SMT-Based Verification of Concurrent Critical Systems

Matthias Güdemann[1]

Abstract: Petri nets are a widely used formalism to describe and analyze critical systems. It is in particular well suited for systems with concurrency like cache coherence protocols, fault-tolerant distributed systems or security critical protocols. The verification approaches for Petri nets are most often based on enumerative approaches which allow for analyzing complex, often temporal, properties.

Dataflow languages are widely used in safety critical systems. There are several state-of-the-art model-checkers for these languages. While the properties that can be verified are generally limited to invariants, it is possible to encode some interesting properties of Petri nets as invariants which makes them accessible for powerful analysis methods based on modern SMT and SAT solvers.

The SpiNat approach transforms Petri net into synchronous dataflow language models. This allows for using predicate abstraction and the theory of unbounded integers allows to analyze the potentially unbounded markings of Petri nets using model-checking tools for languages like Lustre. The presented approach is orthogonal to enumeration based approaches for Petri net analysis and allows benefiting from any increase in efficiency of industrial strength SMT-based model-checkers like kind2 and JKind.

1 Introduction

In many critical systems there is some inherent concurrency, in particular if a system contains distributed components or uses a protocol. The behavior of such systems is often hard to understand as the different execution sequences make it easy to build systems that might deadlock or have parts which are never activated. Due to the complex possible behavior, it is most often not possible to apply testing in an exhaustive way.

To overcome these problems there are different formalisms to describe and analyze critical systems with concurrent behavior. Process algebras like CSP [Ho78] and languages like LNT [GLS17] are higher level description mechanisms which allow specification of concurrent systems. Their formal analysis is often conducted via transformation into Petri nets. Such formalisms have been used, e.g., to analyze safety of fault-tolerant distributed systems [Am11], TLS handshake [Bo18], fault-tolerant routing on a network-on-chip system [Zh16], system-on-chip cache coherence protocols [KS15] or autonomous cloud coordination managers [ASDP17]. The programming language Go [La] takes its use of channels from process algebras. The language Erlang [Er] uses a mechanism similar to synchronization in process algebras to exchange messages between concurrent light-weight processes. Many of the approaches to analyze such models of concurrent systems use Petri

[1] Munich University of Applied Science HM, Department of Computer Science and Mathematics, Lothstrasse 64, 80335 München, Germany matthias.guedemann@hm.edu

doi:10.18420/sicherheit2022_04

nets and are based on enumeration. While this allows for the analysis of complex properties it also makes the analyses susceptible to the state-space explosion problem.

Symbolic verification based on SAT and SMT solvers has become common in HW verification. Bounded model checking is very successful in bit-precise Software Model-checking [KT14]. On the one hand there is the drawback that these approaches are mainly limited to invariant properties. On the other hand, several interesting properties of concurrent systems can be expressed as invariant or reachability problems using observers. In these cases, it is possible to make use of the advanced algorithms, automated theorem provers and industrial strength model-checking tools for dataflow languages.

The main contribution of this work is the SMT-based Petri Net Analysis Technique (SpiNat) to analyze concurrent systems using synchronous languages. It is meant to be an approach orthogonal to other analysis methods of these systems. In its current form, it could be used in a portfolio approach to verification where different algorithms are used in parallel. The overall result is simply the first finished analysis. The first experimental results show its applicability to Petri net models of the Model Checking Contest (MCC).

2 Background

2.1 Petri Nets

Petri nets are a low-level formalism to describe concurrent systems. For example the CADP [Ga13] toolbox translates the LNT [GLS17] language into Petri nets for further analysis. Petri nets, also called place transition or P/T nets.

Definition 1 (Petri Net) *A Petri net is a tuple $N = (P, T, F)$ with a set of places P, a set of transitions T and a relation of arcs F where $F \subseteq (P \times T) \cup (T \times P)$*

Definition 2 (Marking) *For a Petri net $N = (P, T, F)$ a marking M is a mapping $M : P \mapsto \mathbb{N}$ which represents the number of tokens at each place.*

A marking of a Petri net therefore expresses the number of token on each place of the net.

Definition 3 (Pre- and post-set) *For a Petri net $N = (P, T, F)$ the pre-set is ${}^\bullet x := \{y|yFx\}$ and the post-set is $x^\bullet := \{y|xFy\}$.*

Definition 4 (Labelling and Enabled Transitions) *A labelling is a function $L : F \mapsto \mathbb{N}$ which assigns a natural number to each arc.*

A transition $t \in T$ is enabled *iff*

$$\forall p \in {}^{\bullet}t : L(p,t) \leq M(p)$$

This means that a transition t is *enabled* if each incoming arc (p,t) is labelled with a number less than or equal to the marking of the place p. Informally this means that each place in the pre-set of t must hold enough tokens to enable the transition.

Definition 5 (Transition Firing) *If a transition t is enabled and the current marking is M, then the transition t can* fire *which changes the marking to M'. This is often written as $M \xrightarrow{t} M'$, where for new marking M' the following holds*

$$\begin{aligned}\forall p :&(p \in {}^{\bullet}t \rightarrow M'(p) = M(p) - L(p,t))\\ \wedge&(p \in t^{\bullet} \rightarrow M'(p) = M(p) + L(t,p))\\ \wedge&(p \in P \setminus ({}^{\bullet}t \cup t^{\bullet}) \rightarrow M'(p) = M(p))\end{aligned}$$

Informally this means that when a transition t fires, the current marking changes by removing tokens from the places in the pre-set of t and adding tokens to the post-set of t. The number of tokens is equal to the label of the incoming (respective outgoing) arc of t.

Every Petri net has an initial marking M_I, i.e., an initial distribution of tokens on places. A marking M is *reachable* if there exists a finite sequence of transition firings such that the initial marking is transformed into M, i.e., $M_I \xrightarrow{t_1} \cdots \xrightarrow{t_n} M$. For more details see [Re13].

2.2 Model Checking using Satisfiability

Many of the most efficient approaches to verification use satisfiability or constraint problem formulation and rely on efficient SAT or SMT solvers to find solutions. SMT allows for a richer modeling language and can also use higher level reasoning similar to automated theorem provers. These tools support bit-precise formal analysis for fixed width bitvectors as well as additional decidable theories, e.g., the theory of unbounded integers for infinite state systems [KT14, CG12, La19].

The basic approach of bounded model-checking (BMC) unrolls the transition relation until a state that violates the property is reached. This method is incomplete but can be made complete using k-induction [BC00]. Newer approaches use for example the ic3/pdr [Br11] (property directed reachability) method to strengthen the property and to make it 1-inductive. In this paper we will use the verification of invariants or reachability checking to verify properties of Petri nets.

2.3 Lustre Dataflow Language

Dataflow languages are well studied and widely used for safety critical systems modeling and implementation. Lustre is a synchronous dataflow language which has a long history of use in safety critical systems in both commercial applications and academia. A program is built from hierarchical nodes. Special temporal operators allow the use of state variables with a defined initial value and a function to calculate the value for the next time step. At each time step the outermost node reads its inputs and uses these values to calculate simultaneously all values of each inner node and the output value of the outermost node. For details see [JRH16].

Several verification tools for Lustre have been developed, including commercial ones. There are modern open source industrial grade model checkers, kind2 [Ch16] and JKind [Ga18]. These tools allow for using different state of the art SMT engines, e.g., Z3 [DMB08] or CVC4 [Ba11] and use different algorithms in parallel in a portfolio approach to verification.

3 Expressing the Semantics of Petri Nets in Lustre

Here, the expression of Petri net semantics in Lustre follows an example driven explanation. A fully formal specification is under current development. In the SpiNat approach, Lustre state variables are used to encode the places of Petri nets. Every state variable is of type integer (unbounded) and holds the number of tokens at the corresponding place. Transition firing changes the marking of the Petri net, adjusting the value of the state variables corresponding to the impacted places.

Two transitions are *independent* if their post-sets (or pre-sets) have no common element. If progress is possible, then at least one transition of a set of independent transitions fires. This is ensured by defining one Boolean *activation* input parameter for each transition. This uses the *assert* mechanism of Lustre to add the constraint that if progress is possible, then exactly one transition set can have activated transitions. Each activated transition in a set of independent transitions fires if it is enabled, i.e., its incoming places hold enough tokens. Via an additional *assert* it is ensured that at each step if there is a transition that is enabled there also exists an activated and enabled transition, guaranteeing progress if possible.

3.1 Example Petri Net

Figure 1 shows an example Petri net [wi]. In the initial marking the place p_1 contains one token. All other places are empty, each arc is labelled with a weight of value 1. The encoding of the Petri net in Lustre works as follows. For each of the four transitions the *main* node of the Lustre model has one Boolean input parameter in the form of *act_t* for a transition t. Each of these parameters models the potential *activation* of a transition, an activated transition can fire if it is *enabled*, too. In the later analysis, a model-checker can used non-deterministic values for each activation input parameter of the transitions. The output of the *main* node is the content of the three places modeled as integers and the information which transition fires.

```
node main(act_t0, act_t1, act_t2, act_t3 : bool)
returns (p1,p2,p3 : int; firing_t0,firing_t1,firing_t2,firing_t3 : bool);
```

For each transition there exists a Boolean flag which is true iff the transition is enabled. For t_0 for example this means that p_1 and p_3 hold at least one token.

```
enable_t0 = p1 >= 1 and p3 >= 1;
```

As no transitions are independent, each transition set only contains a single element and each transition set is activated iff the contained transition is activated. Correct transition firing is modelled as follows via Lustre *assert*. This ensures that if a transition is enabled, then exactly one set of independent transitions has activated transitions. It also ensures that at least one activated transition is also enabled.

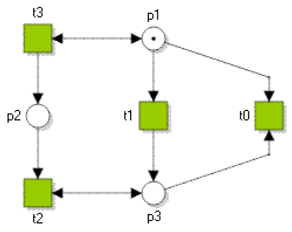

Fig. 1: Example Petri Net

```
trans_set0 = act_t0;
trans_set1 = act_t1;
trans_set2 = act_t2;
trans_set3 = act_t3;

assert (not (enable_t0 or enable_t1 or enable_t2 or enable_t3)
         or (bool2int(trans_set0) + bool2int(trans_set1) +
             bool2int(trans_set2) + bool2int(trans_set3) = 1));
assert ((enable_t0 or not act_t0) and (enable_t1 or not act_t1)
         and (enable_t2 or not act_t2) and (enable_t3 or not act_t3));
```

The first constraint models that if at least one transition is enabled, then exactly one independent set can contain activated transitions. In this example each transition set contains exactly one transition, the *bool2int* function equals 1 for true and 0 for false. Note, that if no transition is enabled a deadlock is reached and no progress is possible any more. The second constraint models that if there is an activated transition there is also enabled transition which fires.

For each transition t there exists a Boolean flag which represents that t fires, e.g., for transition t_0 in the example:

```
firing_t0 = act_t0 and enable_t0;
```

Places are modelled as state variables. The initial value of p_1 is 1, the value of p_1 in the next time step is defined by the expression after the arrow operator ->. The `pre` operator represents the value in the previous time step. A firing of transition t_3 removes a token from p_1 and adds a token to it. Therefore, in that case the change is the difference between the weight of the incoming arc and the weight of the outgoing arc which is 0. For the place p_1 in the example this translates to (cf. Def. 5):

```
p1 = 1 -> if pre firing_t0 then pre p1 - 1
  else if pre firing_t1 then pre p1 - 1
  else if pre firing_t3 then pre p1 - 0
  else  pre p1;
```

3.2 Petri Net Properties for SpiNat

There are some standard properties of Petri nets which can be directly expressed as invariants. For other properties full temporal logic is necessary and those are not yet supported in SpiNat.

- *no dead transition / quasi-liveness* in a marking is a transition property which states that every transition can be enabled eventually.
- *no dead places* is a property on places which states that no place exists which never holds a token.
- *deadlock* in a marking states that no transition is enabled and no more progress is possible.
- *n-safety* is a property on the number of tokens at each place. A net is called *n-safe* if there are at most n tokens on each place for each reachable marking. An often used special case of this property is *1-safety*.

Quasi-liveness is checked via recording whether at least one transition is never enabled. Any counterexample to this invariant proves quasi-liveness. Dead places exist if the invariant holds that at least one place has value zero in all reachable markings. This requires observing whether each place holds a non-zero value of tokens eventually. A deadlock corresponds to observing that no transition is enabled.

```
enabled_t0 = false -> if enable_t0 then true else pre enabled_t0;
marked_p1 = false -> if 0 < p1 then true else pre marked_p1;
deadlock = not (enable_t0 or enable_t1 or enable_t2 or enable_t3);
```

The *n-safety* property is the conjunction of the boundedness of the places. In the example 1-safety of the net is formulated as follows:

```
prop_oneSafe = 1 >= p1 and 1 >= p2 and 1 >= p3;
```

In the example no dead places exist, i.e., for each place p there is a reachable marking where p holds at least one token. The net is not 1-safe because after two firings of the transition t_3 there are 2 tokens at place p_2. There exists a deadlock, i.e., if t_1 fires in the initial marking, no other transition can fire afterwards. The transition t_0 can never be enabled, either p_1 or p_3 holds a token, but never both. For the example net in Fig. 1 we get the following analysis results for the above properties.

```
<Failure> prop_noDeadlock is invalid by PDR for k=1 after 0.035s.
<Failure> prop_oneSafe is invalid by BMC for k=2 after 0.035s.
<Failure> prop_deadPlaceExists is invalid by BMC for k=2 after 0.035s.
<Success> prop_deadTransExists is valid by PDR after 0.100s.
```

The above properties are generated automatically. This is done by generating the necessary observers states and the necessary Boolean variables during the transformation from the Petri net description to Lustre. Obviously there can be other interesting properties, but most often application and net specific properties which can be modelled manually.

4 Model Checking Examples

The Model Checking Contest (MCC) [mc] provides example Petri nets in the XML-based Petri Net Modeling Language (PNML) [Hi10] format. Many of the models are parametric and can therefore be analyzed in different sizes and/or difficulties. For this experimental study, a small subset of the available models and their parametric variants was selected and analyzed. It it interesting to compare different analysis tools for Lustre among each other and then also compare SpiNat with an analysis tool for Petri nets.

4.1 SpiNat Variants and Results

There exist different model checkers which can check Lustre programs. The most modern versions of these are JKind and kind2. JKind is written in Java and allows for using the built-in SMTInterpol [CHN12] SMT solver or an external one based on the standardized SMTLIB2 format [BFT17]. kind2 is a reimplementation of the pkind model checker. It is written in OCaml and supports several SMT solvers with Z3 being the standard choice. Both model checkers implement different analysis engines with the main ones being bounded model-checking, k-induction, invariant generation and pdr/ic3.

The analysis runs were executed with a 5-minute timeout using JKind version 4.4.4 using Z3 version 4.8.1[2] as external solver and kind2 1.4.0 with Z3 version 4.8.12 on a quad-core i7-8665U with 1.90 GHz. Timeouts are marked with —, ✓ marks a valid property and ∅ marks an invalid property. The elapsed time is shown in parentheses after the result. The numbers beside the name indicate the variant of the model. For the *no dead transitions* / *no dead places* properties, a validity means reaching a marking and an property corresponds to an invariant. For *no deadlock* and *1-safety* a valid property is proven as an invariant and an invalid one via reaching a marking.

Table 1 shows the results for some selected model variants, comparing the results of JKind and kind2. Both analysis tools implement a portfolio approach for their analyses and are capable to use multiple processors. Neither tool is strictly superior to the other. For each one there are models where it is better than the other and vice-versa. For example, JKind is significantly faster for the referendum-10 model whereas kind2 is significantly faster for the 2 PhaseLocking model. For the viral 03 1 1 2 model JKind manages to strictly analyze more properties than kind2, while for the satellite 100-3 model the opposite is true.

[2] More recent versions of Z3 are currently not supported because of a format change.

Name	no dead transitions	no dead places	no deadlock	1-safety
JKind				
GPU FP 40 a	∅$2m8.686s$	∅$4m1.755s$	∅$23.715ss$	—
CloudOps	∅$1.355s$	∅$1.355s$	—	∅$0.329s$
Satellite 100-3	—	✓$0.298s$	—	∅$0.235s$
Referendum 10	∅$6.587s$	∅$12.533s$	∅$0.336s$	✓$12.533s$
Election 2020	✓$0.754s$	✓$1.358s$	—	∅$0.429s$
2 PhaseLocking	✓$0.321s$	✓$0.321s$	∅$1m5.75s$	∅$0.228s$
Raft 02	∅$2m50.069s$	∅$2m50.069s$	—	—
viral 03 1 1 2	∅$1m25.486s$	$4m40.131s$	∅$1.271s$	—
kind2				
GPU FP 40 a	—	—	∅$51.575s$	—
CloudOps	∅$1.915s$	∅$1.915s$	—	∅$0.083s$
Satellite 100-3	—	✓$0.183s$	✓$22.284s$	∅$0.053s$
Referendum 10	∅$39.335s$	∅$39.335s$	∅$0.224s$	✓$3m34.42s$
Election 2020	✓$1.429s$	✓$3.345s$	—	∅$0.517s$
2 PhaseLocking	✓$0.242s$	✓$0.242s$	∅$23.010s$	∅$0.054s$
Raft 02	∅$3m9.877s$	∅$3m9.877s$	—	—
viral 03 1 1 2	—	—	∅$2.388s$	—

Tab. 1: Results for different SpiNat solver variants

4.2 Comparing SpiNat to TINA

Here we present more results with different models, comparing the results of an analysis using SpiNat with TINA [BRV04], a specialized Petri Net analysis tool which has been successful in the MCC.

The set of generated models for different examples of Petri nets from MCC is available[3]. The computer setup is the same as for the analyses in the previous section and used the same timeout of 5 minutes. The times reported for SpiNat are the minimum of runs with kind2 and JKind. The reasoning here is that using both solvers and keeping the first result is easy to do in a portfolio approach.

Whether it takes longer to prove an invariant or compute a reachable marking depends on how the net is structured. Generally SAT / SMT based analysis is well suited to prove invariants via inductive arguments and finding reachable markings which are *not far* from the initial marking. *Deep* markings which require a specific, long sequence of transition firings can be costly to compute, either in terms of memory for BMC or in terms of time for pdr.

[3] `https://guedemann.org/downloads/pnml_in_lustre.zip`

TINA has direct support for 1-safety and deadlock freeness, other properties need to be expressed explicitly in its modeling language. Table 2 shows the comparison of analysis results using TINA and SpiNat for deadlocks and 1-safety. For TINA this was done using the *sift* tool using the `-dead` and `-f safe` option.

	no deadlock			1-safety		
Name	valid	SpiNat	TINA	valid	SpiNat	TINA
GPU FP 24 a	∅	8.073s	0.013s	—	—	—
GPU FP 28 a	∅	15.27s	0.017s	—	—	—
GPU FP 32 a	∅	13.643s	0.029s	—	—	—
GPU FP 36 a	∅	18.204s	0.067s	—	—	—
GPU FP 40 a	∅	23.715s	0.071s	—	—	—
CloudOps	—	—	—	∅	0.083s	0.00s
Satellite 100-3	✓	22.284s	0.160s	∅	0.053s	0.00s
Satellite 1000-32	✓	28.325s	25.791s	∅	0.053s	0.00s
Satellite 1500-46	✓	27.352s	1m2.946s	∅	0.084s	0.00s
Satellite 3000-94	✓	27.159s	3m36.982s	∅	0.074s	0.00s
Satellite 65535-2048	✓	28.814s	—	∅	0.074s	0.00s
Referendum 10	∅	0.224s	0.349s	✓	12.533s	0.361s
Referendum 15	∅	0.307s	2m13.568s	✓	34.352s	2m15.971s
Referendum 20	∅	0.417s	—	✓	1m12.267s	—
Referendum 50	∅	1.036s	—	—	—	—
Noc3x3 1 a	✓	3m24.345s	—	—	—	—
Election 2020	—	—	—	∅	0.429s	0.00s
2 PhaseLocking	∅	23.01s	0.002s	∅	0.054s	0.00s
Raft 02	✓	—	0.045s	✓	—	0.026s
viral 03 1 1 02	∅	2.388s	3.851s	—	—	—
viral 04 1 1 02	∅	2.858s	—	—	—	—
viral 04 1 1 03	∅	27.464s	—	—	—	—
viral 08 1 1 02	∅	1m52.008s	—	∅	1.375s	0.00s

Tab. 2: Comparing SpiNat and TINA on Selected Petri Net Models

It is clear that in some cases TINA is more efficient in its analysis and in other cases SpiNat is faster. If a counterexample can be found close to the initial marking TINA is efficient, almost independent of the size of the model variant, e.g., as seen for the GPU model variants. In contrast, for the satellite model the whole state space must be constructed for the valid deadlock property. The state space grows quickly, increasing the analysis time for TINA. For this model the induction based verification of SpiNat shows almost no increase in run-time for the deadlock property.

5 Related Work

Most Petri net analysis and verification methods are based on enumerative approaches. A lot of interesting properties are expressed in temporal logic which is more expressive than the simple invariants here. In [PCM14] the authors apply SMT based verification on a constrained subset of timed Petri nets. They use BMC which allows for checking reachable markings but cannot prove invariants. BMC also has the challenge that with every time step the problem for the underlying solver gets bigger and more complex.

In [ABDZ21] the authors combine SMT-based verification with an abstraction technique called polyhedral abstraction for Petri nets. They show that this works well together and allows for solving a lot of instances of the MCC competition. A similar approach is followed in [TM20] and the corresponding ITS-Tools. It would be interesting to see how to combine these approaches with the one described here.

In [BSS09] the authors propose a related approach which expresses the semantics of Lustre in an asynchronous framework called BIP (Behavior, Interaction, Priority). BIP models are then executed via a transformation into a special form of Petri nets. It could be interesting to see if it is possible to combine the framework in with SpiNat to allow for analysis of systems with both synchronous and asynchronous behavior.

6 Conclusion and Outlook

This work shows that it is possible to express Petri nets in the synchronous dataflow language Lustre. This allows for using tools like JKind and kind2 to analyze certain properties of critical systems with concurrency. The approach is orthogonal to other verification methods. First experiments show that there are properties and Petri nets where the required analysis time is comparable or even better than specialized Petri net tools.[4] Using SMT-based verification for Petri nets is not new but to my knowledge dataflow languages as intermediate representation have not yet been proposed.

There is room to increase the efficiency of the current implementation. For larger Lustre models the parsers of the model checkers sometimes have problems. Therefore it might make sense to also provide the possibility to use them as libraries and use their API instead of text files to exchange models.

The current implementation supports basic Petri nets. A possible extension would be support for colored Petri nets and hierarchic Petri nets. To increase efficiency of the analyses it would make sense to support annotations, e.g., convert the information that a system is *n-safe* into a system constraint in order to speed up verification of other properties. By using *incremental inductive CTL* [HBS12] it would be possible to support temporal logic properties. This technique would have to be integrated into the model checkers directly.

[4] **Threat to validity** The author is no expert on using the TINA toolbox. Therefore, It is possible that with different options and / or some pre-processing, TINA can be faster than reported in the table.

Bibliography

[ABDZ21] Amat, Nicolas; Berthomieu, Bernard; Dal Zilio, Silvano: , On the Combination of Polyhedral Abstraction and SMT-based Model Checking for Petri nets, 2021.

[Am11] Ameur-Boulifa, Rabéa; Halalai, Raluca; Henrio, Ludovic; Madelaine, Eric: Verifying Safety of Fault-Tolerant Distributed Components – Extended Version. Research Report RR-7717, INRIA, September 2011.

[ASDP17] Abid, Rim; Salaün, Gwen; De Palma, Noel: Asynchronous synthesis techniques for coordinating autonomic managers in the cloud. Science of Computer Programming, 146:87–103, 2017.

[Ba11] Barrett, Clark; Conway, Christopher L; Deters, Morgan; Hadarean, Liana; Jovanović, Dejan; King, Tim; Reynolds, Andrew; Tinelli, Cesare: CVC4. In: International Conference on Computer Aided Verification. Springer, pp. 171–177, 2011.

[BC00] Bjesse, Per; Claessen, Koen: SAT-based verification without state space traversal. In: International Conference on Formal Methods in Computer-Aided Design. Springer, pp. 409–426, 2000.

[BFT17] Barrett, Clark; Fontaine, Pascal; Tinelli, Cesare: The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[Bo18] Bozic, Josip; Marsso, Lina; Mateescu, Radu; Wotawa, Franz: A formal TLS handshake model in LNT. arXiv preprint arXiv:1803.10319, 2018.

[Br11] Bradley, Aaron R: SAT-based model checking without unrolling. In: International Workshop on Verification, Model Checking, and Abstract Interpretation. Springer, pp. 70–87, 2011.

[BRV04] Berthomieu, Bernard; Ribet, P-O; Vernadat, François: The tool TINA–construction of abstract state spaces for Petri nets and time Petri nets. International journal of production research, 42(14):2741–2756, 2004.

[BSS09] Bozga, Marius; Sfyrla, Vassiliki; Sifakis, Joseph: Modeling Synchronous Systems in BIP. In (Chakraborty, Samarjit; Halbwachs, Nicolas, eds): 9th ACM & IEEE International conference on Embedded software, EMSOFT 2009. ACM, Grenoble, France, pp. 77–86, October 2009.

[CG12] Cimatti, Alessandro; Griggio, Alberto: Software model checking via IC3. In: International Conference on Computer Aided Verification. Springer, pp. 277–293, 2012.

[Ch16] Champion, Adrien; Mebsout, Alain; Sticksel, Christoph; Tinelli, Cesare: The Kind 2 model checker. In: International Conference on Computer Aided Verification. Springer, pp. 510–517, 2016.

[CHN12] Christ, Jürgen; Hoenicke, Jochen; Nutz, Alexander: SMTInterpol: An interpolating SMT solver. In: International SPIN Workshop on Model Checking of Software. Springer, pp. 248–254, 2012.

[DMB08] De Moura, Leonardo; Bjørner, Nikolaj: Z3: An efficient SMT solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, pp. 337–340, 2008.

[Er] Erlang: , The Erlang Programming Language. https://www.erlang.org/.

[Ga13] Garavel, Hubert; Lang, Frédéric; Mateescu, Radu; Serwe, Wendelin: CADP 2011: a toolbox for the construction and analysis of distributed processes. International Journal on Software Tools for Technology Transfer, 15(2):89–107, 2013.

[Ga18] Gacek, Andrew; Backes, John; Whalen, Mike; Wagner, Lucas; Ghassabani, Elaheh: The JKind model checker. In: International Conference on Computer Aided Verification. Springer, pp. 20–27, 2018.

[GLS17] Garavel, Hubert; Lang, Frédéric; Serwe, Wendelin: From LOTOS to LNT. In: ModelEd, TestEd, TrustEd, pp. 3–26. Springer, 2017.

[HBS12] Hassan, Zyad; Bradley, Aaron R; Somenzi, Fabio: Incremental, inductive CTL model checking. In: International Conference on Computer Aided Verification. Springer, pp. 532–547, 2012.

[Hi10] Hillah, Lom-Messan; Kordon, Fabrice; Petrucci, Laure; Treves, Nicolas: PNML Framework: an extendable reference implementation of the Petri Net Markup Language. In: Proceedings of Petri Nets. Springer, pp. 318–327, 2010.

[Ho78] Hoare, Charles Antony Richard: Communicating sequential processes. Communications of the ACM, 21(8):666–677, 1978.

[JRH16] Jahier, Erwan; Raymond, Pascal; Halbwachs, Nicolas: The Lustre V6 reference manual. Verimag, Grenoble, Dec, 2016.

[KS15] Kriouile, Abderahman; Serwe, Wendelin: Using a formal model to improve verification of a cache-coherent system-on-chip. In: Proceedings of TACAS. Springer, 2015.

[KT14] Kroening, Daniel; Tautschnig, Michael: CBMC–C bounded model checker. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, pp. 389–391, 2014.

[La] Language, Go Programming: , The Go Programming Language. https://golang.org/.

[La19] Lange, Tim; Neuhäußer, Martin R; Noll, Thomas; Katoen, Joost-Pieter: IC3 software model checking. International Journal on Software Tools for Technology Transfer, pp. 1–27, 2019.

[mc] mcc: , Model Checking Contest. https://mcc.lip6.fr/models.php.

[PCM14] Półrola, Agata; Cybula, Piotr; Męski, Artur: SMT-based reachability checking for bounded time Petri nets. Fundamenta Informaticae, 135(4):467–482, 2014.

[Re13] Reisig, Wolfgang: Understanding Petri Nets: Modeling techniques, analysis methods, case studies. Springer, 2013.

[Sa] Satellite: , Memory Model. https://mcc.lip6.fr/pdf/SatelliteMemory-form.pdf.

[TM20] Thierry-Mieg, Yann: Structural Reductions Revisited. In: International Conference on Applications and Theory of Petri Nets and Concurrency. Springer, pp. 303–323, 2020.

[wi] wikipedia: , Petri Net Example. https://en.wikipedia.org/wiki/Petri_net.

[Zh16] Zhang, Zhen; Serwe, Wendelin; Wu, Jian; Yoneda, Tomohiro; Zheng, Hao; Myers, Chris: An improved fault-tolerant routing algorithm for a network-on-chip derived with formal analysis. Science of Computer Programming, 118:24–39, 2016.

A Satellite Model Example

The Satellite Memory model is an industrial case study from the aerospace domain. It has been submitted to MCC as an example model and is described in [Sa]. It is a Petri net with multiple variants. Figure 2 shows the initial marking for the variant $X = 100$ and $Y = 6$.

This models a satellite which contains memory in the form of a circular buffer. The size of the buffer in the number of available sectors is modelled as the parameter X. The circular buffer has a write pointer and a read pointer. The parameter Y models the minimal difference between the sector currently used for writing and the sector used for reading. More details on the case study can be found in [Sa].

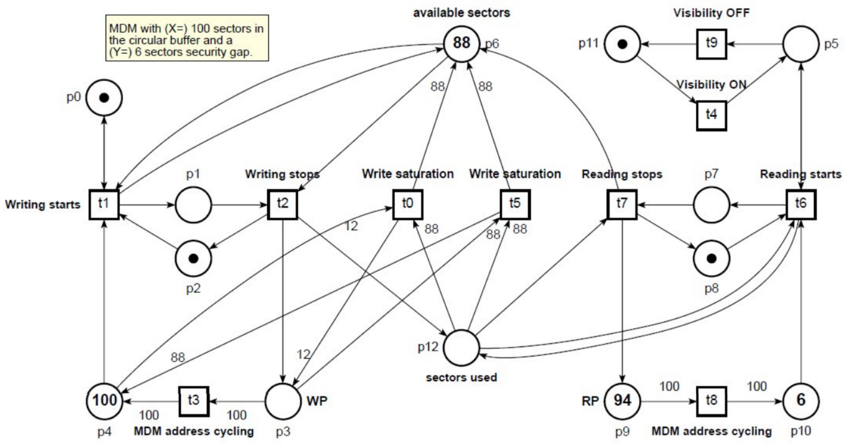

Fig. 2: Satellite Memory Model Variant $X = 100, Y = 6$ [Sa]

A.1 Lustre version of the Satellite Model

```
node main(act_t0, act_t1, act_t2, act_t3, act_t4, act_t5, act_t6,
   act_t7, act_t8, act_t9 : bool)

returns (p0, p1, p10, p11, p12, p2, p3, p4, p5, p6, p7, p8, p9 : int;
  firing_t0, firing_t1, firing_t2, firing_t3, firing_t4, firing_t5,
  firing_t6, firing_t7, firing_t8, firing_t9 : bool);

var
deadlock : bool;

trans_set0, trans_set1, trans_set2, trans_set3, trans_set4 : bool;

enabled_t0, enabled_t1, enabled_t2, enabled_t3, enabled_t4,
   enabled_t5, enabled_t6, enabled_t7, enabled_t8, enabled_t9 : bool;
```

```
enable_t0, enable_t1, enable_t2, enable_t3, enable_t4, enable_t5,
   enable_t6, enable_t7, enable_t8, enable_t9 : bool;

marked_p0, marked_p1, marked_p10, marked_p11, marked_p12, marked_p2, marked_p3,
  marked_p4, marked_p5, marked_p6, marked_p7, marked_p8, marked_p9 : bool;

prop_oneSafe : bool;
prop_deadTransExists : bool;
prop_deadPlaceExists  : bool;
prop_noDeadlock : bool;

let
trans_set0 = act_t0 or act_t4 or act_t8;
trans_set1 = act_t1 or act_t6;
trans_set2 = act_t2;
trans_set3 = act_t3 or act_t7 or act_t9;
trans_set4 = act_t5;

assert (not (enable_t0 or enable_t1 or enable_t2 or enable_t3 or enable_t4
          or enable_t5 or enable_t6 or enable_t7 or enable_t8 or enable_t9)
        or (bool2int(trans_set0) + bool2int(trans_set1) +  bool2int(trans_set2)
            + bool2int(trans_set3) + bool2int(trans_set4) = 1));

assert ((enable_t0 or not act_t0) and (enable_t1 or not act_t1) and
        (enable_t2 or not act_t2) and (enable_t3 or not act_t3) and
        (enable_t4 or not act_t4) and (enable_t5 or not act_t5) and
        (enable_t6 or not act_t6) and (enable_t7 or not act_t7) and
        (enable_t8 or not act_t8) and (enable_t9 or not act_t9));

enable_t0 = true and p12 >= 94 and p4 >= 6;
enable_t1 = true and p0 >= 1 and p2 >= 1 and p4 >= 1 and p6 >= 1;
enable_t2 = true and p1 >= 1 and p6 >= 1;
enable_t3 = true and p3 >= 100;
enable_t4 = true and p11 >= 1;
enable_t5 = true and p12 >= 94 and p3 >= 94;
enable_t6 = true and p10 >= 1 and p12 >= 1 and p5 >= 1 and p8 >= 1;
enable_t7 = true and p12 >= 1 and p7 >= 1;
enable_t8 = true and p9 >= 100;
enable_t9 = true and p5 >= 1;

deadlock = not (enable_t0 or enable_t1 or enable_t2 or enable_t3 or enable_t4
             or enable_t5 or enable_t6 or enable_t7 or enable_t8 or enable_t9);

p1 = 0 -> if pre firing_t1 then pre p1 + 1
  else if pre firing_t2 then pre p1 - 1
  else  pre p1;

p12 = 0 -> if pre firing_t0 then pre p12 - 94
  else if pre firing_t2 then pre p12 + 1
  else if pre firing_t5 then pre p12 - 94
  else if pre firing_t6 then pre p12 - 0
  else if pre firing_t7 then pre p12 - 1
  else  pre p12;
```

```
p3 = 0 -> if pre firing_t0 then pre p3 + 6
  else if pre firing_t2 then pre p3 + 1
  else if pre firing_t3 then pre p3 - 100
  else if pre firing_t5 then pre p3 - 94
  else  pre p3;

p5 = 0 -> if pre firing_t4 then pre p5 + 1
  else if pre firing_t6 then pre p5 - 0
  else if pre firing_t9 then pre p5 - 1
  else  pre p5;

p7 = 0 -> if pre firing_t6 then pre p7 + 1
  else if pre firing_t7 then pre p7 - 1
  else  pre p7;

p0 = 1 -> if pre firing_t1 then pre p0 - 0
  else  pre p0;

p11 = 1 -> if pre firing_t4 then pre p11 - 1
  else if pre firing_t9 then pre p11 + 1
  else  pre p11;

p2 = 1 -> if pre firing_t1 then pre p2 - 1
  else if pre firing_t2 then pre p2 + 1
  else  pre p2;

p8 = 1 -> if pre firing_t6 then pre p8 - 1
  else if pre firing_t7 then pre p8 + 1
  else  pre p8;

p10 = 3 -> if pre firing_t6 then pre p10 - 1
  else if pre firing_t8 then pre p10 + 100
  else  pre p10;

p6 = 94 -> if pre firing_t0 then pre p6 + 94
  else if pre firing_t1 then pre p6 - 0
  else if pre firing_t2 then pre p6 - 1
  else if pre firing_t5 then pre p6 + 94
  else if pre firing_t7 then pre p6 + 1
  else  pre p6;

p9 = 97 -> if pre firing_t7 then pre p9 + 1
  else if pre firing_t8 then pre p9 - 100
  else  pre p9;

p4 = 100 -> if pre firing_t0 then pre p4 - 6
  else if pre firing_t1 then pre p4 - 1
  else if pre firing_t3 then pre p4 + 100
  else if pre firing_t5 then pre p4 + 94
  else  pre p4;

firing_t0 = act_t0 and p12 >= 94 and p4 >= 6;
firing_t1 = act_t1 and p0 >= 1 and p2 >= 1 and p4 >= 1 and p6 >= 1;
```

```
firing_t2 = act_t2 and p1 >= 1 and p6 >= 1;
firing_t3 = act_t3 and p3 >= 100;
firing_t4 = act_t4 and p11 >= 1;
firing_t5 = act_t5 and p12 >= 94 and p3 >= 94;
firing_t6 = act_t6 and p10 >= 1 and p12 >= 1 and p5 >= 1 and p8 >= 1;
firing_t7 = act_t7 and p12 >= 1 and p7 >= 1;
firing_t8 = act_t8 and p9 >= 100;
firing_t9 = act_t9 and p5 >= 1;

enabled_t0 = false -> if enable_t0 then true else pre enabled_t0;
enabled_t1 = false -> if enable_t1 then true else pre enabled_t1;
enabled_t2 = false -> if enable_t2 then true else pre enabled_t2;
enabled_t3 = false -> if enable_t3 then true else pre enabled_t3;
enabled_t4 = false -> if enable_t4 then true else pre enabled_t4;
enabled_t5 = false -> if enable_t5 then true else pre enabled_t5;
enabled_t6 = false -> if enable_t6 then true else pre enabled_t6;
enabled_t7 = false -> if enable_t7 then true else pre enabled_t7;
enabled_t8 = false -> if enable_t8 then true else pre enabled_t8;
enabled_t9 = false -> if enable_t9 then true else pre enabled_t9;

marked_p0 = false -> if 0 < p0 then true else pre marked_p0;
marked_p1 = false -> if 0 < p1 then true else pre marked_p1;
marked_p10 = false -> if 0 < p10 then true else pre marked_p10;
marked_p11 = false -> if 0 < p11 then true else pre marked_p11;
marked_p12 = false -> if 0 < p12 then true else pre marked_p12;
marked_p2 = false -> if 0 < p2 then true else pre marked_p2;
marked_p3 = false -> if 0 < p3 then true else pre marked_p3;
marked_p4 = false -> if 0 < p4 then true else pre marked_p4;
marked_p5 = false -> if 0 < p5 then true else pre marked_p5;
marked_p6 = false -> if 0 < p6 then true else pre marked_p6;
marked_p7 = false -> if 0 < p7 then true else pre marked_p7;
marked_p8 = false -> if 0 < p8 then true else pre marked_p8;
marked_p9 = false -> if 0 < p9 then true else pre marked_p9;

prop_noDeadlock = not deadlock;

prop_deadTransExists = (not enabled_t0) or (not enabled_t1) or (not enabled_t2)
 or (not enabled_t3) or (not enabled_t4) or (not enabled_t5) or (not enabled_t6)
 or (not enabled_t7) or (not enabled_t8) or (not enabled_t9);

prop_deadPlaceExists = (not marked_p0) or (not marked_p1) or (not marked_p10)
    or (not marked_p11) or (not marked_p12) or (not marked_p2) or (not marked_p3)
    or (not marked_p4) or (not marked_p5) or (not marked_p6) or (not marked_p7)
    or (not marked_p8) or (not marked_p9);

prop_oneSafe = 1 >= p0 and 1 >= p1 and 1 >= p10 and 1 >= p11 and
  1 >= p12 and 1 >= p2 and 1 >= p3 and 1 >= p4 and 1 >= p5 and
  1 >= p6 and 1 >= p7 and 1 >= p8 and 1 >= p9;

--%PROPERTY prop_noDeadlock;
--%PROPERTY prop_oneSafe;
--%PROPERTY prop_deadTransExists;
--%PROPERTY prop_deadPlaceExists;
tel;
```