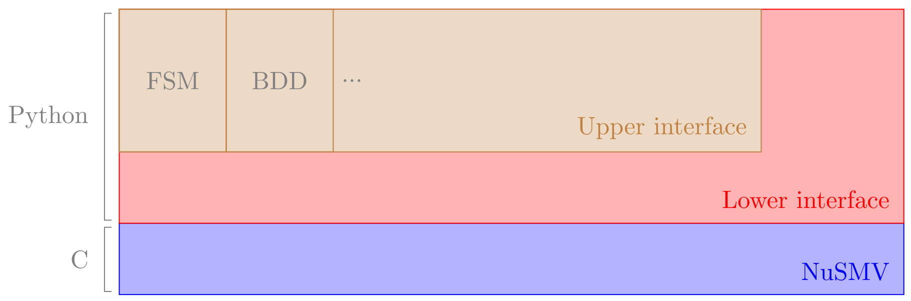

Fig. 2.1: PyNuSMV three-layer architecture

2.2.1 NuSMV

The version of NuSMV used in PyNuSMV is the version 2.5.4. NuSMV code has been kept unchanged, except for very small details; the details of which can be seen (and is explained) in the several *.patch* files we apply before building the NuSMV sources. For instance:

- some functions and macro declarations have been commented because they were defined twice;
- some static keywords have been removed to allow exporting the functions;
- an assertion check was removed from *src/cmd/cmdMisc.c* since it made the NuSMV initalization/deinitialization impossible.

2.2.2 Lower Interface

The lower interface is composed of a set Python modules generated by SWIG. For every NuSMV package, i.e. for every sub-directory in the *src/* directory of NuSMV, there is a SWIG interface and a Python module that provide wrappers for functions and data structures of the package. This section briefly discusses the structure and content of the lower interface and presents its limitations.

Structure

The structure of the lower interface is a copy of the one of NuSMV. Let's consider as a NuSMV package any sub-directory of the *src/* directory of NuSMV sources. For example, NuSMV contains the *mc/* and *fsm/bdd/* packages. The structure of the lower interface is the same. The lower interface is located in the `pynusmv_lower_interface` Python package. Every NuSMV package gets its PyNuSMV package. For example, the *prop/* NuSMV package is wrapped into the `pynusmv_lower_interface.nusmv.prop` Python package; the *compile/symb_table/* NuSMV package is wrapped into the `pynusmv_lower_interface.nusmv.compile.symb_table` package. Furthermore, every wrapped function is automatically documented by SWIG with the corresponding C function signature. It allows the developer to know what types of arguments the wrapped function takes.

Content

The goal of the lower interface is to provide a wrapper for every function of NuSMV. In practice, for every package, only the set of functions that are considered as public are provided. This means that, for every package, all the headers are exported, except the ones with a name ending with *Int.h*, *_int.h* or *_private.h* (these are explicitly referred to as 'internal' or 'private' headers in the NuSMV codebase).

Limitations

The lower interface has some limitations. First, it does not wrap all the functions, but only the ones present in the public headers, as described in the previous section.

Furthermore, there are some exceptions:

- the *utils/lsort.h* header is not wrapped because SWIG cannot process it.
- A set of functions, from different packages, are not wrapped because they have no concrete implementation so far.

2.2.3 Upper Interface

The upper interface is composed of Python classes representing data structures of NuSMV as well as additional modules giving access to main functionalities that do not belong to a data structure, like CTL model checking. Each instance of these classes contains a pointer to the corresponding NuSMV data structure and provides a set of methods on this pointer. This section explains the way all pointers to data structures are wrapped, how the memory is managed and presents an overview of the classes and modules currently defined.

Wrapping pointers

Every pointer to a NuSMV data structure is wrapped into a Python class that is a subclass of the *PointerWrapper* class. This class contains a `_ptr` attribute (the wrapped pointer) and implements the `__del__` destructor. All the other functionalities are left to subclasses. This provides a uniform way of wrapping all NuSMV pointers.

Garbage Collection

In PyNuSMV, we distinguish two types of pointers to NuSMV data structures: the pointers that have to be freed and the ones that do not. For example, a pointer to a BDD has to be freed after usage (with `bdd_free`) while a pointer to the main FSM do not, because NuSMV frees it when deinitializing.

In addition to the wrapped pointer, the PointerWrapper class contains a flag called `_freeit` that tells whether the pointer has to be freed when destroying the wrapper. If needed, the destructor calls the `_free` method, that does the work. The `_free` method of *PointerWrapper* class does nothing. It is the responsibility of subclasses to reimplement this `_free` method if the pointer has to be freed. In fact, *PointerWrapper* cannot say how to free the pointer since the NuSMV function to call depends on the wrapped pointer (BDDs have to be freed with `bdd_free`, other pointers need other functions).

Furthermore, we define the following conventions:

- wrappers containing pointers that do not have to be freed do not have to reimplement the `_free` method.
- pointers that do not have to be freed can be shared between any number of wrappers. Since these pointers are not freed, there is no problem.
- wrappers containing pointers that have to be freed must reimplement the `_free` method to free the pointer when needed.
- there must exist at most one wrapper for any pointer that has to be freed. This ensures that the pointer will be freed only once.
- if no wrapper is created to wrap a pointer, it is the responsibility of the one who got the pointer to free it.

By following these conventions, PyNuSMV can manage the memory and free it when needed.

Thanks to the specific `_free` method implementations, pointers can be correctly freed when the wrapper is destroyed by Python. But pointers must not be freed after deinitializing NuSMV. So we need a way to free every pointer before deinitializing NuSMV. To achieve this garbage collection, PyNuSMV comes with a specific module *pynusmv.init* that allows to initialize and deinitialize NuSMV, with the *init_nusmv* and *deinit_nusmv* functions. Before using PyNuSMV, `init_nusmv` must be called; after using PyNuSMV, it is necessary to deinitializing NuSMV by calling `deinit_nusmv`. Furthermore, `init_nusmv` creates a new list in which every newly created *PointerWrapper* (or subclass of it) is registered. When `deinit_nusmv` is called, all the wrappers of the list are freed before deinitializing NuSMV. This ensures that all NuSMV data pointers wrapped by PyNuSMV classes are freed before deinitializing NuSMV.

Classes and Modules

PyNuSMV is composed of several modules, each one proposing some NuSMV functionalities:

- *init* contains all the functions needed to initialize and close NuSMV. These functions need to be used before any other access to PyNuSMV.
- *glob* provides functionalities to read and build a model from an SMV source file.
- *model* provides functionalities to define NuSMV models in Python.
- *node* provides a wrapper to NuSMV *node* structures.
- *fsm* contains all the FSM-related structures like BDD-represented FSM, BDD-represented transition relation, BDD encoding and symbols table.
- *prop* defines structures related to propositions of a model; this includes CTL specifications.
- *dd* provides BDD-related structures like generic BDD, lists of BDDs and BDD-represented states, input values and cubes.
- *parser* gives access to NuSMV parser to parse simple expressions of the SMV language.
- *mc* contains model checking features.
- *exception* groups all the PyNuSMV-related exceptions.
- *utils* contains some side functionalities.
- *sat* contains classes and functions related to the operation and manipulation of the different sat solvers available in PyNuSMV.
- *bmc.glob* serves as a reference entry point for the bmc-related functions (commands) and global objects. It defines amongst other the function *bmc_setup* wich must be called before using any of the BMC related features + the class *BmcSupport* which acts as a context manager and frees you from the need of explicitly calling *bmc_setup*.
- *bmc.ltlspec* contains all the functionalities related to the bounded model checking of LTL properties: from end to end property verification to the translation of formulas to boolean expressions corresponding to the SAT problem necessary to verify these using LTL bounded semantics of the dumping of problem to file (in DIMACS format).
- *bmc.invarspec* contains all the functionalities related to the verification of INVARSPEC properties using a technique close to that of SAT-based bounded model checking for LTL. (See *Niklas Een and Niklas Sorensson. "Temporal induction by incremental sat solving."* for further details).
- *bmc.utils* contains bmc related utility functions.
- *be.expression* contains classes and functions related to the operation and manipulation of the boolean expressions.
- *be.fsm* contains classes and functions related to PyNuSMV's description of an FSM when it is encoded in terms of boolean expressions.

- *be.encoder* provides the boolean expression encoder capabilities that make the interaction with a SAT solver easy.
- *be.manager* contains classes and functions related to the management of boolean expressions (conversion to reduced boolean circuits. Caveat: RBC representation is not exposed to the upper interface).
- *collections* impements pythonic wrappers around the internal collections and iterator structures used in NuSMV.
- *wff* encapsulates the notion of well formed formula as specified per the input language of NuSMV. It is particularly useful in the scope of BMC.
- *trace* defines the classes Trace and TraceStep which serve the purpose of representing traces (executions) in a PyNuSMV model.
- *sexp.fsm* contains a representation of the FSM in terms of simple expressions.

2.3 Limitations

PyNuSMV has some limitations. Two major ones are the exposed functionalities and error management.

2.3.1 Exposed functionalities

Since the upper interface of PyNuSMV is written by hand, it needs some work to implement its functionalities (compared to the lower interface generated with SWIG) and therefore, the framework might be missing some functionalities. If one such functionality is of interest to you, feel free to either hack the code yourself or get in touch with us to get some help.

2.3.2 Error Management

NuSMV can react in various ways when an error occurs. It can output a message at `stderr` and returns an error flag, e.g. when executing a command. It also integrates a try/fail mechanism using `lonjmp` functionalities. And it can also abruptly exit using the `exit()` function.

For now, there is little error management in PyNuSMV. When possible, the try/fail mechanism has been used to avoid NuSMV to completely `exit()` when there is an error. Instead, exceptions are raised, with sometimes error messages from NuSMV. In some cases, errors are correctly raised but a message is printed at `stderr` by NuSMV itself. Some future work on PyNuSMV includes a better error management.

CHAPTER 3

PyNuSMV Reference

3.1 `pynusmv.__init__` Module

PyNuSMV is a Python framework for experimenting and prototyping BDD-based model checking algorithms based on NuSMV. It gives access to main BDD-related NuSMV functionalities, like model 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.

PyNuSMV is composed of several modules, each one proposing some functionalities:

- *`init`* contains all the functions needed to initialize and close NuSMV. These functions need to be used before any other access to PyNuSMV.
- *`glob`* provides functionalities to read and build a model from an SMV source file.
- *`model`* provides functionalities to define NuSMV models in Python.
- *`node`* provides a wrapper to NuSMV *node* structures.
- *`fsm`* contains all the FSM-related structures like BDD-represented FSM, BDD-represented transition relation, BDD encoding and symbols table.
- *`prop`* defines structures related to propositions of a model; this includes CTL specifications.
- *`dd`* provides BDD-related structures like generic BDD, lists of BDDs and BDD-represented states, input values and cubes.
- *`parser`* gives access to NuSMV parser to parse simple expressions of the SMV language.
- *`mc`* contains model checking features.
- *`exception`* groups all the PyNuSMV-related exceptions.
- *`utils`* contains some side functionalities.
- *`sat`* contains classes and functions related to the operation and manipulation of the different sat solvers available in PyNuSMV.

- *bmc.glob* serves as a reference entry point for the bmc-related functions (commands) and global objects. It defines amongst other the function *bmc_setup* wich must be called before using any of the BMC related features + the class *BmcSupport* which acts as a context manager and frees you from the need of explicitly calling *bmc_setup*.
- *bmc.ltlspec* contains all the functionalities related to the bounded model checking of LTL properties: from end to end property verification to the translation of formulas to boolean expressions corresponding to the SAT problem necessary to verify these using LTL bounded semantics of the dumping of problem to file (in DIMACS format).
- *bmc.invarspec* contains all the functionalities related to the verification of INVARSPEC properties using a technique close to that of SAT-based bounded model checking for LTL. (See *Niklas Een and Niklas Sorensson. "Temporal induction by incremental sat solving."* for further details).
- *bmc.utils* contains bmc related utility functions.
- *be.expression* contains classes and functions related to the operation and manipulation of the boolean expressions.
- *be.fsm* contains classes and functions related to PyNuSMV's description of an FSM when it is encoded in terms of boolean expressions.
- *be.encoder* provides the boolean expression encoder capabilities that make the interaction with a SAT solver easy.
- *be.manager* contains classes and functions related to the management of boolean expressions (conversion to reduced boolean circuits. Caveat: RBC representation is not exposed to the upper interface).
- *collections* impements pythonic wrappers around the internal collections and iterator structures used in NuSMV.
- *wff* encapsulates the notion of well formed formula as specified per the input language of NuSMV. It is particularly useful in the scope of BMC.
- *trace* defines the classes Trace and TraceStep which serve the purpose of representing traces (executions) in a PyNuSMV model.
- *sexp.fsm* contains a representation of the FSM in terms of simple expressions.

Warning: Before using any PyNuSMV functionality, make sure to call *init_nusmv* function to initialize NuSMV; do not forget to also call *deinit_nusmv* when you do not need PyNuSMV anymore to clean everything needed by NuSMV to run.

3.2 `pynusmv.collections` Module

This module implements wrappers around the NuSMV list types:

- ***Slist*** **which represents a singly linked list as used in many internal** functions of NuSMV
- *NodeList* which is a wrapper for NuSMV's internal NodeList class
- ***Assoc*** **which stands for NuSMV's internal associative array** (hash_ptr and st_table*)

class `pynusmv.collections.`**`Conversion`**(*p2o*, *o2p*)

Bases: `object`

This class wraps the functions used to perform the back and forth conversion between types of the the lower interface (pointers) and types of the higher interface (python objects)

to_object (*pointer*)
Returns an higher level (and meaningful) representation of *pointer*

Parameters **pointer** – a raw (low level) pointer that needs to be mapped to something more meaningful

Returns an higher level (and meaningful) representation of *pointer*

to_pointer (*obj*)
Returns a low level pointer representing *obj*

Parameters **obj** – an high level object that needs to be translated to a C pointer

Returns a low level pointer representing *obj*

class pynusmv.collections.**IntConversion**
Bases: *pynusmv.collections.Conversion*

A conversion object able to wrap/unwrap int to void* and vice versa

class pynusmv.collections.**NodeConversion**
Bases: *pynusmv.collections.Conversion*

A conversion object able to wrap/unwrap Node to void* and vice versa

class pynusmv.collections.**Slist** (*ptr*, *conversion*, *freeit=True*)
Bases: *pynusmv.utils.PointerWrapper*, collections.abc.Iterable

This class implements an high level pythonic interface to Slist which is a NuSMV-defined simply linked list.

Although this type is implemented in C, some of its operation have a slow O(n) performance. For instance the __getitem__ and __delitem__ which correspond to x = lst[y] and del lst[z] are O(n) operation despite their indexed-looking syntax. In case many such operations are required or whenever you need more advanced list operations, you are encouraged to cast this list to a builtin python list with the list(lst) operator. The inverse conversion is possible using Slist.fromlist(lst) but requires however to provide the element conversion as an object of type util.Conversion

static empty (*conversion*)
Returns a new empty Slist

Parameters **conversion** (*pynusmv.util.Conversion*) – the object encapsulating the conversion to and from pointer

Returns a new empty list

static from_list (*lst*, *conversion*)
Returns a new Slist corresponding to the *lst* given as first argument

Parameters

- **lst** (*any collection that can be iterated*) – an iterable from which to create a new Slist
- **conversion** (*pynusmv.util.Conversion*) – the object encapsulating the conversion to and from pointer

Returns a new list containing the same elements (in the same order) than *lst*

copy ()

Returns a copy of this Slist

reverse ()
Reverses the order of the elements in this list

push(*o*)
Prepends o tho the list

pop()

Returns returns and remove the top (first element) of the list

top()

Returns the first element of the list w/o removing it

is_empty()

Returns True iff this list is empty

extend(*other*)
Appends one list to this one

Parameters `other` – the other list to append to thisone

remove(*item*)
Removes all occurences of *item* in this list

Parameters `item` – the item to remove

..note: This method should not be used as the NuSMV implementation it relies on is buggy.

clear()
Removes all items from the list

class `pynusmv.collections.`**SlistIterator**(*slist*)
Bases: `collections.abc.Iterator`

This class defines a pythonic iterator to iterate over NuSMV Slist objects

class `pynusmv.collections.`**Sentinel**
Bases: `object`

This class implements a sentinel value

class `pynusmv.collections.`**NodeIterator**(*ptr*)
Bases: `collections.abc.Iterator`

This class implements an useful iterator to iterate over nodes or wrap them to python lists.

static from_node(*node*)
Creates an iterator from an high level Node :param node: the Node representing the list to iterate :type node: Node :return: an iterator iterating over the node considered as a linked list

static from_pointer(*ptr*)
Creates an iterator from a low level pointer to a node_ptr :param ptr: the pointer to a node representing the list to iterate :type node: node_ptr :return: an iterator iterating over the node considered as a linked list

class `pynusmv.collections.`**NodeList**(*ptr*, *conversion=<pynusmv.collections.NodeConversion object>*, *freeit=False*)
Bases: *pynusmv.utils.PointerWrapper*, `collections.abc.Iterable`

This class implements a pythonic interface to NuSMV's internal version of a doubly linked list

Note: The following apis have not been exposed since they require pointer to function (in C) which are considered too low level for this pythonic interface. However, these apis are accessible using pynusmv_lower_interface.nusmv.utils.utils.<TheAPI> and passing nodelst._ptr instead of nodelist.

- NodeList_remove_elems

- NodeList_search
- NodeList_foreach
- NodeList_map
- NodeList_filter

---

static `empty`(*conversion=<pynusmv.collections.NodeConversion object>*, *freeit=True*)
Creates a new empty list

Parameters

- **`conversion`** – the conversion object allowing the transformation from pointer to object and vice versa
- **`freeit`** – a flag indicating whether or not this list should be freed upon garbage collection

static `from_list`(*lst*, *conversion=<pynusmv.collections.NodeConversion object>*, *freeit=True*)

Returns a NodeList equivalent to the pythonic list *lst*

Parameters

- **`lst`** – the list which shall serve as basis for the created one
- **`conversion`** – the object encapsulating the conversion back and forth from and to pointer
- **`freeit`** – a flag indicating whether or not this list should be freed upon garbage collection

`copy`(*freeit=True*)

Parameters **`freeit`** – a flag indicating whether or not the copy shoudl be freed upon garbage collection

Returns Returns a copy of this list

`append`(*node*)
Adds the given node at the end of the list

Parameters **`node`** – the node to append to the list

`prepend`(*node*)
Adds the given node at the beginning of the list

Parameters **`node`** – the node to append to the list

`reverse`()
inverts the order of the items in the list

`extend`(*other*, *unique=False*)
Appends all the iems of *other* to this list. If param unique is set to true, the items are only appended if not already present in the list.

Parameters

- **`other`** – the other list to concatenate to this one
- **`unique`** – a flag to conditionally add the elements of the other list

`count`(*node*)

Returns the number of occurences of *node*

Parameters **`node`** – the node whose number of occurrences is being counted

insert_before (*iterator*, *node*)
inserts *node* right before the position pointed by iterator :param iterator: the iterator pointing the position where to insert :param node: the node to insert in the list

insert_after (*iterator*, *node*)
inserts *node* right after the position pointed by iterator :param iterator: the iterator pointing the position where to insert :param node: the node to insert in the list

insert_at (*idx*, *node*)
inserts *node* right before the node at position *idx* :param idx: the the position where to insert

print_nodes (*stdio_file*)
Prints the list node to the given stream. :param stdio_file: an instance of StdioFile wrapping an open C stream

class `pynusmv.collections.`**NodeListIter** (*lst*)
Bases: `collections.abc.Iterator`

An iterator to iterate over NodeList.

Note: Despite the fact that it wraps a pointer, this class does not extend the PointerWrapper class since there is no need to free the pointer.

class `pynusmv.collections.`**Assoc** (*ptr*, *key_conversion=<pynusmv.collections.NodeConversion object>*, *value_conversion=<pynusmv.collections.NodeConversion object>*, *freeit=False*)
Bases: `pynusmv.utils.PointerWrapper`

This class implements a pythonic abstraction to the NuSMV associative array encapsulated in st_table aka hash_ptr which is often used in the NuSMV internals.

Note: I couldn't find any documentation about the ST_PFSR type. As a consequence of this, I couldn't implement the iterable protocol

Warning: BOTH the key AND the value are supposed to be of type Node. Hence, the conversion method must take care to return the nodes and objects of the right types.

static empty (*key_conversion=<pynusmv.collections.NodeConversion object>*, *value_conversion=<pynusmv.collections.NodeConversion object>*, *initial_capa=0*, *freeit=False*)
Creates an empty assoc

Parameters

- **key_conversion** – the conversion for the key (object <–> pointer)
- **value_conversion** – the conversion of the value (object <–> pointer)
- **capa** (*initial*) – the initial capacity of the associative array
- **freeit** – a flag indicating whether or not this object should be freed upon garbage collection

Returns a new empty Assoc

static **from_dict**(*dico*, *key_conversion=<pynusmv.collections.NodeConversion object>*, *value_conversion=<pynusmv.collections.NodeConversion object>*, *freeit=False*)
Creates an assoc from a pythonic dict

Parameters

- **dico** – python dictionary
- **key_conversion** – the conversion for the key (object <–> pointer)
- **value_conversion** – the conversion of the value (object <–> pointer)
- **freeit** – a flag indicating whether or not this object should be freed upon garbage collection

Returns an assoc with the same contents as the given dico

copy()
Creates a copy of this Assoc

clear()
Empties the container

3.3 `pynusmv.dd` Module

The `pynusmv.dd` module provides some BDD-related structures:

- `BDD` represents a BDD.
- `BDDList` represents a list of BDDs.
- `State` represents a particular state of the model.
- `Inputs` represents input variables values, i.e. a particular action of the model.
- `StateInputs` represents a particular state-inputs pair of the model.
- `Cube` represents a particular cube of variables the model.
- `DDManager` represents a NuSMV DD manager.

It also provides global methods to work on BDD variables reordering: `enable_dynamic_reordering()`, `disable_dynamic_reordering()`, `dynamic_reordering_enabled()`, `reorder()`.

pynusmv.dd.**enable_dynamic_reordering**(*DDmanager=None*, *method='sift'*)
Enable dynamic reordering of BDD variables under control of *DDmanager* with the given *method*.

Parameters

- **DDmanager** (`DDManager`) – the concerned DD manager; if None, the global DD manager is used instead.
- **method** (`str`) – the method to use for reordering: *sift (default method)*, *random*, *random_pivot*, *sift_converge*, *symmetry_sift*, *symmetry_sift_converge*, *window{2, 3, 4}*, *window{2, 3, 4}_converge*, *group_sift*, *group_sift_converge*, *annealing*, *genetic*, *exact*, *linear*, *linear_converge*, *same* (the previously chosen method)

Raise a `MissingManagerError` if the manager is missing

Note: For more information on reordering methods, see NuSMV manual.

`pynusmv.dd.`**`disable_dynamic_reordering`**(*DDmanager=None*)

Disable dynamic reordering of BDD variables under control of *DDmanager*.

Parameters **`DDmanager`** (*`DDManager`*) – the concerned DD manager; if None, the global DD manager is used instead.

Raise a *`MissingManagerError`* if the manager is missing

`pynusmv.dd.`**`dynamic_reordering_enabled`**(*DDmanager=None*)

Return the dynamic reordering method used if reordering is enabled for BDD under control of *DDmanager*, None otherwise.

Parameters **`DDmanager`** (*`DDManager`*) – the concerned DD manager; if None, the global DD manager is used instead.

Return type None, or a the name of the method used

Raise a *`MissingManagerError`* if the manager is missing

`pynusmv.dd.`**`reorder`**(*DDmanager=None*, *method='sift'*)

Force a reordering of BDD variables under control of *DDmanager*.

Parameters

- **`DDmanager`** (*`DDManager`*) – the concerned DD manager; if None, the global DD manager is used instead.
- **`method`** (`str`) – the method to use for reordering: *sift (default method)*, *random*, *random_pivot*, *sift_converge*, *symmetry_sift*, *symmetry_sift_converge*, *window{2, 3, 4}*, *window{2, 3, 4}_converge*, *group_sift*, *group_sift_converge*, *annealing*, *genetic*, *exact*, *linear*, *linear_converge*, *same* (the previously chosen method)

Raise a *`MissingManagerError`* if the manager is missing

Note: For more information on reordering methods, see NuSMV manual.

class `pynusmv.dd.`**`BDD`**(*ptr*, *dd_manager=None*, *freeit=True*)

Bases: *`pynusmv.utils.PointerWrapper`*

Python class for BDD structure.

The BDD represents a BDD in NuSMV and provides a set of operations on this BDD. Thanks to operator overloading, it is possible to write compact expressions on BDDs. The available operations are:

- `a + b` and `a | b` compute the disjunction of `a` and `b`
- `a * b` and `a & b` compute the conjunction of `a` and `b`
- `~a` and `-a` compute the negation of `a`
- `a - b` computes `a & ~b`
- `a ^ b` computes the exclusive-OR (XOR) of `a` and `b`
- `a == b`, `a <= b`, `a < b`, `a > b` and `a >= b` compare `a` and `b`

Any BDD operation raises a *`MissingManagerError`* whenever the manager of the BDD is None and a manager is needed to perform the operation.

`size`

The number of BDD nodes of this BDD.

`equal`(*other*)

Determine whether this BDD is equal to *other* or not.

Parameters **`other`** (*BDD*) – the BDD to compare

`dup`()
Return a copy of this BDD.

`is_true`()
Determine whether this BDD is true or not.

`is_false`()
Determine whether this BDD is false or not.

`isnot_true`()
Determine whether this BDD is not true.

`isnot_false`()
Determine whether this BDD is not false.

`entailed`(*other*)
Determine whether this BDD is included in *other* or not.

Parameters **`other`** (*BDD*) – the BDD to compare

`intersected`(*other*)
Determine whether the intersection between this BDD and *other* is not empty.

Parameters **`other`** (*BDD*) – the BDD to compare

`leq`(*other*)
Determine whether this BDD is less than or equal to *other*.

Parameters **`other`** (*BDD*) – the BDD to compare

`not_`()
Compute the complement of this BDD.

`and_`(*other*)
Compute the conjunction of this BDD and *other*.

Parameters **`other`** (*BDD*) – the other BDD

`or_`(*other*)
Compute the conjunction of this BDD and *other*.

Parameters **`other`** (*BDD*) – the other BDD

`xor`(*other*)
Compute the exclusive-OR of this BDD and *other*.

Parameters **`other`** (*BDD*) – the other BDD

`iff`(*other*)
Compute the IFF operation on this BDD and *other*.

Parameters **`other`** (*BDD*) – the other BDD

`imply`(*other*)
Compute the IMPLY operation on this BDD and *other*.

Parameters **`other`** (*BDD*) – the other BDD

`diff`(*other*)

`union`(*other*)

`intersection`(*other*)

forsome(*cube*)
Existentially abstract all the variables in cube from this BDD.

Parameters **cube** (*BDD*) – the cube

forall(*cube*)
Universally abstract all the variables in cube from this BDD.

Parameters **cube** (*BDD*) – the cube

minimize(*c*)
Restrict this BDD with c, as described in Coudert et al. ICCAD90.

Parameters **c** (*BDD*) – the BDD used to restrict this BDD

Note: Always returns a BDD not larger than the this BDD.

static true(*manager_or_fsm=None*)
Return the TRUE BDD.

Parameters **manager_or_fsm** (*DDManager* or *BddFsm*) – if not *None*, the manager of the returned BDD or the FSM; otherwise, the global FSM is used.

static false(*manager_or_fsm=None*)
Return the FALSE BDD.

Parameters **manager_or_fsm** (*DDManager* or *BddFsm*) – if not *None*, the manager of the returned BDD or the FSM; otherwise, the global FSM is used.

class pynusmv.dd.**BDDList**(*ptr*, *ddmanager=None*, *freeit=True*)
Bases: *pynusmv.utils.PointerWrapper*

A BDD list stored as NuSMV nodes.

The BDDList class implements a NuSMV nodes-based BDD list and can be used as any Python list.

to_tuple()
Return a tuple containing all BDDs of self. The returned BDDs are copies of the ones of self.

static from_tuple(*bddtuple*)
Create a node-based list from the Python tuple *bddtuple*.

Parameters **bddtuple** – a Python tuple of BDDs

Return a *BDDList* representing the given tuple, using NuSMV nodes. All BDDs are assumed from the same DD manager; the created list contains the DD manager of the first non-*None* BDD. If all elements of *bddtuple* are *None*, the manager of the created *BDDList* is *None*.

class pynusmv.dd.**State**(*ptr*, *fsm*, *freeit=True*)
Bases: *pynusmv.dd.BDD*

Python class for State structure.

A State is a *BDD* representing a single state of the model.

get_str_values(*layers=None*)
Return a dictionary of the (variable, value) pairs of this State.

Parameters **layers** – if not *None*, the set of names of the layers from which picking the string values

Return type a dictionary of pairs of strings.

static **`from_bdd`**(*bdd*, *fsm*)
Return a new State of fsm from bdd.

Parameters

- **`bdd`** (*`BDD`*) – a BDD representing a single state
- **`fsm`** (*`BddFsm`*) – the FSM from which the BDD comes from

class `pynusmv.dd.`**`Inputs`**(*ptr*, *fsm*, *freeit=True*)
Bases: *`pynusmv.dd.BDD`*

Python class for inputs structure.

An Inputs is a *`BDD`* representing a single valuation of the inputs variables of the model, i.e. an action of the model.

`get_str_values`(*layers=None*)
Return a dictionary of the (variable, value) pairs of these Inputs.

Parameters **`layers`** – if not *None*, the set of names of the layers from which picking the string values

Return type a dictionary of pairs of strings.

static **`from_bdd`**(*bdd*, *fsm*)
Return a new Inputs of fsm from bdd.

Parameters

- **`bdd`** (*`BDD`*) – a BDD representing a single inputs variables valuation
- **`fsm`** (*`BddFsm`*) – the FSM from which the BDD comes from

class `pynusmv.dd.`**`StateInputs`**(*ptr*, *fsm*, *freeit=True*)
Bases: *`pynusmv.dd.BDD`*

Python class for State and Inputs structure.

A StateInputs is a *`BDD`* representing a single state/inputs pair of the model.

`get_str_values`()
Return a dictionary of the (variable, value) pairs of this StateInputs.

Return type a dictionary of pairs of strings.

class `pynusmv.dd.`**`Cube`**(*ptr*, *dd_manager=None*, *freeit=True*)
Bases: *`pynusmv.dd.BDD`*

Python class for Cube structure.

A Cube is a *`BDD`* representing a BDD cube of the model.

`diff`(*other*)
Compute the difference between this cube and *other*

Parameters **`other`** (*`BDD`*) – the other cube

`intersection`(*other*)
Compute the intersection of this Cube and *other*.

Parameters **`other`** (*`BDD`*) – the other Cube

`union`(*other*)
Compute the union of this Cube and *other*.

Parameters **`other`** (*`Cube`*) – the other Cube

class `pynusmv.dd.`**`DDManager`**(*pointer*, *freeit=False*)
Bases: *`pynusmv.utils.PointerWrapper`*

Python class for NuSMV BDD managers.

`size`
The number of variables handled by this manager.

`reorderings`
Returns the number of times reordering has occurred in this manager.

3.4 `pynusmv.exception` Module

The *`pynusmv.exception`* module provides all the exceptions used in PyNuSMV. Every particular exception raised by a PyNuSMV function is a sub-class of the *`PyNuSMVError`* class, such that one can catch all PyNuSMV by catching *`PyNuSMVError`* exceptions.

exception `pynusmv.exception.`**`PyNuSMVError`**
Bases: `Exception`

A generic PyNuSMV Error, superclass of all PyNuSMV Errors.

exception `pynusmv.exception.`**`MissingManagerError`**
Bases: *`pynusmv.exception.PyNuSMVError`*

Exception for missing BDD manager.

exception `pynusmv.exception.`**`NuSMVLexerError`**
Bases: *`pynusmv.exception.PyNuSMVError`*

Exception for NuSMV lexer error.

exception `pynusmv.exception.`**`NuSMVNoReadModelError`**
Bases: *`pynusmv.exception.PyNuSMVError`*

Exception raised when no SMV model has been read yet.

exception `pynusmv.exception.`**`NuSMVModelAlreadyReadError`**
Bases: *`pynusmv.exception.PyNuSMVError`*

Exception raised when a model is already read.

exception `pynusmv.exception.`**`NuSMVCannotFlattenError`**
Bases: *`pynusmv.exception.PyNuSMVError`*

Exception raised when no SMV model has been read yet.

exception `pynusmv.exception.`**`NuSMVModelAlreadyFlattenedError`**
Bases: *`pynusmv.exception.PyNuSMVError`*

Exception raised when the model is already flattened.

exception `pynusmv.exception.`**`NuSMVNeedFlatHierarchyError`**
Bases: *`pynusmv.exception.PyNuSMVError`*

Exception raised when the model must be flattened.

exception `pynusmv.exception.`**`NuSMVModelAlreadyEncodedError`**
Bases: *`pynusmv.exception.PyNuSMVError`*

Exception raised when the model is already encoded.

exception `pynusmv.exception.`**`NuSMVFlatModelAlreadyBuiltError`**

Bases: *`pynusmv.exception.PyNuSMVError`*

Exception raised when the flat model is already built.

exception `pynusmv.exception.`**`NuSMVNeedFlatModelError`**

Bases: *`pynusmv.exception.PyNuSMVError`*

Exception raised when the model must be flattened.

exception `pynusmv.exception.`**`NuSMVModelAlreadyBuiltError`**

Bases: *`pynusmv.exception.PyNuSMVError`*

Exception raised when the BDD model is already built.

exception `pynusmv.exception.`**`NuSMVNeedVariablesEncodedError`**

Bases: *`pynusmv.exception.PyNuSMVError`*

Exception raised when the variables of the model must be encoded.

exception `pynusmv.exception.`**`NuSMVInitError`**

Bases: *`pynusmv.exception.PyNuSMVError`*

NuSMV initialisation-related exception.

exception `pynusmv.exception.`**`NuSMVParserError`**

Bases: *`pynusmv.exception.PyNuSMVError`*

Exception raised when an error occured while parsing a string with NuSMV.

exception `pynusmv.exception.`**`NuSMVTypeCheckingError`**

Bases: *`pynusmv.exception.PyNuSMVError`*

Exception raised when an expression is wrongly typed.

exception `pynusmv.exception.`**`NuSMVFlatteningError`**

Bases: *`pynusmv.exception.PyNuSMVError`*

Exception raised when an error occured while flattening some expression.

exception `pynusmv.exception.`**`NuSMVBddPickingError`**

Bases: *`pynusmv.exception.PyNuSMVError`*

Exception raised when an error occured while picking a state/inputs from a BDD.

exception `pynusmv.exception.`**`NuSMVParsingError`**(*errors*)

Bases: *`pynusmv.exception.PyNuSMVError`*

A *`NuSMVParsingError`* is a NuSMV parsing exception. Contains several errors accessible through the *`errors`* attribute.

static **`from_nusmv_errors_list`**(*errors*)

Create a new NuSMVParsingError from the given list of NuSMV errors.

Parameters **`errors`** – the list of errors from NuSMV

`errors`

The tuple of errors of this exception. These errors are tuples *(line, token, message)* representing the line, the token and the message of the error.

exception `pynusmv.exception.`**`NuSMVModuleError`**

Bases: *`pynusmv.exception.PyNuSMVError`*

Exception raised when an error occured while creating a module.

exception `pynusmv.exception.`**`NuSMVSymbTableError`**
Bases: *`pynusmv.exception.PyNuSMVError`*

Exception raised when an error occured while working with symbol tables.

exception `pynusmv.exception.`**`NuSMVNeedBooleanModelError`**
Bases: *`pynusmv.exception.PyNuSMVError`*

Exception raised when the boolean model must be created.

exception `pynusmv.exception.`**`NuSMVBmcAlreadyInitializedError`**
Bases: *`pynusmv.exception.PyNuSMVError`*

Exception raised when the bmc sub system is already initialized

exception `pynusmv.exception.`**`NuSMVBeFsmMasterInstanceNotInitializedError`**
Bases: *`pynusmv.exception.PyNuSMVError`*

Exception raised when the one tries to access the global master BeFsm while it is not initialized

exception `pynusmv.exception.`**`NuSMVWffError`**
Bases: *`pynusmv.exception.PyNuSMVError`*

Exception raised when one tampers with a WFF in an unauthorized way

exception `pynusmv.exception.`**`NuSmvIllegalTraceStateError`**
Bases: *`pynusmv.exception.PyNuSMVError`*

Exception raised when an operation is made on a trace which is not in an appropriate state (for instance forcing a step to be considered loopback while the parent trace is frozen).

exception `pynusmv.exception.`**`BDDDumpFormatError`**
Bases: *`pynusmv.exception.PyNuSMVError`*

Exception raised when an error occurs while loading a dumped BDD.

3.5 `pynusmv.fsm` Module

The *`pynusmv.fsm`* module provides some functionalities about FSMs represented and stored by NuSMV:

- *`BddFsm`* represents the model encoded into BDDs. This gives access to elements of the FSM like BDD encoding, initial states, reachable states, transition relation, pre and post operations, etc.
- *`BddTrans`* represents a transition relation encoded with BDDs. It provides access to pre and post operations.
- *`BddEnc`* represents the BDD encoding, with some functionalities like getting the state mask or the input variables mask.
- *`SymbTable`* represents the symbols table of the model.

class `pynusmv.fsm.`**`BddFsm`**(*ptr*, *freeit=False*)
Bases: *`pynusmv.utils.PointerWrapper`*

Python class for FSM structure, encoded into BDDs.

The BddFsm provides some functionalities on the FSM: getting initial and reachable states as a BDD, getting or replacing the transition relation, getting fairness, state and inputs constraints, getting pre and post images of BDDs, possibly through particular actions, picking and counting states and actions of given BDDs.

`bddEnc`
The BDD encoding of this FSM.

init
The BDD of initial states of this FSM.

trans
The transition relation (*BddTrans*) of this FSM. Can also be replaced.

state_constraints
The BDD of states satisfying the invariants of the FSM.

inputs_constraints
The BDD of inputs satisfying the invariants of the FSM.

fairness_constraints
The list of fairness constraints, as BDDs.

reachable_states
The set of reachable states of this FSM, represented as a BDD.

deadlock_states
The set of reachable states of the system with no successor.

fair_states
The set of fair states of this FSM, represented as a BDD.

pre(*states*, *inputs=None*)
Return the pre-image of *states* in this FSM. If *inputs* is not *None*, it is used as constraints to get pre-states that are reachable through these inputs.

Parameters

- **states** (*BDD*) – the states from which getting the pre-image
- **inputs** (*BDD*) – the inputs through which getting the pre-image

Return type *BDD*

weak_pre(*states*)
Return the weak pre-image of *states* in this FSM. This means that it returns a BDD representing the set of states with corresponding inputs <s,i> such that there is a state in *state* reachable from s through i.

Parameters **states** (*BDD*) – the states from which getting the weak pre-image

Return type *BDD*

post(*states*, *inputs=None*)
Return the post-image of *states* in this FSM. If *inputs* is not *None*, it is used as constraints to get post-states that are reachable through these inputs.

Parameters

- **states** (*BDD*) – the states from which getting the post-image
- **inputs** (*BDD*) – the inputs through which getting the post-image

Return type *BDD*

pick_one_state(*bdd*)
Return a BDD representing a state of *bdd*.

Return type *State*

Raise a *NuSMVBddPickingError* if *bdd* is false or an error occurs while picking one state

pick_one_state_random(*bdd*)
Return a BDD representing a state of *bdd*, picked at random.

Return type `State`

Raise a `NuSMVBddPickingError` if *bdd* is false or an error occurs while picking one state

pick_one_inputs(*bdd*)
Return a BDD representing an inputs of *bdd*.

Return type `Inputs`

Raise a `NuSMVBddPickingError` if *bdd* is false or an error occurs while picking one inputs

pick_one_inputs_random(*bdd*)
Return a BDD representing an inputs of *bdd*, picked at random.

Return type `Inputs`

Raise a `NuSMVBddPickingError` if *bdd* is false or an error occurs while picking one inputs

pick_one_state_inputs(*bdd*)
Return a BDD representing a state/inputs pair of *bdd*.

Return type `StateInputs`

Raise a `NuSMVBddPickingError` if *bdd* is false or an error occurs while picking one pair

pick_one_state_inputs_random(*bdd*)
Return a BDD representing a state/inputs pair of *bdd*, picked at random.

Return type `StateInputs`

Raise a `NuSMVBddPickingError` if *bdd* is false or an error occurs while picking one pair

get_inputs_between_states(*current*, *next_*)
Return the BDD representing the possible inputs between *current* and *next_*.

Parameters

- **current** (*BDD*) – the source states
- **next** (*BDD*) – the destination states

Return type `BDD`

count_states(*bdd*)
Return the number of states of the given BDD.

Parameters **bdd** (*BDD*) – the concerned BDD

count_inputs(*bdd*)
Return the number of inputs of the given BDD

Parameters **bdd** (*BDD*) – the concerned BDD

count_states_inputs(*bdd*)
Return the number of state/inputs pairs of the given BDD

Parameters **bdd** (*BDD*) – the concerned BDD

pick_all_states(*bdd*)
Return a tuple of all states belonging to *bdd*.

Parameters **bdd** (*BDD*) – the concerned BDD

Return type tuple(`State`)

Raise a `NuSMVBddPickingError` if something is wrong

pick_all_inputs(*bdd*)
Return a tuple of all inputs belonging to *bdd*.

Parameters **bdd** (*BDD*) – the concerned BDD

Return type tuple(*Inputs*)

Raise a *NuSMVBddPickingError* if something is wrong

pick_all_states_inputs(*bdd*)
Return a tuple of all states/inputs pairs belonging to *bdd*.

Parameters **bdd** (*BDD*) – the concerned BDD

Return type tuple(*StateInputs*)

Raise a *NuSMVBddPickingError* if something is wrong

static from_filename(*filepath*)
Return the FSM corresponding to the model in *filepath*.

Parameters **filepath** – the path to the SMV model

static from_string(*model*)
Return the FSM corresponding to the model defined by the given string.

Parameters **model** – a String representing the SMV model

static from_modules(**modules*)
Return the FSM corresponding to the model defined by the given list of modules.

Parameters **modules** (a list of *Module* subclasses) – the modules defining the NuSMV model. Must contain a *main* module.

class pynusmv.fsm.**BddTrans**(*ptr*, *enc=None*, *manager=None*, *freeit=True*)
Bases: *pynusmv.utils.PointerWrapper*

Python class for transition relation encoded with BDDs.

A BddTrans represents a transition relation and provides pre and post operations on BDDs, possibly restricted to given actions.

monolithic
This transition relation represented as a monolithic BDD.

Return type *BDD*

pre(*states*, *inputs=None*)
Compute the pre-image of *states*, through *inputs* if not *None*.

Parameters

- **states** (*BDD*) – the concerned states
- **inputs** (*BDD*) – possible inputs

Return type *BDD*

post(*states*, *inputs=None*)
Compute the post-image of *states*, through *inputs* if not *None*.

Parameters

- **states** (*BDD*) – the concerned states
- **inputs** (*BDD*) – possible inputs

Return type *BDD*

classmethod **`from_trans`**(*symb_table*, *trans*, *context=None*)
Return a new BddTrans from the given trans.

Parameters

- **`symb_table`** (*`SymbTable`*) – the symbols table used to flatten the trans
- **`trans`** – the parsed string of the trans, not flattened
- **`context`** – an additional parsed context, in which trans will be flattened, if not None

Return type *`BddTrans`*

Raise a *`NuSMVFlatteningError`* if *trans* cannot be flattened under *context*

classmethod **`from_string`**(*symb_table*, *strtrans*, *strcontext=None*)
Return a new BddTrans from the given strtrans, in given strcontex.

Parameters

- **`symb_table`** (*`SymbTable`*) – the symbols table used to flatten the trans
- **`strtrans`** (*`str`*) – the string representing the trans
- **`strcontext`** – an additional string representing a context, in which trans will be flattened, if not None

Return type *`BddTrans`*

Raise a *`NuSMVTypeCheckingError`* if *strtrans* is wrongly typed under *context*

class `pynusmv.fsm.`**`BddEnc`**(*pointer*, *freeit=False*)
Bases: *`pynusmv.utils.PointerWrapper`*

Python class for BDD encoding.

A BddEnc provides some basic functionalities like getting the DD manager used to manage BDDs, the symbols table or the state and inputs masks.

`DDmanager`
The DD manager of this encoding.

Return type *`DDManager`*

`symbTable`
The symbols table of this encoding.

Return type *`SymbTable`*

`statesMask`
The mask for all state variables, represented as a BDD.

Return type *`BDD`*

`inputsMask`
The mask for all input variables, represented as a BDD.

Return type *`BDD`*

`statesInputsMask`
The mask for all input and state variables, represented as a BDD.

Return type *`BDD`*

`statesCube`
The cube for all state variables, represented as a BDD.

Return type *`BDD`*

inputsCube
The cube for all input variables, represented as a BDD.

Return type *BDD*

cube_for_inputs_vars(*variables*)
Return the cube for the given input variables.

Parameters **variables** – a list of input variable names

Return type *BDD*

cube_for_state_vars(*variables*)
Return the cube for the given state variables.

Parameters **variables** – a list of state variable names

Return type *BDD*

inputsVars
Return the set of inputs variables names.

Return type frozenset(str)

stateVars
Return the set of state variables names.

Return type frozenset(str)

definedVars
Return the set of defined variables names.

Return type frozenset(str)

get_variables_ordering(*var_type='scalar'*)
Return the order of variables.

Parameters **var_type** – the type of variables needed; *"scalar"* for only scalar variables (one variable per model variable), *"bits"* for bits for each scalar variables (default: "scalar")

Return type tuple(str)

force_variables_ordering(*order*)
Reorder variables based on the given order.

Parameters **order** – a list of variables names (scalar and/or bits) of the system; variables that are not part of the system are ignored (a warning is printed), variables of the system that are not in order are put at the end of the new order.

..note:: For more information on variables orders, see NuSMV documentation.

dump(*bdd*, *file_*)
Dump the given BDD into the given file.

Parameters

- **bdd** – the BDD to dump.
- **file** – the file object in which the BDD is dumped.

Note: The content of the file is composed of:

- the list of variables appearing in the BDD, one variable name per line;
- the BDD itself, where each line is:

- TRUE: for the TRUE node
- FALSE: for the FALSE node
- VAR COMP IDTHEN IDELSE: for any other node where VAR is the index of the variable of the node in the list above, COMP is 0 or 1 depending on whether the node is complemented (1) or not (0), and IDTHEN and IDELSE are the indices of the then and else children of the node in the list of nodes (starting at 0 with the first dumped node).

The lines are ordered according to an inverse topological order of the DAG represented by the BDD. The two parts of the file are separated by an empty line.

`load`(*file_*)

Load and return the BDD stored in the given file.

Parameters **`file`** – the file object in which the BDD is dumped.

Raise a *`BDDDumpFormatError`* if some error occurs while loading the BDD.

Note: The content of the file is composed of:

- the list of variables appearing in the BDD, one variable name per line;
- the BDD itself, where each line is:
  - TRUE: for the TRUE node
  - FALSE: for the FALSE node
  - VAR COMP IDTHEN IDELSE: for any other node where VAR is the index of the variable of the node in the list above, COMP is 0 or 1 depending on whether the node is complemented (1) or not (0), and IDTHEN and IDELSE are the indices of the then and else children of the node in the list of nodes (starting at 0 with the first dumped node).

The lines are ordered according to an inverse topological order of the DAG represented by the BDD. The two parts of the file are separated by an empty line.

class `pynusmv.fsm.`**`SymbTable`**(*pointer*, *freeit=False*)

Bases: *`pynusmv.utils.PointerWrapper`*

Python class for symbols table.

`ins_policies` **= {'SYMB_LAYER_POS_FORCE_TOP': <MagicMock name='mock.symb_table.SYMB_LAYER_POS**

`SYMBOL_STATE_VAR` **= <MagicMock name='mock.symb_table.SYMBOL_STATE_VAR' id='139958098407040'>**

`SYMBOL_FROZEN_VAR` **= <MagicMock name='mock.symb_table.SYMBOL_FROZEN_VAR' id='139958098423592'>**

`SYMBOL_INPUT_VAR` **= <MagicMock name='mock.symb_table.SYMBOL_INPUT_VAR' id='139958098431952'>**

`layer_names`

The names of the layers of this symbol table.

`create_layer`(*layer_name*, *ins_policy=<MagicMock name='mock.symb_table.SYMB_LAYER_POS_DEFAULT' id='139958113302512'>*)

Create a new layer in this symbol table.

Parameters

- **`layer_name`** (`str`) – the name of the created layer
- **`ins_policy`** – the insertion policy for inserting the new layer

get_variable_type(*variable*)
Return the type of the given variable.

Parameters **variable** (`Node`) – the name of the variable

Return type a NuSMV *SymbType_ptr*

Warning: The returned pointer must not be altered or freed.

can_declare_var(*layer*, *variable*)
Return whether the given *variable* name can be declared in *layer*.

Parameters

- **layer** (`str`) – the name of the layer
- **variable** (*Node*) – the name of the variable

Return type `bool`

declare_input_var(*layer*, *ivar*, *type_*)
Declare a new input variable in this symbol table.

Parameters

- **layer** (`str`) – the name of the layer in which insert the variable
- **ivar** (*Node*) – the name of the input variable
- **type** (*Node*) – the type of the declared input variable

Raise a *NuSMVSymbTableError* if the variable is already defined in the given layer

Warning: *type_* must be already resolved, that is, the body of *type_* must be leaf values.

declare_state_var(*layer*, *var*, *type_*)
Declare a new state variable in this symbol table.

Parameters

- **layer** (`str`) – the name of the layer in which insert the variable
- **var** (*Node*) – the name of the state variable
- **type** (*Node*) – the type of the declared state variable

Raise a *NuSMVSymbTableError* if the variable is already defined in the given layer

Warning: *type_* must be already resolved, that is, the body of *type_* must be leaf values.

declare_frozen_var(*layer*, *fvar*, *type_*)
Declare a new frozen variable in this symbol table.

Parameters

- **layer** (`str`) – the name of the layer in which insert the variable
- **fvar** (*Node*) – the name of the frozen variable
- **type** (*Node*) – the type of the declared frozen variable

Raise a *NuSMVSymbTableError* if the variable is already defined in the given layer

> **Warning:** *type_* must be already resolved, that is, the body of *type_* must be leaf values.

declare_var(*layer*, *name*, *type_*, *kind*)
Declare a new variable in this symbol table.

Parameters

- **layer** (str) – the name of the layer in which insert the variable
- **name** (*Node*) – the name of the variable
- **type** (*Node*) – the type of the declared variable
- **kind** – the kind of the declared variable

Raise a *NuSMVSymbTableError* if the variable is already defined in the given layer

> **Warning:** *type_* must be already resolved, that is, the body of *type_* must be leaf values.

is_input_var(*ivar*)
Return whether the given *var* name is a declared input variable.

Parameters **ivar** (*Node*) – the name of the input variable

Raise a *NuSMVSymbTableError* if the variable is not defined in this symbol table

is_state_var(*var*)
Return whether the given *var* name is a declared state variable.

Parameters **var** (*Node*) – the name of the state variable

Raise a *NuSMVSymbTableError* if the variable is not defined in this symbol table

is_frozen_var(*fvar*)
Return whether the given *var* name is a declared frozen variable.

Parameters **fvar** (*Node*) – the name of the frozen variable

Raise a *NuSMVSymbTableError* if the variable is not defined in this symbol table

3.6 `pynusmv.glob` Module

The *pynusmv.glob* module provide some functions to access global NuSMV functionalities. These functions are used to feed an SMV model to NuSMV and build the different structures representing the model, like flattening the model, building its BDD encoding and getting the BDD-encoded FSM.

Besides the functions, this module provides an access to main globally stored data structures like the flat hierarchy, the BDD encoding, the symbols table and the propositions database.

pynusmv.glob.**load**(**model*)
Load the given model. This model can be of several forms:

- a file path; in this case, the model is loaded from the file;
- NuSMV modelling code; in this case, *model* is the code for the model;
- a list of modules (list of *Module* subclasses); in this case, the model is represented by the set of modules.

`pynusmv.glob.`**`flatten_hierarchy`**(*keep_single_enum=False*)

Flatten the read model and store it in global data structures.

Parameters **`keep_single_enum`** (`bool`) – whether or not enumerations with single values should be converted into defines

Raise a *`NuSMVNoReadModelError`* if no model is read yet

Raise a *`NuSMVCannotFlattenError`* if an error occurred during flattening

Raise a *`NuSMVModelAlreadyFlattenedError`* if the model is already flattened

> **Warning:** In case of type checking errors, a message is printed at stderr and a *`NuSMVCannotFlattenError`* is raised.

`pynusmv.glob.`**`symb_table`**()

Return the main symbols table of the current model.

Return type *`SymbTable`*

`pynusmv.glob.`**`encode_variables`**(*layers=None*, *variables_ordering=None*)

Encode the BDD variables of the current model and store it in global data structures. If variables_ordering is provided, use this ordering to encode the variables; otherwise, the default ordering method is used.

Parameters

- **`layers`** (`set`) – the set of layers variables to encode
- **`variables_ordering`** (*`path to file`*) – the file containing a custom ordering

Raise a *`NuSMVNeedFlatHierarchyError`* if the model is not flattened

Raise a *`NuSMVModelAlreadyEncodedError`* if the variables are already encoded

`pynusmv.glob.`**`encode_variables_for_layers`**(*layers=None*, *init=False*)

Encode the BDD variables of the given layers and store them in global data structures.

Parameters

- **`layers`** (`set`) – the set of layers variables to encode
- **`init`** (`bool`) – whether or not initialize the global encodings

`pynusmv.glob.`**`bdd_encoding`**()

Return the main bdd encoding of the current model.

Return type `BddEnc`

`pynusmv.glob.`**`build_flat_model`**()

Build the Sexp FSM (Simple Expression FSM) of the current model and store it in global data structures.

Raise a *`NuSMVNeedFlatHierarchyError`* if the model is not flattened

Raise a *`NuSMVFlatModelAlreadyBuiltError`* if the Sexp FSM is already built

`pynusmv.glob.`**`build_model`**()

Build the BDD FSM of the current model and store it in global data structures.

Raise a *`NuSMVNeedFlatModelError`* if the Sexp FSM of the model is not built yet

Raise a *`NuSMVNeedVariablesEncodedError`* if the variables of the model are not encoded yet

Raise a *`NuSMVModelAlreadyBuiltError`* if the BDD FSM of the model is already built `pynusmv.glob.`**`build_boolean_model`**(*force=False*)

Compiles the flattened hierarchy into a boolean model (SEXP) and stores it it a global variable.

Note: This function is subject to the following requirements:

- hierarchy must already be flattened (*`flatten_hierarchy()`*)
- encoding must be already built (*`encode_variables()`*)
- boolean model must not exist yet (or the force flag must be on)

Parameters **`force`** – a flag telling whether or not the boolean model must be built even though the cone of influence option is turned on.

Raises

- ***`NuSMVNeedFlatHierarchyError`*** – if the hierarchy wasn't flattened yet.
- ***`NuSMVNeedVariablesEncodedError`*** – if the variables are not yet encoded
- ***`NuSMVModelAlreadyBuiltError`*** – if the boolean model is already built and force=False

`pynusmv.glob.`**`flat_hierarchy`**()

Return the global flat hierarchy.

Return type *`FlatHierarchy`*

`pynusmv.glob.`**`prop_database`**()

Return the global properties database.

Return type *`PropDb`*

`pynusmv.glob.`**`compute_model`**(*variables_ordering=None*, *keep_single_enum=False*)

Compute the read model and store its parts in global data structures. This function is a shortcut for calling all the steps of the model building that are not yet performed. If variables_ordering is not None, it is used as a file containing the order of variables used for encoding the model into BDDs.

Parameters

- **`variables_ordering`** (*`path to file`*) – the file containing a custom ordering
- **`keep_single_enum`** (`bool`) – whether or not enumerations with single values should be converted into defines

3.7 `pynusmv.init` Module

The *`pynusmv.init`* module provides functions to initialize and quit NuSMV.

The *`init_nusmv()`* function can be used as a context manager for the *with* Python statement:

```
with init_nusmv():
    ...
```

Warning: *`init_nusmv()`* should be called before any other call to pynusmv functions; *`deinit_nusmv()`* should be called after using pynusmv.

`pynusmv.init.`**`init_nusmv`**(*collecting=True*)

Initialize NuSMV. Must be called only once before calling *`deinit_nusmv()`*.

Parameters **`collecting`** – Whether or not collecting pointer wrappers to free them before deinit-ing nusmv.

`pynusmv.init.`**`deinit_nusmv`**(*ddinfo=False*)

Quit NuSMV. Must be called only once, after calling *`init_nusmv()`*.

Parameters **`ddinfo`** – Whether or not display Decision Diagrams statistics.

`pynusmv.init.`**`reset_nusmv`**()

Reset NuSMV, i.e. deinit it and init it again. Cannot be called before *`init_nusmv()`*.

`pynusmv.init.`**`is_nusmv_init`**()

Return whether NuSMV is initialized.

3.8 `pynusmv.mc` Module

The *`pynusmv.mc`* module provides some functions of NuSMV dealing with model checking, like CTL model checking.

`pynusmv.mc.`**`check_ltl_spec`**(*spec*)

Return whether the loaded SMV model satisfies or not the LTL given *spec*. That is, return whether all initial states of le model satisfies *spec* or not.

Parameters **`spec`** (*`Spec`*) – a specification

Return type *bool*

`pynusmv.mc.`**`check_explain_ltl_spec`**(*spec*)

Return whether the loaded SMV model satisfies or not the LTL given *spec*, that is, whether all initial states of le model satisfies *spec* or not. Return also an explanation for why the model does not satisfy *spec*, if it is the case, or *None* otherwise.

The result is a tuple where the first element is a boolean telling whether *spec* is satisfied, and the second element is either *None* if the first element is *True*, or a path of the SMV model violating *spec* otherwise.

The explanation is a tuple of alternating states and inputs, starting and ennding with a state. The path is looping if the last state is somewhere else in the sequence. States and inputs are represented by dictionaries where keys are state and inputs variable of the loaded SMV model, and values are their value.

Parameters **`spec`** (*`Spec`*) – a specification

Return type tuple

`pynusmv.mc.`**`check_ctl_spec`**(*fsm*, *spec*, *context=None*)

Return whether the given *fsm* satisfies or not the given *spec* in *context*, if specified. That is, return whether all initial states of *fsm* satisfies *spec* in context or not.

Parameters

- **`fsm`** (*`BddFsm`*) – the concerned FSM
- **`spec`** (*`Spec`*) – a specification about *fsm*
- **`context`** (*`Spec`*) – the context in which evaluate *spec*

Return type *bool*

`pynusmv.mc.`**`eval_simple_expression`**(*fsm*, *sexp*)

Return the set of states of *fsm* satisfying *sexp*, as a BDD. *sexp* is first parsed, then evaluated on *fsm*.

Parameters

- **`fsm`** (*`BddFsm`*) – the concerned FSM
- **`sexp`** – a simple expression, as a string

Return type *`BDD`*

`pynusmv.mc.`**`eval_ctl_spec`**(*fsm*, *spec*, *context=None*)
Return the set of states of *fsm* satisfying *spec* in *context*, as a BDD.

Parameters

- **`fsm`** (*`BddFsm`*) – the concerned FSM
- **`spec`** (*`Spec`*) – a specification about *fsm*
- **`context`** (*`Spec`*) – the context in which evaluate *spec*

Return type *`BDD`*

`pynusmv.mc.`**`ef`**(*fsm*, *states*)
Return the set of states of *fsm* satisfying *EF states*, as a BDD.

Parameters

- **`fsm`** (*`BddFsm`*) – the concerned FSM
- **`states`** (*`BDD`*) – a set of states of *fsm*

Return type *`BDD`*

`pynusmv.mc.`**`eg`**(*fsm*, *states*)
Return the set of states of *fsm* satisfying *EG states*, as a BDD.

Parameters

- **`fsm`** (*`BddFsm`*) – the concerned FSM
- **`states`** (*`BDD`*) – a set of states of *fsm*

Return type *`BDD`*

`pynusmv.mc.`**`ex`**(*fsm*, *states*)
Return the set of states of *fsm* satisfying *EX states*, as a BDD.

Parameters

- **`fsm`** (*`BddFsm`*) – the concerned FSM
- **`states`** (*`BDD`*) – a set of states of *fsm*

Return type *`BDD`*

`pynusmv.mc.`**`eu`**(*fsm*, *s1*, *s2*)
Return the set of states of *fsm* satisfying *E[s1 U s2]*, as a BDD.

Parameters

- **`fsm`** (*`BddFsm`*) – the concerned FSM
- **`s1`** (*`BDD`*) – a set of states of *fsm*
- **`s2`** (*`BDD`*) – a set of states of *fsm*

Return type *`BDD`*

`pynusmv.mc.`**`au`**(*fsm*, *s1*, *s2*)
Return the set of states of *fsm* satisfying *A[s1 U s2]*, as a BDD.

Parameters

- **fsm** (*BddFsm*) – the concerned FSM
- **s1** (*BDD*) – a set of states of *fsm*
- **s2** (*BDD*) – a set of states of *fsm*

Return type *BDD*

`pynusmv.mc.`**`explain`**(*fsm*, *state*, *spec*, *context=None*)

Explain why *state* of *fsm* satisfies *spec* in *context*.

Parameters

- **state** (*State*) – a state of *fsm*
- **fsm** (*BddFsm*) – the system
- **spec** (*Spec*) – a specification about *fsm*
- **context** (*Spec*) – the context in which evaluate *spec*

Return a tuple *t* composed of states (*State*) and inputs (*Inputs*), such that *t[0]* is *state* and *t* represents a path in *fsm* explaining why *state* satisfies *spec* in *context*. The returned path is looping if the last state of path is equal to a previous state along the path.

`pynusmv.mc.`**`explainEX`**(*fsm*, *state*, *a*)

Explain why *state* of *fsm* satisfies *EX phi*, where *a* is the set of states of *fsm* satisfying *phi*, represented by a BDD.

Parameters

- **fsm** (*BddFsm*) – the system
- **state** (*State*) – a state of *fsm*
- **a** (*BDD*) – the set of states of *fsm* satisfying *phi*

Return *(s, i, s')* tuple where *s* (*State*) is the given state, *s'* (*State*) is a successor of *s* belonging to *a* and *i* (*Inputs*) is the inputs to go from *s* to *s'* in *fsm*.

`pynusmv.mc.`**`explainEU`**(*fsm*, *state*, *a*, *b*)

Explain why *state* of *fsm* satisfies *E[phi U psi]*, where *a is the set of states of 'fsm* satisfying *phi* and *b* is the set of states of *fsm* satisfying *psi*, both represented by BDDs.

Parameters

- **fsm** (*BddFsm*) – the system
- **state** (*State*) – a state of *fsm*
- **a** (*BDD*) – the set of states of *fsm* satisfying *phi*
- **b** (*BDD*) – the set of states of *fsm* satisfying *psi*

Return a tuple *t* composed of states (*State*) and inputs (*Inputs*), such that *t[0]* is *state*, *t[-1]* belongs to *b*, and every other state of *t* belongs to *a*. The states of *t* are separated by inputs. Furthermore, *t* represents a path in *fsm*.

`pynusmv.mc.`**`explainEG`**(*fsm*, *state*, *a*)

Explain why *state* of *fsm* satisfies *EG phi*, where *a* the set of states of *fsm* satisfying *phi*, represented by a BDD.

Parameters

- **fsm** (*BddFsm*) – the system
- **state** (*State*) – a state of *fsm*

- **a** (*BDD*) – the set of states of *fsm* satisfying *phi*

Return a tuple *(t, (i, loop))* where *t* is a tuple composed of states (*State*) and inputs (*Inputs*), such that *t[0]* is state and every other state of *t* belongs to *a*. The states of *t* are separated by inputs. Furthermore, *t* represents a path in *fsm*. *loop* represents the start of the loop contained in *t*, i.e. *t[-1]* can lead to *loop* through *i*, and *loop* is a state of *t*.

3.9 `pynusmv.model` Module

The *pynusmv.model* module provides a way to define NuSMV modules in Python. The module is composed of several classes that fall in five sections:

- `Expression` sub-classes represent elements of expressions of the NuSMV modelling language. NuSMV expressions can be defined by combining these classes (e.g. `Add(Identifier("c"), 1)`), by using `Expression` methods (e.g. `Identifier("c").add(1)`) or by using built-in operators (e.g. `Identifier("c") + 1`).
- `Type` sub-classes represent types of NuSMV variables.
- `Section` sub-classes represent the different sections (VAR, IVAR, TRANS, etc.) of a NuSMV module.
- `Declaration` sub-classes are used in the declaration of a module to allow a more pythonic way of declaring NuSMV variables.
- *Module*: the *Module* class represents a generic NuSMV module, and must be subclassed to define specific NuSMV modules. See the documentation of the *Module* class to get more information on how to declare a NuSMV module with this class.

`pynusmv.model.`**`Comment`**(*element*, *string*)

Attach the given comment to the given element.

Parameters

- **element** (*Element*) – the element to attach the comment to.
- **string** (*str*) – the comment to attach.

Returns the element itself.

class `pynusmv.model.`**`Identifier`**(*name*, **args*, ***kwargs*)

Bases: `pynusmv.model.Expression`

An identifier.

class `pynusmv.model.`**`Self`**(**args*, ***kwargs*)

Bases: *pynusmv.model.Identifier*

The *self* identifier.

class `pynusmv.model.`**`Dot`**(*instance*, *element*, **args*, ***kwargs*)

Bases: `pynusmv.model.ComplexIdentifier`

Access to a part of a module instance.

class `pynusmv.model.`**`ArrayAccess`**(*array*, *index*, **args*, ***kwargs*)

Bases: `pynusmv.model.ComplexIdentifier`

Access to an index of an array.

class `pynusmv.model.`**`NumericalConst`**(*value*, **args*, ***kwargs*)

Bases: `pynusmv.model.Constant`

A numerical constant.

class `pynusmv.model.`**`Trueexp`**(**args*, ***kwargs*)
Bases: `pynusmv.model.BooleanConst`

The TRUE constant.

class `pynusmv.model.`**`Falseexp`**(**args*, ***kwargs*)
Bases: `pynusmv.model.BooleanConst`

The FALSE constant.

class `pynusmv.model.`**`NumberWord`**(*value*, **args*, ***kwargs*)
Bases: `pynusmv.model.Constant`

A word constant.

class `pynusmv.model.`**`RangeConst`**(*start*, *stop*, **args*, ***kwargs*)
Bases: `pynusmv.model.Constant`

A range of integers.

class `pynusmv.model.`**`Conversion`**(*target_type*, *value*, **args*, ***kwargs*)
Bases: `pynusmv.model.Function`

Converting an expression into a specific type.

class `pynusmv.model.`**`WordFunction`**(*function*, *value*, *size*, **args*, ***kwargs*)
Bases: `pynusmv.model.Function`

A function applied on a word.

class `pynusmv.model.`**`Count`**(*values*, **args*, ***kwargs*)
Bases: `pynusmv.model.Function`

A counting function.

class `pynusmv.model.`**`Next`**(*value*, **args*, ***kwargs*)
Bases: `pynusmv.model.Expression`

A next expression.

class `pynusmv.model.`**`Smallinit`**(*value*, **args*, ***kwargs*)
Bases: `pynusmv.model.Expression`

An init() expression.

class `pynusmv.model.`**`Case`**(*values*, **args*, ***kwargs*)
Bases: `pynusmv.model.Expression`

A case expression.

class `pynusmv.model.`**`Subscript`**(*array*, *index*, **args*, ***kwargs*)
Bases: `pynusmv.model.Expression`

Array subscript.

class `pynusmv.model.`**`BitSelection`**(*word*, *start*, *stop*, **args*, ***kwargs*)
Bases: `pynusmv.model.Expression`

Word bit selection.

class `pynusmv.model.`**`Set`**(*elements*, **args*, ***kwargs*)
Bases: `pynusmv.model.Expression`

A set.

class `pynusmv.model.`**`Not`**(*value*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

A negated (-) expression.

class `pynusmv.model.`**`Concat`**(*left*, *right*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

A concatenation (*::*) of expressions.

class `pynusmv.model.`**`Minus`**(*value*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

Minus (-) expression.

class `pynusmv.model.`**`Mult`**(*left*, *right*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

A multiplication (*) of expressions.

class `pynusmv.model.`**`Div`**(*left*, *right*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

A division (/) of expressions.

class `pynusmv.model.`**`Mod`**(*left*, *right*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

A modulo (%) of expressions.

class `pynusmv.model.`**`Add`**(*left*, *right*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

An addition (+) of expressions.

class `pynusmv.model.`**`Sub`**(*left*, *right*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

A subtraction (-) of expressions.

class `pynusmv.model.`**`LShift`**(*left*, *right*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

A left shift (<<) of expressions.

class `pynusmv.model.`**`RShift`**(*left*, *right*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

A right shift (>>) of expressions.

class `pynusmv.model.`**`Union`**(*left*, *right*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

A union (*union*) of expressions.

class `pynusmv.model.`**`In`**(*left*, *right*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

The *in* expression.

class `pynusmv.model.`**`Equal`**(*left*, *right*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

The = expression.

class `pynusmv.model.`**`NotEqual`**(*left*, *right*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

The *!=* expression.

class `pynusmv.model.`**`Lt`**(*left*, *right*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

The < expression.

class `pynusmv.model.`**`Gt`**(*left*, *right*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

The > expression.

class `pynusmv.model.`**`Le`**(*left*, *right*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

The <= expression.

class `pynusmv.model.`**`Ge`**(*left*, *right*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

The >= expression.

class `pynusmv.model.`**`And`**(*left*, *right*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

The *&* expression.

class `pynusmv.model.`**`Or`**(*left*, *right*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

The | expression.

class `pynusmv.model.`**`Xor`**(*left*, *right*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

The *xor* expression.

class `pynusmv.model.`**`Xnor`**(*left*, *right*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

The *xnor* expression.

class `pynusmv.model.`**`Ite`**(*condition*, *left*, *right*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

The *? :* expression.

class `pynusmv.model.`**`Iff`**(*left*, *right*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

The <-> expression.

class `pynusmv.model.`**`Implies`**(*left*, *right*, **args*, ***kwargs*)
Bases: `pynusmv.model.Operator`

The -> expression.

class `pynusmv.model.`**`ArrayExpr`**(*array*, **args*, ***kwargs*)
Bases: `pynusmv.model.Element`

An array define expression.

class `pynusmv.model.`**`Boolean`**(*comments=None*)
Bases: `pynusmv.model.SimpleType`

A boolean type.

class `pynusmv.model.`**`Word`**(*size*, *sign=None*, **args*, ***kwargs*)
Bases: `pynusmv.model.SimpleType`

A word type.

class `pynusmv.model.`**`Scalar`**(*values*, **args*, ***kwargs*)
Bases: `pynusmv.model.SimpleType`

An enumeration type.

class `pynusmv.model.`**`Range`**(*start*, *stop*, **args*, ***kwargs*)
Bases: `pynusmv.model.SimpleType`

A range type.

class `pynusmv.model.`**`Array`**(*start*, *stop*, *elementtype*, **args*, ***kwargs*)
Bases: `pynusmv.model.SimpleType`

An array type.

class `pynusmv.model.`**`Modtype`**(*modulename*, *arguments*, *process=False*, **args*, ***kwargs*)
Bases: `pynusmv.model.Type`

A module instantiation.

class `pynusmv.model.`**`Variables`**(*variables*, **args*, ***kwargs*)
Bases: `pynusmv.model.MappingSection`

Declaring variables.

class `pynusmv.model.`**`InputVariables`**(*ivariables*, **args*, ***kwargs*)
Bases: `pynusmv.model.MappingSection`

Declaring input variables.

class `pynusmv.model.`**`FrozenVariables`**(*fvariables*, **args*, ***kwargs*)
Bases: `pynusmv.model.MappingSection`

Declaring frozen variables.

class `pynusmv.model.`**`Defines`**(*defines*, **args*, ***kwargs*)
Bases: `pynusmv.model.MappingSection`

Declaring defines.

class `pynusmv.model.`**`Assigns`**(*assigns*, **args*, ***kwargs*)
Bases: `pynusmv.model.MappingSection`

Declaring assigns.

class `pynusmv.model.`**`Constants`**(*constants*, **args*, ***kwargs*)
Bases: `pynusmv.model.ListingSection`

Declaring constants.

class `pynusmv.model.`**`Trans`**(*body*, **args*, ***kwargs*)
Bases: `pynusmv.model.ListingSection`

A TRANS section.

class `pynusmv.model.`**`Init`**(*body*, **args*, ***kwargs*)
Bases: `pynusmv.model.ListingSection`

An INIT section.

class `pynusmv.model.`**`Invar`**(*body*, **args*, ***kwargs*)
Bases: `pynusmv.model.ListingSection`

An INVAR section.

class `pynusmv.model.`**`Fairness`**(*body*, **args*, ***kwargs*)
Bases: `pynusmv.model.ListingSection`

A FAIRNESS section.

class `pynusmv.model.`**`Justice`**(*body*, **args*, ***kwargs*)
Bases: `pynusmv.model.ListingSection`

A Justice section.

class `pynusmv.model.`**`Compassion`**(*body*, **args*, ***kwargs*)
Bases: `pynusmv.model.ListingSection`

A COMPASSION section.

class `pynusmv.model.`**`Var`**(*type_*, *name=None*, **args*, ***kwargs*)
Bases: `pynusmv.model.Declaration`

A declared VAR.

class `pynusmv.model.`**`IVar`**(*type_*, *name=None*, **args*, ***kwargs*)
Bases: `pynusmv.model.Declaration`

A declared IVAR.

class `pynusmv.model.`**`FVar`**(*type_*, *name=None*, **args*, ***kwargs*)
Bases: `pynusmv.model.Declaration`

A declared FROZENVAR.

class `pynusmv.model.`**`Def`**(*type_*, *name=None*, **args*, ***kwargs*)
Bases: `pynusmv.model.Declaration`

A declared DEFINE.

class `pynusmv.model.`**`Module`**(**args*, ***kwargs*)
Bases: *`pynusmv.model.Modtype`*

A generic module.

To create a new module, the user must subclass the *`Module`* class and add class attributes with names corresponding to sections of NuSMV module definitions: *VAR*, *IVAR*, *FROZENVAR*, *DEFINE*, *CONSTANTS*, *ASSIGN*, *TRANS*, *INIT*, *INVAR*, *FAIRNESS*, *JUSTICE*, *COMPASSION*.

In addition to these attributes, the *ARGS*, *NAME* and *COMMENT* attributes can be defined:

- If *NAME* is defined, it overrides module name for the NuSMV module name.
- If *ARGS* is defined, it must be a sequence object where each element's string representation is an argument of the module.
- If *COMMENT* is defined, it will be used as the module's header, that is, it will be added as a NuSMV comment just before the module declaration.

Treatment of the section depends of the type of the section and the value of the corresponding attribute.

CONSTANTS section If the value of the section is a string (`str`), it is parsed as the body of the constants declaration. Otherwise, the value must be a sequence and it is parsed as the defined constants.

VAR, IVAR, FROZENVAR, DEFINE, ASSIGN sections If the value of the section is a string (`str`), it is parsed as the body of the declaration. If it is a dictionary (`dict`), keys are parsed as names of variables (or input variables, define, etc.) if they are strings, or used as they are otherwise, and values are parsed as bodies of the declaration (if strings, kept as they are otherwise). Otherwise, the value must be a sequence, and each element is treated separately:

- if the element is a string (`str`), it is parsed as a declaration;
- otherwise, the element must be a sequence, and the first element is used as the name of the variable (or input variable, define, etc.) and parsed if necessary, and the second one as the body of the declaration.

TRANS, INIT, INVAR, FAIRNESS, JUSTICE, COMPASSION sections If the value of the section is a string (`str`), it is parsed as the body of the section. Otherwise, it must be a sequence and the representation (parsed if necessary) of the elements of the sequence are declared as different sections.

In addition to these sections, the class body can contain instances of `pynsumv.model.Declaration`. These ones take the name of the corresponding variable, and are added to the corresponding section (*VAR*, *IVAR*, *FROZENVAR* or *DEFINE*) when creating the class.

For example, the class

```python
class TwoCounter(Module):
    COMMENT = "Two asynchronous counters"
    NAME = "twoCounter"
    ARGS = ["run"]
    c1 = Range(0, 2)
    VAR = {"c2": "0..2"}
    INIT = [c1 == 0 & "c2 = 0"]
    TRANS = [Next(c1) == Ite("run", (c1 + 1) % 2, c1),
             "next(c2) = run ? c2+1 mod 2 : c2"]
```

defines the module

```
-- Two asynchronous counters
MODULE twoCounter(run)
    VAR
        c1 : 0..2;
        c2 : 0..2;
    INIT
        c1 = 0 & c2 = 0
    TRANS
        next(c1) = run ? c1+1 mod 2 : c1
    TRANS
        next(c2) = run ? c2+1 mod 2 : c2
```

After creation, module sections satisfy the following patterns:

- pair-based sections such as VAR, IVAR, FROZENVAR, DEFINE and ASSIGN are mapping objects (dictionaries) where keys are identifiers and values are types (for VAR, IVAR and FROZENVAR) or expressions (for DEFINE and ASSIGN).
- list-based sections such as CONSTANTS are enumerations composed of elements of the section.
- expression-based sections such as TRANS, INIT, INVAR, FAIRNESS, JUSTICE and COMPASSION are enumerations composed of expressions.

ARGS = []

COMMENT = ‘’

NAME = 'Module'

members = ('__module__', '__qualname__', '__doc__', '__init__', 'NAME', 'COMMENT', 'ARGS')

source = None

3.10 `pynusmv.node` Module

The *pynusmv.node* module provides classes representing NuSMV internal nodes, as well as a class *FlatHierarchy* to represent a NuSMV flat hierarchy.

`pynusmv.node.`**`find_hierarchy`**(*node*)

Traverse the hierarchy represented by *node* and transfer it to the node hash table.

Parameters **node** (*a SWIG wrapper for a NuSMV node_ptr*) – the node

class `pynusmv.node.`**`Node`**(*left*, *right*, *type_=None*)

Bases: *pynusmv.utils.PointerWrapper*

A generic NuSMV node.

car

The left branch of this node.

cdr

The right branch of this node.

static **`from_ptr`**(*ptr*, *freeit=False*)

class `pynusmv.node.`**`Module`**(*left*, *right*, *type_=None*)

Bases: *pynusmv.node.Node*

class `pynusmv.node.`**`Section`**(*left*, *right*, *type_=None*)

Bases: *pynusmv.node.Node*

A generic section.

class `pynusmv.node.`**`Trans`**(*left*, *right*, *type_=None*)

Bases: *pynusmv.node.Section*

class `pynusmv.node.`**`Init`**(*left*, *right*, *type_=None*)

Bases: *pynusmv.node.Section*

class `pynusmv.node.`**`Invar`**(*left*, *right*, *type_=None*)

Bases: *pynusmv.node.Section*

class `pynusmv.node.`**`Assign`**(*left*, *right*, *type_=None*)

Bases: *pynusmv.node.Section*

class `pynusmv.node.`**`Fairness`**(*left*, *right*, *type_=None*)

Bases: *pynusmv.node.Section*

class `pynusmv.node.`**`Justice`**(*left*, *right*, *type_=None*)

Bases: *pynusmv.node.Section*

class `pynusmv.node.`**`Compassion`**(*left*, *right*, *type_=None*)

Bases: *pynusmv.node.Section*

class `pynusmv.node.`**`Spec`**(*left*, *right*, *type_=None*)

Bases: *pynusmv.node.Section*

class `pynusmv.node.`**`Ltlspec`**(*left*, *right*, *type_=None*)

Bases: *pynusmv.node.Section*

class `pynusmv.node.`**`Pslspec`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Section`*

class `pynusmv.node.`**`Invarspec`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Section`*

class `pynusmv.node.`**`Compute`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Section`*

class `pynusmv.node.`**`Define`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Section`*

class `pynusmv.node.`**`ArrayDef`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Node`*

class `pynusmv.node.`**`Isa`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Section`*

class `pynusmv.node.`**`Constants`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Section`*

class `pynusmv.node.`**`Var`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Section`*

class `pynusmv.node.`**`Frozenvar`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Section`*

class `pynusmv.node.`**`Ivar`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Section`*

class `pynusmv.node.`**`Type`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Node`*

A generic type node.

class `pynusmv.node.`**`Boolean`**
Bases: *`pynusmv.node.Type`*

The boolean type.

class `pynusmv.node.`**`UnsignedWord`**(*length*)
Bases: *`pynusmv.node.Type`*

An unsigned word type.

`length`

class `pynusmv.node.`**`Word`**(*length*)
Bases: *`pynusmv.node.UnsignedWord`*

An unsigned word type.

class `pynusmv.node.`**`SignedWord`**(*length*)
Bases: *`pynusmv.node.Type`*

A signed word type.

`length`

class `pynusmv.node.`**`Range`**(*start*, *stop*)
Bases: *`pynusmv.node.Type`*

A range type.

`start`

stop

class pynusmv.node.**ArrayType**(*start*, *stop*, *elementtype*)

Bases: *pynusmv.node.Type*

An array type.

start

stop

elementtype

class pynusmv.node.**Scalar**(*values*)

Bases: *pynusmv.node.Type*

The enumeration type.

values

The values of this enumration.

class pynusmv.node.**Modtype**(*name*, *arguments*)

Bases: *pynusmv.node.Type*

A module instantiation type.

name

arguments

class pynusmv.node.**Process**(*left*, *right*, *type_=None*)

Bases: *pynusmv.node.Type*

The process type.

Warning: This type is deprecated.

class pynusmv.node.**Integer**(*left*, *right*, *type_=None*)

Bases: *pynusmv.node.Type*

The integer number type.

Warning: This node type is not supported by NuSMV.

class pynusmv.node.**Real**(*left*, *right*, *type_=None*)

Bases: *pynusmv.node.Type*

The real number type.

Warning: This node type is not supported by NuSMV.

class pynusmv.node.**Wordarray**(*left*, *right*, *type_=None*)

Bases: *pynusmv.node.Type*

The word array type.

Warning: This type is not documented in NuSMV documentation.

class `pynusmv.node.`**`Expression`**(*left*, *right*, *type_=None*)

Bases: *`pynusmv.node.Node`*

A generic expression node.

`in_context`(*context*)

`array`(*index*)

`twodots`(*stop*)

`ifthenelse`(*true*, *false*)

`implies`(*expression*)

`iff`(*expression*)

`or_`(*expression*)

`xor`(*expression*)

`xnor`(*expression*)

`and_`(*expression*)

`not_`()

`equal`(*expression*)

`notequal`(*expression*)

`lt`(*expression*)

`gt`(*expression*)

`le`(*expression*)

`ge`(*expression*)

`union`(*expression*)

`setin`(*expression*)

`in_`(*expression*)

`mod`(*expression*)

`plus`(*expression*)

`minus`(*expression*)

`times`(*expression*)

`divide`(*expression*)

`uminus`()

`next`()

`dot`(*expression*)

`lshift`(*expression*)

`rshift`(*expression*)

`lrotate`(*expression*)

`rrotate`(*expression*)

`bit_selection`(*start*, *stop*)

`concatenation`(*expression*)

`concat`(*expression*)

`castbool`()

`bool`()

`castword1`()

`word1`()

`castsigned`()

`signed`()

`castunsigned`()

`unsigned`()

`extend`(*size*)

`waread`(*expression*)

`read`(*expression*)

`wawrite`(*second*, *third*)

`write`(*second*, *third*)

`uwconst`(*expression*)

`swconst`(*expression*)

`wresize`(*size*)

`resize`(*size*)

`wsizeof`()

`sizeof`()

`casttoint`()

`toint`()

static `from_string`(*expression*)

Parse the string representation of the given expression and return the corresponding node.

Parameters **`expression`** – the string to parse

Return type *`Expression`* subclass

class `pynusmv.node.`**`Leaf`**(*left*, *right*, *type_=None*)

Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Failure`**(*message*, *kind*)

Bases: *`pynusmv.node.Leaf`*

A FAILURE node.

`message`

`kind`

class `pynusmv.node.`**`Falseexp`**

Bases: *`pynusmv.node.Leaf`*

The FALSE expression.

class `pynusmv.node.`**`Trueexp`**

Bases: *`pynusmv.node.Leaf`*

The TRUE expression.

class `pynusmv.node.`**`Self`**

Bases: *`pynusmv.node.Leaf`*

The *self* expression.

class `pynusmv.node.`**`Atom`**(*name*)

Bases: *`pynusmv.node.Leaf`*

An ATOM node.

`name`

class `pynusmv.node.`**`Number`**(*value*)

Bases: *`pynusmv.node.Leaf`*

A node containing an integer.

`value`

class `pynusmv.node.`**`NumberUnsignedWord`**(*value*)

Bases: *`pynusmv.node.Leaf`*

A node containing an unsigned word value.

`value`

class `pynusmv.node.`**`NumberSignedWord`**(*value*)

Bases: *`pynusmv.node.Leaf`*

A node containing a signed word value.

`value`

class `pynusmv.node.`**`NumberFrac`**(*left*, *right*, *type_=None*)

Bases: *`pynusmv.node.Leaf`*

A rational number.

Warning: This node type is not supported by NuSMV.

class `pynusmv.node.`**`NumberReal`**(*left*, *right*, *type_=None*)

Bases: *`pynusmv.node.Leaf`*

A real number.

Warning: This node type is not supported by NuSMV.

class `pynusmv.node.`**`NumberExp`**(*left*, *right*, *type_=None*)

Bases: *`pynusmv.node.Leaf`*

An exponential-formed number.

> **Warning:** This node type is not supported by NuSMV.

class `pynusmv.node.`**`Context`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

A CONTEXT node.

`context`

`expression`

class `pynusmv.node.`**`Array`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

An ARRAY node.

`array`

`index`

class `pynusmv.node.`**`Twodots`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

A range of integers.

`start`

`stop`

class `pynusmv.node.`**`Case`**(*values*)
Bases: *`pynusmv.node.Expression`*

A set of cases.

`values`
The mapping values of this Case expression.

> **Warning:** The returned mapping should not be modified. Modifying the returned mapping will not change the actual NuSMV values of this node.

class `pynusmv.node.`**`Ifthenelse`**(*condition*, *true*, *false*)
Bases: *`pynusmv.node.Expression`*

The *cond ? truebranch : falsebranch* expression.

`condition`

`true`

`false`

class `pynusmv.node.`**`Implies`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Iff`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Or`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Xor`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Xnor`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`And`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Not`**(*expression*)
Bases: *`pynusmv.node.Expression`*

A NOT expression.

`expression`

class `pynusmv.node.`**`Equal`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Notequal`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Lt`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Gt`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Le`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Ge`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Union`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Setin`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Mod`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Plus`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Minus`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Times`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Divide`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Uminus`**(*expression*)
Bases: *`pynusmv.node.Expression`*

A unitary minus expression.

`expression`

class `pynusmv.node.`**`Next`**(*expression*)
Bases: *`pynusmv.node.Expression`*

A NEXT expression.

`expression`

class `pynusmv.node.`**`Dot`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Lshift`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Rshift`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Lrotate`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Rrotate`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`BitSelection`**(*word*, *start*, *stop*)
Bases: *`pynusmv.node.Expression`*

A Bit selection node.

`word`

`start`

`stop`

class `pynusmv.node.`**`Concatenation`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`CastBool`**(*expression*)
Bases: *`pynusmv.node.Expression`*

A boolean casting node.

`expression`

class `pynusmv.node.`**`CastWord1`**(*expression*)
Bases: *`pynusmv.node.Expression`*

A word-1 casting node.

`expression`

class `pynusmv.node.`**`CastSigned`**(*expression*)
Bases: *`pynusmv.node.Expression`*

A signed number casting node.

`expression`

class `pynusmv.node.`**`CastUnsigned`**(*expression*)
Bases: *`pynusmv.node.Expression`*

An unsigned number casting node.

`expression`

class `pynusmv.node.`**`Extend`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Waread`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Wawrite`**(*first*, *second*, *third*)
Bases: *`pynusmv.node.Expression`*

A WAWRITE node.

class `pynusmv.node.`**`Uwconst`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Swconst`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Wresize`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Wsizeof`**(*expression*)
Bases: *`pynusmv.node.Expression`*

A size-of-word node.

`expression`

class `pynusmv.node.`**`CastToint`**(*expression*)
Bases: *`pynusmv.node.Expression`*

An integer casting node.

`expression`

class `pynusmv.node.`**`Count`**(*values*)
Bases: *`pynusmv.node.Expression`*

A set expression.

`values`
The values of this count.

class `pynusmv.node.`**`CustomExpression`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

A generic custom expression.

class `pynusmv.node.`**`Set`**(*values*)
Bases: *`pynusmv.node.CustomExpression`*

A set expression.

`values`
The values of this set.

class `pynusmv.node.`**`Identifier`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.CustomExpression`*

A custom identifier.

static `from_string`(*identifier*)
Return the node representation of identifier.

:param `str` identifier: the string representation of an identifier

class `pynusmv.node.`**`Property`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Expression`*

class `pynusmv.node.`**`Eu`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`Au`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`Ew`**(*left*, *right*, *type_=None*)
Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`Aw`**(*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`Ebu`**(*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`Abu`**(*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`Minu`**(*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`Maxu`**(*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`Ex`**(*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`Ax`**(*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`Ef`**(*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`Af`**(*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`Eg`**(*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`Ag`**(*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`Since`**(*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`Until`**(*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`Triggered`**(*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`Releases`**(*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`Ebf`**(*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`Ebg`**(*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`Abf`**(*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`Abg`**(*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`OpNext`**(*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`OpGlobal`**(*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`OpFuture`** (*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`OpPrec`** (*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`OpNotprecnot`** (*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`OpHistorical`** (*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`OpOnce`** (*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Property`*

class `pynusmv.node.`**`Cons`** (*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Node`*

class `pynusmv.node.`**`Pred`** (*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Node`*

class `pynusmv.node.`**`Attime`** (*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Node`*

class `pynusmv.node.`**`PredsList`** (*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Node`*

class `pynusmv.node.`**`Mirror`** (*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Node`*

class `pynusmv.node.`**`SyntaxError_`** (*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Node`*

class `pynusmv.node.`**`Simpwff`** (*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Node`*

class `pynusmv.node.`**`Nextwff`** (*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Node`*

class `pynusmv.node.`**`Ltlwff`** (*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Node`*

class `pynusmv.node.`**`Ctlwff`** (*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Node`*

class `pynusmv.node.`**`Compwff`** (*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Node`*

class `pynusmv.node.`**`Compid`** (*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Node`*

class `pynusmv.node.`**`Bdd`** (*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Node`*

class `pynusmv.node.`**`Semi`** (*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Node`*

class `pynusmv.node.`**`Eqdef`** (*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Node`*

class `pynusmv.node.`**`Smallinit`** (*left*, *right*, *type_=None*)
    Bases: *`pynusmv.node.Node`*

class `pynusmv.node.`**`Bit`**(*left*, *right*, *type_=None*)
  Bases: *`pynusmv.node.Node`*

class `pynusmv.node.`**`Nfunction`**(*left*, *right*, *type_=None*)
  Bases: *`pynusmv.node.Node`*

class `pynusmv.node.`**`Goto`**(*left*, *right*, *type_=None*)
  Bases: *`pynusmv.node.Node`*

class `pynusmv.node.`**`Constraint`**(*left*, *right*, *type_=None*)
  Bases: *`pynusmv.node.Node`*

class `pynusmv.node.`**`Lambda`**(*left*, *right*, *type_=None*)
  Bases: *`pynusmv.node.Node`*

class `pynusmv.node.`**`Comma`**(*left*, *right*, *type_=None*)
  Bases: *`pynusmv.node.Node`*

class `pynusmv.node.`**`Colon`**(*left*, *right*, *type_=None*)
  Bases: *`pynusmv.node.Node`*

class `pynusmv.node.`**`Declaration`**(*type_*, *section*, *name=None*)
  Bases: *`pynusmv.node.Atom`*

  A Declaration behaves like an atom, except that it knows which type it belongs to. Furthermore, it does not know its name for sure, and cannot be printed without giving it a name.

  `name`
    The name of the declared identifier.

class `pynusmv.node.`**`DVar`**(*type_*, *name=None*)
  Bases: *`pynusmv.node.Declaration`*

  A declared VAR.

class `pynusmv.node.`**`DIVar`**(*type_*, *name=None*)
  Bases: *`pynusmv.node.Declaration`*

  A declared IVAR.

class `pynusmv.node.`**`DFVar`**(*type_*, *name=None*)
  Bases: *`pynusmv.node.Declaration`*

  A declared FROZENVAR.

class `pynusmv.node.`**`DDef`**(*type_*, *name=None*)
  Bases: *`pynusmv.node.Declaration`*

  A declared DEFINE.

class `pynusmv.node.`**`FlatHierarchy`**(*ptr*, *freeit=False*)
  Bases: *`pynusmv.utils.PointerWrapper`*

  Python class for flat hiearchy. The flat hierarchy is a NuSMV model where all the modules instances are reduced to their variables.

  A FlatHierarchy is used to store information obtained after flattening module hierarchy. It stores:

  - the list of TRANS, INIT, INVAR, ASSIGN, SPEC, COMPUTE, LTLSPEC, PSLSPEC, INVARSPEC, JUSTICE, COMPASSION,
  - a full list of variables declared in the hierarchy,
  - a hash table associating variables to their assignments and constraints.

Note: There are a few assumptions about the content stored in this class:

1. All expressions are stored in the same order as in the input file (in module body or module instantiation order).
2. Assigns are stored as a list of pairs (process instance name, assignments in it).
3. Variable list contains only vars declared in this hierarchy.
4. The association var->assignments should be for assignments of this hierarchy only.
5. The association var->constraints (init, trans, invar) should be for constraints of this hierarchy only.

`symbTable`
The symbolic table of the hierarchy.

`variables`
The set of variables declared in this hierarchy.

Warning: The returned variables must not be altered.

`init`
The INIT section of the flat hierarchy.

`invar`
The INVAR section of the flat hierarchy.

`trans`
The TRANS section of the flat hierarchy.

`justice`
The JUSTICE section of the flat hierarchy.

`compassion`
The COMPASSION section of the flat hierarchy.

3.11 `pynusmv.parser` Module

The *`pynusmv.parser`* module provides functions to parse strings and return corresponding ASTs. This module includes three types of functionalities:

- *`parse_simple_expression()`*, *`parse_next_expression()`*, *`parse_identifier()`* and *`parse_ctl_spec()`* are direct access to NuSMV parser, returning wrappers to NuSMV internal data structures representing the language AST.
- `identifier`, `simple_expression`, `constant`, `next_expression`, `type_identifier`, `var_section`, `ivar_section`, `frozenvar_section`, `define_section`, `constants_section`, `assign_constraint`, `init_constraint`, `trans_constraint`, `invar_constraint`, `fairness_constraint`, `justice_constraint`, `compassion_constraint`, `module` and `model` are pyparsing parsers parsing the corresponding elements of a NuSMV model (see NuSMV documentation for more information on these elements of the language).
- *`parseAllString()`* is a helper function to directly return ASTs for strings parsed with pyparsing parsers.

`pynusmv.parser.`**`parse_simple_expression`**`(`*expression*`)`

Parse a simple expression.

Parameters **`expression`** (*`string`*) – the expression to parse

Raise a *`NuSMVParsingError`* if a parsing error occurs

Warning: Returned value is a SWIG wrapper for the NuSMV node_ptr. It is the responsibility of the caller to manage it.

`pynusmv.parser.`**`parse_next_expression`**`(`*expression*`)`

Parse a "next" expression.

Parameters **`expression`** (*`string`*) – the expression to parse

Raise a *`NuSMVParsingError`* if a parsing error occurs

Warning: Returned value is a SWIG wrapper for the NuSMV node_ptr. It is the responsibility of the caller to manage it.

`pynusmv.parser.`**`parse_identifier`**`(`*expression*`)`

Parse an identifier

Parameters **`expression`** (*`string`*) – the identifier to parse

Raise a *`NuSMVParsingError`* if a parsing error occurs

Warning: Returned value is a SWIG wrapper for the NuSMV node_ptr. It is the responsibility of the caller to manage it.

`pynusmv.parser.`**`parse_ctl_spec`**`(`*spec*`)`

Parse a CTL specification

Parameters **`spec`** (*`string`*) – the specification to parse

Raise a *`NuSMVParsingError`* if a parsing error occurs

Warning: Returned value is a SWIG wrapper for the NuSMV node_ptr. It is the responsibility of the caller to manage it.

`pynusmv.parser.`**`parse_ltl_spec`**`(`*spec*`)`

Parse a LTL specification

Parameters **`spec`** (*`string`*) – the specification to parse

Raise a *`NuSMVParsingError`* if a parsing error occurs

Warning: Returned value is a SWIG wrapper for the NuSMV node_ptr. It is the responsibility of the caller to manage it.

`pynusmv.parser.`**`parseAllString`**(*parser*, *string*)

Parse *string* completely with *parser* and set source of the result to *string*. *parser* is assumed to return a one-element list when parsing *string*.

Parameters

- **`parser`** – a pyparsing parser
- **`string`** (`str`) – the string to parse

3.12 `pynusmv.prop` Module

The *`pynusmv.prop`* module contains classes and functions dealing with properties and specifications of models.

`pynusmv.prop.`**`propTypes`** **= {'Invariant': <MagicMock name='mock.prop.Prop_Invar' id='139958097689344'>, 'NoType'**

The possible types of properties. This gives access to NuSMV internal types without dealing with *pynusmv.nusmv* modules.

class `pynusmv.prop.`**`Prop`**(*pointer*, *freeit=False*)

Bases: *`pynusmv.utils.PointerWrapper`*

Python class for properties.

Properties are NuSMV data structures containing specifications but also pointers to models (FSM) and other things.

`type`

The type of this property. It is one element of *`propTypes`*.

`status`

The status of this property. It is one element of `propStatuses`.

`name`

The name of this property, as a string.

`expr`

The expression of this property.

Return type *`Spec`*

`exprcore`

The core expression of this property

Return type *`Spec`*

`bddFsm`

The BDD-encoded FSM of this property.

Return type *`BddFsm`*

`beFsm`

The generic boolean (SEXP) FSM of this property.

Return type *`BeFsm`*

`scalarFsm`

The generic scalar (SEXP) FSM of this property.

Return type *`SexpFsm`*

`booleanFsm`

The generic boolean (SEXP) FSM of this property.

Return type *SexpFsm*

need_rewriting
Check if the given property needs rewriting to be checked.

Returns true iff the property is an invariant that needs to be rewritten

class pynusmv.prop.**PropDb**(*pointer*, *freeit=False*)
Bases: *pynusmv.utils.PointerWrapper*

Python class for property database.

A property database is just a list of properties (*Prop*). Any PropDb can be used as a Python list.

master
The master property of this database.

Return type *Prop*

get_prop_at_index(*index*)
Return the property stored at *index*.

Return type *Prop*

get_size()
Return the number of properties stored in this database.

get_props_of_type(*prop_type*)
Return the list of properties of the given *prop_type* in the database

Parameters **prop_type** – one of the value of *propTypes* used to filter the content of the database

Returns list of properties of the given *prop_type* in the database

class pynusmv.prop.**Spec**(*ptr*, *freeit=False*)
Bases: *pynusmv.utils.PointerWrapper*

A CTL specification.

The Spec class implements a NuSMV nodes-based specification. No check is made to insure that the node is effectively a specification, i.e. the stored pointer is not checked against spec types.

type
The type of this specification.

car
The left child of this specification.

Return type *Spec*

cdr
The right child of this specification.

Return type *Spec*

pynusmv.prop.**true**()
Return a new specification corresponding to *TRUE*.

Return type *Spec*

pynusmv.prop.**false**()
Return a new specification corresponding to *FALSE*.

Return type *Spec*

`pynusmv.prop.`**`not_`**(*spec*)
Return a new specification corresponding to *NOT spec*.

Return type `Spec`

`pynusmv.prop.`**`and_`**(*left*, *right*)
Return a new specification corresponding to *left AND right*.

Return type `Spec`

`pynusmv.prop.`**`or_`**(*left*, *right*)
Return a new specification corresponding to *left OR right*.

Return type `Spec`

`pynusmv.prop.`**`imply`**(*left*, *right*)
Return a new specification corresponding to *left IMPLIES right*.

Return type `Spec`

`pynusmv.prop.`**`iff`**(*left*, *right*)
Return a new specification corresponding to *left IFF right*.

Return type `Spec`

`pynusmv.prop.`**`ex`**(*spec*)
Return a new specification corresponding to *EX spec*.

Return type `Spec`

`pynusmv.prop.`**`eg`**(*spec*)
Return a new specification corresponding to *EG spec*.

Return type `Spec`

`pynusmv.prop.`**`ef`**(*spec*)
Return a new specification corresponding to *EF spec*.

Return type `Spec`

`pynusmv.prop.`**`eu`**(*left*, *right*)
Return a new specification corresponding to *E[left U right]*.

Return type `Spec`

`pynusmv.prop.`**`ew`**(*left*, *right*)
Return a new specification corresponding to *E[left W right]*.

Return type `Spec`

`pynusmv.prop.`**`ax`**(*spec*)
Return a new specification corresponding to *AX spec*.

Return type `Spec`

`pynusmv.prop.`**`ag`**(*spec*)
Return a new specification corresponding to *AG spec*.

Return type `Spec`

`pynusmv.prop.`**`af`**(*spec*)
Return a new specification corresponding to *AF spec*.

Return type `Spec`

`pynusmv.prop.`**`au`**(*left*, *right*)
Return a new specification corresponding to *A[left U right]*.

Return type *Spec*

`pynusmv.prop.`**`aw`**`(`*left*, *right*`)`

Return a new specification corresponding to *A[left W right]*.

Return type *Spec*

`pynusmv.prop.`**`atom`**`(`*strrep*, *type_checking=True*`)`

Return a new specification corresponding to the given atom. *strrep* is parsed and type checked on the current model. A model needs to be read and with variables encoded to be able to type check the atomic proposition. If type_checking is *False*, type checking is not performed and a model is not needed anymore.

Parameters

- **`strrep`** – the string representation of the atom
- **`type_checking`** (`bool`) – whether or not type check the parsed string (default: True)

Return type *Spec*

3.13 `pynusmv.sat` Module

The *`pynusmv.sat`* module contains classes and functions related to the operation and manipulation of the different sat solvers available in PyNuSMV.

class `pynusmv.sat.`**`SatSolverResult`**

Bases: `enum.IntEnum`

This result represents the possible outcomes of a sat solving.

`INTERNAL_ERROR` = 1

`TIMEOUT` = 1

`MEMOUT` = 1

`SATISFIABLE` = 1

`UNSATISFIABLE` = 1

`UNAVAILABLE` = 1

class `pynusmv.sat.`**`Polarity`**

Bases: `enum.IntEnum`

In general, a polarity is assigned to a formula and its variables. If this were not done, this would potentially slow down the solver because the solver would unnecessarily try to assign values to many variables.

`POSITIVE` = 1

`NEGATIVE` = -1

`NOT_SET` = 0

class `pynusmv.sat.`**`SatSolverFactory`**

Bases: `object`

static **`normalize_name`**`(`*name*`)`

Returns a normalized solver name corresponding to the given *name*. (Only case should be changed)

Parameters **`name`** – the name (string) of the solver to normalize.

Raise ValueError whenever the name corresponds to none of the available solvers

static available_solvers()

> **Returns** a list with the name of the solvers that can be instantiated

static print_available_solvers(*stdio_file=<pynusmv.utils.StdioFile object>*)
prints the list of available SAT solvers to the given stdio file

static create(*name='MiniSat'*, *incremental=True*, *proof=True*)
Creates a new sat solver corresponding to the given name and capabilities :param name: the name of the solver to instanciate :param incremental: a flag indicating whether the instanciated solver should have incremental capabilities :param proof: a flag indicating whether the instanciated solver should have proof logging capability. :return: a new sat solver corresponding to the given name and capabilities

> **Raise** Given that ZChaff does not support proof logging, this method raises a ValueError when prooflogging is turned on and zchaff is passed as name parameter.

class pynusmv.sat.**SatSolver**(*ptr*, *freeit=True*)
Bases: *pynusmv.utils.PointerWrapper*

This class encapsulates the capabilities any sat solver should provide

name

> **Returns** the name of the instanciated solver

last_solving_time

> **Returns** the time of the last solving

permanent_group

> **Returns** the permanent group of this class instance

Every solver has one permanent group that can not be destroyed. This group may has more efficient representation and during invocations of any 'solve' functions, the permanent group will always be included into the groups to be solved.

random_mode
Enables or disables random mode for polarity. (useful to perform sat based simulation)

If given seed is != 0, then random polarity mode is enabled with given seed, otherwise random mode is disabled.

> **Parameters** **seed** – a double serving to initialize the PRNG or zero to disable random mode

add(*cnf*)
Adds a CNF formula to the set of CNF to be solved (more specifically to the permanent group of this solver).

The function does not specify the polarity of the formula. This should be done using the polarity function of this solver. In general, if polarity is not set any value can be assigned to the formula and its variables (this may potentially slow down the solver because there is a number of variables whose value can be any and solver will try to assign values to them though it is not necessary). Moreover, some solver (such as ZChaff) can deal with non-redundant clauses only, so the input clauses must be non-redundant: no variable can be in the same clause twice. CNF formula may be a constant.

> **Parameters** **cnf** – a BeCnf representing a boolean expression encoded in CNF

polarity(*be_cnf*, *polarity*, *group=None*)
sets the polarity mode of the solver for the given group and formula

> **Parameters**
>
> - **be_cnf** – a BeCnf formula whose polarity in the group is to be set

- **`polarity`** – the new polarity
- **`group`** – the group on which the polarity applies

`solve`()

Tries to solve all the clauses of the (permanent group of the) solver and returns the flag.

Returns the outcome of the solving (value in SatSolverResult)

`model`

Returns a list of values in dimacs form that satisfy the set of formulas

The previous solving call should have returned SATISFIABLE. The returned list is a list of values in dimac form (positive literal is included as the variable index, negative literal as the negative variable index, if a literal has not been set its value is not included).

Returns a list of values in dimac form that satisfy the set of formulas

class `pynusmv.sat.`**`SatIncSolver`**(*ptr*, *freeit=True*)

Bases: *`pynusmv.sat.SatSolver`*

This class encapsulates the capabilities of an incremental sat solver (ie: manipulate groups)

`create_group`()

Creates a new group and returns its ID

Returns the id of the newly created group

`destroy_group`(*group*)

Destroy an existing group and all formulas in it. :param group: the group to destroy :raise: ValueError if the given group is the solver's permanent group

`move_to_permanent`(*group*)

Moves all formulas from a group into the permanent group of the solver and then destroy the given group. Permanent group may have more efficient implementation, but cannot be destroyed

Parameters **`group`** – the group whose formulas are to be moved to the permanent group.

`add_to_group`(*cnf*, *group*)

Adds a CNF formula to a group

The function does not specify the polarity of the formula. This should be done using the set_group_polarity function of this solver. In general, if polarity is not set any value can be assigned to the formula and its variables (this may potentially slow down the solver because there is a number of variables whose value can be any and solver will try to assign values to them though it is not necessary). Moreover, some solver (such as ZChaff) can deal with non-redundant clauses only, so the input clauses must be non-redundant: no variable can be in the same clause twice. CNF formula may be a constant.

Parameters

- **`cnf`** – a BeCnf representing a boolean expression encoded in CNF
- **`group`** – the solving group to which add the cnf formula

`solve_groups`(*groups*)

Tries to solve formulas from the groups in the list.

Note:

- The permanent group is automatically added to the list.
- the model property may be accessed iff this function returns SatSolverResult.SATISFIABLE

Returns a flag whether the solving was successful.

`solve_without_groups`(*groups*)
Tries to solve formulas in groups belonging to the solver except the groups in the given list *groups_olist*

Note:

- The permanent group may not be in the groups_olist
- the model property may be accessed iff this function returns SatSolverResult.SATISFIABLE

Returns a flag whether the solving was successful.

`solve_all_groups`()
Solves all groups belonging to the solver and returns the flag

Returns the outcome of the solving (value in SatSolverResult)

class `pynusmv.sat.`**`SatIncProofSolver`**(*ptr*, *freeit=True*)
Bases: *`pynusmv.sat.SatIncSolver`*

This type is simply a 'marker' type meant to show that this kind of solver has both incremental sat solving and proof logging capability.

class `pynusmv.sat.`**`SatProofSolver`**(*ptr*, *freeit=True*)
Bases: *`pynusmv.sat.SatSolver`*

This type is simply a 'marker' type meant to show that this kind of solver has proof logging capability.

3.14 `pynusmv.trace` Module

The module *`pynusmv.trace`* defines the classes *`Trace`* and *`TraceStep`* which serve the purpose of representing traces (executions) in a PyNuSMV model.

For instance, these classes are used to represent a counter example in the scope of LTL verification via bounded model checking.

class `pynusmv.trace.`**`TraceType`**
Bases: `enum.IntEnum`

The possible types of traces

`UNSPECIFIED` = **1**

`COUNTER_EXAMPLE` = **1**

`SIMULATION` = **1**

`EXECUTION` = **1**

`END` = **1**

class `pynusmv.trace.`**`Trace`**(*ptr*, *freeit=False*)
Bases: *`pynusmv.utils.PointerWrapper`*, `collections.abc.Iterable`

Encapsulates the details of a counter example trace.

static **`create`**(*description*, *trace_type*, *symb_table*, *symbols_list*, *is_volatile*)
Creates a new (empty trace)

Parameters

- **`description`** – a text describing what the trace is describing
- **`trace_type`** – an enumeration value (*`TraceType`*) describing how the trace should be interpreted
- **`symb_table`** – the symbol table used to associate an human meaningful symbol to the internal representation of the trace
- **`symb_list`** – a NodeList (*`pynusmv.collections.NodeList`*) containing the various symbols that may appear in the trace. Note, this is not a regular python list but you can obtain a NodeList using *NodeList.from_list* if you need to. In case you just use SexpFsm (*`pynusmv.sexp.fsm.SexpFsm.symbols_list()`*) then no conversion is required as it already yields a NodeList
- **`is_volatile`** – a flag indicating whether or not the created insrance should be responsible of the symbol table reference it owns

`concat`(*other*)
Concatenates all the content from *other* to self and destroys other.

Warning: The initial state of *other* is not copied over to self.

Parameters **`other`** – the other trace to append to self

`id`
An unique identifier for this trace (a non-negative number)

Returns an unique identifier for this trace

`description`

Returns this trace description in a human friendly format

`type`
Returns the TraceType (*`TraceType`*) explaining how this trace should be interpreted

Returns the trace type of this trace

`length`
Length for a trace is defined as the number of the transitions in it. Thus, a trace consisting only of an initial state is a 0-length trace. A trace with two states is a 1-length trace and so forth.

Returns the length of this trace

`is_empty`
Tests this trace for emptiness

Returns True iff this trace is empty (that is to say it has length==0)

`is_volatile`
A trace is volatile if it is not the owner of its symbol table reference

Returns a flag indicating whether or not the trace is volatile

`is_registered`

Returns true iff the trace is registered with a trace manager

`register`(*identifier*)
sets the id of the trace (to be called by the trace manager when the trace gets registered in that context)

Parameters `identifier` – an id for this trace

`unregister()`
De-associates this trace from the trace manager it was previously registered with

`is_frozen`
A frozen trace holds explicit information about loopbacks and can not be appended a step, or added a variable value.

Warning: After freezing no automatic looback calculation will be performed: it is up to the owner of the trace to manually add loopback information.

Returns True iff this trace is frozen.

`freeze()`
Forces this trace to enter the frozen state so as to be able to add loopback information on this trace.

A frozen trace holds explicit information about loopbacks. Its length and assignments are immutable, that is it cannot be appended more steps, nor can it accept more values that those already stored in it.

Still it is possible to register/unregister the trace and to change its type or description.

Warning: After freezing no automatic looback calculation is performed: it is up to the owner of the trace to manually add loopback information.

`is_thawed`
A thawed trace holds no explicit information about loopbacks and can be appended a step or added a variable value.

Note: As the name suggests, thawed <-> ! frozen.

Warning: After thawing the trace will not persistently retain any loopback information. In particular it is *illegal* to force a loopback on a thawed trace.

`thaw()`
Forces this trace to enter the thawed state so as to enable the addition of steps or variables.

Note: As the name suggests, thawed <-> ! frozen.

Warning: After thawing the trace will not persistently retain any loopback information. In particular it is *illegal* to force a loopback on a thawed trace.

`equals(`*other*`)`
Two traces are equals iff:

1. They're the same object or None.

2. They have exactly the same language, length, assignments for all variables in all times and the same loopbacks.

   (Defines are not taken into account for equality.)

Note: In order to be considered equal, the two traces need not be both frozen/thawed, and to both have the same registration status. (Of course two traces *cannot* have the same ID).

Warning: This test implements an 'equals logic', not an 'is same' logic since the id field of the trace is not considered in the comparison.

Hence, this equality test is inconsistent with the result of __hash__

`append_step`()

Creates and return a new step which is appended to the current trace

Returns a new trace step which corresponds to the last step of the trace.

`symbol_table`

:return the symbol table associated to this trace

`symbols`

Returns a NodeList (*`pynusmv.collections.NodeList`*) exposing the symbols of the trace language.

Returns a NodeList exposing the symbols of the trace language

`state_vars`

Returns a NodeList (*`pynusmv.collections.NodeList`*) exposing the state variables that exist in the trace language

Returns a NodeList containing the state variables of the trace language

`state_frozen_vars`

Returns a NodeList (*`pynusmv.collections.NodeList`*) exposing the state and frozen variables that exist in the trace language

Returns a NodeList containing the state and frozen variables of the trace language

`input_vars`

Returns a NodeList (*`pynusmv.collections.NodeList`*) exposing the input variables that exist in the trace language

Returns a NodeList containing the input variables of the trace language

`language_contains`(*symbol_node*)

Tests whether the given symbol represented by *symbol_node* (*`pynusmv.node.Node`*) belongs to the trace language.

Note: A more pythonic accessor is foreseen for the same purpose. If you prefer, you may perfectly use *symb_node* in *self* to get the exact same result.

Returns True iff this symbol_node belongs to the trace language.

`is_complete`(*vars_nlist*, *report=False*)

Checks if a Trace is complete on the given set of vars

A Trace is complete iff in every node, all vars are given a value

Note:

- Only input and state section are taken into account. Input vars are not taken into account in the first step. Defines are not taken into account at all.
- If result is false and parameter 'report' is true then a message will be output in nusmv_stderr with some explanation of why the trace is not complete

Parameters **`vars_nlist`** – a NodeList of variable symbols that need to have a value in order for the trace to be considered complete. (*`pynusmv.collections.NodeList`*)

Returns True iff the trace has a value associated to each of the vars in vars_nlist.

`steps`

class `pynusmv.trace.`**`TraceStep`**(*trace*, *step_ptr*)

Bases: `object`

Encapsulates the details of what step is in a trace. In the context of a trace, a step is considered to be a container for incoming input and next state (i.e. it has the form <i, S>)

`is_loopback`

Tests whether the state denoted by this step is a loopback state w.r.t the last state in the parent trace.

This function behaves accordingly to two different modes a trace can be: frozen or thawed(default).

If the trace is frozen, permanent loopback information is used to determine if this step has a loopback state and no further loopback computation is made.

If the trace is thawed, dynamic loopback calculation takes place, using a variant of Rabin-Karp pattern matching algorithm.

Note: No matter the configuration, the last step of a trace is always seen as *NOT* being a loopback step.

`force_loopback`()

Forces this step to be considered as a loopback step using explicit loopback information (trace must be frozen)

Use this function to store explicit loopback information in a frozen trace. The trace will retain loopback data until it is thawed again.

Raises ***`NuSmvIllegalTraceStateError`*** – if the parent trace is not frozen

`assign`(*symbol_node*, *value_node*)

Stores an assignment into a trace step

Warning: Assignments to symbols not in trace language are silently ignored.

Parameters

- **`symbol_node`** – a Node (*`pynusmv.node.Node`*) representing the symbol to which a value is assigned
- **`value_node`** – a Node (*`pynusmv.node.Node`*) representing the value being assigned to the symbol

Returns true iff the assignment worked smoothly.

Raises *NuSmvIllegalTraceStateError* – if the parent trace is frozen

value

3.15 `pynusmv.utils` Module

The *pynusmv.utils* module contains some secondary functions and classes used by PyNuSMV internals.

class `pynusmv.utils.`**`PointerWrapper`**(*pointer*, *freeit=False*)

Bases: `object`

Superclass wrapper for NuSMV pointers.

Every pointer to a NuSMV structure is wrapped in a PointerWrapper or in a subclass of PointerWrapper. Every subclass instance takes a pointer to a NuSMV structure as constructor parameter.

It is the responsibility of PointerWrapper and its subclasses to free the wrapped pointer. Some pointers have to be freed like *bdd_ptr*, but other do not have to be freed since NuSMV takes care of this; for example, *BddFrm_ptr* does not have to be freed. To ensure that a pointer is freed only once, PyNuSMV ensures that any pointer is wrapped by only one PointerWrapper (or subclass of it) if the pointer have to be freed.

`pynusmv.utils.`**`fixpoint`**(*funct*, *start*)

Return the fixpoint of *funct*, as a BDD, starting with *start* BDD.

Return type *BDD*

Note: mu Z.f(Z) least fixpoint is implemented with *fixpoint(funct, false)*. nu Z.f(Z) greatest fixpoint is implemented with *fixpoint(funct, true)*.

`pynusmv.utils.`**`update`**(*old*, *new*)

Update *old* with *new*. *old* is assumed to have the *extend* or *update* method, and *new* is assumed to be a good argument for the corresponding method.

Parameters

- **`old`** – the data to update.
- **`new`** – the date to update with.

class `pynusmv.utils.`**`StdioFile`**(*fname*, *mode*)

Bases: `object`

Wrapper class that provides a context manager to access a FILE* whenever the lower interface needs one. This makes for a more pythonic way to interact with APIs that need a standard file handle without having to deal with the low level open/close instructions. Example:

```
# opens an arbitrary file of your choice.
with StdioFile.for_name('my_output_file', 'w') as f:
    lower_interface_do_something_smart(f)
```

This wrapper also gives you access to stdin, stdout and stderr which, are never closed despite the fact that they may be used with a *with* statement:

```
# stdio is ALREADY open at this time
with StdioFile.stdout() as out:
    lower_interface_do_something_smart(out)
# stdio is STILL open here
```

static for_name (*fname=None*, *mode='w'*)
This function acts like a generic factory that either return a handle for standard file if the name is specified or to stdin or stdout if the name is not specified (it depends on the mode)

Returns a stdiofile for the given name or stdin/stdout if no name is specified depending on the value of the mode

static stdin ()
Standard input

static stdout ()
standard output

static stderr ()
standard error

handle

Returns a FILE* handle to the opened stream

class pynusmv.utils.**writeonly** (*fn*)
Bases: object

writeonly provides a write only decorator for properties that do not have a getter accessor. This makes for pythonic property-lik APIs where your class defines should have defined a setter. Example:

```
class Dummy(PointerWrapper):
  # .. code elided ...

  @writeonly
  def config_tweak(self, new_value_of_tweak):
    lower_interface_set_tweak(self._ptr, new_value_of_tweak)
```

Can be used the following way:

```
d = Dummy()
# this is now perfectly OK
d.config_tweak = 42

# this will however fail since no getter was defined
d.config_tweak
```

class pynusmv.utils.**indexed** (*target*, *fget=None*, *fset=None*, *fdel=None*)
Bases: object

indexed provides a set of decorators that enable the use of 'pythonic' indexed get/setters. These give you the possibility to automagically add syntax sugar to the classes you write.

The easiest (and most flexible way) to get started with the indexed series of decorator is to use *@indexed.property(<name>)*. But if you are after something more limited, you might want to give a look to the other decorators that are provided: namely, *@indexed.getter*, *@indexed.setter* and *@indexed.deleter*.

static getter (*fn*)
Wraps a function *fn* and turns it into pythonic indexed-like acessor.

Parameters **fn** – the function to use to perform the keyed-lookup

Example usage:

```
class GetterOnly:
    # ... code elided ...

    # using @indexed or @indexed.getter is perfectly equivalent although
    # the use of @indexed.getter is considered slightly cleaner
    @indexed.getter
    def clause(self, index):
      return lower_interface_get_clause_at(self._ptr, index)

# example of use:
g = GetterOnly()
g.clause[42] # returns the 42th clause
```

static `setter` (*fn*)
wraps a function *fn* and turns it into pythonic indexed-like acessor.

Parameters **`fn`** – the function to use to perform the keyed-assignment

Example usage:

```
class SetterOnly:
    # ... code elided ...

    @indexed.setter
    def clause(self, index, new_value):
      lower_interface_set_clause_at(self._ptr, index, new_value)

# example of use:
s = GetterOnly()
s.clause[42] = another_clause # changes the value of the clause
```

static `deleter` (*fn*)
wraps a function *fn* and turns it into pythonic indexed-like deleter.

Parameters **`fn`** – the function to use to perform the keyed-lookup

Example usage:

```
class DeleterOnly:
    # ... code elided ...

    @indexed.deleter
    def clause(self, index, new_value):
      return lower_interface_delete_clause_at(self._ptr, index)

# example of use:
d = DeleterOnly()
del d.clause[42] # 42th clause has been deleted
```

static `property` (*name*, ***kwargs*)
Wraps the constructor of the decorated class to add a virtual indexed property called *name*

By **default**, the generated indexed getter, indexed setter and indexed deleted are assumed to be called respectively:

- get_'name'
- set_'name'
- del_'name'

However, these names are not enforced and can be customized if you pass the keywords fget=<the_name_of_your_getter_fn>, fset=<the_name_of_your_setter_fn> and/or fdel=<the_name_of_your_deleter_fn>.

Note: The keyword parameters also let you provide a docstring for the virtual property you define. To this end, simply use the *doc* keyword.

Warning: The getter, setter and deleter functions MUST BE CALLABLE objects ! This means, you MAY NOT decorate any of the functions you intend to use in your virtual property with any of the @property, @indexed.getter, @indexed.setter or @indexed.deleter since you resulting property would simply not work.

Simple example:

```
@indexed.property('smartlst')
class Cls:
    def __init__(self):
        self._lst   = [4,5,6]

    def get_smartlst(self, idx):
        return self._lst[idx]

    def set_smartlst(self, idx, v):
        self._lst[idx] = v

    def del_smartlst(self, idx):
        del self._lst[idx]

# Usage:
c = Cls()
c.smartlst[1]      # calls _get_smartlst and returns 5
c.smartlst[1]=42   # calls _set_smartlst and changes _slt to be [4,42,6]
del c.smartlst[1] # calls _del_smartlst and changes _slt to be [4, 6]
```

Example with custom property names:

```
@indexed.property('smartlst', fget='glst', fset='slst', fdel='dlst')
class Cls:
    def __init__(self):
        self._lst   = [4,5,6]

    def glst(self, idx):
        return self._lst[idx]

    def slst(self, idx, v):
        self._lst[idx] = v

    def dlst(self, idx):
        del self._lst[idx]

# Usage:
c = Cls()
c.smartlst[1]      # calls glst and returns 5
c.smartlst[1]=42   # calls slst and changes _slt to be [4,42,6]
```

```
del c.smartlst[1] # calls dlst and changes _slt to be [4, 6]
```

If you don't like to use the decorator 'magic' but still want to define a virtual property with very little effort: you should then use the indexed constructor itself as such:

```
class Dummy:
    def __init__(self):
        self.clause= indexed(self, fget=self.get_clause, fset=self.set_clause,
↪fdel=self.del_clause)

    @indexed.getter
    def get_clause(self, clause_idx):
        return lower_interface_get_clause_at(self._ptr, index)

    @indexed.setter
    def set_clause(self, clause_idx, value):
        lower_interface_set_clause_at(self._ptr, index, new_value)

    @indexed.deleter
    def del_clause(self, clause_idx):
        lower_interface_delete_clause_at(self._ptr, index)

# example of use:
d = Dummy()
d.clause[42]                 # returns the 42th clause
d.clause[42] = other_clause  # updates the 42th clause
del d.clause[42]             # drops the 42th clause
```

3.16 `pynusmv.wff` Module

This module defines the `Wff` which encapsulates the notion of well formed formula as specified per the input language of NuSMV. It is particularly useful in the scope of BMC as it

class `pynusmv.wff.`**`Wff`**(*ptr*, *freeit=False*)

Bases: `pynusmv.utils.PointerWrapper`

The `Wff` (Well Formed [Temporal] Formula) encapsulates the structure of a specification in various supported logics as of the NuSMV input language

static **`decorate`**(*node_like*)

Creates a new instance from a node_like object (`pynusmv.node`).

Parameters **`node_like`** – an object which behaves as a node (typically subclass) which will be wrapped to be considered as a Wff

Returns *node_like* but wrapped in a Wff overlay.

static **`true`**()

static **`false`**()

`depth`

Returns the modal depth of the given formula]

Returns 0 for propositional formulae, 1 or more for temporal formulae

Raises **`NuSMVWffError`** – when this wff is not a valid LTL formula

`to_negation_normal_form`()

`to_node`()

`to_boolean_wff`(*bdd_enc=None*)
Converts this scalar expression to a boolean equivalent

Note: Uses the determinizing layer and can therefore introduce new determination variable.

`to_be`(*be_enc*)
Converts this WFF to the BE format.

Warning: This *DOES NOT WORK FOR TEMPORAL FORMULAS*, if you pass one, NuSMV will complain and crash with an error message on stderr. In order to generate the BMC problem for this particular formula, (if it is a temporal one) you should instead, use a the *bounded semantic* of your choice to that end (LTL semantic is already defined).

Parameters **`be_enc`** – the boolean expression encoder that will serve to back the conversion process.

Returns a BE (rbc) representation of this formula.

`not_`()

`and_`(*other*)

`or_`(*other*)

`implies`(*other*)

`iff`(*other*)

`next_`()

`next_times`(*x*)

`opnext`()

`opprec`()

`opnotprecnot`()

`globally`()

`historically`()

`eventually`()

`once`()

`until`(*other*)

`since`(*other*)

`releases`(*other*)

`triggered`(*other*)

3.17 `pynusmv.be.__init__` Module

The `pynusmv.be` module regroups the modules related to the treatment of boolean expressions (BE) in pynusmv. In particular, it provides an access to:

- *`expression`* containing classes and functions related to the BE themselves.
- *`fsm`* containing classes and functions to represent the model FSM encoded in terms of boolean variables only.
- *`encoder`* containg classes to represent boolean variables and the way to encode them so as to represent a *timeline*, a path in the fsm.
- *`manager`* which provides an access to lower level functions and classes related to the management and physical representation of the boolean expressions.

3.18 `pynusmv.be.encoder` Module

The *`pynusmv.be.encoder`* module provides the BE encoder related functionalities

- *`BeWrongVarType`* a kind of exception thrown when the type of a variable does not correspond to what the specification expects.
- *`BeVarType`* an enum describing the possible types of variables (These can be combined)
- *`BeEnc`* which provides the encoder related functionalities (i.e: shifts)

Note: Most of the documentation comes either from the NuSMV source code BeEnc.c or the NuSMV-2.5 User Manual

url http://nusmv.fbk.eu/NuSMV/userman/v25/nusmv.pdf

exception `pynusmv.be.encoder.`**`BeWrongVarType`**`(`*msg*`)`
Bases: `Exception`

Thrown whenever an error happens because the type of variable is not the expected one.

class `pynusmv.be.encoder.`**`BeVarType`**
Bases: `enum.IntEnum`

Used to classify a be variable within 4 main categories:

- current state variables
- frozen variables
- input variables
- next state variables

These values can be combined when for example the iteration is performed. In this way it is possible to iterate through the set of current and next state vars, skipping the inputs.

`CURR` **= 1**

`FROZEN` **= 1**

`INPUT` **= 1**

`NEXT` **= 1**

`ALL` **= 1**

ERROR = 1

class `pynusmv.be.encoder.`**`BeEnc`**(*ptr*, *freeit=False*)

Bases: *`pynusmv.utils.PointerWrapper`*

Pythonic wrapper for the BE encoder class of NuSMV.

Note: The timed and untimed variable notions are used a lot in the context of this classs. It is therefore worthwile to explain what these notions encompass.

An *untimed* variable, is an 'abstract' variable that does not belong to any time block. It can therefore be seen as a prototypical variable meant to help you retrieve the variable which is actually used to model this 'variable' at some point of time in a computation. That second type of variable (which are associated to a time block) are therefore called *timed* variables.

---

static **`global_singleton_instance`**()

Currently, in NuSMV, the be_enc is a singleton global private variable which is shared between all the BE FSMs.

Returns the global singleton be_encoder instance

Raises **`NuSMVBeEncNotInitializedError`** – if the global encoding is not initialized

`symbol_table`

Returns the symbol table used by this encoder

`manager`

Returns the Boolean Expressions manager (`BeManager`) contained into the variable manager, to be used by any operation on BEs

`max_time`

Returns the maximum allocated time

`num_of_vars`

Returns number of input and state variables currently handled by the encoder

`num_of_state_vars`

Returns the number of state variables in the encoded model.

A state of the model is an assignment of values to a set of state and frozen variables. State variables (and also instances of modules) are declared by the notation:

var_declaration :: VAR var_list

Returns number of state variables currently handled by the encoder

`num_of_frozen_vars`

Returns the number of frozen variables in the encoded model.

FROZENVAR s (frozen variables) are variables that retain their initial value throughout the evo- lution of the state machine; this initial value can be constrained in the same ways as for normal state variables. Similar to input variables the difference between the syntax for the frozen and state variables declarations is the keyword indicating the beginning of a declaration:

frozenvar_declaration :: FROZENVAR simple_var_list

The semantics of some frozen variable a is that of a state variable accompanied by an assignment that keeps its value constant (it is handled more efficiently, though):

ASSIGN next(a) := a;

Returns number of frozen variables currently handled by the encoder

num_of_input_vars
Returns the number of input variables in the encoded model.

IVAR s (input variables) are used to label transitions of the Finite State Machine. The difference between the syntax for the input and state variables declarations is the keyword indicating the beginning of a declaration:

ivar_declaration :: IVAR simple_var_list

Returns number of input variables currently handled by the encoder

iterator (*var_type=<BeVarType.CURR: 1>*, *randomized=False*, *rnd_offset=1*)

Returns an iterator to walk through all the untimed variables

Parameters

- **var_type** – the kind of variables to be taken into account while iterating
- **randomized** – a flag indicating whether or not the variables should be walked in a random order.
- **rnd_offset** – the random offset to use when iterating in random order

at_index

by_name

by_expr

untimed_variables
Returns the list of all the untimed variable

Returns the list of (all) untimed variables

curr_variables
Returns the list of all the (untimed) current state variables

Returns the list of the current state untimed variables

frozen_variables
Returns the list of the frozen variables.

FROZENVAR s (frozen variables) are variables that retain their initial value throughout the evo- lution of the state machine; this initial value can be constrained in the same ways as for normal state variables. Similar to input variables the difference between the syntax for the frozen and state variables declarations is the keyword indicating the beginning of a declaration:

frozenvar_declaration :: FROZENVAR simple_var_list

The semantics of some frozen variable a is that of a state variable accompanied by an assignment that keeps its value constant (it is handled more efficiently, though):

ASSIGN next(a) := a;

Returns the list of the frozen variables

input_variables

Returns the list of the (untimed) input variables

IVAR s (input variables) are used to label transitions of the Finite State Machine. The difference between the syntax for the input and state variables declarations is the keyword indicating the beginning of a declaration:

ivar_declaration :: IVAR simple_var_list

Returns the list of the (untimed) input variables

next_variables

Returns the list of the (untimed) next state variables

shift_curr_to_next (*expr*)

Returns an *untimed* Be expression corresponding to *expr* in which all variables v have been shifted to next(v). Example:

v == True & w == False becomes next(v) == True & next(w) == False

Note: Despite the fact that this operation performs a shift of the variables it remains in the *untimed* block. (next of untimed vars are also untimed vars). Hence the returned expression is an *untimed* expression. Therefore, in order to use it in (ie a transition relation unrolling), it must be shifted again to a time block using one of :

- *shift_to_time()*
- *shift_to_times()*
- *or_interval()*
- *and_interval()*

Warning: argument 'expr' must contain only untimed current state variables and untimed frozen variables, otherwise results will be unpredictable. Unfortunately, there is no way to preemptively check that a given expression contains only untimed variable so it is up to the programmer to make sure he calls this method in an appropriate way.

Parameters `expr` – the expression to shift

Returns an expression equivalent to expr but with the variables shifted to the next-state portion of the block.

shift_to_time (*expr*, *time*)

Returns a *timed* Be expression corresponding to *expr* in which all variables v have been shifted to the given *time* block. Natually, the variables of the *next* sub-block are shifted to time t+1 (which corresponds to what one would intuitively expect).

Warning: argument 'expr' must contain only untimed current state variables and untimed frozen variables, otherwise results will be unpredictable. Unfortunately, there is no way to preemptively check that a given expression contains only untimed variable so it is up to the programmer to make sure he calls this method in an appropriate way.

Parameters

- **`expr`** – the expression to shift
- **`time`** – the time to shift the expression to

Returns an expression equivalent to *expr* but with the variables shifted to the given *time* block.

`shift_to_times`(*expr*, *curr_time*, *frozen_time*, *ivar_time*, *next_time*)

Returns a *timed* Be expression corresponding to *expr* in which:

- all the current state variables are shifted to time *curr_time*
- all the frozen variables are shifted to time *frozen_time*
- all the input variables are shifted to time *ivar_time*
- all the next state variables are shifted to time *next_time*

Warning: argument 'expr' must contain only untimed current state variables and untimed frozen variables, otherwise results will be unpredictable. Unfortunately, there is no way to preemptively check that a given expression contains only untimed variable so it is up to the programmer to make sure he calls this method in an appropriate way.

Parameters

- **`expr`** – the expression to shift
- **`curr_time`** – the time to shift the current variables to
- **`frozen_time`** – the time to shift the frozen variables to
- **`ivar_time`** – the time to shift the input variables to
- **`next_time`** – the time to shift the next state variables to.

Returns an expression equivalent to *expr* but with the sets of variables shifted to the time blocks.

`and_interval`(*expr*, *start*, *end*)

This method is an utility meant to let you easily compute the conjunction of the given *expr* shifted at all the times in the interval [start, end]. Mathematically, this corresponds to the following formula:

$$\bigwedge_{t=start}^{t=end} shift\_to\_time(expr, t)$$

`or_interval`(*expr*, *start*, *end*)

This method is an utility meant to let you easily compute the disjunction of the given *expr* shifted at all the times in the interval [start, end]. Mathematically, this corresponds to the following formula:

$$\bigvee_{t=start}^{t=end} shift\_to\_time(expr, t)$$

`encode_to_bits`(*name_node*)

Returns the list of bits variable names used to encode the SMV variable denoted by *name_node* in the boolean model.

Parameters **`name_node`** – the node symbol representing the expression to break down to bits.

Returns the list of bits names (in the form of nodes) which are used to encode *name_node*.

decode_value (*list_of_bits_and_value*)

Returns a node (`pynusmv.node.Node`) corresponding to the value of the variable encoded by the list of bits and values.

Parameters **list_of_bits_and_value** – a sequence of tuples (BeVar, BooleanValue) which represent a bit and its value.

Returns an intelligible value node corresponding to what these bits means when interpreted in the context of the SMV model.

decode_sat_model (*sat_model*)

Decodes the given *sat_model* and translates it in a sequence of valuations. Concretely, the returned value is a multi-level dictionary with the following structure: time_block -> scalar_name -> decoded_value

Parameters **sat_model** – the dimacs model generated by a sat solver to satisfy some given property.

Returns a multi-level map time_block -> scalar_name -> decoded_value representing the content of the sat_model

3.19 `pynusmv.be.expression` Module

The `pynusmv.be.expression` module contains classes and functions related to the operation and manipulation of the boolean expressions in PyNuSMV.

In particular it contains:

- `Be`
- `BeCnf`
- other utility functions

class `pynusmv.be.expression.`**Be** (*ptr*, *be_manager*)

Bases: `object`

This is the interface of the boolean expression type. For obvious reasons, the function names have been kept as close to its BDD counterpart. The 'dsl' has also been kept. Hence:

- `a + b` and `a | b` compute the disjunction of `a` and `b`
- `a * b` and `a & b` compute the conjunction of `a` and `b`
- `~a` and `-a` compute the negation of `a`
- `a - b` computes `a & ~b`
- `a ^ b` computes the exclusive-OR (XOR) of `a` and `b`

However, no comparison operators such as (lt, ge, ...) are provided.

Note: For the sake of compactness and efficient memory use, the boolean expressions (Be) are encoded in the form of reduced boolean circuits (directed acyclic graphs) because this representation is up to exponentially smaller than the classic tree representation. See slides 9-10 and 19 of http://disi.unitn.it/~rseba/DIDATTICA/fm2016/03_TEMPORAL_LOGICS_SLIDES.pdf to get more information about this.

However, the details of the reduced boolean circuit are (so far) not accessible throught PyNuSMV since it was considered as a too low level concern.

static true(*be_manager*) → pynusmv.be.expression.Be
Creates a constant with the meaning 'True'

Parameters **be_manager** (BeManager) – the manager responsible for this expression

Returns a constant boolean expression meaning True

static false(*be_manager*) → pynusmv.be.expression.Be
Creates a constant with the meaning 'False'

Parameters **be_manager** (BeManager) – the manager responsible for this expression

Returns a constant boolean expression meaning False

static if_then_else(*be_manager*, *_if*, *_then*, *_else*) → pynusmv.be.expression.Be
Creates an if then else operation

Parameters

- **be_manager** (BeManager) – the manager responsible for this expression
- **_if** (Be) – the 'if' condition expression of the if then else
- **_then** (Be) – the 'then' part of the expression
- **_else** (Be) – the alternative

Returns a constant boolean expression meaning True

is_true() → bool
Returns true iff the expression can be evaluated to True

Returns true iff the expression can be evaluated to True

is_false() → bool
Returns true iff the expression can be evaluated to False

Returns true iff the expression can be evaluated to False

is_constant() → bool
Returns true if the given be is a constant value, such as either False or True

Returns true if the given be is a constant value, such as either False or True

not_() → pynusmv.be.expression.Be
Negates the current expression

Returns an expression (Be) corresponding to the negation of *self*

and_(*other*) → pynusmv.be.expression.Be
Returns an expression (Be) corresponding to the conjunction of *self* and *other*

Parameters **other** – a Be that will be conjuncted with self.

Returns Returns an expression (Be) corresponding to the conjunction of *self* and *other*

or_(*other*) → pynusmv.be.expression.Be
Returns an expression (Be) corresponding to the disjunction of *self* and *other*

Parameters **other** – a Be that will be disjuncted with self.

Returns Returns an expression (Be) corresponding to the disjunction of *self* and *other*

xor(*other*) → pynusmv.be.expression.Be
Returns an expression (Be) corresponding (*self* exclusive or *other*)

Parameters **other** – a Be that will be xor'ed with self.

Returns Returns an expression (Be) corresponding (*self* exclusive or *other*)

imply(*other*) → pynusmv.be.expression.Be
Returns an expression (Be) corresponding $(self \implies other)$. That is to say: $(\neg self \lor other)$

Parameters **other** – a Be that will have to be implied by *self*

Returns Returns an expression (Be) corresponding $(self \implies other)$

iff(*other*) → pynusmv.be.expression.Be
Returns an expression (Be) corresponding $(self \iff other)$. That is to say: $(self \implies other) \land (other \implies self)$

Parameters **other** – a Be that will have to be equivalent to *self*

Returns Returns an expression (Be) corresponding $(self \iff other)$

inline(*add_conj*)
Performs the inlining of this expression, either including or not the conjuction set.

If add_conj is true, the conjuction set is included, otherwise only the inlined formula is returned for a lazy SAT solving.

Parameters **conj** – a flag to tell whether or not to include the conjunction set

Returns a copy of this expression with the inlining performed.

to_cnf(*polarity=<Polarity.POSITIVE: 1>*)
Converts this boolean expression to a corresponding CNF

Note: By default, the POSITIVE polarity is used since it corresponds to what one would intuitively imagine when converting a formula to CNF.

'polarity' is used to determine if the clauses generated should represent the RBC positively, negatively, or both (1, -1 or 0 respectively). For an RBC that is known to be true, the clauses that represent it being false are not needed (they would be removed anyway by propogating the unit literal which states that the RBC is true). Similarly for when the RBC is known to be false. This parameter is only used with the compact cnf conversion algorithm, and is ignored if the simple algorithm is used.

Parameters **polarity** (*integer*) – the polarity of the expression

Returns a CNF equivalent of this boolean expression

class pynusmv.be.expression.**BeCnf**(*ptr*, *be_manager*, *freeit=False*)
Bases: object

This class implements the CNF representation of a boolean expression

original_problem

Returns the original BE problem this CNF was created from

vars_list
Returns the list of independent variables in the CNF representation.

:return:the independent variables list in the CNF representation

clauses_list

Returns a list of lists which contains the CNF-ed formula]

Each list in the list is a set of integers which represents a single clause. Any integer value depends on the variable name and the time which the variable is considered in, whereas the integer sign is the variable polarity in the CNF-ed representation.

vars_number

Returns Returns the number of independent variables in the given BeCnf structure

clauses_number

Returns the number of clauses in the given Be_Cnf structure

max_var_index

Returns the maximum variable index in the list of clauses

formula_literal

Returns the literal assigned to the whole formula. It may be negative. If the formula is a constant unspecified value is returned

remove_duplicates()

Removes any duplicate literal appearing in single clauses

print_stats(*file*, *prefix*)

Prints some statistics

Print out, in this order: the clause number, the var number, the highest variable index, the average clause size, the highest clause size

Parameters

- **file** – the output StdioFile where to write the stats
- **prefix** – the prefix to prepend the output lines with

3.20 `pynusmv.be.fsm` Module

The *pynusmv.be.fsm* module contains classes and functions related to PyNuSMV's description of a BE encoded FSM

In particular it contains: *BeFsm* which is the sole implementation of a BE FSM

class pynusmv.be.fsm.**BeFsm**(*ptr*, *freeit=False*)

Bases: *pynusmv.utils.PointerWrapper*

This class wraps the public interface of a BE FSM as defined in NuSMV with the BeFsm_ptr type.

Its role is to give an access to the properties of FSM as represented with boolean expressions (BE). For instance, it gives access to the initial states the invariants and the transition relation encoded in terms of BE. Moreover it gives the possibility to compute the synchronous product of two FSMs.

static global_master_instance()

Returns the boolean FSM in BE stored in the master prop.

Raises ***NuSMVBeFsmMasterInstanceNotInitializedError*** – when the global BE FSM is null in the global properties database (ie when coi is enabled).

static create(*enc*, *init*, *invar*, *trans*, *fairness*, *freeit=False*)

Creates a new BeFsm instance using the given encoder, initial states, invariants, transition relation and list of fairness.

Parameters

- **enc** – the encoder used to represent the variables of the model.
- **init** – the boolean expression representing the initial states of the FSM

- **invar** – the boolean expression representing the invariants of the model encoded in this FSM
- **fairness** – a list of boolean expression representing the fairness constraints of the model.
- **freeit** – a flag indicating whether or not the system resources should be freed upon garbage collection.

Pqram trans the boolean expression representing the transition relation of the model encoded in this FSM

Returns a new instance of BeFsm that gets its init, invar, transition relation and the list of fairness in Boolean Expression format

static **create_from_sexp**(*enc*, *sexp_fsm*)
Creates a new instance of BeFsm extracting the necessary information from the given *sexp_fsm* of type `BoolSexpFsm`

Parameters

- **enc** – the encoder used to represent the variables of the model.
- **sexp_fsm** – the BoolSexpFsm which contains the automaton information

copy(*freeit=True*)
Creates a new independent copy of the FSM. :param freeit: a flag indicating whether or not the system resources should be freed upon garbage collection.

encoding
The BE encoding of this FSM

init
The BE representing the initial states of this FSM

invariants
The boolean expression representing the invariants of this FSM

trans
The boolean expression representing the transition relation of the FSM

Note: Transition expression shifted at time zero is what brings you to state one. Hence:

```
shift_to_time(init, 0) & shift_to_time(trans, 0) == STATE_1
```

fairness_list
The list of fairness constraints of this model encoded in BE format.

Note: accessing this property is not free: use fairness_iterator instead if you don't need to manipulate the list as a list.

fairness_iterator()

Returns an iterator to iterate over the fairness list

apply_synchronous_product(*other*)
Apply the synchronous product between self and other modifying self. :param other: the other to compute the synchronous product with

3.21 `pynusmv.be.manager` Module

The *`pynusmv.be.manager`* module contains classes and functions related to the management of boolean expressions in PyNuSMV

In particular it contains:

- *`BeManager`* which is the abstract interface of a manager
- *`BeRbcManager`* which is the sole implementation of the BE manager

class `pynusmv.be.manager.`**`BeManager`**(*ptr*, *freeit=False*)

Bases: *`pynusmv.utils.PointerWrapper`*

The manager is the data structure that serves to physically store the variables used to construct boolean expressions. As such, it is seen as rather low level of abstraction meant to offer services to an higher level encoding of the variables. As a consequence, the objects manipulated through its interface are fairly low level as well (integers instead of variables). If you are not implementing a new boolean encoding, you will probably want to focus and use the boolean encoding (*`pynusmv.be.encoder.BeEnc`*) which offers roughly the same services but with an higher level interface.

Note: whenever an 'index' is mention, it is used to denote the canonical identifier that permits to establish a correspondence between a be literal and a cnf literal.

Warning: Subclassing this class imposes to implement _free

See also:

`pynusmv.be.encoder.BeEnc`

`be_index_to_var`(*index*)

Retrieves the BE variable (expression) corresponding to the given index (index may be retrieved from the literals managed by this manager)

Parameters **`index`** – the index

Returns the be corresponding to this index

`be_var_to_index`(*expression*)

Returns the BE index which corresponding to the given expression which can later be used to identify the BE literal or the CNF literal corres- ponding to this expression.

Note: This is the function you need to call in order to gain access to the BE or CNF literals (managed by this manager) corresponding to the given expression.

Exemple:

```
# assuming that variable a was declared in the SMV model and
# converted to cnf
idx = self.mgr.be_var_to_index(a)
blit= self.mgr.be_index_to_literal(idx)
clit= self.mgr.be_literal_to_cnf_literal(blit)
```

Parameters **`expression`** – the expression whose index needs to be found

Returns the BE index of the expression

be_literal_to_index(*literal*)
Retrieves the BE index corresponding to the given BE literal. That BE index can later be used to identify the corresponding CNF literal.

Parameters **literal** – the literal (may not be zero)

Returns converts a BE literal to its index

Raise ValueError if *literal* is zero

be_index_to_literal(*index*)
Retrieves the BE literal stored at the given index.

Parameters **index** – the index (may not be smaller than zero)

Returns Converts a BE index into a BE literal (always positive)

Raise ValueError if the given *index* is < 0

be_index_to_cnf_literal(*index*)
Retrieves the CNF literal corresponding to the given index.

Parameters **index** – the index

Returns Returns a CNF literal (always positive) associated with a given index

be_literal_to_cnf_literal(*literal*)
Converts a BE literal into a CNF literal (sign is taken into account)

Parameters **literal** (`integer`) – the be literal

Returns the CNF literal corresponding to the BE literal `literal`

cnf_to_be_model(*slist*)
Converts the given CNF model (dimacs obtained from *solver.model* into an equivalent model.

Parameters **slist** – the cnf model in the form of a slist (as is the case from *solver.model*).

Returns Converts the given CNF model into BE model

cnf_literal_to_be_literal(*literal*)
Converts a CNF literal into a BE literal

The function returns 0 if there is no BE index associated with the given CNF index. A given CNF literal should be created by given BE manager

Parameters **literal** (`integer`) – the cnf literal (may not be zero)

Returns the BE literal corresponding to the cnf literal `literal`

Raise ValueError if *literal* is zero

cnf_literal_to_index(*literal*)
Retrieves the index corresponding to the given CNF literal. That index can later be used to identify the corresponding BE literal.

Parameters **literal** – the literal (may not be zero)

Returns converts a CNF literal to its corresponding index

Raise ValueError if *literal* is zero

dump_davinci(*be*, *file*)
Dumps the BE to the given *file* in davinci format

Parameters

- **`be`** – the boolean expression
- **`file`** – the output StdioFile used to dump the information

`dump_gdl` (*be*, *file*)
Dumps the BE to the given *file* in gdl format

Parameters

- **`be`** – the boolean expression
- **`file`** – the output StdioFile used to dump the information

`dump_sexpr` (*be*, *file*)
Dumps the BE to the given *file*

Parameters

- **`be`** – the boolean expression
- **`file`** – the output StdioFile used to dump the information

class `pynusmv.be.manager.`**`BeRbcManager`** (*ptr*, *freeit=False*)
Bases: *`pynusmv.be.manager.BeManager`*

This is (at the time being) the sole implementation of the BeManager. It uses RBC as the underlying format to represent the boolean expressions but these are (so far) only exposed as an opaque pointer.

Note: RBC stands for Reduced Boolean Circuit which is used to encode (rewrite and shorten) boolean expressions in the form of an directed acyclic graph.

This form of representation is currently not available to PyNuSMV.

static **`with_capacity`** (*capacity: int*, *freeit=True*) → pynusmv.be.manager.BeRbcManager
Creates a BeRbcManager with the capacity to store ‘capacity’ variables.

Args:

param capacity the variable capacity of this rbc manager

type capacity integer

Returns: A fresh instance of `BeRbcManager`

`reserve` (*size*)
Changes the maximum number of variables the rbc manager can handle.

Args:

param size the new maximum number of variables that can be handled by this manager.

type size integer

`reset` ()
Resets the RBC cache

3.22 `pynusmv.bmc.__init__` Module

The `pynusmv.bmc` module regroups the modules related to the bounded model checking features of pynusmv. In particular, it provides an access to the following modules:

- `glob` which provides initialisation facilities for the bmc sub system and global structures related to BMC in NuSMV.
- `utils` which provides a set of utility functions convenient when implementing a SAT based bounded model checker.
- `ltlspec` which provides functions relative to the implementation of BMC verification for linear time logic (see [BCC+03] for further information about BMC).
- `invarspec` which provides a set of features relative to *temporal induction* using sat solvers which is a technique conceptually close to BMC. (For the full details, check [ES03]).

3.22.1 References

3.23 `pynusmv.bmc.glob` Module

The module `pynusmv.bmc.glob` serves as a reference entry point for the bmc-related functions (commands) and global objects.

It has explicitly not been integrated with `pynusmv.glob` in order to keep a clear distinction between the functions that are BDD and BMC related. It however depends on some of the functions of that module and these could be merged (or at least grouped under a common package) in the future.

pynusmv.bmc.glob.**bmc_setup**(*force=False*)

Initializes the bmc sub-system, and builds the model in a Boolean Expression format. This function must be called before the use of any other bmc-related functionalities. Only one call per session is required.

If you don't intend to do anything special, you might consider using *go_bmc* which is a shortcut for the whole bmc initialization process.

Note: This function is subject to the following requirements:

- a model must be loaded (`pynusmv.glob.load()`)
- hierarchy must already be flattened (`pynusmv.glob.flatten_hierarchy()`)
- encoding must be already built (`pynusmv.glob.encode_variables()`)
- **boolean model must be already built (`pynusmv.glob.build_boolean_model()`)** except if cone of influence is enabled and force is false

Parameters **force** – a flag telling whether or not the boolean model must exist despite the cone of influence being enabled

Raises ***NuSMVNeedBooleanModelError*** – if the boolean model wasn't created

pynusmv.bmc.glob.**bmc_exit**()

Releases all resources associated to the bmc model manager. If you want to do bmc again after calling this, you will have to call `bmc_setup()` or `go_bmc()` again.

pynusmv.bmc.glob.**build_master_be_fsm**()

Creates the BE fsm from the Sexpr FSM. Currently the be_enc is a singleton global private variable which is shared between all the BE FSMs. If not previoulsy committed (because a boolean encoder was not available at the time due to the use of coi) the determinization layer will be committed to the be_encoder

Raises **NuSMVBeEncNotInitializedError** – if the global BeEnc singleton is not initialized

pynusmv.bmc.glob.**master_be_fsm**()

Returns the boolean FSM in BE stored in the master prop.

Raises `NuSMVBeFsmMasterInstanceNotInitializedError` – when the global BE FSM is null in the global properties database (ie when coi is enabled).

`pynusmv.bmc.glob.`**`go_bmc`**(*force=False*)

Performs all the necessary steps to use loaded model and be able to perform bmc related operations on it.

Raises `NuSMVNoReadModelError` – if no module was read (`pynusmv.glob.load()`) before this method was called.

class `pynusmv.bmc.glob.`**`BmcSupport`**

Bases: `object`

This class implements a context manager (an object that can be used with a 'with' statement) that initializes and deinitialises BMC and its submodules.

Note: The 'support' part in the name of this class does not bear the sometimes used signification of 'what makes a formula true'. Instead, it is present to make the code read easily (and be easily understood when read)

Example:

```
with init_nusmv():
    load(model)

    with BmcSupport():
        # do something smart like verifying LTL properties.
```

3.24 `pynusmv.bmc.invarspec` Module

The *`pynusmv.bmc.invarspec`* contains all the functionalities related to the verification of INVARSPEC properties using a technique close to that of SAT-based bounded model checking for LTL.

Most of the techniques exposed in this module have been described in *[ES03]*. Therefore, the reading of this paper is highly recommended in order to understand the purpose, ins and outs of the algorithms exposed hereunder.

`pynusmv.bmc.invarspec.`**`check_invar_induction`**(*invar_prop, solve=True, dump_type=<DumpType.NONE: 1>, fname_template=None*)

High level function that performs the verification of an INVAR property (INVARSPEC property as obtained from the `pynusmv.prop.PropDb`) using an inductive technique.

This function performs an end to end verification of the given property and prints the outcome (satisfaction or violation result) to standard output

Parameters

- **`invar_prop`** – the property to be verified. This should be an instance of Prop similar to what you obtain querying PropDb (`pynusmv.glob.prop_database()`)
- **`solve`** – a flag indicating whether or not the verification should actually be performed. (when this flag is turned off, no sat solver is not used to perform the verification and the function can serve to simply dump the problem to file).
- **`dump_type`** – the format in which to perform a dump of the generated sat problem (ie dimacs). By default, this parameter takes the value `pynusmv.bmc.utils.DumpType.NONE` which means that the problem is not dumped to file. Should you want to change this behavior, then this parameter is used to specify a file format in conjunction with

fname_template which is used to specify the name of the location where the file will be output.

- **fname_template** – the file name template of the location where to output the dump file.

Raises

- **NuSmvSatSolverError** – when the verification could not be performed because of a problem related to the sat solver (solver could not be created)
- **ValueError** – when the values of *dump_type* and *fname_template* are not consistent with each other (if one of them is None, both have to be None).

`pynusmv.bmc.invarspec.`**`check_invar_een_sorensson`**`(`*invar_prop*, *max_bound*, *dump_type=<DumpType.NONE: 1>*, *fname_template=None*, *use_extra_step=False*`)`

High level function that performs the verification of an INVAR property (INVARSPEC property as obtained from the *`pynusmv.prop.PropDb`*) using a technique called 'temporal induction' proposed by N. Een and N. Sorensson.

This function performs an end to end verification of the given property and prints the outcome (satisfaction or violation result) to standard output

Note: This approach to invariant verification is described in *[ES03]* .

This algorithm is *NOT* incremental and performs its verification by the means of temporal induction. With this technique (as is the case for regular inductive proof), the proof depends on a base case and an induction step. However, in order to make this technique complete, the requirements are hardened with two extra constraints:

- all states encountered must be different.
- the base case is assumed to hold for n consecutive steps

Parameters

- **invar_prop** – the property to be verified. This should be an instance of Prop similar to what you obtain querying PropDb (*`pynusmv.glob.prop_database()`*)
- **max_bound** – the maximum length of a trace considered in the generated SAT problem.
- **dump_type** – the format in which to perform a dump of the generated sat problem (ie dimacs). By default, this parameter takes the value *`pynusmv.bmc.utils.DumpType.NONE`* which means that the problem is not dumped to file. Should you want to change this behavior, then this parameter is used to specify a file format in conjunction with *fname_template* which is used to specify the name of the location where the file will be output.
- **fname_template** – the file name template of the location where to output the dump file.

Raises

- **NuSmvSatSolverError** – when the verification could not be performed because of a problem related to the sat solver (solver could not be created)
- **ValueError** – when the values of *dump_type* and *fname_template* are not consistent with each other (if one of them is None, both have to be None).

class `pynusmv.bmc.invarspec.`**`InvarClosureStrategy`**

Bases: `enum.IntEnum`

An enumeration to parameterize the direction of the problem generation either forward or backward

FORWARD = 1

BACKWARD = 1

pynusmv.bmc.invarspec.**check_invar_incrementally_dual**(*invar_prop*, *max_bound*, *closure_strategy*)

High level function that performs the verification of an INVAR property (INVARSPEC property as obtained from the *pynusmv.prop.PropDb*) using one of the variants of a technique called 'temporal induction' proposed by N. Een and N. Sorensson.

This function performs an end to end verification of the given property and prints the outcome (satisfaction or violation result) to standard output

Concretely, the dual algorithm is used an configure to 'increment' the SAT problem as specified by the *closure_strategy* which may either be forward or backward.

Note: This approach to invariant verification is described in *[ES03]* .

This algorithm is incremental and performs its verification by the means of temporal induction. With this technique (as is the case for regular inductive proof), the proof depends on a base case and an induction step. However, in order to make this technique complete, the requirements are hardened with two extra constraints:

- all states encountered must be different.
- the base case is assumed to hold for n consecutive steps

Parameters

- **invar_prop** – the property to be verified. This should be an instance of Prop similar to what you obtain querying PropDb (*pynusmv.glob.prop_database()*)
- **max_bound** – the maximum length of a trace considered in the generated SAT problem.
- **closure_strategy** – an enum value that configures the way the problem generation is performed.

Raises NuSmvSatSolverError – when the verification could not be performed because of a problem related to the sat solver (solver could not be created)

pynusmv.bmc.invarspec.**check_invar_incrementally_zigzag**(*invar_prop*, *max_bound*)

High level function that performs the verification of an INVAR property (INVARSPEC property as obtained from the *pynusmv.prop.PropDb*) using one of the variants of a technique called 'temporal induction' proposed by N. Een and N. Sorensson in *[ES03]* .

This function performs an end to end verification of the given property and prints the outcome (satisfaction or violation result) to standard output

Concretely, the zigzag algorithm alternates between the two problem expansion directions of the dual approach (either forward as in Algorithm 2:'Extending Base' or backward as in Algorithm 3:'Extending Step'

Note: This approach to invariant verification is described in *[ES03]* .

This algorithm is incremental and performs its verification by the means of temporal induction alternating between a forward and backward strategy. With this technique (as is the case for regular inductive proof), the proof depends on a base case and an induction step. However, in order to make this technique complete, the requirements are hardened with two extra constraints:

- all states encountered must be different.

- the base case is assumed to hold for n consecutive steps

Parameters

- **`invar_prop`** – the property to be verified. This should be an instance of Prop similar to what you obtain querying PropDb (*`pynusmv.glob.prop_database()`*)
- **`max_bound`** – the maximum length of a trace considered in the generated SAT problem.

Raises `NuSmvSatSolverError` – when the verification could not be performed because of a problem related to the sat solver (solver could not be created)

`pynusmv.bmc.invarspec.`**`check_invar_incrementally_falsification`**(*invar_prop*, *max_bound*)

High level function that performs the verification of an INVAR property (INVARSPEC property as obtained from the *`pynusmv.prop.PropDb`*) using one of the variants of a technique called 'temporal induction' proposed by N. Een and N. Sorensson.

This function performs an end to end verification of the given property and prints the outcome (satisfaction or violation result) to standard output

Concretely, the falsification algorithm is used which expands the base case.

Note: This approach to invariant verification is described in *[ES03]* .

This algorithm is incremental and performs its verification by the means of temporal induction alternating between a forward and backward strategy. With this technique (as is the case for regular inductive proof), the proof depends on a base case and an induction step. However, in order to make this technique complete, the requirements are hardened with two extra constraints:

- all states encountered must be different.
- the base case is assumed to hold for n consecutive steps

Parameters

- **`invar_prop`** – the property to be verified. This should be an instance of Prop similar to what you obtain querying PropDb (*`pynusmv.glob.prop_database()`*)
- **`max_bound`** – the maximum length of a trace considered in the generated SAT problem.

Raises `NuSmvSatSolverError` – when the verification could not be performed because of a problem related to the sat solver (solver could not be created)

`pynusmv.bmc.invarspec.`**`generate_invar_problem`**(*be_fsm*, *prop_node*)

Builds and returns the invariant problem of the given propositional formula

Concretely, this is the negation of (which needs to be satisfiable):

$$(I0 \implies P0) \wedge ((P0 \wedge R01) \implies P1)$$

Parameters

- **`be_fsm`** – the BeFsm object that represents the model against which the property will be verified. (if in doubt, it can be obtained via *`pynusmv.bmc.glob.master_be_fsm()`*)

- **prop_node** – the property for which to generate a verification problem represented in a 'node' format (subclass of *pynusmv.node.Node*) which corresponds to the format obtained from the ast. (remark: if you need to manipulate [ie negate] the formula before passing it, it is perfectly valid to pass a node decorated by *Wff.decorate*).

Returns the invariant problem of the given propositional formula

pynusmv.bmc.invarspec.**generate_base_step**(*be_fsm*, *prop_node*)

Builds and returns the boolean expression corresponding to the base step of the invariant problem to be generated for the given invar problem.

Concretely, this is:

```
I0 -> P0, where I0 is the init and invar at time 0,
          and   P0 is the given formula  at time 0
```

Parameters

- **be_fsm** – the BeFsm object that represents the model against which the property will be verified. (if in doubt, it can be obtained via *pynusmv.bmc.glob.master_be_fsm()*)
- **prop_node** – the property for which to generate a verification problem represented in a 'node' format (subclass of *pynusmv.node.Node*) which corresponds to the format obtained from the ast. (remark: if you need to manipulate [ie negate] the formula before passing it, it is perfectly valid to pass a node decorated by *Wff.decorate*).

Returns the invariant problem of the given propositional formula

pynusmv.bmc.invarspec.**generate_inductive_step**(*be_fsm*, *prop_node*)

Builds and returns the boolean expression corresponding to the inductive step of the invariant problem to be generated for the given invar problem.

Concretely, this is:

```
(P0 and R01) -> P1, where P0 is the formula at time 0,
                          R01 is the transition (without init) from time 0 to 1,
                      and P1 is the formula at time 1
```

Parameters

- **be_fsm** – the BeFsm object that represents the model against which the property will be verified. (if in doubt, it can be obtained via *pynusmv.bmc.glob.master_be_fsm()*)
- **prop_node** – the property for which to generate a verification problem represented in a 'node' format (subclass of *pynusmv.node.Node*) which corresponds to the format obtained from the ast. (remark: if you need to manipulate [ie negate] the formula before passing it, it is perfectly valid to pass a node decorated by *Wff.decorate*).

Returns the invariant problem of the given propositional formula

pynusmv.bmc.invarspec.**dump_dimacs_filename**(*be_enc*, *be_cnf*, *fname*)

Opens a new file named filename, then dumps the given INVAR problem in DIMACS format

Note: Calling this function is strictly equivalent to the following snippet:

with StdioFile.for_name(fname) as f: dump_dimacs(be_enc, be_cnf, f.handle)

Parameters

- **be_enc** – the encoding of the problem (typically fsm.encoding)
- **be_cnf** – the LTL problem represented in CNF
- **fname** – the name of the file in which to dump the DIMACS output.

`pynusmv.bmc.invarspec.`**`dump_dimacs`**(*be_enc*, *be_cnf*, *stdio_file*)

Dumps the given INVAR problem in DIMACS format to the designated *stdio_file*

Parameters

- **be_enc** – the encoding of the problem (typically fsm.encoding)
- **be_cnf** – the LTL problem represented in CNF
- **stdio_file** – the the file in which to dump the DIMACS output.

3.25 `pynusmv.bmc.ltlspec` Module

The `pynusmv.bmc.ltlspec` contains all the functionalities related to the bounded model checking of LTL properties: from end to end property verification to the translation of formulas to boolean expressions corresponding to the SAT problem necessary to verify these using LTL bounded semantics of the dumping of problem to file (in DIMACS format)

`pynusmv.bmc.ltlspec.`**`check_ltl`**(*ltl_prop*, *bound=10*, *loop=<MagicMock name='mock.bmc.Bmc_Utils_GetAllLoopbacks()' id='139958058740480'>*, *one_problem=False*, *solve=True*, *dump_type=<DumpType.NONE: 1>*, *fname_template=None*)

High level function that performs the verification of an LTL property (LTLSPEC property as obtained from the `pynusmv.prop.PropDb`).

This function performs an end to end verification of the given LTL property and prints the outcome (satisfaction or violation result) to standard output

Formally, it tries to determine if the problem $[[M, f]]_k$ is satisfiable. This problem is generated as

$$[[M]]_k \wedge \neg((\neg L_k \wedge [[ltl\_prop]]_k) \vee {}_l[[ltl\_prop]]_k^l)$$

Parameters

- **ltl_prop** – the LTL property to be verified. This should be an instance of Prop similar to what you obtain querying PropDb (`pynusmv.glob.prop_database()`)
- **bound** – the bound of the problem, that is to say the maximum number of times the problem will be unrolled. This parameter corresponds to the value *k* used in the formal definitions of a bmc problem.
- **loop** – a loop definition. This is an integer value corresponding to the moment in time where the loop might be starting (the parameter *l* in the formal definitions). However, this value is not as 'crude' as an absolute moment in time since it may indicate:
  - an absolute moment in time (obviously) when the value is positive
  - indicate a relative moment in time (when it is a negative number (for instance value -2 indicates that the loops are supposed to start 2 states ago)
  - that NO loop at all must be considered (ignore infinite behaviors) when this parameter takes the special value defined in `pynusmv.bmc.utils.no_loopback()`

– that ALL possible loops in the model must be taken into account when this parameter takes the special value defined in `pynusmv.bmc.utils.all_loopback()` (this is the default)

- **`one_problem`** – a flag indicating whether the problem should be verified for all possible execution lengths UP TO *bound* or if it should be evaluated only for executions that have the exact length *bound*. By default this flag is OFF and all problem lengths up to *bound* are verified.
- **`solve`** – a flag indicating whether or not the verification should actually be performed. (when this flag is turned off, no sat solver is not used to perform the bmc verification and the function can serve to simply dump the ltl problem to files).
- **`dump_type`** – the format in which to perform a dump of the generated sat problem (ie dimacs). By default, this parameter takes the value *`pynusmv.bmc.utils.DumpType.NONE`* which means that the problem is not dumped to file. Should you want to change this behavior, then this parameter is used to specify a file format in conjunction with *fname_template* which is used to specify the name of the location where the file will be output.
- **`fname_template`** – the file name template of the location where to output the dump file.

Raises

- **`NuSmvSatSolverError`** – when the verification could not be performed because of a problem related to the sat solver (solver could not be created)
- **`ValueError`** – when the bound is infeasible (negative value) or when the loop and bound values are inconsistent (loop is greater than the bound but none of the special values described above)

`pynusmv.bmc.ltlspec.`**`check_ltl_incrementally`**(*ltl_prop*, *bound=10*, *loop=<MagicMock name='mock.bmc.Bmc_Utils_GetAllLoopbacks()' id='139958058740480'>*, *one_problem=False*)

Performs the same end to end LTL property verification as *check_ltl* but generates the problem /incrementally/ instead of doing it all at once.

Concretely, this means that it does not compute the complete unrolling of the transition relation $[[M]]_k$ up front but computes each unrolling step separately and adds it to a group of the incremental sat solver.

The bounded semantics conversion of *ltl_prop* is done the exact same way as in *check_ltl*. So the real gain of calling this function resides in the avoidance of the regeneration of the formula representing the unrolled transition relation for lengths < bound. (and thus in the reduction of the size of the generated formula that needs to be solved).

Parameters

- **`ltl_prop`** – the LTL property to be verified. This should be an instance of Prop similar to what you obtain querying PropDb (*`pynusmv.glob.prop_database()`*)
- **`bound`** – the bound of the problem, that is to say the maximum number of times the problem will be unrolled. This parameter corresponds to the value *k* used in the formal definitions of a bmc problem.
- **`loop`** – a loop definition. This is an integer value corresponding to the moment in time where the loop might be starting (the parameter *l* in the formal definitions). However, this value is not as 'crude' as an absolute moment in time since it may indicate:

  – an absolute moment in time (obviously) when the value is positive

  - indicate a relative moment in time (when it is a negative number (for instance value -2 indicates that the loops are supposed to start 2 states ago)
  - that NO loop at all must be considered (ignore infinite behaviors) when this parameter takes the special value defined in `pynusmv.bmc.utils.no_loopback()`
  - that ALL possible loops in the model must be taken into account when this parameter takes the special value defined in `pynusmv.bmc.utils.all_loopback()` (this is the default)
- **one_problem** – a flag indicating whether the problem should be verified for all possible execution lengths UP TO *bound* or if it should be evaluated only for executions that have the exact length *bound*. By default this flag is OFF and all problem lengths up to *bound* are verified.

Raises

- **NuSmvSatSolverError** – when the verification could not be performed because of a problem related to the sat solver (solver could not be created)
- **ValueError** – when the bound is infeasible (negative value) or when the loop and bound values are inconsistent (loop is greater than the bound but none of the special values described above)

`pynusmv.bmc.ltlspec.`**`generate_ltl_problem`**(*fsm*, *prop_node*, *bound=10*, *loop=<MagicMock name='mock.bmc.Bmc_Utils_GetAllLoopbacks()' id='139958058740480'>*)

Generates a (non-incremental) Be expression corresponding to the SAT problem denoted by $[[fsm, prop\_node]]_{bound}^{loop}$

That is to say it generates the problem that combines both the formula and and the model to perform the verification. Put another way, this problem can be read as:

$$[[fsm]]_{bound} \wedge \neg((\neg L_k \wedge [[\neg prop\_node]]_k) \vee {}_l[[\neg prop\_node]]_k^l)$$

Parameters

- **fsm** – the BeFsm object that represents the model against which the property will be verified. (if in doubt, it can be obtained via `pynusmv.bmc.glob.master_be_fsm()`)
- **prop_node** – the property for which to generate a verification problem represented in a 'node' format (subclass of `pynusmv.node.Node`) which corresponds to the format obtained from the ast. (remark: if you need to manipulate [ie negate] the formula before passing it, it is perfectly valid to pass a node decorated by *Wff.decorate*).
- **bound** – the bound of the problem, that is to say the maximum number of times the problem will be unrolled. This parameter corresponds to the value *k* used in the formal definitions of a bmc problem.
- **loop** – a loop definition. This is an integer value corresponding to the moment in time where the loop might be starting (the parameter *l* in the formal definitions). However, this value is not as 'crude' as an absolute moment in time since it may indicate:
  - an absolute moment in time (obviously) when the value is positive
  - indicate a relative moment in time (when it is a negative number (for instance value -2 indicates that the loops are supposed to start 2 states ago)
  - that NO loop at all must be considered (ignore infinite behaviors) when this parameter takes the special value defined in `pynusmv.bmc.utils.no_loopback()`

– that ALL possible loops in the model must be taken into account when this parameter takes the special value defined in `pynusmv.bmc.utils.all_loopback()` (this is the default)

Returns a Be boolean expression representing the satisfiability problem of for the verification of this property.

Raises **`ValueError`** – when the bound is infeasible (negative value) or when the loop and bound values are inconsistent (loop is greater than the bound but none of the special values described above)

`pynusmv.bmc.ltlspec.`**`bounded_semantics`**(*fsm*, *prop_node*, *bound=10*, *loop=<MagicMock name='mock.bmc.Bmc_Utils_GetAllLoopbacks()' id='139958058740480'>*)

Generates a Be expression corresponding to the bounded semantics of the given LTL formula. It combines the bounded semantics of the formula when there is a loop and when there is none with the loop condition.

In the literature, the resulting formula would be denoted as

$$[[f]]_k := (\neg L_k \wedge [[f]]^0_k) \vee (\bigvee_{j=l}^{k} {}_jL_k \wedge {}_j[[f]]^0_k)$$

where l is used to denote *loop*, f for *prop_node* and k for the *bound*.

Note: Fairness is taken into account in the generation of the resulting expression

Parameters

- **`fsm`** – the fsm against which the formula will be evaluated. It is not directly relevant to the generation of the formula for *prop_node* but is used to determine to generate fairness constraints for this model which are combined with *prop_node* constraint.
- **`prop_node`** – the property for which to generate a verification problem represented in a 'node' format (subclass of *`pynusmv.node.Node`*) which corresponds to the format obtained from the ast.(remark: if you need to manipulate [ie negate] the formula before passing it, it is perfectly valid to pass a node decorated by *Wff.decorate*).
- **`bound`** – the bound of the problem, that is to say the maximum number of times the problem will be unrolled. This parameter corresponds to the value *k* used in the formal definitions of a bmc problem.
- **`optimized`** – a flag indicating whether or not the use of the optimisation for formulas of depth 1 is desired.

Returns a boolean expression corresponding to the bounded semantics of *prop_node*

Raises **`ValueError`** – when the bound is infeasible (negative value) or when the loop and bound values are inconsistent (loop is greater than the bound but none of the special values described above)

`pynusmv.bmc.ltlspec.`**`bounded_semantics_without_loop`**(*fsm*, *prop_node*, *bound*)

Generates a Be expression corresponding to the bounded semantics of the given LTL formula in the case where the formula is evaluated against paths that contain no loop and have a maximal length of *bound*.

Note: This function proves to be very useful since the bounded semantics of LTL depends on two cases: (a) when the encountered path contains loops (in that case the unbounded semantics of LTL can be maintained since there exists infinite paths) and (b) the case where there are no possible loops (and the semantics has to be altered slightly).

In the literature, the expression generated by this function is denoted $[[f]]^0_k$

With f used to represent the formula *prop_node*, and k for *bound*

Note: Fairness is taken into account in the generation of the resulting expression

---

Parameters

- **fsm** – the fsm against which the formula will be evaluated. It is not directly relevant to the generation of the formula for *prop_node* but is used to determine to generate fairness constraints for this model which are combined with *prop_node* constraint.
- **prop_node** – the property for which to generate a verification problem represented in a 'node' format (subclass of `pynusmv.node.Node`) which corresponds to the format obtained from the ast.(remark: if you need to manipulate [ie negate] the formula before passing it, it is perfectly valid to pass a node decorated by *Wff.decorate*).
- **bound** – the bound of the problem, that is to say the maximum number of times the problem will be unrolled. This parameter corresponds to the value *k* used in the formal definitions of a bmc problem.

Returns a boolean expression corresponding to the bounded semantics of *prop_node* in the case where there is no loop on the path.

Raises ValueError – when the specified problem bound is negative

`pynusmv.bmc.ltlspec.`**`bounded_semantics_single_loop`**(*fsm*, *prop_node*, *bound*, *loop*)

Generates a Be expression corresponding to the bounded semantics of the given LTL formula in the case where the formula is evaluated against a path that contains one single loop starting at position *loop*.

In the literature, the resulting formula would be denoted as ${}_lL_k \wedge {}_l[[f]]^0_k$

where l is used to denote *loop*, f for *prop_node* and k for the *bound*.

In other words, the generated boolean expression is the conjunction of the constraint imposing that there be a k-l loop from *bound* to *loop* and that the formula is evaluated at time 0 out of *bound*.

Note: Fairness is taken into account in the generation of the resulting expression

---

Parameters

- **fsm** – the fsm against which the formula will be evaluated. It is not directly relevant to the generation of the formula for *prop_node* but is used to determine to generate fairness constraints for this model which are combined with *prop_node* constraint.
- **prop_node** – the property for which to generate a verification problem represented in a 'node' format (subclass of `pynusmv.node.Node`) which corresponds to the format obtained from the ast.(remark: if you need to manipulate [ie negate] the formula before passing it, it is perfectly valid to pass a node decorated by *Wff.decorate*).
- **bound** – the bound of the problem, that is to say the maximum number of times the problem will be unrolled. This parameter corresponds to the value *k* used in the formal definitions of a bmc problem.

Returns a boolean expression corresponding to the bounded semantics of *prop_node* in the case where there is a loop from bound to loop

Raises `ValueError` – when the bound is infeasible (negative value) or when the loop and bound values are inconsistent (loop is greater than the bound but none of the special values described above)

`pynusmv.bmc.ltlspec.`**`bounded_semantics_all_loops_optimisation_depth1`**(*fsm*, *prop_node*, *bound*)

Generates a Be expression corresponding to the bounded semantics of the given LTL formula in the case where the formula is evaluated against a path that contains a loop at any of the positions in the range [0; bound] and *the 'depth'(:attr:'pynusmv.wff.Wff.depth') of the formula is 1 and no fairness constraint comes into play*.

Note: Unless you know precisely why you are using this function, it is probably safer to just use bounded_semantics_all_loops with the optimized flag turned on.

Parameters

- **`fsm`** – the fsm against which the formula will be evaluated. It is not directly relevant to the generation of the formula for *prop_node* but is used to determine to generate fairness constraints for this model which are combined with *prop_node* constraint.
- **`prop_node`** – the property for which to generate a verification problem represented in a 'node' format (subclass of `pynusmv.node.Node`) which corresponds to the format obtained from the ast.(remark: if you need to manipulate [ie negate] the formula before passing it, it is perfectly valid to pass a node decorated by *Wff.decorate*).
- **`bound`** – the bound of the problem, that is to say the maximum number of times the problem will be unrolled. This parameter corresponds to the value *k* used in the formal definitions of a bmc problem.

Returns a boolean expression corresponding to the bounded semantics of *prop_node* in the case where there is may be a loop anywhere on the path between the positions *loop* and *bound* and the formula has a depth of exactly one.

Raises `ValueError` – when the specified propblem bound is negative

`pynusmv.bmc.ltlspec.`**`bounded_semantics_all_loops`**(*fsm*, *prop_node*, *bound*, *loop*, *optimized=True*)

Generates a Be expression corresponding to the bounded semantics of the given LTL formula in the case where the formula is evaluated against a path that contains a loop at any of the positions in the range [loop; bound]

In the literature, the resulting formula would be denoted as

$$\bigvee_{j=l}^{k} {}_jL_k \wedge {}_j[[f]]_k^0$$

where l is used to denote *loop*, f for *prop_node* and k for the *bound*.

Note: Fairness is taken into account in the generation of the resulting expression

Parameters

- **fsm** – the fsm against which the formula will be evaluated. It is not directly relevant to the generation of the formula for *prop_node* but is used to determine to generate fairness constraints for this model which are combined with *prop_node* constraint.
- **prop_node** – the property for which to generate a verification problem represented in a 'node' format (subclass of `pynusmv.node.Node`) which corresponds to the format obtained from the ast.(remark: if you need to manipulate [ie negate] the formula before passing it, it is perfectly valid to pass a node decorated by *Wff.decorate*).
- **bound** – the bound of the problem, that is to say the maximum number of times the problem will be unrolled. This parameter corresponds to the value *k* used in the formal definitions of a bmc problem.
- **optimized** – a flag indicating whether or not the use of the optimisation for formulas of depth 1 is desired.

Returns a boolean expression corresponding to the bounded semantics of *prop_node* in the case where there is may be a loop anywhere on the path between the positions *loop* and *bound*

Raises **ValueError** – when the bound is infeasible (negative value) or when the loop and bound values are inconsistent (loop is greater than the bound but none of the special values described above)

`pynusmv.bmc.ltlspec.`**bounded_semantics_without_loop_at_offset**(*fsm*, *formula*, *time*, *bound*, *offset*)

Generates the Be $[[formula]]_{bound}^{time}$ corresponding to the bounded semantic of *formula* when there is no loop on the path but encodes it with an *offset* long shift in the timeline of the encoder.

Note: This code was first implemented in Python with PyNuSMV but, since the Python implementation proved to be a huge performance bottleneck (profiling revealed that useless memory management was dragging the whole system behind), it has been translated back to C to deliver much better perf. results.

Note: This function plays the same role as *bounded_semantics_without_loop* but allows to position the time blocks at some place we like in the encoder timeline. This is mostly helpful if you want to devise verification methods that need to have multiple parallel verifications. (ie. diagnosability).

Note however, that the two implementations are different.

Warning: So far, the only supported temporal operators are F, G, U, R, X

Parameters

- **fsm** – the BeFsm for which the property will be verified. Actually, it is only used to provide the encoder used to assign the variables to some time blocks. The api was kept this ways to keep uniformity with its non-offsetted counterpart.
- **formula** – the property for which to generate a verification problem represented in a 'node' format (subclass of `pynusmv.node.Node`) which corresponds to the format obtained from the ast. (remark: if you need to manipulate [ie negate] the formula before passing it, it is perfectly valid to pass a node decorated by *Wff.decorate*).
- **time** – the logical time at which the semantics is to be evaluated. (Leave out the offset for this param. If you intend the 3rd state of a trace, say time 2).

- **bound** – the logical time bound to the problem. (Leave out the offset for this param: if you intend to have a problem with at most 10 steps, say bound=10)
- **offset** – the time offset in the encoding block where the sem of this formula will be generated.

Returns a Be corresponding to the semantics of *formula* at *time* for a problem with a maximum of *bound* steps encoded to start at time *offset* in the *fsm* encoding timeline.

pynusmv.bmc.ltlspec.**bounded_semantics_with_loop_at_offset**(*fsm*, *formula*, *time*, *bound*, *loop*, *offset*)

Generates the Be ${}_{loop}[[formula]]_{bound}^{time}$ corresponding to the bounded semantic of *formula* when a loop starts at time 'loop' on the path but encodes it with an *offset* long shift in the timeline of the encoder.

Note: This code was first implemented in Python with PyNuSMV but, since the Python implementation proved to be a huge performance bottleneck (profiling revealed that useless memory management was dragging the whole system behind), it has been translated back to C to deliver much better perf. results.

Note: This function plays the same role as *bounded_semantics_with_loop* but allows to position the time blocks at some place we like in the encoder timeline. This is mostly helpful if you want to devise verification methods that need to have multiple parallel verifications. (ie. diagnosability).

Note however, that the two implementations are different.

Warning: So far, the only supported temporal operators are F, G, U, R, X

Parameters

- **fsm** – the BeFsm for which the property will be verified. Actually, it is only used to provide the encoder used to assign the variables to some time blocks. The api was kept this ways to keep uniformity with its non-offsetted counterpart.
- **formula** – the property for which to generate a verification problem represented in a 'node' format (subclass of *pynusmv.node.Node*) which corresponds to the format obtained from the ast. (remark: if you need to manipulate [ie negate] the formula before passing it, it is perfectly valid to pass a node decorated by *Wff.decorate*).
- **time** – the logical time at which the semantics is to be evaluated. (Leave out the offset for this param. If you intend the 3rd state of a trace, say time 2).
- **bound** – the logical time bound to the problem. (Leave out the offset for this param: if you intend to have a problem with at most 10 steps, say bound=10)
- **loop** – the logical time at which a loop starts on the path. (Leave out the offset for this param. If you intend to mean that loop starts at 2nd state of the trace, say loop=2)
- **offset** – the time offset in the encoding block where the sem of this formula will be generated.

Returns a Be corresponding to the semantics of *formula* at *time* for a problem with a maximum of *bound* steps encoded to start at time *offset* in the *fsm* encoding timeline.

pynusmv.bmc.ltlspec.**bounded_semantics_at_offset**(*fsm*, *formula*, *bound*, *offset*, *fairness=True*)

Generates the Be $[[formula]]_{bound}$ corresponding to the bounded semantic of *formula* but encodes it with an *offset* long shift in the timeline of the encoder.

Note: This function plays the same role as *bounded_semantics_all_loops* but allows to position the time blocks at some place we like in the encoder timeline. This is mostly helpful if you want to devise verification methods that need to have multiple parallel verifications. (ie. diagnosability).

Note however, that the two implementations are different.

Warning: So far, the only supported temporal operators are F, G, U, R, X

Parameters

- **fsm** – the BeFsm for which the property will be verified. Actually, it is only used to provide the encoder used to assign the variables to some time blocks. The api was kept this ways to keep uniformity with its non-offsetted counterpart.
- **formula** – the property for which to generate a verification problem represented in a 'node' format (subclass of *pynusmv.node.Node*) which corresponds to the format obtained from the ast. (remark: if you need to manipulate [ie negate] the formula before passing it, it is perfectly valid to pass a node decorated by *Wff.decorate*).
- **bound** – the logical time bound to the problem. (Leave out the offset for this param: if you intend to have a problem with at most 10 steps, say bound=10)
- **offset** – the time offset in the encoding block where the sem of this formula will be generated.
- **fairness** – a flag indicating whether or not to take the fairness constraint into account.

Returns a Be corresponding to the semantics of *formula* for a problem with a maximum of *bound* steps encoded to start at time *offset* in the *fsm* encoding timeline.

pynusmv.bmc.ltlspec.**dump_dimacs_filename**(*be_enc*, *be_cnf*, *bound*, *fname*)

Opens a new file named filename, then dumps the given LTL problem in DIMACS format

Note: The bound of the problem is used only to generate a the readable version of the mapping table as comment at beginning of the file.

Note: Calling this function is strictly equivalent to the following snippet:

```
with StdioFile.for_name(fname) as f:
    dump_dimacs(be_enc, be_cnf, bound, f.handle)
```

Parameters

- **be_enc** – the encoding of the problem (typically fsm.encoding)
- **be_cnf** – the LTL problem represented in CNF
- **bound** – the bound of the problem

- **fname** – the name of the file in which to dump the DIMACS output.

pynusmv.bmc.ltlspec.**dump_dimacs**(*be_enc*, *be_cnf*, *bound*, *stdio_file*)
Dumps the given LTL problem in DIMACS format to the designated *stdio_file*

Note: The bound of the problem is used only to generate a the readable version of the mapping table as comment at beginning of the file.

Parameters

- **be_enc** – the encoding of the problem (typically fsm.encoding)
- **be_cnf** – the LTL problem represented in CNF
- **bound** – the bound of the problem
- **stdio_file** – the the file in which to dump the DIMACS output.

3.26 `pynusmv.bmc.utils` Module

The module *pynusmv.bmc.utils* contains bmc related utility functions. These are roughly organized in six categories:

- loop related utility functions (include the generation of loop condition)
- inlining of boolean expressions
- ast nodes manipulations and normalization
- BMC model / unrolling
- problem dumping to file
- counter example (trace) generation.

pynusmv.bmc.utils.**all_loopbacks**()

Returns the integer value corresponding to the all loopback specification.

pynusmv.bmc.utils.**no_loopback**()

Returns the integer value corresponding to the no loopback specification.

pynusmv.bmc.utils.**is_all_loopbacks**(*loop*)

Returns true iff the given loop number corresponds to *all_loopbacks()*

pynusmv.bmc.utils.**is_no_loopback**(*loop*)

Returns true iff the given loop number corresponds to *no_loopback()*

pynusmv.bmc.utils.**loop_from_string**(*loop_text*)
Given a string representing a possible loopback specification (an integer value, * (all) or x (none), returns the corresponding integer.

Returns the integer value corresponding to the given loop spec.

pynusmv.bmc.utils.**check_consistency**(*bound*, *loop*)
This function raises an exception ValueError when the given bound and loop are not consistent: either bound is negative or the loop is greater than the bound and is none of the special values for the loop (all_loopbacks or no_loopbacks)

Parameters

- **bound** – the bound of the problem, that is to say the maximum number of times the problem will be unrolled. This parameter corresponds to the value *k* used in the formal definitions of a bmc problem.
- **loop** – a loop definition. This is an integer value corresponding to the moment in time where the loop might be starting (the parameter *l* in the formal definitions). However, this value is not as 'crude' as an absolute moment in time since it may indicate:
  - an absolute moment in time (obviously) when the value is positive
  - indicate a relative moment in time (when it is a negative number (for instance value -2 indicates that the loops are supposed to start 2 states ago)
  - that NO loop at all must be considered (ignore infinite behaviors) when this parameter takes the special value defined in *pynusmv.bmc.utils.no_loopback()*
  - that ALL possible loops in the model must be taken into account when this parameter takes the special value defined in `pynusmv.bmc.utils.all_loopback()` (this is the default)

Raises ValueError – when the *bound* and *loop* are not consistent with one another.

`pynusmv.bmc.utils.`**`convert_relative_loop_to_absolute`**`(l, k)`

Converts a relative loop value (wich can also be an absolute loop value) to an absolute loop value.

Example:

```
For example the -4 value when k is 10 is the value 6,
but the value 4 (absolute loop value) is still 4
```

Note: No check is made to prevent you from entering inconsistent values. For instance l=-12 and k=10 will get you -2 which does not mean anything Similarly, l=12 and k=10 will get you 12 which should be forbidden by BMC semantics.

If you need such consistency, check *check_consistency()*

Parameters

- **l** – the relative loop value (which may actually be absolute)
- **k** – the bound on the considered problem

Returns the absolute value for the loop

Raises ValueError – when the given *k* and *l* are not consistent with each other or when the bound *k* is negative.

`pynusmv.bmc.utils.`**`loop_condition`**`(enc, k, l)`

This function generates a Be expression representing the loop condition which is necessary to determine that k->l is a backloop.

Formally, the returned constraint is denoted ${}_{l}L_{k}$

Because the transition relation is encoded in Nusmv as formula (and not as a relation per-se), we determine the existence of a backloop between l < k and forall var, var(i) == var(k)

That is to say: if it is possible to encounter two times the same state (same state being all variables have the same value in both states) we know there is a backloop on the path

Note: This code was first implemented in Python with PyNuSMV but, since the Python implementation proved to be a huge performance bottleneck (profiling revealed that useless memory management was dragging the whole system behind), it has been translated back to C to deliver much better perf. results.

Parameters

- **`fsm`** – the fsm on which the condition will be evaluated
- **`k`** – the highest time
- **`l`** – the time where the loop is assumed to start

Returns a Be expression representing the loop condition that verifies that k-l is a loop path.

Raises **`ValueError`** – when the given *k* and *l* are not consistent with each other or when the bound *k* is negative.

`pynusmv.bmc.utils.`**`fairness_constraint`**(*fsm*, *k*, *l*)

Computes a step of the constraint to be added to the loop side of the BE when one wants to take fairness into account for the case where we consider the existence of a k-l loop (between k and l obviously).

Note: This code was first implemented in Python with PyNuSMV but, since the Python implementation proved to be a huge performance bottleneck (profiling revealed that useless memory management was dragging the whole system behind), it has been translated back to C to deliver much better perf. results.

Parameters

- **`fsm`** – the fsm whose transition relation must be unrolled
- **`k`** – the maximum (horizon/bound) time of the problem
- **`l`** – the time where the loop starts

Returns a step of the fairness constraint to force fair execution on the k-l loop.

Raises **`ValueError`** – when the given *k* and *l* are not consistent with each other or when the bound *k* is negative.

`pynusmv.bmc.utils.`**`successor`**(*time*, *k*, *l*)

Returns the successor time of *time* in the context of a (loopy) trace (k-l loop) on the interval [loop; bound].

For a complete definition of the successor relation, check defintion 6 in *[BCC+03]* .

Note: In the particular case where the value of *l* equal to *no_loopback()*, then the sucessor is simply *time* + 1. If on top of that, *time* is equal to *k*. Then there is no sucessor and the value None is returned.

Warning: To be consistent with the way the loop condition is implemented (equiv of all the state variables). In the case of a loopy path (k-l loop) we have that walking 'k' steps means to be back at step 'l'. Hence, the value of i can only vary from 0 to k-1 (and will repeat itself in the range [l; k-1])

Parameters

- **`time`** – the time whose successor needs to be computed.

- **k** – the highest time
- **l** – the time where the loop is assumed to start

Returns the successor of *time* in the context of a k-l loop.

Raises **ValueError** – when the *time* or the bound *k* is negative

`pynusmv.bmc.utils.`**`apply_inlining`**(*be_expr*)

Performs the inlining of *be_expr* (same effect as *pynusmv.be.expression.Be.inline()*) but uses the global user's settings in order to determine the value that should be given to the *add_conj* parameter.

Parameters **be_expr** – a Be expression (*pynusmv.be.expression.Be*) that needs to be inlined.

Returns a boolean expression (*pynusmv.be.expression.Be*) equivalent to *be_expr* but inlined according to the user's preferences.

`pynusmv.bmc.utils.`**`apply_inlining_for_incremental_algo`**(*be_expr*)

Performs the inlining of *be_expr* in a way that guarantees soundness of incremental algorithms.

Note: Calling this function is strictly equivalent to calling *be_expr.inline(True)*.

Parameters **be_expr** – a Be expression (*pynusmv.be.expression.Be*) that needs to be inlined.

Returns a boolean expression (*pynusmv.be.expression.Be*) equivalent to *be_expr* but inlined according to the user's preferences.

`pynusmv.bmc.utils.`**`make_nnf_boolean_wff`**(*prop_node*)

Decorates the property identified by *prop_node* to become a boolean WFF, and converts the resulting formula to negation normal form. (negation sign on literals only).

`pynusmv.bmc.utils.`**`make_negated_nnf_boolean_wff`**(*prop_node*)

Decorates the property identified by *prop_node* to become a boolean WFF, negates it and converts the resulting formula to negation normal form. (negation sign on literals only).

`pynusmv.bmc.utils.`**`is_constant_expr`**(*node*)

Returns True iff the given node type corresponds to a constant expression (true or false).

Parameters **node** – the expression in node format (that is to say, the format obtained after parsing an expression *pynusmv.node.Node*) for which we want to determinate whether or not it is a constant expression.

Returns True iff the given node represents a constant expression.

`pynusmv.bmc.utils.`**`is_variable`**(*node*)

Returns True iff the given node type corresponds to a variable expression.

Parameters **node** – the expression in node format (that is to say, the format obtained after parsing an expression *pynusmv.node.Node*) for which we want to determinate whether or not it denotes a variable.

Returns True iff the given node represents a variable.

`pynusmv.bmc.utils.`**`is_past_operator`**(*node*)

Returns True iff the given node type corresponds to a expression using a past operator.

Parameters **`node`** – the expression in node format (that is to say, the format obtained after parsing an expression *`pynusmv.node.Node`*) for which we want to determinate whether or not it denotes an expression using a past operator.

Returns True iff the given node represents a past operator expression.

`pynusmv.bmc.utils.`**`is_binary_operator`**`(`*node*`)`

Returns True iff the given node denotes a binary expression.

Parameters **`node`** – the expression in node format (that is to say, the format obtained after parsing an expression *`pynusmv.node.Node`*) for which we want to determinate whether or not it denotes binary expression.

Returns True iff the given node represents a binary expression.

class `pynusmv.bmc.utils.`**`OperatorType`**

Bases: `enum.IntEnum`

An enumeration to classify the kind of operator we are dealing with.

`UNKNOWN_OP` = **1**

`CONSTANT_EXPR` = **1**

`LITERAL` = **1**

`PROP_CONNECTIVE` = **1**

`TIME_OPERATOR` = **1**

`pynusmv.bmc.utils.`**`operator_class`**`(`*node*`)`

Determines the kind of expression represented by the given *node*.

Note: In this context, NOT is considered as negated literal and receives thus the LITERAL class. It is therefore not considered as as propositional connective.

Parameters **`node`** – the expression in node format (that is to say, the format obtained after parsing an expression *`pynusmv.node.Node`*) for whose kind needs to be determined.

Returns the *`OperatorType`* corresponding to the expression represented by *node*.

class `pynusmv.bmc.utils.`**`BmcModel`**`(`*be_fsm=None*`)`

Bases: `object`

The *`BmcModel`* defines a wrapper providing an higher level interface to the BeFsm object. This is the object that must be used to generate the LTL problems.

`init`

`invar`

`trans`

`unrolling``(`*j, k*`)`

Unrolls the transition relation from j to k, taking into account of invars.

Parameters

- **`j`** – the start time
- **`k`** – the end time

Returns a Be representing the unrolling of the fsm from time i to k

Raises **`ValueError`** – if one of the specified times (k,j) is negative and when k < j

`unrolling_fragment`

`path` (*k*, *with_init=True*)

Returns the path for the model from 0 to k. If the flag *with_init* is off, only the invariants are taken into account (and no init) otherwise both are taken into account.

Parameters

- **`k`** – the end time
- **`with_init`** – a flag indicating whether or not to consider the init

Retunr a Be representing the paths in the model from times 0 to *k*.

Raises **`ValueError`** – if the specified time k is negative.

`fairness` (*k*, *l*)

Generates and returns an expression representing all fairnesses in a conjunctioned form.

Parameters

- **`k`** – the maximum length of the considered problem
- **`l`** – the time when a loop may start

Returns an expression representing all fairnesses in a conjunctioned form.

Raises **`ValueError`** – when the k and l parameters are incorrect (namely, when one says the loop must start after the problem bound).

`invar_dual_forward_unrolling` (*invarspec*, *i*)

Performs one step in the unrolling of the invarspec property.

In terms of pseudo code, this corresponds to:

```
if i == 0 :
    return Invar[0]
else
    return Trans[i-1] & Invar[i] & Property[i-1]
```

Note: this is specific to the INVARSPEC verification

Parameters

- **`invarspec`** – a booleanized, NNF formula representing an invariant property.
- **`i`** – the time step for which the unrolling is generated.

Returns Trans[i-1] & Invar[i] & Property[i-1]

Raises **`ValueError`** – in case the given parameters are incorrect.

class `pynusmv.bmc.utils.`**`DumpType`**

Bases: `enum.IntEnum`

An enumeration to specify the format in which a dump should be performed

`NONE` = **1**

`DA_VINCI` = **1**

`GDL` = **1**

DIMACS = 1

`pynusmv.bmc.utils.`**`dump_problem`**(*be_enc*, *be_cnf*, *prop*, *bound*, *loop*, *dump_type*, *fname*)

Dumps the given problem (LTL or INVAR) in the specified format to the designated file.

> **Warning:** In order to call this function, *prop MUST* be a property as returned from the *PropDb* (*`pynusmv.prop.PropDb`*). That is to say, it should correspond to a property which was specified in the SMV input text as LTLSPEC or INVARSPEC.

Parameters

- **`be_enc`** – the encoding of the problem (typically fsm.encoding)
- **`be_cnf`** – the problem represented in CNF (may be LTL or INVAR problem)
- **`prop_node`** – the property being verified (the translation of be_cnf) represented in a 'Prop' format (subclass of *`pynusmv.prop.Prop`*) which corresponds to the format obtained from the *PropDb* (*`pynusmv.glob.prop_database()`*)
- **`bound`** – the bound of the problem
- **`loop`** – a loop definition. This is an integer value corresponding to the moment in time where the loop might be starting (the parameter *l* in the formal definitions). However, this value is not as 'crude' as an absolute moment in time since it may indicate:
  - an absolute moment in time (obviously) when the value is positive
  - indicate a relative moment in time (when it is a negative number (for instance value -2 indicates that the loops are supposed to start 2 states ago)
  - that NO loop at all must be considered (ignore infinite behaviors) when this parameter takes the special value defined in *`pynusmv.bmc.utils.no_loopback()`*
  - that ALL possible loops in the model must be taken into account when this parameter takes the special value defined in `pynusmv.bmc.utils.all_loopback()` (this is the default)
- **`dump_type`** – the format in which to output the data. (*`DumpType`*)
- **`fname`** – a template of the name of the file where the information will be dumped.

Raises **`ValueError`** – in case the given parameters are incorrect.

`pynusmv.bmc.utils.`**`print_counter_example`**(*fsm*, *problem*, *solver*, *k*, *descr='BMC counter example'*)

Prints a counter example for *problem* evaluated against *fsm* as identified by *solver* (problem has a length *k*) to standard output.

> **Note:** If you are looking for something more advanced, you might want to look at *`pynusmv.be.encoder.BeEnc.decode_sat_model()`* which does the same thing but is more complete.

Parameters

- **`fsm`** – the FSM against which problem was evaluated
- **`problem`** – the SAT problem used to identify a counter example
- **`solver`** – the SAT solver that identified a counter example
- **`k`** – the length of the generated problem (length in terms of state)

- **descr** – a description of what the generated counter example is about

Raises ValueError – whenever the problem or the solver is None or when the problem bound *k* is negative.

pynusmv.bmc.utils.**generate_counter_example**(*fsm*, *problem*, *solver*, *k*, *descr='BMC counter example'*)

Generates a counter example for *problem* evaluated against *fsm* as identified by *solver* (problem has a length *k*) but prints nothing.

Note: If you are looking for something more advanced, you might want to look at `pynusmv.be.encoder.BeEnc.decode_sat_model()` which does the same thing but is more complete.

Parameters

- **fsm** – the FSM against which problem was evaluated
- **problem** – the SAT problem used to identify a counter example
- **solver** – the SAT solver that identified a counter example
- **k** – the length of the generated problem (length in terms of state)
- **descr** – a description of what the generated counter example is about

Raises ValueError – whenever the problem or the solver is None or when the problem bound *k* is negative.

pynusmv.bmc.utils.**fill_counter_example**(*fsm*, *solver*, *k*, *trace*)

Uses the given sat solver instance to fill the details of the trace and store it all in *trace*.

Note: If you are looking for something more advanced, you might want to look at `pynusmv.be.encoder.BeEnc.decode_sat_model()` which does the same thing but is more complete.

Note: The given *trace must* be empty, otherwise an exception is raised.

Parameters

- **fsm** – the FSM against which problem was evaluated
- **solver** – the SAT solver that identified a counter example
- **k** – the length of the generated problem (length in terms of state)
- **trace** – the trace to be populated with the details read from the sat solver.

Returns *trace* but populated with the counter example extracted from the solver model.

Raises

- ***NuSmvIllegalTraceStateError*** – if the given *trace* is not empty.
- **ValueError** – whenever the fsm, solver or trace is None or when the problem bound *k* is negative.

3.27 `pynusmv.sexp.__init__` Module

The `pynusmv.sexp` module regroups the modules related to the treatment of simple expressions (SEXP) in pynusmv. As a matter of fact, it only provides an access to the *`fsp`* module which provides an abstract representation of the FSM encoded in terms of simple expressions (but not BE yet).

Concretely, it is highly likely that you'll use this module in conjunction with *`pynusmv.be.fsm`*.

3.28 `pynusmv.sexp.fsm` Module

This module defines the classes wrapping the SEXP FSM structures. In particular:

- *`SexpFsm`* which wraps the basic SEXP fsm
- *`BoolSexpFsm`* which wraps a boolean SEXP fsm

class `pynusmv.sexp.fsm.`**`SexpFsm`**(*ptr*, *freeit=True*)

Bases: *`pynusmv.utils.PointerWrapper`*

This class encapsulates a generic SEXP FSM

`copy`()

Creates a copy of this object

Returns a copy of this object

`symbol_table`

Returns the symbol table associated to this FSM.

`is_boolean`

Returns true iff this fsm is a boolean SEXP FSM

`hierarchy`

Returns the flat hierarchy associated to this object

`init`

Returns an Expression that collects init states for all handled vars.

`invariants`

Returns an Expression that collects invar states for all handled vars.

`trans`

Returns an Expression that collects all next states for all variables

`input`

Returns an Expression that collects all input states for all variables

`justice`

The list of sexp expressions defining the set of justice constraints for this FSM.

Note: NUSMV supports two types of fairness constraints, namely justice constraints and com- passion constraints. A justice constraint consists of a formula f, which is assumed to be true infinitely often in all the fair paths. In NUSMV, justice constraints are identified by keywords JUSTICE and, for backward compatibility, FAIRNESS.

Returns the list of sexp expressions defining the set of justice constraints for this FSM.

`compassion`

The list of sexp expressions defining the set of compassion constraints for this FSM.

Note: NUSMV supports two types of fairness constraints, namely justice constraints and compassion constraints. A justice constraint consists of a formula f, which is assumed to be true infinitely often in all the fair paths. A compassion constraint consists of a pair of formulas (p,q); if property p is true infinitely often in a fair path, then also formula q has to be true infinitely often in the fair path. In NUSMV, compassion constraints are identified by keyword COMPASSION. If compassion constraints are used, then the model must not contain any input variables. Currently, NUSMV does not enforce this so it is the responsibility of the user to make sure that this is the case.

Returns the list of sexp expressions defining the set of compassion constraints for this FSM.

`variables_list`

Returns the set of variables in the FSM

`symbols_list`

Returns the set of symbols in the FSM

`is_syntactically_universal`

Checks if the SexpFsm is syntactically universal: Checks INIT, INVAR, TRANS, INPUT, JUSTICE, COMPASSION to be empty (ie: True Expr). In this case returns true, false otherwise

Returns true iff this fsm has no INIT, INVAR, TRANS, INPUT, JUSTICE or COMPASSION.

class `pynusmv.sexp.fsm.`**`BoolSexpFsm`**(*ptr*, *freeit=True*)

Bases: *`pynusmv.sexp.fsm.SexpFsm`*

This class encapsulates a boolean encoded SEXP FSM.

Note: Since it defines the same interface as the regular SexpFSM, the purpose of this class is to correctly redefine the _free function and override the _as_SexpFsm_ptr function so as to leverage the inheritance defined in C.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Bibliography

[BCC+03] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. “Bounded model checking.” In Ad- vances in Computers, volume 58. Academic Press, 2003.

[ES03] Niklas Een and Niklas Sorensson. “Temporal induction by incremental sat solving.” in Ofer Strichman and Armin Biere, editors, Electronic Notes in Theoretical Computer Science, volume 89. Elsevier, 2004.

[ES03] Niklas Een and Niklas Sorensson. “Temporal induction by incremental sat solving.” in Ofer Strichman and Armin Biere, editors, Electronic Notes in Theoretical Computer Science, volume 89. Elsevier, 2004.

[BCC+03] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. “Bounded model checking.” In Ad- vances in Computers, volume 58. Academic Press, 2003.

Python Module Index

p

pynusmv.__init__, 17
pynusmv.be.__init__, 83
pynusmv.be.encoder, 83
pynusmv.be.expression, 88
pynusmv.be.fsm, 91
pynusmv.be.manager, 93
pynusmv.bmc.__init__, 95
pynusmv.bmc.glob, 96
pynusmv.bmc.invarspec, 97
pynusmv.bmc.ltlspec, 102
pynusmv.bmc.utils, 111
pynusmv.collections, 18
pynusmv.dd, 23
pynusmv.exception, 28
pynusmv.fsm, 30
pynusmv.glob, 38
pynusmv.init, 40
pynusmv.mc, 41
pynusmv.model, 44
pynusmv.node, 51
pynusmv.parser, 64
pynusmv.prop, 66
pynusmv.sat, 69
pynusmv.sexp.__init__, 119
pynusmv.sexp.fsm, 119
pynusmv.trace, 72
pynusmv.utils, 77
pynusmv.wff, 81

Index

A

Abf (class in pynusmv.node), 61
Abg (class in pynusmv.node), 61
Abu (class in pynusmv.node), 61
Add (class in pynusmv.model), 46
add() (pynusmv.sat.SatSolver method), 70
add_to_group() (pynusmv.sat.SatIncSolver method), 71
Af (class in pynusmv.node), 61
af() (in module pynusmv.prop), 68
Ag (class in pynusmv.node), 61
ag() (in module pynusmv.prop), 68
ALL (pynusmv.be.encoder.BeVarType attribute), 83
all_loopbacks() (in module pynusmv.bmc.utils), 111
And (class in pynusmv.model), 47
And (class in pynusmv.node), 58
and_() (in module pynusmv.prop), 68
and_() (pynusmv.be.expression.Be method), 89
and_() (pynusmv.dd.BDD method), 25
and_() (pynusmv.node.Expression method), 54
and_() (pynusmv.wff.Wff method), 82
and_interval() (pynusmv.be.encoder.BeEnc method), 87
append() (pynusmv.collections.NodeList method), 21
append_step() (pynusmv.trace.Trace method), 75
apply_inlining() (in module pynusmv.bmc.utils), 114
apply_inlining_for_incremental_algo() (in module pynusmv.bmc.utils), 114
apply_synchronous_product() (pynusmv.be.fsm.BeFsm method), 92
ARGS (pynusmv.model.Module attribute), 50
arguments (pynusmv.node.Modtype attribute), 53
Array (class in pynusmv.model), 48
Array (class in pynusmv.node), 57
array (pynusmv.node.Array attribute), 57
array() (pynusmv.node.Expression method), 54
ArrayAccess (class in pynusmv.model), 44
ArrayDef (class in pynusmv.node), 52
ArrayExpr (class in pynusmv.model), 47
ArrayType (class in pynusmv.node), 53
Assign (class in pynusmv.node), 51
assign() (pynusmv.trace.TraceStep method), 76
Assigns (class in pynusmv.model), 48
Assoc (class in pynusmv.collections), 22
at_index (pynusmv.be.encoder.BeEnc attribute), 85
Atom (class in pynusmv.node), 56
atom() (in module pynusmv.prop), 69
Attime (class in pynusmv.node), 62
Au (class in pynusmv.node), 60
au() (in module pynusmv.mc), 42
au() (in module pynusmv.prop), 68
available_solvers() (pynusmv.sat.SatSolverFactory static method), 69
Aw (class in pynusmv.node), 60
aw() (in module pynusmv.prop), 69
Ax (class in pynusmv.node), 61
ax() (in module pynusmv.prop), 68

B

BACKWARD (pynusmv.bmc.invarspec.InvarClosureStrategy attribute), 99
BDD (class in pynusmv.dd), 24
Bdd (class in pynusmv.node), 62
bdd_encoding() (in module pynusmv.glob), 39
BDDDumpFormatError, 30
BddEnc (class in pynusmv.fsm), 34
bddEnc (pynusmv.fsm.BddFsm attribute), 30
BddFsm (class in pynusmv.fsm), 30
bddFsm (pynusmv.prop.Prop attribute), 66
BDDList (class in pynusmv.dd), 26
BddTrans (class in pynusmv.fsm), 33
Be (class in pynusmv.be.expression), 88
be_index_to_cnf_literal() (pynusmv.be.manager.BeManager method), 94
be_index_to_literal() (pynusmv.be.manager.BeManager method), 94
be_index_to_var() (pynusmv.be.manager.BeManager method), 93
be_literal_to_cnf_literal() (pynusmv.be.manager.BeManager method), 94

be_literal_to_index() (pynusmv.be.manager.BeManager method), 94
be_var_to_index() (pynusmv.be.manager.BeManager method), 93
BeCnf (class in pynusmv.be.expression), 90
BeEnc (class in pynusmv.be.encoder), 84
BeFsm (class in pynusmv.be.fsm), 91
beFsm (pynusmv.prop.Prop attribute), 66
BeManager (class in pynusmv.be.manager), 93
BeRbcManager (class in pynusmv.be.manager), 95
BeVarType (class in pynusmv.be.encoder), 83
BeWrongVarType, 83
Bit (class in pynusmv.node), 62
bit_selection() (pynusmv.node.Expression method), 55
BitSelection (class in pynusmv.model), 45
BitSelection (class in pynusmv.node), 59
bmc_exit() (in module pynusmv.bmc.glob), 96
bmc_setup() (in module pynusmv.bmc.glob), 96
BmcModel (class in pynusmv.bmc.utils), 115
BmcSupport (class in pynusmv.bmc.glob), 97
bool() (pynusmv.node.Expression method), 55
Boolean (class in pynusmv.model), 47
Boolean (class in pynusmv.node), 52
booleanFsm (pynusmv.prop.Prop attribute), 66
BoolSexpFsm (class in pynusmv.sexp.fsm), 120
bounded_semantics() (in module pynusmv.bmc.ltlspec), 105
bounded_semantics_all_loops() (in module pynusmv.bmc.ltlspec), 107
bounded_semantics_all_loops_optimisation_depth1() (in module pynusmv.bmc.ltlspec), 107
bounded_semantics_at_offset() (in module pynusmv.bmc.ltlspec), 109
bounded_semantics_single_loop() (in module pynusmv.bmc.ltlspec), 106
bounded_semantics_with_loop_at_offset() (in module pynusmv.bmc.ltlspec), 109
bounded_semantics_without_loop() (in module pynusmv.bmc.ltlspec), 105
bounded_semantics_without_loop_at_offset() (in module pynusmv.bmc.ltlspec), 108
build_boolean_model() (in module pynusmv.glob), 39
build_flat_model() (in module pynusmv.glob), 39
build_master_be_fsm() (in module pynusmv.bmc.glob), 96
build_model() (in module pynusmv.glob), 39
by_expr (pynusmv.be.encoder.BeEnc attribute), 85
by_name (pynusmv.be.encoder.BeEnc attribute), 85

C

can_declare_var() (pynusmv.fsm.SymbTable method), 37
car (pynusmv.node.Node attribute), 51
car (pynusmv.prop.Spec attribute), 67
Case (class in pynusmv.model), 45
Case (class in pynusmv.node), 57
CastBool (class in pynusmv.node), 59
castbool() (pynusmv.node.Expression method), 55
CastSigned (class in pynusmv.node), 59
castsigned() (pynusmv.node.Expression method), 55
CastToint (class in pynusmv.node), 60
casttoint() (pynusmv.node.Expression method), 55
CastUnsigned (class in pynusmv.node), 59
castunsigned() (pynusmv.node.Expression method), 55
CastWord1 (class in pynusmv.node), 59
castword1() (pynusmv.node.Expression method), 55
cdr (pynusmv.node.Node attribute), 51
cdr (pynusmv.prop.Spec attribute), 67
check_consistency() (in module pynusmv.bmc.utils), 111
check_ctl_spec() (in module pynusmv.mc), 41
check_explain_ltl_spec() (in module pynusmv.mc), 41
check_invar_een_sorensson() (in module pynusmv.bmc.invarspec), 98
check_invar_incrementally_dual() (in module pynusmv.bmc.invarspec), 99
check_invar_incrementally_falsification() (in module pynusmv.bmc.invarspec), 100
check_invar_incrementally_zigzag() (in module pynusmv.bmc.invarspec), 99
check_invar_induction() (in module pynusmv.bmc.invarspec), 97
check_ltl() (in module pynusmv.bmc.ltlspec), 102
check_ltl_incrementally() (in module pynusmv.bmc.ltlspec), 103
check_ltl_spec() (in module pynusmv.mc), 41
clauses_list (pynusmv.be.expression.BeCnf attribute), 90
clauses_number (pynusmv.be.expression.BeCnf attribute), 91
clear() (pynusmv.collections.Assoc method), 23
clear() (pynusmv.collections.Slist method), 20
cnf_literal_to_be_literal() (pynusmv.be.manager.BeManager method), 94
cnf_literal_to_index() (pynusmv.be.manager.BeManager method), 94
cnf_to_be_model() (pynusmv.be.manager.BeManager method), 94
Colon (class in pynusmv.node), 63
Comma (class in pynusmv.node), 63
COMMENT (pynusmv.model.Module attribute), 50
Comment() (in module pynusmv.model), 44
Compassion (class in pynusmv.model), 49
Compassion (class in pynusmv.node), 51
compassion (pynusmv.node.FlatHierarchy attribute), 64
compassion (pynusmv.sexp.fsm.SexpFsm attribute), 120
Compid (class in pynusmv.node), 62
Compute (class in pynusmv.node), 52
compute_model() (in module pynusmv.glob), 40
Compwff (class in pynusmv.node), 62
Concat (class in pynusmv.model), 46

concat() (pynusmv.node.Expression method), 55
concat() (pynusmv.trace.Trace method), 73
Concatenation (class in pynusmv.node), 59
concatenation() (pynusmv.node.Expression method), 55
condition (pynusmv.node.Ifthenelse attribute), 57
Cons (class in pynusmv.node), 62
CONSTANT_EXPR (pynusmv.bmc.utils.OperatorType attribute), 115
Constants (class in pynusmv.model), 48
Constants (class in pynusmv.node), 52
Constraint (class in pynusmv.node), 63
Context (class in pynusmv.node), 57
context (pynusmv.node.Context attribute), 57
Conversion (class in pynusmv.collections), 18
Conversion (class in pynusmv.model), 45
convert_relative_loop_to_absolute() (in module pynusmv.bmc.utils), 112
copy() (pynusmv.be.fsm.BeFsm method), 92
copy() (pynusmv.collections.Assoc method), 23
copy() (pynusmv.collections.NodeList method), 21
copy() (pynusmv.collections.Slist method), 19
copy() (pynusmv.sexp.fsm.SexpFsm method), 119
Count (class in pynusmv.model), 45
Count (class in pynusmv.node), 60
count() (pynusmv.collections.NodeList method), 21
count_inputs() (pynusmv.fsm.BddFsm method), 32
count_states() (pynusmv.fsm.BddFsm method), 32
count_states_inputs() (pynusmv.fsm.BddFsm method), 32
COUNTER_EXAMPLE (pynusmv.trace.TraceType attribute), 72
create() (pynusmv.be.fsm.BeFsm static method), 91
create() (pynusmv.sat.SatSolverFactory static method), 70
create() (pynusmv.trace.Trace static method), 72
create_from_sexp() (pynusmv.be.fsm.BeFsm static method), 92
create_group() (pynusmv.sat.SatIncSolver method), 71
create_layer() (pynusmv.fsm.SymbTable method), 36
Ctlwff (class in pynusmv.node), 62
Cube (class in pynusmv.dd), 27
cube_for_inputs_vars() (pynusmv.fsm.BddEnc method), 35
cube_for_state_vars() (pynusmv.fsm.BddEnc method), 35
CURR (pynusmv.be.encoder.BeVarType attribute), 83
curr_variables (pynusmv.be.encoder.BeEnc attribute), 85
CustomExpression (class in pynusmv.node), 60

D

DA_VINCI (pynusmv.bmc.utils.DumpType attribute), 116
DDef (class in pynusmv.node), 63
DDManager (class in pynusmv.dd), 27
DDmanager (pynusmv.fsm.BddEnc attribute), 34
deadlock_states (pynusmv.fsm.BddFsm attribute), 31
Declaration (class in pynusmv.node), 63
declare_frozen_var() (pynusmv.fsm.SymbTable method), 37
declare_input_var() (pynusmv.fsm.SymbTable method), 37
declare_state_var() (pynusmv.fsm.SymbTable method), 37
declare_var() (pynusmv.fsm.SymbTable method), 38
decode_sat_model() (pynusmv.be.encoder.BeEnc method), 88
decode_value() (pynusmv.be.encoder.BeEnc method), 87
decorate() (pynusmv.wff.Wff static method), 81
Def (class in pynusmv.model), 49
Define (class in pynusmv.node), 52
definedVars (pynusmv.fsm.BddEnc attribute), 35
Defines (class in pynusmv.model), 48
deinit_nusmv() (in module pynusmv.init), 41
deleter() (pynusmv.utils.indexed static method), 79
depth (pynusmv.wff.Wff attribute), 81
description (pynusmv.trace.Trace attribute), 73
destroy_group() (pynusmv.sat.SatIncSolver method), 71
DFVar (class in pynusmv.node), 63
diff() (pynusmv.dd.BDD method), 25
diff() (pynusmv.dd.Cube method), 27
DIMACS (pynusmv.bmc.utils.DumpType attribute), 116
disable_dynamic_reordering() (in module pynusmv.dd), 23
Div (class in pynusmv.model), 46
DIVar (class in pynusmv.node), 63
Divide (class in pynusmv.node), 58
divide() (pynusmv.node.Expression method), 54
Dot (class in pynusmv.model), 44
Dot (class in pynusmv.node), 58
dot() (pynusmv.node.Expression method), 54
dump() (pynusmv.fsm.BddEnc method), 35
dump_davinci() (pynusmv.be.manager.BeManager method), 94
dump_dimacs() (in module pynusmv.bmc.invarspec), 102
dump_dimacs() (in module pynusmv.bmc.ltlspec), 111
dump_dimacs_filename() (in module pynusmv.bmc.invarspec), 101
dump_dimacs_filename() (in module pynusmv.bmc.ltlspec), 110
dump_gdl() (pynusmv.be.manager.BeManager method), 95
dump_problem() (in module pynusmv.bmc.utils), 117
dump_sexpr() (pynusmv.be.manager.BeManager method), 95
DumpType (class in pynusmv.bmc.utils), 116
dup() (pynusmv.dd.BDD method), 25
DVar (class in pynusmv.node), 63
dynamic_reordering_enabled() (in module pynusmv.dd), 24

E

Ebf (class in pynusmv.node), 61
Ebg (class in pynusmv.node), 61
Ebu (class in pynusmv.node), 61
Ef (class in pynusmv.node), 61
ef() (in module pynusmv.mc), 42
ef() (in module pynusmv.prop), 68
Eg (class in pynusmv.node), 61
eg() (in module pynusmv.mc), 42
eg() (in module pynusmv.prop), 68
elementtype (pynusmv.node.ArrayType attribute), 53
empty() (pynusmv.collections.Assoc static method), 22
empty() (pynusmv.collections.NodeList static method), 21
empty() (pynusmv.collections.Slist static method), 19
enable_dynamic_reordering() (in module pynusmv.dd), 23
encode_to_bits() (pynusmv.be.encoder.BeEnc method), 87
encode_variables() (in module pynusmv.glob), 39
encode_variables_for_layers() (in module pynusmv.glob), 39
encoding (pynusmv.be.fsm.BeFsm attribute), 92
END (pynusmv.trace.TraceType attribute), 72
entailed() (pynusmv.dd.BDD method), 25
Eqdef (class in pynusmv.node), 62
Equal (class in pynusmv.model), 46
Equal (class in pynusmv.node), 58
equal() (pynusmv.dd.BDD method), 24
equal() (pynusmv.node.Expression method), 54
equals() (pynusmv.trace.Trace method), 74
ERROR (pynusmv.be.encoder.BeVarType attribute), 83
errors (pynusmv.exception.NuSMVParsingError attribute), 29
Eu (class in pynusmv.node), 60
eu() (in module pynusmv.mc), 42
eu() (in module pynusmv.prop), 68
eval_ctl_spec() (in module pynusmv.mc), 42
eval_simple_expression() (in module pynusmv.mc), 41
eventually() (pynusmv.wff.Wff method), 82
Ew (class in pynusmv.node), 60
ew() (in module pynusmv.prop), 68
Ex (class in pynusmv.node), 61
ex() (in module pynusmv.mc), 42
ex() (in module pynusmv.prop), 68
EXECUTION (pynusmv.trace.TraceType attribute), 72
explain() (in module pynusmv.mc), 43
explainEG() (in module pynusmv.mc), 43
explainEU() (in module pynusmv.mc), 43
explainEX() (in module pynusmv.mc), 43
expr (pynusmv.prop.Prop attribute), 66
exprcore (pynusmv.prop.Prop attribute), 66
Expression (class in pynusmv.node), 54
expression (pynusmv.node.CastBool attribute), 59
expression (pynusmv.node.CastSigned attribute), 59
expression (pynusmv.node.CastToint attribute), 60
expression (pynusmv.node.CastUnsigned attribute), 59
expression (pynusmv.node.CastWord1 attribute), 59
expression (pynusmv.node.Context attribute), 57
expression (pynusmv.node.Next attribute), 58
expression (pynusmv.node.Not attribute), 58
expression (pynusmv.node.Uminus attribute), 58
expression (pynusmv.node.Wsizeof attribute), 60
Extend (class in pynusmv.node), 59
extend() (pynusmv.collections.NodeList method), 21
extend() (pynusmv.collections.Slist method), 20
extend() (pynusmv.node.Expression method), 55

F

Failure (class in pynusmv.node), 55
fair_states (pynusmv.fsm.BddFsm attribute), 31
Fairness (class in pynusmv.model), 49
Fairness (class in pynusmv.node), 51
fairness() (pynusmv.bmc.utils.BmcModel method), 116
fairness_constraint() (in module pynusmv.bmc.utils), 113
fairness_constraints (pynusmv.fsm.BddFsm attribute), 31
fairness_iterator() (pynusmv.be.fsm.BeFsm method), 92
fairness_list (pynusmv.be.fsm.BeFsm attribute), 92
false (pynusmv.node.Ifthenelse attribute), 57
false() (in module pynusmv.prop), 67
false() (pynusmv.be.expression.Be static method), 89
false() (pynusmv.dd.BDD static method), 26
false() (pynusmv.wff.Wff static method), 81
Falseexp (class in pynusmv.model), 45
Falseexp (class in pynusmv.node), 55
fill_counter_example() (in module pynusmv.bmc.utils), 118
find_hierarchy() (in module pynusmv.node), 51
fixpoint() (in module pynusmv.utils), 77
flat_hierarchy() (in module pynusmv.glob), 40
FlatHierarchy (class in pynusmv.node), 63
flatten_hierarchy() (in module pynusmv.glob), 38
for_name() (pynusmv.utils.StdioFile static method), 78
forall() (pynusmv.dd.BDD method), 26
force_loopback() (pynusmv.trace.TraceStep method), 76
force_variables_ordering() (pynusmv.fsm.BddEnc method), 35
formula_literal (pynusmv.be.expression.BeCnf attribute), 91
forsome() (pynusmv.dd.BDD method), 25
FORWARD (pynusmv.bmc.invarspec.InvarClosureStrategy attribute), 99
freeze() (pynusmv.trace.Trace method), 74
from_bdd() (pynusmv.dd.Inputs static method), 27
from_bdd() (pynusmv.dd.State static method), 26
from_dict() (pynusmv.collections.Assoc static method), 22

from_filename() (pynusmv.fsm.BddFsm static method), 33
from_list() (pynusmv.collections.NodeList static method), 21
from_list() (pynusmv.collections.Slist static method), 19
from_modules() (pynusmv.fsm.BddFsm static method), 33
from_node() (pynusmv.collections.NodeIterator static method), 20
from_nusmv_errors_list() (pynusmv.exception.NuSMVParsingError static method), 29
from_pointer() (pynusmv.collections.NodeIterator static method), 20
from_ptr() (pynusmv.node.Node static method), 51
from_string() (pynusmv.fsm.BddFsm static method), 33
from_string() (pynusmv.fsm.BddTrans class method), 34
from_string() (pynusmv.node.Expression static method), 55
from_string() (pynusmv.node.Identifier static method), 60
from_trans() (pynusmv.fsm.BddTrans class method), 33
from_tuple() (pynusmv.dd.BDDList static method), 26
FROZEN (pynusmv.be.encoder.BeVarType attribute), 83
frozen_variables (pynusmv.be.encoder.BeEnc attribute), 85
Frozenvar (class in pynusmv.node), 52
FrozenVariables (class in pynusmv.model), 48
FVar (class in pynusmv.model), 49

G

GDL (pynusmv.bmc.utils.DumpType attribute), 116
Ge (class in pynusmv.model), 47
Ge (class in pynusmv.node), 58
ge() (pynusmv.node.Expression method), 54
generate_base_step() (in module pynusmv.bmc.invarspec), 101
generate_counter_example() (in module pynusmv.bmc.utils), 118
generate_inductive_step() (in module pynusmv.bmc.invarspec), 101
generate_invar_problem() (in module pynusmv.bmc.invarspec), 100
generate_ltl_problem() (in module pynusmv.bmc.ltlspec), 104
get_inputs_between_states() (pynusmv.fsm.BddFsm method), 32
get_prop_at_index() (pynusmv.prop.PropDb method), 67
get_props_of_type() (pynusmv.prop.PropDb method), 67
get_size() (pynusmv.prop.PropDb method), 67
get_str_values() (pynusmv.dd.Inputs method), 27
get_str_values() (pynusmv.dd.State method), 26
get_str_values() (pynusmv.dd.StateInputs method), 27
get_variable_type() (pynusmv.fsm.SymbTable method), 36
get_variables_ordering() (pynusmv.fsm.BddEnc method), 35
getter() (pynusmv.utils.indexed static method), 78
global_master_instance() (pynusmv.be.fsm.BeFsm static method), 91
global_singleton_instance() (pynusmv.be.encoder.BeEnc static method), 84
globally() (pynusmv.wff.Wff method), 82
go_bmc() (in module pynusmv.bmc.glob), 97
Goto (class in pynusmv.node), 63
Gt (class in pynusmv.model), 47
Gt (class in pynusmv.node), 58
gt() (pynusmv.node.Expression method), 54

H

handle (pynusmv.utils.StdioFile attribute), 78
hierarchy (pynusmv.sexp.fsm.SexpFsm attribute), 119
historically() (pynusmv.wff.Wff method), 82

I

id (pynusmv.trace.Trace attribute), 73
Identifier (class in pynusmv.model), 44
Identifier (class in pynusmv.node), 60
if_then_else() (pynusmv.be.expression.Be static method), 89
Iff (class in pynusmv.model), 47
Iff (class in pynusmv.node), 57
iff() (in module pynusmv.prop), 68
iff() (pynusmv.be.expression.Be method), 90
iff() (pynusmv.dd.BDD method), 25
iff() (pynusmv.node.Expression method), 54
iff() (pynusmv.wff.Wff method), 82
Ifthenelse (class in pynusmv.node), 57
ifthenelse() (pynusmv.node.Expression method), 54
Implies (class in pynusmv.model), 47
Implies (class in pynusmv.node), 57
implies() (pynusmv.node.Expression method), 54
implies() (pynusmv.wff.Wff method), 82
imply() (in module pynusmv.prop), 68
imply() (pynusmv.be.expression.Be method), 90
imply() (pynusmv.dd.BDD method), 25
In (class in pynusmv.model), 46
in_() (pynusmv.node.Expression method), 54
in_context() (pynusmv.node.Expression method), 54
index (pynusmv.node.Array attribute), 57
indexed (class in pynusmv.utils), 78
Init (class in pynusmv.model), 48
Init (class in pynusmv.node), 51
init (pynusmv.be.fsm.BeFsm attribute), 92
init (pynusmv.bmc.utils.BmcModel attribute), 115
init (pynusmv.fsm.BddFsm attribute), 30
init (pynusmv.node.FlatHierarchy attribute), 64
init (pynusmv.sexp.fsm.SexpFsm attribute), 119
init_nusmv() (in module pynusmv.init), 40

inline() (pynusmv.be.expression.Be method), 90
INPUT (pynusmv.be.encoder.BeVarType attribute), 83
input (pynusmv.sexp.fsm.SexpFsm attribute), 119
input_variables (pynusmv.be.encoder.BeEnc attribute), 85
input_vars (pynusmv.trace.Trace attribute), 75
Inputs (class in pynusmv.dd), 27
inputs_constraints (pynusmv.fsm.BddFsm attribute), 31
inputsCube (pynusmv.fsm.BddEnc attribute), 34
inputsMask (pynusmv.fsm.BddEnc attribute), 34
inputsVars (pynusmv.fsm.BddEnc attribute), 35
InputVariables (class in pynusmv.model), 48
ins_policies (pynusmv.fsm.SymbTable attribute), 36
insert_after() (pynusmv.collections.NodeList method), 22
insert_at() (pynusmv.collections.NodeList method), 22
insert_before() (pynusmv.collections.NodeList method), 21
IntConversion (class in pynusmv.collections), 19
Integer (class in pynusmv.node), 53
INTERNAL_ERROR (pynusmv.sat.SatSolverResult attribute), 69
intersected() (pynusmv.dd.BDD method), 25
intersection() (pynusmv.dd.BDD method), 25
intersection() (pynusmv.dd.Cube method), 27
Invar (class in pynusmv.model), 49
Invar (class in pynusmv.node), 51
invar (pynusmv.bmc.utils.BmcModel attribute), 115
invar (pynusmv.node.FlatHierarchy attribute), 64
invar_dual_forward_unrolling() (pynusmv.bmc.utils.BmcModel method), 116
InvarClosureStrategy (class in pynusmv.bmc.invarspec), 98
invariants (pynusmv.be.fsm.BeFsm attribute), 92
invariants (pynusmv.sexp.fsm.SexpFsm attribute), 119
Invarspec (class in pynusmv.node), 52
is_all_loopbacks() (in module pynusmv.bmc.utils), 111
is_binary_operator() (in module pynusmv.bmc.utils), 115
is_boolean (pynusmv.sexp.fsm.SexpFsm attribute), 119
is_complete() (pynusmv.trace.Trace method), 75
is_constant() (pynusmv.be.expression.Be method), 89
is_constant_expr() (in module pynusmv.bmc.utils), 114
is_empty (pynusmv.trace.Trace attribute), 73
is_empty() (pynusmv.collections.Slist method), 20
is_false() (pynusmv.be.expression.Be method), 89
is_false() (pynusmv.dd.BDD method), 25
is_frozen (pynusmv.trace.Trace attribute), 74
is_frozen_var() (pynusmv.fsm.SymbTable method), 38
is_input_var() (pynusmv.fsm.SymbTable method), 38
is_loopback (pynusmv.trace.TraceStep attribute), 76
is_no_loopback() (in module pynusmv.bmc.utils), 111
is_nusmv_init() (in module pynusmv.init), 41
is_past_operator() (in module pynusmv.bmc.utils), 114
is_registered (pynusmv.trace.Trace attribute), 73
is_state_var() (pynusmv.fsm.SymbTable method), 38
is_syntactically_universal (pynusmv.sexp.fsm.SexpFsm attribute), 120
is_thawed (pynusmv.trace.Trace attribute), 74
is_true() (pynusmv.be.expression.Be method), 89
is_true() (pynusmv.dd.BDD method), 25
is_variable() (in module pynusmv.bmc.utils), 114
is_volatile (pynusmv.trace.Trace attribute), 73
Isa (class in pynusmv.node), 52
isnot_false() (pynusmv.dd.BDD method), 25
isnot_true() (pynusmv.dd.BDD method), 25
Ite (class in pynusmv.model), 47
iterator() (pynusmv.be.encoder.BeEnc method), 85
IVar (class in pynusmv.model), 49
Ivar (class in pynusmv.node), 52

J

Justice (class in pynusmv.model), 49
Justice (class in pynusmv.node), 51
justice (pynusmv.node.FlatHierarchy attribute), 64
justice (pynusmv.sexp.fsm.SexpFsm attribute), 119

K

kind (pynusmv.node.Failure attribute), 55

L

Lambda (class in pynusmv.node), 63
language_contains() (pynusmv.trace.Trace method), 75
last_solving_time (pynusmv.sat.SatSolver attribute), 70
layer_names (pynusmv.fsm.SymbTable attribute), 36
Le (class in pynusmv.model), 47
Le (class in pynusmv.node), 58
le() (pynusmv.node.Expression method), 54
Leaf (class in pynusmv.node), 55
length (pynusmv.node.SignedWord attribute), 52
length (pynusmv.node.UnsignedWord attribute), 52
length (pynusmv.trace.Trace attribute), 73
leq() (pynusmv.dd.BDD method), 25
LITERAL (pynusmv.bmc.utils.OperatorType attribute), 115
load() (in module pynusmv.glob), 38
load() (pynusmv.fsm.BddEnc method), 36
loop_condition() (in module pynusmv.bmc.utils), 112
loop_from_string() (in module pynusmv.bmc.utils), 111
Lrotate (class in pynusmv.node), 59
lrotate() (pynusmv.node.Expression method), 54
LShift (class in pynusmv.model), 46
Lshift (class in pynusmv.node), 59
lshift() (pynusmv.node.Expression method), 54
Lt (class in pynusmv.model), 47
Lt (class in pynusmv.node), 58
lt() (pynusmv.node.Expression method), 54
Ltlspec (class in pynusmv.node), 51
Ltlwff (class in pynusmv.node), 62

M

make_negated_nnf_boolean_wff() (in module pynusmv.bmc.utils), 114
make_nnf_boolean_wff() (in module pynusmv.bmc.utils), 114
manager (pynusmv.be.encoder.BeEnc attribute), 84
master (pynusmv.prop.PropDb attribute), 67
master_be_fsm() (in module pynusmv.bmc.glob), 96
max_time (pynusmv.be.encoder.BeEnc attribute), 84
max_var_index (pynusmv.be.expression.BeCnf attribute), 91
Maxu (class in pynusmv.node), 61
members (pynusmv.model.Module attribute), 51
MEMOUT (pynusmv.sat.SatSolverResult attribute), 69
message (pynusmv.node.Failure attribute), 55
minimize() (pynusmv.dd.BDD method), 26
Minu (class in pynusmv.node), 61
Minus (class in pynusmv.model), 46
Minus (class in pynusmv.node), 58
minus() (pynusmv.node.Expression method), 54
Mirror (class in pynusmv.node), 62
MissingManagerError, 28
Mod (class in pynusmv.model), 46
Mod (class in pynusmv.node), 58
mod() (pynusmv.node.Expression method), 54
model (pynusmv.sat.SatSolver attribute), 71
Modtype (class in pynusmv.model), 48
Modtype (class in pynusmv.node), 53
Module (class in pynusmv.model), 49
Module (class in pynusmv.node), 51
monolithic (pynusmv.fsm.BddTrans attribute), 33
move_to_permanent() (pynusmv.sat.SatIncSolver method), 71
Mult (class in pynusmv.model), 46

N

NAME (pynusmv.model.Module attribute), 51
name (pynusmv.node.Atom attribute), 56
name (pynusmv.node.Declaration attribute), 63
name (pynusmv.node.Modtype attribute), 53
name (pynusmv.prop.Prop attribute), 66
name (pynusmv.sat.SatSolver attribute), 70
need_rewriting (pynusmv.prop.Prop attribute), 67
NEGATIVE (pynusmv.sat.Polarity attribute), 69
Next (class in pynusmv.model), 45
Next (class in pynusmv.node), 58
NEXT (pynusmv.be.encoder.BeVarType attribute), 83
next() (pynusmv.node.Expression method), 54
next_() (pynusmv.wff.Wff method), 82
next_times() (pynusmv.wff.Wff method), 82
next_variables (pynusmv.be.encoder.BeEnc attribute), 86
Nextwff (class in pynusmv.node), 62
Nfunction (class in pynusmv.node), 63
no_loopback() (in module pynusmv.bmc.utils), 111
Node (class in pynusmv.node), 51
NodeConversion (class in pynusmv.collections), 19
NodeIterator (class in pynusmv.collections), 20
NodeList (class in pynusmv.collections), 20
NodeListIter (class in pynusmv.collections), 22
NONE (pynusmv.bmc.utils.DumpType attribute), 116
normalize_name() (pynusmv.sat.SatSolverFactory static method), 69
Not (class in pynusmv.model), 45
Not (class in pynusmv.node), 58
not_() (in module pynusmv.prop), 67
not_() (pynusmv.be.expression.Be method), 89
not_() (pynusmv.dd.BDD method), 25
not_() (pynusmv.node.Expression method), 54
not_() (pynusmv.wff.Wff method), 82
NOT_SET (pynusmv.sat.Polarity attribute), 69
NotEqual (class in pynusmv.model), 46
Notequal (class in pynusmv.node), 58
notequal() (pynusmv.node.Expression method), 54
num_of_frozen_vars (pynusmv.be.encoder.BeEnc attribute), 84
num_of_input_vars (pynusmv.be.encoder.BeEnc attribute), 85
num_of_state_vars (pynusmv.be.encoder.BeEnc attribute), 84
num_of_vars (pynusmv.be.encoder.BeEnc attribute), 84
Number (class in pynusmv.node), 56
NumberExp (class in pynusmv.node), 56
NumberFrac (class in pynusmv.node), 56
NumberReal (class in pynusmv.node), 56
NumberSignedWord (class in pynusmv.node), 56
NumberUnsignedWord (class in pynusmv.node), 56
NumberWord (class in pynusmv.model), 45
NumericalConst (class in pynusmv.model), 44
NuSMVBddPickingError, 29
NuSMVBeFsmMasterInstanceNotInitializedError, 30
NuSMVBmcAlreadyInitializedError, 30
NuSMVCannotFlattenError, 28
NuSMVFlatModelAlreadyBuiltError, 28
NuSMVFlatteningError, 29
NuSmvIllegalTraceStateError, 30
NuSMVInitError, 29
NuSMVLexerError, 28
NuSMVModelAlreadyBuiltError, 29
NuSMVModelAlreadyEncodedError, 28
NuSMVModelAlreadyFlattenedError, 28
NuSMVModelAlreadyReadError, 28
NuSMVModuleError, 29
NuSMVNeedBooleanModelError, 30
NuSMVNeedFlatHierarchyError, 28
NuSMVNeedFlatModelError, 29
NuSMVNeedVariablesEncodedError, 29
NuSMVNoReadModelError, 28
NuSMVParserError, 29

NuSMVParsingError, 29
NuSMVSymbTableError, 29
NuSMVTypeCheckingError, 29
NuSMVWffError, 30

O

once() (pynusmv.wff.Wff method), 82
operator_class() (in module pynusmv.bmc.utils), 115
OperatorType (class in pynusmv.bmc.utils), 115
OpFuture (class in pynusmv.node), 61
OpGlobal (class in pynusmv.node), 61
OpHistorical (class in pynusmv.node), 62
OpNext (class in pynusmv.node), 61
opnext() (pynusmv.wff.Wff method), 82
OpNotprecnot (class in pynusmv.node), 62
opnotprecnot() (pynusmv.wff.Wff method), 82
OpOnce (class in pynusmv.node), 62
OpPrec (class in pynusmv.node), 62
opprec() (pynusmv.wff.Wff method), 82
Or (class in pynusmv.model), 47
Or (class in pynusmv.node), 57
or_() (in module pynusmv.prop), 68
or_() (pynusmv.be.expression.Be method), 89
or_() (pynusmv.dd.BDD method), 25
or_() (pynusmv.node.Expression method), 54
or_() (pynusmv.wff.Wff method), 82
or_interval() (pynusmv.be.encoder.BeEnc method), 87
original_problem (pynusmv.be.expression.BeCnf attribute), 90

P

parse_ctl_spec() (in module pynusmv.parser), 65
parse_identifier() (in module pynusmv.parser), 65
parse_ltl_spec() (in module pynusmv.parser), 65
parse_next_expression() (in module pynusmv.parser), 65
parse_simple_expression() (in module pynusmv.parser), 64
parseAllString() (in module pynusmv.parser), 65
path() (pynusmv.bmc.utils.BmcModel method), 116
permanent_group (pynusmv.sat.SatSolver attribute), 70
pick_all_inputs() (pynusmv.fsm.BddFsm method), 32
pick_all_states() (pynusmv.fsm.BddFsm method), 32
pick_all_states_inputs() (pynusmv.fsm.BddFsm method), 33
pick_one_inputs() (pynusmv.fsm.BddFsm method), 32
pick_one_inputs_random() (pynusmv.fsm.BddFsm method), 32
pick_one_state() (pynusmv.fsm.BddFsm method), 31
pick_one_state_inputs() (pynusmv.fsm.BddFsm method), 32
pick_one_state_inputs_random() (pynusmv.fsm.BddFsm method), 32
pick_one_state_random() (pynusmv.fsm.BddFsm method), 31
Plus (class in pynusmv.node), 58
plus() (pynusmv.node.Expression method), 54
PointerWrapper (class in pynusmv.utils), 77
Polarity (class in pynusmv.sat), 69
polarity() (pynusmv.sat.SatSolver method), 70
pop() (pynusmv.collections.Slist method), 20
POSITIVE (pynusmv.sat.Polarity attribute), 69
post() (pynusmv.fsm.BddFsm method), 31
post() (pynusmv.fsm.BddTrans method), 33
pre() (pynusmv.fsm.BddFsm method), 31
pre() (pynusmv.fsm.BddTrans method), 33
Pred (class in pynusmv.node), 62
PredsList (class in pynusmv.node), 62
prepend() (pynusmv.collections.NodeList method), 21
print_available_solvers() (pynusmv.sat.SatSolverFactory static method), 70
print_counter_example() (in module pynusmv.bmc.utils), 117
print_nodes() (pynusmv.collections.NodeList method), 22
print_stats() (pynusmv.be.expression.BeCnf method), 91
Process (class in pynusmv.node), 53
Prop (class in pynusmv.prop), 66
PROP_CONNECTIVE (pynusmv.bmc.utils.OperatorType attribute), 115
prop_database() (in module pynusmv.glob), 40
PropDb (class in pynusmv.prop), 67
Property (class in pynusmv.node), 60
property() (pynusmv.utils.indexed static method), 79
propTypes (in module pynusmv.prop), 66
Pslspec (class in pynusmv.node), 51
push() (pynusmv.collections.Slist method), 19
pynusmv.__init__ (module), 17
pynusmv.be.__init__ (module), 83
pynusmv.be.encoder (module), 83
pynusmv.be.expression (module), 88
pynusmv.be.fsm (module), 91
pynusmv.be.manager (module), 93
pynusmv.bmc.__init__ (module), 95
pynusmv.bmc.glob (module), 96
pynusmv.bmc.invarspec (module), 97
pynusmv.bmc.ltlspec (module), 102
pynusmv.bmc.utils (module), 111
pynusmv.collections (module), 18
pynusmv.dd (module), 23
pynusmv.exception (module), 28
pynusmv.fsm (module), 30
pynusmv.glob (module), 38
pynusmv.init (module), 40
pynusmv.mc (module), 41
pynusmv.model (module), 44
pynusmv.node (module), 51
pynusmv.parser (module), 64
pynusmv.prop (module), 66

pynusmv.sat (module), 69
pynusmv.sexp.__init__ (module), 119
pynusmv.sexp.fsm (module), 119
pynusmv.trace (module), 72
pynusmv.utils (module), 77
pynusmv.wff (module), 81
PyNuSMVError, 28

R

random_mode (pynusmv.sat.SatSolver attribute), 70
Range (class in pynusmv.model), 48
Range (class in pynusmv.node), 52
RangeConst (class in pynusmv.model), 45
reachable_states (pynusmv.fsm.BddFsm attribute), 31
read() (pynusmv.node.Expression method), 55
Real (class in pynusmv.node), 53
register() (pynusmv.trace.Trace method), 73
Releases (class in pynusmv.node), 61
releases() (pynusmv.wff.Wff method), 82
remove() (pynusmv.collections.Slist method), 20
remove_duplicates() (pynusmv.be.expression.BeCnf method), 91
reorder() (in module pynusmv.dd), 24
reorderings (pynusmv.dd.DDManager attribute), 28
reserve() (pynusmv.be.manager.BeRbcManager method), 95
reset() (pynusmv.be.manager.BeRbcManager method), 95
reset_nusmv() (in module pynusmv.init), 41
resize() (pynusmv.node.Expression method), 55
reverse() (pynusmv.collections.NodeList method), 21
reverse() (pynusmv.collections.Slist method), 19
Rrotate (class in pynusmv.node), 59
rrotate() (pynusmv.node.Expression method), 54
RShift (class in pynusmv.model), 46
Rshift (class in pynusmv.node), 59
rshift() (pynusmv.node.Expression method), 54

S

SatIncProofSolver (class in pynusmv.sat), 72
SatIncSolver (class in pynusmv.sat), 71
SATISFIABLE (pynusmv.sat.SatSolverResult attribute), 69
SatProofSolver (class in pynusmv.sat), 72
SatSolver (class in pynusmv.sat), 70
SatSolverFactory (class in pynusmv.sat), 69
SatSolverResult (class in pynusmv.sat), 69
Scalar (class in pynusmv.model), 48
Scalar (class in pynusmv.node), 53
scalarFsm (pynusmv.prop.Prop attribute), 66
Section (class in pynusmv.node), 51
Self (class in pynusmv.model), 44
Self (class in pynusmv.node), 56
Semi (class in pynusmv.node), 62
Sentinel (class in pynusmv.collections), 20
Set (class in pynusmv.model), 45
Set (class in pynusmv.node), 60
Setin (class in pynusmv.node), 58
setin() (pynusmv.node.Expression method), 54
setter() (pynusmv.utils.indexed static method), 79
SexpFsm (class in pynusmv.sexp.fsm), 119
shift_curr_to_next() (pynusmv.be.encoder.BeEnc method), 86
shift_to_time() (pynusmv.be.encoder.BeEnc method), 86
shift_to_times() (pynusmv.be.encoder.BeEnc method), 87
signed() (pynusmv.node.Expression method), 55
SignedWord (class in pynusmv.node), 52
Simpwff (class in pynusmv.node), 62
SIMULATION (pynusmv.trace.TraceType attribute), 72
Since (class in pynusmv.node), 61
since() (pynusmv.wff.Wff method), 82
size (pynusmv.dd.BDD attribute), 24
size (pynusmv.dd.DDManager attribute), 28
sizeof() (pynusmv.node.Expression method), 55
Slist (class in pynusmv.collections), 19
SlistIterator (class in pynusmv.collections), 20
Smallinit (class in pynusmv.model), 45
Smallinit (class in pynusmv.node), 62
solve() (pynusmv.sat.SatSolver method), 71
solve_all_groups() (pynusmv.sat.SatIncSolver method), 72
solve_groups() (pynusmv.sat.SatIncSolver method), 71
solve_without_groups() (pynusmv.sat.SatIncSolver method), 72
source (pynusmv.model.Module attribute), 51
Spec (class in pynusmv.node), 51
Spec (class in pynusmv.prop), 67
start (pynusmv.node.ArrayType attribute), 53
start (pynusmv.node.BitSelection attribute), 59
start (pynusmv.node.Range attribute), 52
start (pynusmv.node.Twodots attribute), 57
State (class in pynusmv.dd), 26
state_constraints (pynusmv.fsm.BddFsm attribute), 31
state_frozen_vars (pynusmv.trace.Trace attribute), 75
state_vars (pynusmv.trace.Trace attribute), 75
StateInputs (class in pynusmv.dd), 27
statesCube (pynusmv.fsm.BddEnc attribute), 34
statesInputsMask (pynusmv.fsm.BddEnc attribute), 34
statesMask (pynusmv.fsm.BddEnc attribute), 34
stateVars (pynusmv.fsm.BddEnc attribute), 35
status (pynusmv.prop.Prop attribute), 66
stderr() (pynusmv.utils.StdioFile static method), 78
stdin() (pynusmv.utils.StdioFile static method), 78
StdioFile (class in pynusmv.utils), 77
stdout() (pynusmv.utils.StdioFile static method), 78
steps (pynusmv.trace.Trace attribute), 76
stop (pynusmv.node.ArrayType attribute), 53
stop (pynusmv.node.BitSelection attribute), 59

stop (pynusmv.node.Range attribute), 52
stop (pynusmv.node.Twodots attribute), 57
Sub (class in pynusmv.model), 46
Subscript (class in pynusmv.model), 45
successor() (in module pynusmv.bmc.utils), 113
Swconst (class in pynusmv.node), 60
swconst() (pynusmv.node.Expression method), 55
symb_table() (in module pynusmv.glob), 39
SYMBOL_FROZEN_VAR (pynusmv.fsm.SymbTable attribute), 36
SYMBOL_INPUT_VAR (pynusmv.fsm.SymbTable attribute), 36
SYMBOL_STATE_VAR (pynusmv.fsm.SymbTable attribute), 36
symbol_table (pynusmv.be.encoder.BeEnc attribute), 84
symbol_table (pynusmv.sexp.fsm.SexpFsm attribute), 119
symbol_table (pynusmv.trace.Trace attribute), 75
symbols (pynusmv.trace.Trace attribute), 75
symbols_list (pynusmv.sexp.fsm.SexpFsm attribute), 120
SymbTable (class in pynusmv.fsm), 36
symbTable (pynusmv.fsm.BddEnc attribute), 34
symbTable (pynusmv.node.FlatHierarchy attribute), 64
SyntaxError_ (class in pynusmv.node), 62

T

thaw() (pynusmv.trace.Trace method), 74
TIME_OPERATOR (pynusmv.bmc.utils.OperatorType attribute), 115
TIMEOUT (pynusmv.sat.SatSolverResult attribute), 69
Times (class in pynusmv.node), 58
times() (pynusmv.node.Expression method), 54
to_be() (pynusmv.wff.Wff method), 82
to_boolean_wff() (pynusmv.wff.Wff method), 82
to_cnf() (pynusmv.be.expression.Be method), 90
to_negation_normal_form() (pynusmv.wff.Wff method), 81
to_node() (pynusmv.wff.Wff method), 81
to_object() (pynusmv.collections.Conversion method), 18
to_pointer() (pynusmv.collections.Conversion method), 19
to_tuple() (pynusmv.dd.BDDList method), 26
toint() (pynusmv.node.Expression method), 55
top() (pynusmv.collections.Slist method), 20
Trace (class in pynusmv.trace), 72
TraceStep (class in pynusmv.trace), 76
TraceType (class in pynusmv.trace), 72
Trans (class in pynusmv.model), 48
Trans (class in pynusmv.node), 51
trans (pynusmv.be.fsm.BeFsm attribute), 92
trans (pynusmv.bmc.utils.BmcModel attribute), 115
trans (pynusmv.fsm.BddFsm attribute), 31
trans (pynusmv.node.FlatHierarchy attribute), 64
trans (pynusmv.sexp.fsm.SexpFsm attribute), 119
Triggered (class in pynusmv.node), 61
triggered() (pynusmv.wff.Wff method), 82
true (pynusmv.node.Ifthenelse attribute), 57
true() (in module pynusmv.prop), 67
true() (pynusmv.be.expression.Be static method), 88
true() (pynusmv.dd.BDD static method), 26
true() (pynusmv.wff.Wff static method), 81
Trueexp (class in pynusmv.model), 44
Trueexp (class in pynusmv.node), 56
Twodots (class in pynusmv.node), 57
twodots() (pynusmv.node.Expression method), 54
Type (class in pynusmv.node), 52
type (pynusmv.prop.Prop attribute), 66
type (pynusmv.prop.Spec attribute), 67
type (pynusmv.trace.Trace attribute), 73

U

Uminus (class in pynusmv.node), 58
uminus() (pynusmv.node.Expression method), 54
UNAVAILABLE (pynusmv.sat.SatSolverResult attribute), 69
Union (class in pynusmv.model), 46
Union (class in pynusmv.node), 58
union() (pynusmv.dd.BDD method), 25
union() (pynusmv.dd.Cube method), 27
union() (pynusmv.node.Expression method), 54
UNKNOWN_OP (pynusmv.bmc.utils.OperatorType attribute), 115
unregister() (pynusmv.trace.Trace method), 74
unrolling() (pynusmv.bmc.utils.BmcModel method), 115
unrolling_fragment (pynusmv.bmc.utils.BmcModel attribute), 116
UNSATISFIABLE (pynusmv.sat.SatSolverResult attribute), 69
unsigned() (pynusmv.node.Expression method), 55
UnsignedWord (class in pynusmv.node), 52
UNSPECIFIED (pynusmv.trace.TraceType attribute), 72
Until (class in pynusmv.node), 61
until() (pynusmv.wff.Wff method), 82
untimed_variables (pynusmv.be.encoder.BeEnc attribute), 85
update() (in module pynusmv.utils), 77
Uwconst (class in pynusmv.node), 59
uwconst() (pynusmv.node.Expression method), 55

V

value (pynusmv.node.Number attribute), 56
value (pynusmv.node.NumberSignedWord attribute), 56
value (pynusmv.node.NumberUnsignedWord attribute), 56
value (pynusmv.trace.TraceStep attribute), 77
values (pynusmv.node.Case attribute), 57
values (pynusmv.node.Count attribute), 60
values (pynusmv.node.Scalar attribute), 53

values (pynusmv.node.Set attribute), 60
Var (class in pynusmv.model), 49
Var (class in pynusmv.node), 52
Variables (class in pynusmv.model), 48
variables (pynusmv.node.FlatHierarchy attribute), 64
variables_list (pynusmv.sexp.fsm.SexpFsm attribute), 120
vars_list (pynusmv.be.expression.BeCnf attribute), 90
vars_number (pynusmv.be.expression.BeCnf attribute), 90

W

Waread (class in pynusmv.node), 59
waread() (pynusmv.node.Expression method), 55
Wawrite (class in pynusmv.node), 59
wawrite() (pynusmv.node.Expression method), 55
weak_pre() (pynusmv.fsm.BddFsm method), 31
Wff (class in pynusmv.wff), 81
with_capacity() (pynusmv.be.manager.BeRbcManager static method), 95
Word (class in pynusmv.model), 48
Word (class in pynusmv.node), 52
word (pynusmv.node.BitSelection attribute), 59
word1() (pynusmv.node.Expression method), 55
Wordarray (class in pynusmv.node), 53
WordFunction (class in pynusmv.model), 45
Wresize (class in pynusmv.node), 60
wresize() (pynusmv.node.Expression method), 55
write() (pynusmv.node.Expression method), 55
writeonly (class in pynusmv.utils), 78
Wsizeof (class in pynusmv.node), 60
wsizeof() (pynusmv.node.Expression method), 55

X

Xnor (class in pynusmv.model), 47
Xnor (class in pynusmv.node), 57
xnor() (pynusmv.node.Expression method), 54
Xor (class in pynusmv.model), 47
Xor (class in pynusmv.node), 57
xor() (pynusmv.be.expression.Be method), 89
xor() (pynusmv.dd.BDD method), 25
xor() (pynusmv.node.Expression method), 54