NKind: a model checker for liveness property verification on Lustre programs

Junjie Wei, Qin Li*
Shanghai Key Laboratory of Trustworthy Computing
East China Normal University, Shanghai, China

***Abstract*—Modeling and verification of real-time reactive systems is getting greater concern in industrial field, especially in safety-critical applications. As a representative language for modeling real-time reactive systems, Lustre has been extensively used in the development of control systems in vehicles and aircraft. Existing model checking tools for Lustre like Kind2 and JKind have good support for verifying safety properties, but they lack explicit support for liveness properties. Thus we present NKind, an SMT-based infinite-state model checker, which accepts models and properties written in Lustre and is capable of verifying both safety and liveness properties. NKind is inspired by many existing model checker and adds liveness support based on their common techniques, which provides more flexibility. It is written in Java, providing good compatibility, and lays emphasis on modularity and extensibility. The results and performance of NKind on benchmark examples demonstrate that it is competitive comparing to other existing tools.**

***Index Terms*—Lustre, Model Checking, Liveness Property**

I. Introduction

As one of the most important measures of ensuring that software meets the expected requirements, model checking is becoming increasingly important in the development of modern systems, especially real-time reactive systems. Many industrial standards like DO-178C, EN50128, ISO26262 etc. require to use formal verification in software design and development for high safety assurance, which demonstrate the bright prospect of wide application of model checking tools in industrial field. As a representative language for modelling real-time reactive systems, Lustre [1] has been extensively used in the development of safety-critical systems like avionic systems and power plant monitoring systems.

Kind2 [2] and JKind [3] are two most popular model checking tools for verifying Lustre programs. Kind2 is a multi-engine model checker and lays emphasis on invariant checking. It uses an extension to Lustre as modelling language, and converts the given model into a state transition system. The property is proved by checking that it holds in all reachable states of the system. JKind provides similar functionalities. It mainly focuses on post-processing and proposes features like inductive validity cores (IVC) and smoothing. JKind and Kind2 support different Lustre language features. For example, JKind lacks the support for automaton structure.

It is worth mentioning that these tools are mostly concentrated on proving safety properties and ignore the liveness properties. This leads to a gap in verifying liveness properties for Lustre. nuXmv [4] is a model checker capable of both finite-state and infinite-state systems. As the successor of NuSmv, it reads models in SMV format extended with infinite-state support. Although nuXmv support liveness property checking, currently there is no available way to make it support Lustre models directly.

*Corresponding author: Qin Li (qli@sei.ecnu.edu.cn)

The main contributions of the paper can be summarized as follows:

- We present NKind, a model checker for Lustre supporting the verification of liveness properties which existing tools for Lustre do not support.
- When verifying liveness properties, the performance gap between NKind and mainstream tools is not significant.

In this paper, we present NKind, an SMT-based infinite-state model checking tool, which is mainly used for proving or disproving properties of synchronous reactive system models written in Lustre. NKind mainly relies on the powerful SMT-solver Z3 [5] to validate or invalidate the properties. For properties that are proved to be invalid, a counterexample will be returned. NKind is inspired by several existing model checkers for Lustre like Kind2 and JKind, and uses similar architecture and techniques. In the mean time, it provides enough extensibility and makes it rather easy to support new features. NKind is written in Java, which offers better multi-platform compatibility and is easy to be integrated to a larger framework as a model checking service provider. To fill the gap in liveness property checking in Lustre, in addition to safety properties, NKind has the capability to verify liveness properties which can have finite-time violations but will finally hold forever. NKind is free to be used for research and evaluation purposes and can be downloaded from https://nkindmodelchecker.github.io.

The rest of the paper is organized as follows. Section II briefly introduces some preliminary concepts involved in NKind. Section III describes the design architecture of NKind and introduces the main algorithms that are used for verification. Section IV provides the capabilities of NKind and is emphasized on the liveness extension. Section V conducts performance benchmarks for NKind and makes comparison with other existing model checking tools Finally, conclusion and future work is given in Section VI .

DOI reference number: 10.18293/SEKE2022-089

II. Preliminaries

A. Lustre language

Real-time reactive systems refer to the systems that continuously accept input and react to the environment in a timely manner. Since the inputs are constantly changing and the systems are expected to respond quickly to the inputs [6], synchronous languages were designed to effectively describe reactive systems. Lustre is a synchronous dataflow language which is widely used in safety-critical control systems like vehicles and aircraft.

Different from imperative languages, dataflow languages focus on data and represent them as infinite sequences of values, i.e. dataflows. This fits well with the usage scenario of real-time reactive systems, which in many cases need to read data from multiple sensors at a fixed frequency as inputs and then calculate the outputs, and the dataflow model can represent this behaviour well. Each dataflow is associated with a clock, and the specific value of a dataflow at a clock time can be uniquely determined by using the clock value as index.

Lustre use a node as a minimal functional module, which has a finite group of inputs and outputs as interface and allows local flow to save its internal state. Functionally speaking, a Lustre node can be regarded as a mapping from an input set to an output set [6], and the outputs are calculated from the current and previous input/output or local flows according to the flow definition. Readers can refer to [1] for detailed grammar of Lustre language.

B. L2SIA-WFR

In comparison with safety properties which have counterexamples of finite length, liveness properties often have counterexamples of infinite length, making it more difficult to verify liveness properties to a large extent. Algorithms like liveness-to-safety [7] and k-Liveness [8] were proposed to solve this problem, but these solutions were mainly restricted in finite-state systems, which were not sufficient to work in infinite-state systems like Lustre. Then an extended version of liveness-to-safety called L2SIA-WFR [9] was presented to handle the infinite state space.

A counterexample of liveness property in the form of $FG\ \neg p$ is often lasso-shaped, which consists of a path starting from the initial state (i.e. stem) and a loop containing at least one state satisfying p. So liveness-to-safety tries to prove the absence of a lasso-shape counterexample as an invariant by ensuring there is no loop paths violating the property. L2SIA-WFR first extend the algorithm by using implicit abstraction. It uses a set of assignments to the predicates to identify multiple concrete states, therefore abstracting the infinite state space to finite state space. Like the idea of CEGAR [10], if spurious counterexamples are found, they will be used to refine the abstraction for next iteration by adding extracted predicates from counterexamples to the predicate set. In order to handle the situations that an abstract loop can be executed finite times, which will not violate the property but prevent the algorithm from terminating, well-founded relations are calculated as a termination proof. In the cases where well-founded relations are available, these relations provides more information of the model for better refining the abstraction and lead to a better performance.

III. NKind Architecture

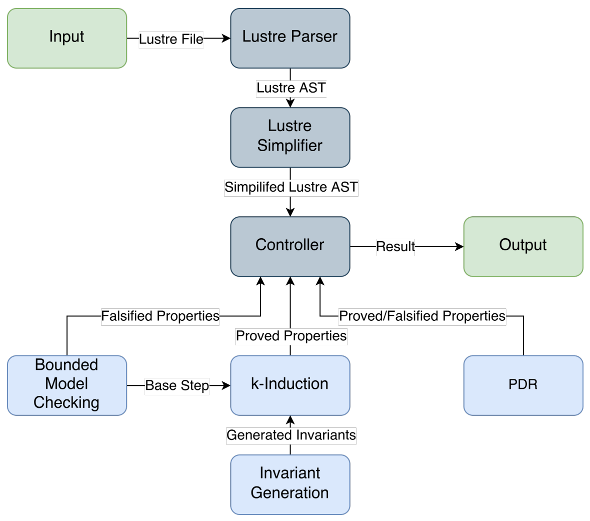

Fig. 1: NKind architecture

The overall architecture of NKind is shown in Fig. 1. It consists of Lustre parser, simplifier, and a controller equipped with several verification engines.

Since there are different dialects of Lustre with different syntax, for scalability reasons, an improved version of visitor pattern is used in the design which provides more flexibility to handle syntactic structures of Lustre and therefore makes it easy to extend functionality.

The traditional version of Lustre, namely Lustre v4, consist of a set of elements which is basic enough and could be considered as Core Lustre. Many features added in more recent versions or dialects can be translated to Core Lustre. For instance, automaton structure, which is available in Kind2 and SCADE [11], introduces a concept similar to state machine and provides the ability to change the behaviour pattern based on external inputs or internal events, allowing a dataflow to have multiple definition in different states. However, such structure can be simulated by converting each state into corresponding nodes without changing the semantics. With this in mind, the Lustre simplifier is used to translate all the complex structure in the given Lustre program before it is converted to transition system and make it possible to leave the process of making transition system unchanged when adding support for new features. It is suitable for implementing syntactic sugar like array iterator introduced in SCADE.

Like several existing model checking tools for Lustre which rely on the expressivity and reasoning capabilities of modern SMT solvers, NKind converts input Lustre program into a transition system with the same semantics in the form of SMT formula. As the core representation of the given model, the transition system is then handed over to solving engines to verify the properties.

NKind follows the practice of many model checkers and uses a set of solving engines which run in parallel for verification. The solving engines of NKind are mainly composed of Bounded Model Checking (BMC), k-Induction, Invariant Generation and Property Directed Reachability (PDR, or IC3).

1) BMC [12] engine checks for counterexamples by unrolling transition relation T step by step. It also provides the proof of base step for k-Induction engine in the mean time.
2) k-Induction [13] is the enhanced form of normal induction. It tries to find a value of k such that the property p holds for all states reachable from initial state I in first k steps (base step) and is preserved by continuous transitions of length k (induction step), i.e. $\forall i \leq k \cdot I \wedge T_0 \wedge T_1 \wedge \cdots \wedge T_k \Rightarrow p_k$ and $\forall n \geq 0 \cdot T_n \wedge p_n \wedge T_{n+1} \wedge p_{n+1} \wedge \cdots \wedge T_{n+k} \Rightarrow p_{n+k}$. If such k exists, it follows inductively that the property holds in all reachable states.
3) Invariant Generation automatically generates some validated auxiliary invariants based on predefined invariant templates [14] according to the given transition relations and are proved by k-Induction. The generated invariants are mainly used for helping the verification process of k-Induction engine in case the given property is not k-inductive.
4) PDR [15] is based on an idea similar to CEGAR to make an over-approximation of the property and iteratively strengthen the approximation until it becomes inductive or meets a counterexample. The original PDR algorithm is only capable of handling finite-state problem, and in order to make it work in infinite-state system, implicit abstraction proposed in [16] to abstract the states into finite ones. The abstraction itself is refined by extracting new predicates from the Craig interpolants of spurious counterexamples.

Once a property is proved or disproved, other engines will be informed to make use of the result. If the verification process is interrupted or the backend SMT solver encounters an error, the property will be marked unknown and returned to user.

IV. Main Features

A. Safety Property

One of the main functionalities of NKind is to verify safety properties of reactive system modelled in Lustre language. Like Kind2, JKind or other Lustre model checkers, NKind attempts to prove that the given properties are invariants in the given system with a set of model checking engines which are described in the previous section, and tries to give a counterexample in case of failure.

B. Liveness Property

Apart from the traditional safety property checking, NKind also introduces some new techniques for liveness property checking, which, to the best of our knowledge, makes NKind the first model checker for Lustre that support liveness properties. In general, if we use Linear Temporal Logic (LTL) to summarize the property that NKind is able to handle, properties in form of $G\ p$ are supported to enable traditional safety property checking. In addition, properties in form of $FG\ p$ are also supported due to the liveness extension. In other words, apart from checking properties that hold forever, properties which have finite-time violations can be allowed as long as the properties will finally holds forever.

1) Liveness Usage: With the liveness support, Lustre becomes more expressive when specifying property. For instance,

```
node main () returns (x: int; f: bool);
var
  N : int;
  e1 : bool;
let
  N = (20 -> pre (N));
  x = (0 -> (pre (x) + 1));
  f = true -> ((pre (x) < 1) or (pre (x) > pre (N)));
  e1 = f;
  --%MAIN;
  --%PROPERTY Live e1;
tel;
```

Fig. 2: Example Lustre program with liveness property

the lustre program shown in Fig. 2 mainly contains three dataflow.

1) N is a constant flow with value 20.
2) x self-increases by 1 per cycle.
3) f indicates the property that either the value of x in the previous cycle is less than 1 or greater than that of N.

It is clear that the property is violated when $1 \leq i \leq 20$. By specifying the property type using keyword ”Live”, NKind can be informed to not simply use invariant proving techniques but to encode the property first. As is mentioned in the previous section, the encoded property is to prove the absence of a loop continuously violating the original property. In this case, after x is increased to 21, x will never be less than N and thus proving the original property.

As a result, the constraints of traditional safety properties can be loosen, and it will be easier to specify qualitative property without giving an explicit value. For example, the program listed in Fig. 3 describes a system where y will finally catch up with x. If we use safety property to specify this property, an explicit gap is needed and thus a counter is introduced. However, we can write a more general property if we use the liveness extension which is displayed in Fig. 4.

It will also be suitable for specifying the system that will enter a stable state after several temporary transition caused by input events from sensors, which is common in various industrial control systems.

2) Liveness implementation: As is mentioned before, the verification process of NKind revolves around the state transition system converted from the given Lustre program. L2SIA-WFR algorithm [9] provides a method to encode liveness properties as safety properties at the level of state transition system.

```
node main() returns(x: int; y: int; f: bool);
var
    e1: bool;
    counter: int;
let
    x = 20 -> (pre x) + 1;
    y = 0 -> (pre y) + 2;
    f = false -> pre x < pre y;
    counter = 0 -> pre counter + 1;
    e1 = (counter > 21) => f;
    --%MAIN;
    --%PROPERTY Safe e1;
tel;
```

Fig. 3: Specifying property with safety property

```
node main() returns(x: int; y: int; f: bool);
var
    e1: bool;
let
    x = 20 -> (pre x) + 1;
    y = 0 -> (pre y) + 2;
    f = false -> pre x < pre y;
    e1 = f;
    --%MAIN;
    --%PROPERTY Live e1;
tel;
```

Fig. 4: Specifying property with liveness property

This commonality in structure makes it possible to introduce support for liveness property in the original framework.

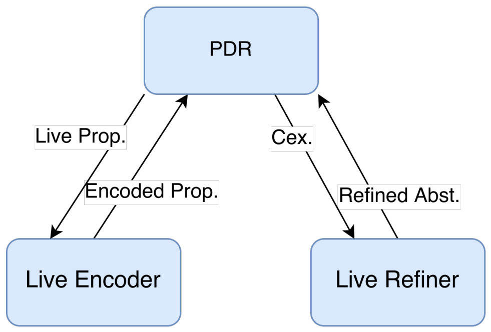

Fig. 5: Embedding liveness extension to PDR process

Since the method monotonically strengthens the original transition system and is a good complement to PDR process [9], the liveness extension in NKind is embedded in PDR engine like Fig. 5 and mainly refers to ic3ia [17], an open-source implementation of the L2SIA-WFR algorithm.

More specifically, a liveness property is first encoded as a safety property using L2SIA algorithm and then put into PDR process. If a counterexample is found and normal refinement of the PDR process failed to make more precise abstraction, Live Refiner will try more liveness-specific methods to make refinement like proving the spuriousness of the counterexample by unrolling the transition relation or attempting to get new ranking relations. Readers can refer to [9] for algorithm details.

Since abstraction refinement in PDR process depends on the calculation of Craig interpolation, an SMT solver which supports such calculation is needed as the backend of the algorithm. For example, JKind chooses to embed SMTInterpol [18], a solver which is devoted to produce interpolants. However, as the input constraints becoming bigger and more complex, this solver may encounter performance degradation. This shortcoming becomes more significant in the liveness extension because the transition system is strengthened monotonically by adding more constraints due to the design of the algorithm, which may lead to variable explosion. Taking the simple Lustre program shown in Fig. 2 as an example, after being translated into transition system, it contains 35 symbols. But after the first liveness encoding, it grows to 284 symbols. Such increment will obviously impose a greater burden on the solver.

Due to the lack of SMT solvers supporting interpolation, it is not easy to simply switch to another solver with higher performance. However, we mentioned the fact that the dependence on interpolation is limited to the part of abstract refinement, which inspired us to use another efficient solver in the main framework of PDR process. Referring to PDR implementation proposed in [19], we use z3 [5] as the main backend SMT solver, and leave the calculation to SMTInterpol only when interpolation is needed by doing a bi-directional conversion between the two solver.

V. Experimental Evaluation

In this section, we evaluate NKind with both safety and liveness benchmarks. The experiments were all conducted on a 64-bit Linux machine running Ubuntu 20.04 with a 20-core Intel Core i9-10900K processor and 32GB of memory.

A. Safety Evaluation

We use a test suite containing 864 Lustre programs from Kind [20], which was also used in the benchmark of Kind2. Two existing and mature tools, Kind2 and JKind, were tested together as a comparison group. The default options for each tool were used and the timeout threshold was set to 300 seconds for each problem.

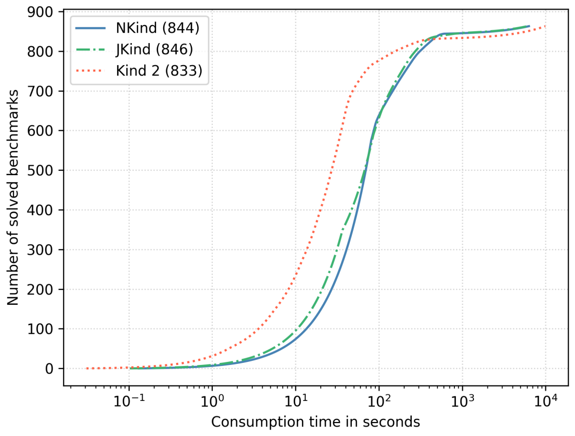

Fig. 6: Verification results on safety property benchmark

The benchmark result is shown in Fig. 6 and the numbers in parentheses represent how many problems were solved in the benchmark. Although there are about 20 programs that cannot be proved or disproved within 5 minutes, the result shows that NKind was capable of proving or invalidating most of the problems in the test suite and had a similar performance to JKind. Relatively speaking, although NKind takes more time when verifying small problem due to the JVM start time, it still has a competitive performance.

B. Liveness Evaluation

Since infinite-state liveness property checking is not as widely supported as safety property is, we first turned to the benchmarks provided together with ic3ia [9]. However, as we have mentioned before, ic3ia accepts a more general form of transition system written in its own vmt format, an extension of the SMT-LIB language. That means not all problems in that benchmark can be translated to a Lustre program with the same semantic, thus making it difficult to accept Lustre program. Due to the lack of effective tools to translate vmt file into Lustre program, we therefore chose 20 tests from the benchmark and converted them in to Lustre programs by hand. Some parameters in the test file were modified to a bigger value to demonstrate the performance of problems in a larger state space.

In order to reflect the effectiveness of the optimizations mentioned in the previous section, a version of NKind using SmtInterpol as backend solver was tested together with the z3 version. Since Kind2 and JKind do not support liveness property, we used ic3ia as a comparison group. The default option and a timeout limit of 300 seconds was used to run the benchmark.

We present some representative test results in detail in Table. I. From the test result, it is clear that the optimization is effective and has a significant performance improvement on most of the test case comparing to the SMTInterpol version. We can see that ic3ia is still the fastest tool in this test, but the gap between ic3ia and NKind is not very significant.

Test name	ic3ia	NKind-z3	NKind-SMTItp
any-down-live	35.77	1.59	25.51
parallel-live	51.68	11.24	32.09
binary-live	0.09	0.8	1.68
piecewise-live	0.11	1.33	2.32
count-nested-live	0.37	1.86	4.42
stabilize-live	5.16	6.57	13.21
count-down-live	1.05	3.81	7.61
swap-dec-live	1.53	3.89	39.37
count-up-to-sym-live	5.84	13.75	3.95
refine_disj_problem	Timeout	Timeout	Timeout

TABLE I: Representative verification results on liveness property benchmark

In order to test with more complex test cases that are closer to the real industrial applications, we leveraged the cases of Kind used as the safety benchmark by converting all the safety properties into liveness properties. Due to the difficulties in conversion from Lustre program to vmt format, we finally use a set of 253 test cases as the benchmark. Fig. 7 shows the benchmark result.

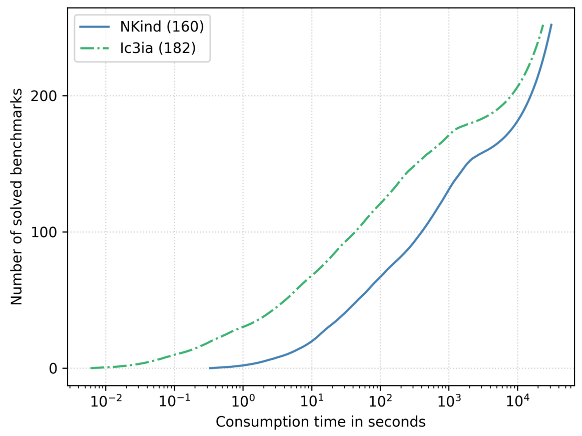

Fig. 7: Verification results on more complex liveness property benchmark

From the benchmark result, NKind is effective in solving liveness properties for Lustre and is able to determine majority of the test cases whether the their properties are valid or not.

However, there is no denying that NKind still has a certain gap compared with the performance of ic3ia. This problem may be caused by the following reason:

1) Due to the use of Java, the JVM start up may consume some time. Since static checking and the translation from Lustre to transition system will be performed before the verification, even small models has a rather long start-up time.

```
node main()
returns
(x : int;f : bool);
var
  N : int;
  e1 : bool;
let
  N = 40 -> pre (N);
  x = 0 -> 1 + pre (x);
  f = false ->  (pre (x) < pre (N));
  e1 = not f;
  --%MAIN;
  --%PROPERTY Live e1;
tel;
```

Fig. 8: Lustre program ”count-up-to-sym-live”

2) Since the liveness extension is mainly based on implicit abstraction version of PDR, we notice that the performance is highly influenced by abstract model. Taking ”count-up-to-sym-live” in the previous benchmark as an example, which is shown in Fig. 8. The model will be verified quickly if the abstraction divide the state space with predicate $x \geq 40$, but will be rather slow if the predicates are $x \leq 1, x \leq 2, \cdots, x \leq 40$. However, we notice that currently the abstraction and the refinement mainly depend on the counterexample returned by the

SMT solver, which is non-deterministic. This may also be the reason why NKind solves less problems. A more guided refinement may be helpful for the performance which need further research.

In general, in spite of such overhead, NKind is capable of handling regular safety properties as well as the liveness ones, which is still believed to be competitive.

VI. Conclusion

In this paper, we presented NKind, an SMT-based model checker for Lustre. Although there are a number of other tools that solve infinite-state model checking problems, NKind integrated their advantages and made the difference. We described its design architecture and functionalities, and emphasized on its extensibility and the support for liveness properties. As far as we know, our liveness extension made NKind the first model checker for Lustre that support both safety and liveness properties, bridging the gap in liveness property checking in Lustre. In the preliminary benchmarks, NKind is proved to be suitable for property verification of industrial control systems, and is rather competitive comparing to the existing model checking tools. Future work includes performing more in-depth optimizations for NKind and improve its performance of model checking by using more guided techniques.

References

[1] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, Sep. 1991, conference Name: Proceedings of the IEEE.

[2] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli, "The Kind 2 Model Checker," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, S. Chaudhuri and A. Farzan, Eds. Cham: Springer International Publishing, 2016, pp. 510–517.

[3] A. Gacek, J. Backes, M. Whalen, L. Wagner, and E. Ghassabani, "The JKind Model Checker," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 20–27.

[4] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuXmv Symbolic Model Checker," in *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Berlin, Heidelberg: Springer-Verlag, Jul. 2014, pp. 334–342. [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9_22

[5] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer, 2008, pp. 337–340.

[6] G. E. Hagen, "Verifying safety properties of lustre programs: an smt-based approach," phd, University of Iowa, USA, 2008, aAI3347220 ISBN-13: 9781109024180.

[7] A. Biere, C. Artho, and V. Schuppan, "Liveness Checking as Safety Checking," *Electronic Notes in Theoretical Computer Science*, vol. 66, pp. 160–177, Dec. 2002.

[8] K. Claessen and N. Sörensson, "A liveness checking algorithm that counts," in *2012 Formal Methods in Computer-Aided Design (FMCAD)*, 2012, pp. 52–59.

[9] J. Daniel, A. Cimatti, A. Griggio, S. Tonetta, and S. Mover, "Infinite-State Liveness-to-Safety via Implicit Abstraction and Well-Founded Relations," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, S. Chaudhuri and A. Farzan, Eds. Cham: Springer International Publishing, 2016, pp. 271–291.

[10] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, pp. 752–794, Sep. 2003.

[11] G. Berry, "SCADE: Synchronous Design and Validation of Embedded Control Software," in *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, S. Ramesh and P. Sampath, Eds. Dordrecht: Springer Netherlands, 2007, pp. 19–33.

[12] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," vol. 4144, Oct. 1999.

[13] M. Sheeran, S. Singh, and G. Stålmarck, "Checking Safety Properties Using Induction and a SAT-Solver," vol. 1954, Nov. 2000, pp. 108–125.

[14] T. Kahsai, P.-L. Garoche, C. Tinelli, and M. Whalen, "Incremental Verification with Mode Variable Invariants in State Machines," vol. 7226, Apr. 2012, pp. 388–402.

[15] A. R. Bradley, "SAT-Based Model Checking without Unrolling," in *Verification, Model Checking, and Abstract Interpretation*, ser. Lecture Notes in Computer Science, R. Jhala and D. Schmidt, Eds. Berlin, Heidelberg: Springer, 2011, pp. 70–87.

[16] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "Infinite-state invariant checking with IC3 and predicate abstraction," *Formal Methods in System Design*, vol. 49, Dec. 2016.

[17] "IC3ia: IC3 Modulo Theories with Implicit Abstraction." [Online]. Available: https://es-static.fbk.eu/people/griggio/ic3ia/index.html

[18] J. Christ, J. Hoenicke, and A. Nutz, "SMTInterpol: an Interpolating SMT Solver," Jul. 2012.

[19] N. Eén, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," *2011 Formal Methods in Computer-Aided Design (FMCAD)*, 2011.

[20] G. Hagen and C. Tinelli, "Scaling Up the Formal Verification of Lustre Programs with SMT-Based Techniques," in *2008 Formal Methods in Computer-Aided Design*, 2008, pp. 1–9.