UNIVERSITÉ LIBRE DE BRUXELLES

INFO-H412
FORMAL VERTIFICATION

NuSMV - SAT based bounded model-checking

Professor:
Jean-Francois RASKIN

Authors:
Ali DHANANI

Contents

1 Introduction

SAT with bounded model checking is a symbolic representation of a transition system and a property are are up to k number of steps in order to obtain a formula that is satisfiable if there exist a counterexample for the property up to length k. The reason for using SAT solvers for BMC is that SAT solvers are able to solve large formulas than the classical techniques on binary decision tree(BDD). In this report we will be discussing about the summary of Bounded model checking and the entities of model checking with SAT with respect to hardware and software system. After having a clear idea of Bounded model checking, we will be looking towards the verification tool NuSMV. With NuSMV, the idea will be to use the tools in link with the Bounded model checking an to verify some basic model and to see how it works.

2 Bounded Model Checking

Bounded Model Checking can be described as the model checking algorithm that checks the model up to a given length k. If we compare that to a finite state machine we can say that it traverses it up to a fix number of steps and then check whether there had been a violation within the steps. It restricts the length of the path and then carries out the checking for the given bounds in the path. Here the bounds can be increased in the number of steps to accommodate the bounds that were not covered in the previous task. Also the bounds can be increased up to the length of the longest loop free path of the model which means it can cover the entire state space[1][2].

2.1 Encode LTL in a BMC

As we already know that LTL(Linear Temporal Logic) properties can always be given in the form of a path. When talking about encoding of LTL in a BMC, the encodings can be different in terms of compactness, ease of implementation, and of course SAT-solving efficiency. We look into how it can be encoded as follows. We take an LTL of the form G_p, where p is a state predicate. In this property, it can be established that p is a global invariant of the system. The counter part for the property of this kind can be given as a finite path that ends with a state s that satisfies ¬p. We can establish the formula as follows:

$$\exists s_0, ..., s_k I(s_0) \wedge \bigwedge_{i=0}^{k-1} \mathbf{T}(s_i, s_{i+1}) \wedge \neg p(s_k)$$

In the above formula we can see that it contains three conjuncts. The first conjunct, which is I(s_0), which says that state s_0 is one of the initial states. The second conjunct encodes shows of the existence of the transition from s_i to s_{i+1} for each i $\in \{0, ..., k-1\}$. This amounts to creating k replicas of the transition relation T. Finally, the conjunct $\neg p(s_k)$ shows that the state s_k satisfies $\neg p$. Here we can see that there is only one existential quantification and thus it can be said that it corresponds to a propositional satisfiability problem [3]. Even if p holds along all the states from $s_0 tos_k$, but there is no back loop from s_k to a previous state, we cannot conclude that we have found a witness for G_p, since p might not hold at s_{k+1}[4].

2.2 Eventuality and Liveness

The idea here is that whether a particular state property is guaranteed to eventually hold. We will be writing this as F_p in LTL, it is very similar to the G_p in LTL that we had seen

earlier. The idea is that all states on the path satisfy ¬p. It can be written as follows:

$$\exists s_0, ..., s_k I(s_0) \wedge \bigwedge_{i=0}^{k-1} \mathbf{T}(s_i, s_{i+1}) \wedge \bigwedge_{i=0}^{k-1} \neg p(s_k) \wedge \bigvee_{i=0}^{k-1} s_k = s_i$$

This formula can first be converted into propositional logic, and after that result can then be passed to a propositional SAT solver. The formula is converted into a suitable automaton that accepts counterexample paths[3].

For liveness, firstly liveness properties are encoded as FG_p properties. Once that is done, then the approach tries to prove that a witness trace for such a property does not exist. If we talk about the situation of finite-state system, a witness trace to FG_p is an infinite path that ends in a loop as the loop contains a state in which p holds. In a situation where FG_p cannot be satisfied, then the prefix of any path satisfies p only for an inifinite number of times. The idea is to count the number of p that occurs and then check that the count is smaller than the bound k. If p is only satisfied at most k times, then FG_p cannot be satisfied on any initialized path. If at any point the property FG_p does not hold for a finite-state system, then this process has to terminate after k reaches the number of states of the system [1][4].

2.3 BMC in Hardware

Here we will speak about the techniques to translate industrial system description languages into BMC. First thing is to verify the design given in the hardware description language. Speaking of the hardware description language, it is used to describe hardware designs in industry where it describes schematics and net-list. It further contains both semantics as well as synthesis semantics for simulation because designers rely heavily on the simulation for building model. The idea is to encoding models given in hardware description languages into SAT. Simulation semantics are based on an event queue, resembling the data structures maintained by event-driven simulators. On the other hand, the synthesis semantics is closer to the actual hardware produced, and may uncover design flaws during simulation. BMC works using synthesis semantics as follows. BMC will be firstly perform some stages of behavioral synthesis to the point where net-list is produced. A net-list is a collection of primitive elements. A way to represent net-lists is to use and-inverter graph (AIG) which is said in the way that the net-list consists of "and" gates, inverters and memory elements referred to as registers. We will look into some examples when we start of with NuSMV[1].

2.4 BMC in Software

The idea of BMC for software is to encode the transition relation of the program into a circuit representation, and then to perform BMC. It works as follows[1]:

- Firstly program counter determines the instruction to be executed next.
- Secondly each instruction is turned separately into a transition relation. To obtain a formula, we firstly convert the arithmetic operation in program into the circuit.

To show the step number 2, we will be providing an example [1], suppose we have an equation as follows:

$$x = y + 1$$

The transition relation of the equation above can be shown as:

$$x = y' + 1 \ \wedge \ y' = y \ \wedge \ z' = z$$

Where x′, y′ and z′ refers to the next state values of state variables. Here the sign = denotes the mathematical equality and not the assignment. The 2nd and the 3rd conjunction shows the value of y and z are not changed. The unwinding phenomena of ”monolithic encoding” states that all program path that traverse k(or fewer) instruction to be explored. The size of the basic unwinding is said to be k times the size of the program. For the large program this phenomena is prohibited, so for that, we will be using certain optimization techniques. The idea of optimization there will be reduction of the size of the encoding by eliminating combinations of control-flow locations which does not correspond to paths through the program. When we talk about sequential program, it is beneficial if we merge all instructions from one basic block to a single big-step instruction.
Previously we looked at ”Monolithic Encodings”, now lets look at ”Path-Based Encodings”, where the idea is that instead of unwinding the entire transition relation, path-based software analyzers perform symbolic simulation alongside specific program paths up to a given depth [1]. The result is then forwarded to SAT solver. The main idea of using it is for checking safety property in achieving particular coverage goal. There are a lot of approaches to pursue from the set of paths the one that has to be explored which can lead to a particular goal. Once we have obtained the satisfying assignment then we can extract the counter example from it. We have to further extract the desirable counter example and then we can use the information from BMC in order to get the root of the error. There are tools for path based encoding one basic tool is to explore each path one at a time. The benefit here is that the formula generated are very simple and can be solved by modern solver. But there is also an issue as the number of paths can be exponential that can create a problem. To resolve the issue of path explosion the paper [1] has introduced a concept of path merging. The idea is to merge the formulas that correspond to two (or more) paths at points of re-converging control flow. By this the number of formula is reduced but on the other hand the formula is large which is hard to solve by the SAT solver.
When we talk about the completeness of the BMC, we can say that it is essentially incomplete, as it searches for property violations only up to a given bound k and never returns “No Errors”. As a result, BMC can be used to prove liveness and safety properties on a particular class of programs if applied in a slightly different way. When we talk the loop structure which is bounded by the maximum number of loop iteration, the algorithm can be applied as follows. A Guess k for the bound on the number of loop iterations is made. The loop is then unrolled up to this bound k using BMC. The property that is checked is that any path exceeding k loop iterations is infeasible. If the property holds, k is established as a sound high-level WCET. Otherwise, there are paths in the program exceeding the bound, and a new guess for the bound is made.
If we deal with **Multi-Threading Program** in the BMC, it can be approached as follows, building a path formulas with thread interleavings. As previous issue such as path explosion, numerous variants for restricting the search until a bound, path merging and compression had been considered. Concurrent programs can be reduced to sequential programs by applying a bound on the number of context switches. It is said that partial-order reduction are beneficial for the verifiers for cocurrent system.
In the light of **Co-Verification of HW/SW**, This approach is the baseline for the broad area of “symbolic co-simulation” of two models, where the software is in C and the hardware is modeled in an HDL.

3 Model Checking with SAT

Lets first describe what is satisfiability(SAT) problem. It can be described as the problem of determining for a given Boolean expression if there exist a certain condition that could satisfy the given Boolean expression. It can be used for bounded model checking in a manner that such that for a Boolean function, SAT would search for the model of values for the variables that could make the formula true. It can be expressed as follows for example, the function f(x,y,z) = $x \wedge y \wedge \neg z$, which can be written as f(x,y,z) = (1,1,0).

3.1 Encoding in SAT

The idea here is to make BMC work for a concrete programming language such as C which is difficult because of the following reasons. First, programming languages have complex syntax and semantics which have to be parsed, analyzed and encoded. Then dealing with memory and in particular pointer arithmetic which requires non-trivial decision procedures for arrays. Lets start with **Encoding Bit Vectors** it is described as encoding of word-level operations, which correspond to the evaluation of arithmetic expressions in programming languages, into bit-level formulas. Now talking about bit-blasting, it is similar to synthesis of hardware models on the register transfer level (RTL) into net-lists. After encoding models into bit vectors and then bit vectors into propositional bit-level logic there remains a last step of encoding bit-level formulas into conjunctive normal form (CNF), which is the common input format of most SAT solvers. To compactly represent formulas we need sharing. For that we use simply combinational circuits that represents generated bit-level formulas and not trees, this can be exponentially large. In order to obtain a formula in CNF from an AIG it is possible to first translate the AIG into negation normal form (NNF), which at most doubles the size of the DAG. Each elimination of a disjunction is quadratic and thus this approach can lead to an exponential blow-up of the resulting CNF. The translating an AIG into CNF by distribution is only feasible for small formulas.
Now, we will be talking about memory as it is an important aspect for not only hardware but also software. By analysing the first-order theory of arrays is powerful enough to express most memory-related properties of practical interest. Here, we can focus on the quantifier-free fragment of arrays over bit vectors. Memory in hardware can be handled by standard decision procedures for arrays. Now talking about the **Encodings with Under- and Over-approximation**, the idea is that a small part of the formula needs to be analyzed to conclude whether it is satisfiable or unsatisfiable. Here the article [1] has used the method of abstraction which is defined as follows that faster decision procedures for bit-vector arithmetic and other theories. The decision process is either working on over or under-approximations. In either cases the desired result is a formula ϕ' that is easier to solve than the original formula ϕ[1].

- **Over Approximation** over-approximation of a decision problem permits more solutions than the original formula. In over-approximation for a satisfiability problem is to replace sub-formulas by new variables. Over-approximation ϕ' is found to be unsatisfiable, then we can conclude that the original formula is unsatisfiable. If Nothing then, we can concluded if ϕ' is satisfiable, since the satisfying assignment for ϕ' need not be a satisfying assignment for ϕ.

- **Under Approximations** under-approximation of a decision problem permits fewer solutions than the original formula. In under-approximation for a satisfiability problem is to add further constraints or to replace sub-formulas by constants. In case an under-approximation ϕ' is found to be satisfiable, we can conclude that the original formula

is satisfiable. Nothing, however, can be concluded if ϕ' is unsatisfiable. A proof of unsatisfiability of ϕ' need not be a proof of unsatisfiability for ϕ.

Under-approximation is applied in the case of expensive bit-vector arithmetic operations such as multiplication.

3.2 Complete Model Check SAT

Now we will be discussing the idea of SAT where it enable proofs that a given property holds for unbounded depth. For completeness threshold, we will perform a strategy of deep searching, where we will search deeper to examine relevant behaviour of the bounded program until the point we realises that more deep search would take us to the state we already examined. The main idea here is to fund the upper bound length of k of counterexample that has to be tried before the property is declared to hold. For the smallest threshold it is hard so for that the one consider the practice of over-approximation. When we apply abstraction techniques such as localization reduction, the completeness threshold can be lowered. For **Image Computation** the BDD-based model checkers perform forward or backward fixed-point iterations using pre and post in order to determine the truth of a property given in temporal logic. As we know that SAT-based techniques are well suited to check whether a given transition system satisfies a given inductive invariant. Let us define this with a formula as follows [1]:

$$(P(s) \wedge T(s, s')) \Rightarrow P(s')$$

Here P is the induction and T is the transition relation. As we can see that both are quantifier-free and can therefore be checked effectively while using the techniques we have described so far. Nothing can be concluded about P if the second condition fails. The main problem is the property that holds should not be inductive. Induction can be made more likely succeeded while checking a state property P' that is stronger than the non-inductive property P. Methods have been proposed for strengthening inductive arguments. But the initial methods relied on careful manual strengthening of properties to make them inductive. The resulting equivalence relation can then be used to simplify the model-checking problem by replacing equivalent nodes by representatives. In the light of **Temporal Decomposition**, the circuit nodes which are initialized to one specific constant value, true or false, and then never change, can be found in the same way. However, nodes only stabilize after a certain number n of steps. So for that issue, model-checking problem should be split into a bounded-model-checking problem for the first n steps, followed by checking a simplified model where the signals fixed after n steps are replaced by constants. Now we will talk about **K-induction**, automated way to increase the strength of the inductive argument is to increase the depth of the unwinding, forming a formula that is very similar to a BMC instance. For k-induction, we first check that there is no counterexample of length k or less. After that we can check no state reachable from a sequence of k-states that satisfy P violates P. Both checks can be performed effectively using a satisfiability decision procedure.

3.3 Model checking with Craig interpolation

It was the first SAT based model-checking technique and is still considered to be one of the most effective techniques in practice. It uses an over-approximation of quantifier elimination, for image computation, which is obtained as an interpolation from a refutation of a BMC run between the first and the remaining states of the considered path.

3.4 Iterative Inductive

Here we talk about the idea of exploited in the seminal algorithm IC3. The basic idea of IC3 is to generate a relative inductive chain $F_0 \subseteq F_1 \subseteq \ \subseteq F_k$ of over-approximations of reachable states. This process is repeated until the chain reaches a fix-point or a bad state is shown to be reachable. The frontier sets F_i are refined by adding restrictions on states reachable in one step backward from a goal state, i.e., a bad state. Initially only bad states are goal states, but after one step backward, the negation of an added clause becomes a goal too. Finding and minimizing these partial models is the most time-consuming part of the algorithm.

3.5 Abstraction Techniques

Predicate abstraction is currently the predominant abstraction technique in software model checking. In predicate abstraction, a sound approximation R' of R is constructed using predicates over program variables. A predicate P partitions the states of a program into two classes: one in which P is true, and one in which it is false. Here each class is an abstract state. Abstractions are automatically constructed using a decision procedure to decide. For n predicates lead to 2^n abstract states.

3.6 Simulation with SAT

The reachability computation above may discover that whether an error state is reachable in the abstract program. Subsequently, a simulation step is used to determine whether an error exists in the program or not. There are two sources of checking accuracy in the abstract model:

- Spurious traces arise because the set of predicates does not have enough information to distinguish between certain states. Spurious traces are eliminated by adding additional predicates, obtained by computing the weakest or strongest precondition of the instructions in the trace.

- Spurious transitions arise because the Cartesian abstraction may contain transitions not in the existential abstraction. Spurious transitions are eliminated by adding constraints to the abstract model. Such transitions are eliminated by restricting the values of the Boolean variables before and after the transition.

Path slicing eliminates from the counterexample instructions that do not contribute to a property violation. Loop detection is used to compute the effect of arbitrary iterations of loops in a counterexample in a single simulation step.

4 NuSMV

4.1 Description

Now we will talk about SAT-based model checking in NUSMV using the reference [5].Here i have used NuSMV 2.4 for the examples. We will be explaining how the language NuSMV works by showing some example in every section. Plus, we will be showing the way to execute it on terminal. For running the file, i have used the mac terminal, it is also supported in linux and windows. The key words are the same in all the operating system. NuSMV model checking, uses the verification technqiue of LTL, CTL and Bounded checking method. We will be starting about the syntax of NuSMV before going forwards with the model checking.

4.2 Basic Syntax

Taking in account an example taking from [6], where we are talking about semaphore.

Semaphore

```
MODULE main
VAR
    semaphore : boolean;
    proc1 : process user(semaphore);
    proc2 : process user(semaphore);
ASSIGN
    init(semaphore) := FALSE;
    SPEC AG ! (proc1.state = critical & proc2.state = critical)
    SPEC AG (proc1.state = entering -> AF proc1.state = critical)

MODULE user(semaphore)
VAR
    state : fidle, entering, critical, exitingg;
ASSIGN
    init(state) := idle;
    next(state) :=
        case
            state = idle : fidle, enteringg;
            state = entering & !semaphore : critical;
            state = critical : fcritical, exitingg;
            state = exiting : idle;
            TRUE : state;
        esac;
    next(semaphore) :=
        case
            state = entering : TRUE;
            state = exiting : FALSE;
            TRUE : semaphore;
        esac;
FAIRNESS
    running
```

Here we can see the example of the Semaphore code where it is causing a transition between 5 states depending on condition. In order to understand a programming language, i used the concept of my previous programming language and try to translate that in here. I will be refereeing to java in order to explain how this language works. **MODULE main**, is just like a class in java, we can have more then one module in a file and we can use the same module again. For example in one module we can call the other module. Taking into account **VAR**, here we define all the variable that we will be using later.

```
VAR location: {l1,l2};
VAR check: boolean;
```

This is not the correct way to write the variable. The variable to be written in a way as follows:

```
VAR location: {l1,l2};
    check: boolean;
```

Now we will talk about the **ASSIGN**, here we set the initial value and the change in the value. The keyword init() is here to specify the initial value of the variable. The keyword next(), is here to specify the next value assign to the variable. The keyword case specify the

cases at which value changes. For example here when the location = l1 and x ¡ 10 at that point the new value of location is l2 otherwise the value l1 remains. Just like in java we work under parentheses, for ”cases”, it has to be written inside case .. esac.

INIT and TRANS

Also as part of NuSMV can can see some more keywords as follows like INIT and TRANS[6].

```
1  INIT
2    output = 0
3  TRANS
4    next(output) = !input | next(output) = output
```

We might have guessed it right that INIT means the initial value of the variable where we can also assign it with the ASSIGN factor. SPEC is used to define the CTL or LTL expression in the code that we want to use for the verification. TRANS value here is chosen non deterministically either being the negation of input or be the same as output.

Execution on Terminal

The NUSMV file are made with the extension of .svm. Inside the nusvm folder we will be going inside the bin folder. There we can find the NuSMV executable. For example we want to execute the file of traffic light, ”./NuSMV -int traffic-light.smv ”. In NuSMV we execute the ”go” command that does ”flatten_hierarchy”, ”encode_variables” and build model for us:

```
1 NuSMV > go
2 NuSMV > pick_state -i
```

The command ”pick state” shows the states, if there is more then one initial state so at that point the terminal will ask to which state do u want to choose as the initial state.

Also there is a command for showing the reachable states which is wrriten as follows:

```
1 NuSMV > print_reachable_states -v
```

If we do any changes in the code or we want to run the other file, so we first have to quit the NuSMV from the terminal by:

```
1 NuSMV > quit
```

Then we can run the file easily just by writing this in the terminal:

```
1 [Terminal] > ./NuSMV -int semaphore.smv
```

4.3 CTL

Computation Tree Logic which is the branching time logic. Here by referring to the tutorial [5] we will express how to use it in NUSMV. Considering an example formula from [5] $AF\ p$, which can be expressed as ”all the paths (A) stating from a state, eventually in the future (F) condition p must hold”. This formula says that from all the branches it will eventually reach p. Similarly there is CTL formula EF states that there exist a path such that in the future p will hold. On the other hand the formula AG p, where G is described as global, that p will globally be true. Also AX p, specify the next condition to be true. In NUSMV a CTL specification is given as CTL formula introduced by the keyword “SPEC”. Whenever a CTL specification is processed, NUSMV checks whether the CTL formula is true in all the

initial states of the model. The tutorial [5] has specified the semaphore example in CTL "AG ! (proc1.state = critical & proc2.state = critical)" which states that "it should never be the case that the two processes proc1 and proc2 are at the same time in the critical state". One other condition which is "AG (proc1.state = entering → AF proc1.state = critical)" which means that "if proc1 wants to enter its critical state, it eventually does".

In the semaphore, we can see the CTL property is defined with SPEC in the ASSIGN. Once you run this file in the terminal, you will see whether this 2 conditions are true or false, if in the condition it is false then it will demonstrate at what condition it is false.

4.4 LTL

Linear Temporal Logic, it characterizes each linear path induced by the FSM (linear-time approach). The example of LTL in NUSMV is as follows: In the statement of "F p", it means that "in the future p", in the future it will be true in one of the future time. The statement "G p" which means "globally p", which mean that for all the in the future it will be true. The statement "p U q", which states that p will be true until q is true in the future by reaching to it. Finally X p, which mean that p is true in the next state [5]. Here we will be using the same Semaphore example as used in the CTL by using the tutorial [5] as follows:

```
1    LTLSPEC G ! (proc1.state = critical & proc2.state = critical)
2    LTLSPEC G (proc1.state = entering -> F proc1.state = critical)
```

Here we can see new Specification as compared to CTL. While in CTL we use the key word "SPEC" to identify the CTL formula, here we use "LTLSPEC" to identify the LTL specification. Here we will be checking the same condition as CTL but in a different way. The condition are as follows that : "G ! (proc1.state = critical & proc2.state = critical)" expressing that the two processes cannot be in the critical region at the same time, and "G (proc1.state = entering → F proc1.state = critical)" expressing that whenever a process wants to enter its critical session, it eventually does. Once we run the code it will specify if the condition holds or not. If not then it will demonstrate the sequence. LTL part of the NUSMV also have some part temporal logic that it recognises as follows that are: "O p" thie means that it reads p once. "H p", it means that "p holds in all previous time instants". "p S q", which means that "p holds since a previous state where condition q holds". And Lastly, "Y p", that means "p holds in the previous time instant"[5].

4.5 Bounded Model Checking

By having a SAT based model checking, BMC has to be connected to the external SAT solver. Currently NuSMV uses 2 external SAT solvers that are MiniSAT and Zchaff.

Checking LTL Spec with BMC

We will be starting with the LTL Spec using the BMC by using a example proprty "LTLSPEC G (y=4 → X y=6)" which is False, but in the terminal we will add a configuration of "-bmc" to allow NuSMV to allow BMC. Once NuSMV finds the condition is false, it will show us the counter example. In general, in BMC mode NUSMV tries to find a counterexample of increasing length, and immediately stops when it succeeds, declaring that the formula is false. The maximum number of iterations can be controlled by using command-line option -bmc length. The default length is 10. If at anypoint the maximum number of iteration has been reached and there exist no counter example then at that point the NuSMV exits by stating that the truth of the formula can not be found. For that we state that the counter example should be longer then the maximum lenght.

Counter Example

There are 2 types of counter examples that can be found here, one of them is the finite sequence of transitions that can be seen from different states. For liveness there are infinite sequence that can be represented as bounded setting. We will be using the above code and the condition ""! G F p"" to demonstrate how it works. To run this we firstly need to setup the bmc as follows: Firstly we need to setup bmc and then run the ltlspec

```
NuSMV > build_boolean_model
NuSMV > bmc_setup
NuSMV> check_ltlspec_bmc onepb -k 9 -l 0
-- no counterexample found with bound 9 and loop at 0 for specification
! G F y = 2
NuSMV> check_ltlspec_bmc onepb -k 8 -l 1
-- no counterexample found with bound 8 and loop at 1 for specification
! G F y = 2
NuSMV> check_ltlspec_bmc onepb -k 9 -l 1
-- specification ! G F y = 2 is false
```

After the last line, it will be demonstrating the execution sequence. Now we will explain what the following commands mean. The command "check ltlspec bmc onepb", it is the fine-grained control of the length and of the loopback condition for the counterexample. The command "-k" represents the length of the counter example. The command "-l" represents the loopback condition[5]. NUSMV did not find a counterexample for cases (k = 9; l = 0) and (k = 8; l = 1). So it can be said that is not possible for NUSMV to satisfy them. Loopback can be checked in the following ways:

- For running the loop for a precise time we just use a natural number as an argument
- For giving a loop length, we give a negative number as a loop length
- If we do not want to give a loopback, we can give "X" as the -l argument and it will not go through finding the infinite counter example.
- If we want to go through all the possible loop, we use "*" as the argument. It will search counterexamples for paths with any possible loopback structure. It is the default value of -l option.

4.6 SAT Solving Category

SAT solver MiniSat will be used for SAT Solver. If no SAT solver has been configured, BMC commands and environment variables will not be available [5].

Here we will be using the Enviroment Vairable.

```
sat_solver
```

For using the sat property, we will be using the command for perfoming SAT based model checking. It takes as argument -l -k and the property to be checked.

```
check_pslspec_bmc
```

On the other hand, the command used for incremental SAT

```
check_pslspec_bmc_inc
```

After the command we execute the trace by using the command "execute traces" with the -e engine of "sat"

Bibliography

[1] Armin Biere and Daniel Kröning. Sat-based model checking.

[2] Bounded model checking in software verification and validation. https://www.youtube.com/watch?v=AbMr7jzKBhE.

[3] Jiang and Ciardo. Improving sat-based bounded model checking for existential ctl through path reuse.

[4] Edmund M. Clarke Ofer Strichman Yunshan Zhu Armin Biere, Alessandro Cimatti. Bounded model checking.

[5] Gavin Keighren Emanuele Olivetti Marco Pistore Roberto Cavada, Alessandro Cimatti and Marco Roveri. Nusmv 2.6 tutorial.

[6] Nusmv 2.0 internet tutorial. http://nusmv.fbk.eu/NuSMV/userman/v20/nusmv_2.html#SEC2.