Nubolic Simulation of AMS Systems with Data Flow and Discrete Event Models

Carna Zivkovic and Christoph Grimm
University of Kaiserslautern, Germany
{zivkovic, grimm}@cs.uni-kl.de

***Abstract*—This paper deals with the performance verification of analog/mixed-signal (AMS) systems by a mixed symbolic/numeric (nubolic) simulation. The approach is to piggyback the symbolic simulation via code-instrumentation on the existing, numeric SystemC AMS simulator. This permits the combination of symbolic and numeric simulation ("nubolic simulation"). The particular focus in the paper is the handling of the symbolic discrete-event process activations. This permits the symbolic simulation of digital parts of AMS systems modeled by discrete event processes. The approach is demonstrated by the symbolic simulation of a dual-charge-pump PLL of an IEEE 802.15.4 RF transceiver that includes an asynchronous digital counter as a frequency divider.**

***Index Terms*—symbolic simulation, affine arithmetic, decision diagrams**

I. Introduction

In today's integrated circuits and system, in particular, the analog/mixed-signal (AMS) parts introduce challenges for verification. According to an industrial evaluation [1], 75% of design risks come from these parts. Performance verification of AMS systems shows that in presence of parameter variations, e.g. process, voltage, temperature variations, the performance properties of a system, e.g. the locking time of a PLL, remain in a specified range. The common approach is the use of multi-run simulation techniques like Monte Carlo (MC) analysis or Corner Case (CC) analysis.

Formal methods, e.g. [2]–[6], promise comprehensive coverage. However, they hardly scale with the complexity and heterogeneity of today's mixed-signal circuits [7]. Furthermore, the use of hybrid automata [2], [3], recurrence equations [4] , or hybrid Petri nets [5] introduces a gap towards the circuit or system level models that are usually given as a SPICE netlist or in modeling languages such as VHDL-AMS or SystemC AMS.

To some extent, compilers can overcome this gap. However, there is a number of drawbacks: First, such compilers support only a usually small subset and suffer from incompatibilities; see [8], [9] for details. Second, due to lack of scalability, it is often useful to focus formal or symbolic methods on some signal paths only and handle other parts of a design by numeric simulation. Then, a tight integration of formal/symbolic methods and numeric simulation is needed.

The approach taken in this paper combines numeric and symbolic simulation to a *nubolic* simulation. For this purpose, we piggyback symbolic methods via code instrumentations on SystemC (AMS) models and use the existing, numerical simulator. [6], [10] describe the symbolic representation by affine arithmetic decision diagrams (AADD). [11] introduces the instrumentation of control flow statements. In [6], [10]–[12] the approach is limited to the timed data flow model of computation or continuous-time models.

The novelty of this paper is the

- support for symbolic activations of discrete event processes, and
- demonstration by the nubolic simulation of a PLL with real complexity.

Note, that some tools for concolic testing of software [13], [14] use a similar approach. Concolic testing piggybacks symbolic methods via code instrumentations on a program to be tested. For example, [14] couples Python with the automated theorem prover Z3. For this purpose, Python's integer objects, operators and functions are replaced by a symbolic type that produces the Z3 input language.

Similar to [14], we introduce symbolic methods by overloading C++'s double data type, operators, functions, and even the if- and while statements. Unlike [14], we have a simulator instead of a compiler, a procedural language with possible side effects instead of a functional language, and we strive to find worst-case properties instead of test cases with comprehensive path coverage.

We give an overview of the nubolic approach in Sec. II. In Sec. III we describe the new method that permits the symbolic execution of discrete event processes. In Sec. IV we describe the nubolic simulation of the PLL of an IEEE 802.15.4 RF transceiver for verification of the worst-case locking time. Sec. V gives a discussion of the results and a conclusion.

II. Nubolic verification of AMS systems

A. Overview of the approach

Figure 1 shows an overview of the verification flow. SystemC [15] resp. SystemC AMS [16] models are generated from a circuit level model via a netlister [17]. The PVT variations from the circuit model can be characterized and added to the SystemC/SystemC AMS model as described in [18] as affine arithmetic decision diagrams (AADDs). [10] describes the modeling of deterministic and probabilistic variations of discrete and continuous nature. Assertions then check the performance properties in a mixed numeric/symbolic simulation run.

In the context of this paper we focus in particular on the following steps:

978-3-9819263-2-3/DATE19/©2019 EDAA

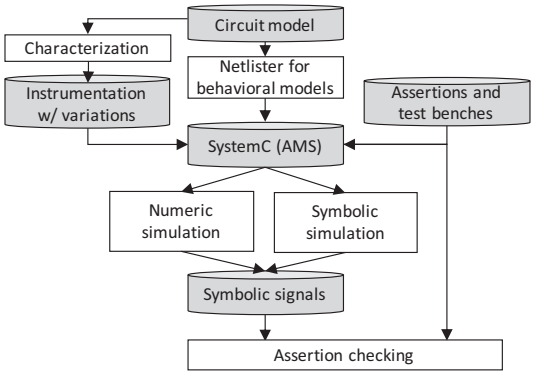

Fig. 1: Overview of nubolic verification of AMS systems.

1) *Instrumentation* specifies which variables or signals are simulated symbolically.
2) *Simulation* is done symbolically where the symbolic types are used; otherwise simulation is numerical. Both cases use the pre-existing SystemC implementation.
3) *Assertion checking* checks performance properties within simulation and reports violations; see [19].

B. Instrumentation

Code instrumentations change the semantics of SystemC resp. SystemC AMS from numeric to a symbolic:

- For variables for which symbolic simulation is desired, the symbolic types `doubleS`, `intS` and `boolS` must be used. Such variables compute reachability by AADDs resp. BDDs as described in Subsec. II-C.
- The header file `aadd.h` provides these types, overloaded operators, and macros that piggyback symbolic simulation on the existing SystemC (AMS).

Note that in symbolic simulation control flow statements must consider all possible outcomes. This is done by the C++ macros ifS, elseS, whileS and endS instead of the respective C++ keywords. The above macros are described in more detail in [11].

Fig. 2 left shows a small example in C++. In line 3, a symbolic variable a is declared that is initialized with the range $[-1, 1]$. The condition in line 4 can hence not be evaluated to true or false. Therefore, the macros `ifS, elseS, endS` (added by preprocessor) will evaluate both paths and not either the if- or the else path.

C. Symbolic computation with AADD

For symbolic computations, we use affine arithmetic decision diagrams (AADD, [6], [12]). AADD combine the properties of affine forms with those of reduced ordered binary decision diagrams (ROBDD).

Affine arithmetic [20] is used to compute with ranges resp. convex spaces. Affine arithmetic represents ranges by affine forms that are a linear model $f^a(\varepsilon_1, \ldots, \varepsilon_n)$ of its dependency from noise symbols $\varepsilon_1, \ldots, \varepsilon_n$. This permits the accurate computation with intervals.

BDD represent the path condition of an AADD:

```
0:#include "aadd.h"
1:void main(){
2:// a is from range [-1, 1]
3: doubleS a(-1, 1);
4: ifS (a>=0) a=a+1;
5: elseS a=a-1; endS}
```

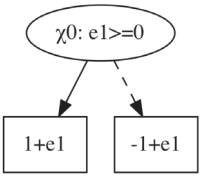

Fig. 2: AADD of a after the control-flow statement.

- Internal nodes are labeled with literals χ_i of the path condition in the form $f^a(\varepsilon_1, \ldots, \varepsilon_n) \geq 0$, and
- Leaves are affine forms that represent the possible ranges of symbolic variables .

As all $f^a(\varepsilon_1, \ldots, \varepsilon_n)$ are linear terms, every path condition is a linear inequation system that can be solved efficiently. This significantly improves scalability.

AADDs are instantiated by the declaration of variables of the type `doubleS` or `intS`. Arithmetic and logic operators of C++ are overloaded to do symbolic operations as described in [6]. The macros ifS, elseS, whileS and endS track the path condition that becomes part of each AADD when it is assigned in a (maybe nested) control flow statement [11]. For example, Fig. 2 right shows the AADD of a after the ifS-elseS statement.

Currently, C++ is supported including nested selection and iteration statements; limitations are goto statements that could easily be added and pointer for which the linear inclusions of AADD would be meaningless. In consequence, SystemC and SystemC AMS models can be simulated as long as there is no symbolic interaction with the simulation kernels. This holds for the statically scheduled timed data flow model of computation of SystemC AMS and some discrete event processes that are sensitive to e.g. a deterministic clock signal.

III. Nubolic simulation of processes with symbolic activation

A. Symbolic signals and symbolic process activation

We first formalize some aspects of signals and the models of computation (MoC) used to implement the simulation. We build on definitions from Jantsch [21] which is equivalent to the tagged signal model [22]. For the nubolic simulation, we have to keep the existing MoC as much as possible unchanged. We assume that time is always a concrete variable and focus on symbolic signals and, maybe symbolic, activation of processes.

Value-symbolic, time-symbolic and concrete signals: A signal is a sequence $\langle e_1, \ldots, e_n \rangle$ of events. An event $e_i = (v_i, t_i), i \in \{1, \ldots, n\}$ consists of a value v_i and a time tag t_i with $t_i > t_{i-1}$ where values are from a set V and time tags from an at least partially ordered set T (time base, simulated time). A signal is value-symbolic if at least one event's value is represented by a symbol, e.g. an AADD or a BDD, that represent more than a single value. It is time-symbolic if at least one time-tag is symbolic and represents more than a single value. Signals that are not value-symbolic and not time-symbolic are concrete.

Symbolic and concrete activation: Each simulator, including analog simulators, can be seen as a network of processes that compute output signals from input signals. More formally, a simulator can be seen as a network of processes P, connected by signals S. Each process $p \in P$ is a tuple $p = (I, O, proc, a)$ where

- $I \subseteq S, O \subseteq S$ are its input resp. output signals.
- $proc$ is a processing method. It uses the input signals I and simulated time T to compute the output signals O, maybe using internal states.
- a is the activation condition; $proc$ is executed for each $t \in T$ where $a = true$.

The activation condition is concrete (deterministic), if its values are either $true$ or $false$. The activation condition is symbolic (uncertain), if at least one of its values is a symbol that represents more than one possible values, i.e. by a BDD.

B. Timed data flow and continuous-time MoC

In the *timed data flow (TDF)* MoC of SystemC AMS, there are n processes that communicate via buffers. A schedule of the process activation is determined statically before the start of the simulation. This schedule is repeated in discrete time steps. Hence, a depends only on the simulated time $t \in T$. We assume that the simulated time and time steps are numeric variables. The activation of processes and the simulation algorithm is independent of inputs, outputs, and states. Therefore, for the nubolic simulation of models in the TDF MoC, it is sufficient to compute $proc$ in a symbolic way using the instrumentations described in Sec. II. No further support for symbolic simulation is needed.

For the *continuous-time (CT)* MoC, the same holds as for the TDF MoC, with restrictions regarding time step control and numerical methods. For SystemC AMS we support CT transfer functions that are embedded in the TDF MoC. They are computed by difference equations that approximate the transfer function $H(s)$; like for the TDF MoC, no further actions are needed.

C. Discrete Event MoC

In the discrete event (DE) MoC we have a more complex simulation algorithm. In the DE simulation we have processes P where:

- the activation condition a of each $p \in P$ is a function $f : I_a \times T \to \mathbb{B}$; T is the simulated time and $I_a \subseteq I$ is the set of input signals to which $p \in P$ is sensitive to.
- $p_j, p_k \in P$ with $a_j = true$ and $a_k = true$ for the same $t \in T$ are executed in a non-deterministic order.

The non-deterministic order of process executions is overcome by the concept of delta-time and 'buffered' signals that synchronously get its assigned values after an update-cycle.

We make the following premises:

1) We assume that all signals are concrete or value-symbolic, but not time-symbolic.
2) We assume that the activation condition a does not use symbolic values for the simulated time.

The first premise is not limiting because a time-symbolic signal can also be written as a value-symbolic signal; the underlying idea is described in [10]. The second premise must be made as otherwise we would have no discrete points in time when to execute a process; this would have serious implications on the overall DE simulation algorithm.

The method for nubolic simulation of DE processes piggybacks symbolic simulation on the DE simulator in two steps:

(1) Before the start of simulation, all DE processes are registered in a global data structure as shown in Listing 1.
(2) Within simulation, at each time at which a value of a symbolic signal $i_a \in I_a$ changes the algorithm shown in Listing 2 is executed.

Listing 1: Symbolic process registry.

```
1: Type SprocessRecord: (Process p, BDD a, set of
    signals I_a)
2: global: set of SprocessRecord PR
```

The process registry PR maintains process records pr with the following elements:

- A reference to each process $p \in P$,
- Its symbolic activation condition a, initialized to a BDD with one leaf node whose value is $false$.
- References to symbolic input signals $I_a \subseteq I$ on which the activation function depends.

In the update-cycles of the DE simulation, an activation condition a is eventually computed. Then, the method in Listing 2 is executed. It will execute the method $proc()$ of each $p \in P$ if its activation condition a is $true$ or symbolic (lines 9-10).

All assignments called within $proc()$ will be made only for $true$ values of activation conditions; this is done using ITE function in the overloaded assignment operators for `AADD` and `BDD` [11]. Hence, symbolic process activation conditions can be treated like conditions in control flow statements [11]:

- A copy of a of $p \in P$ is put on the stack of block conditions.
- ITE function is applied in all assignments within the method $proc()$.
- The copy of a is popped from the stack and the value of a is reset to $false$.

Listing 2: Evaluation of process activation conditions

```
0: METHOD Update()
1:  input: symbolic signal s, time t ∈ T
3:  foreach pr ∈ PR do
4:    foreach i_a ∈ pr.I_a do
5:      if (s == i_a) then
6:        pr.a = pr.a ∨ f(s,t);
7:      break;
8:    end for
9:    if (pr.a ≠ false)
10:     notify_next_cycle();
11:   end for
```

D. Integration in SystemC

In SystemC, DE processes are defined by the macros `SC_METHOD(proc)` and `SC_THREAD(proc)` where `proc` is a method of an SC_MODULE.

Activation conditions are specified by the signal events in a sensitivity list of `SC_METHOD(proc)` or by time constraints in `wait` statements of `SC_THREAD(proc)`. As we assume here that time is always concrete, the process activation in `SC_THREAD(proc)` is not influenced by symbolic signals.

However, for SystemC signal events in the sensitivity list this is not the case. In SystemC the following events can be applied on signals:

- *Value change event*. This event triggers a process $p \in P$ at $t \in T$ if a value of a signal $i_a \in I_a$ at the t changes.
- *Rising and falling edge events*. Rising edge event triggers a process $p \in P$ at $t \in T$ if a value of a signal $i_a \in I_a$ at the t changes and the signal value is 1. Falling edge event triggers a process $p \in P$ at $t \in T$ if a value of a signal $i_a \in I_a$ at the t changes and the signal value is 0.

In order to support SystemC processes to handle symbolic activation conditions, we first implement symbolic signals in SystemC. This is explained in the following.

1) Symbolic Signals in SystemC: In SystemC input and output ports and signals are implemented using template classes `sc_in⟨T⟩`, `sc_out⟨T⟩`, `sc_inout⟨T⟩` and `sc_signal⟨T⟩` where T specifies data type of signal values. Symbolic signal values use the classes `AADD` and `BDD` for data type. E.g. symbolic signals with values of `BDD` data type are defined as shown in Listing 3. All members and methods from the template class are also implemented here, including the methods that create references to rising, falling and value change events `pos()`, `neg()` and `value_changed()`, resp.

Listing 3: Specialization of `sc_signal⟨T⟩` for type BDD

```
template <>
class sc_signal<BDD> { ....
 public:
   void write(const BDD& value);
   void update(); ....};
```

Overloading is done for `write(value)` and `update()` methods. The method `write(value)` assigns a new value $value$ to a signal value if the new value differs from a current signal value. The new value is assigned using the overloaded assignment operator of the class `BDD` [11]. The `update()` method is overloaded using the algorithm shown in Listing 2. This method is called by the SystemC kernel for each signal specified in the process sensitivity list.

2) Symbolic simulation of SC_METHOD(proc): For defining processes with symbolic activation conditions we introduce the macro SC_METHODS(proc) where S stands for symbolic. This macro creates and registers the symbolic process with the method proc() in the global variable PR.

When activation conditions of all processes are evaluated, the kernel call methods $proc()$ of all processes for which the activation conditions are different than false. In order to ensure that assignments in the method $proc()$ of the currently active process use its active condition, the following macros are introduced:

- START_SYMB. This macro checks which process method is currently executed and pushes a copy of its process activation condition on the stack of block conditions $conditions$. The macro should be inserted at the beginning of the main body of the process method $proc()$.
- END_SYMB resets the condition of the active process to $false$ and pops its copy from the stack; it should be inserted at the end of the main body of $proc()$ in order to detect end of scope of $proc()$.

The macros and symbolic signals are implemented in the header file `symb_signals.h`. For symbolic simulation a user needs only to include it in a model with `#include "symb_signals.h"`; there is no need to recompile and reinstall the SystemC library.

IV. Case Study: Phase-locked Loop circuit

The proposed approach is demonstrated on the SystemC model of a charge-pump phase-locked loop (PLL) circuit exported from its Cadence schematic using the netlister [17]. The block diagram of the circuit is shown in Fig. 3.

The phase frequency detector (PFD) compares phases of the reference input signal with the output signal whose frequency is divided by the counter with a ratio N. For this circuit $N = 110$ and the frequency of the reference signal is $f_{ref} = 32MHz$. The standard structure of the PFD is composed of two D flip-flops and a logic AND circuit. The following code shows the SystemC model of the D flip-flops of the PFD, implemented following the approach described in Section III-D.

```
SC_MODULE(DFF){
 sc_in<boolS> CLK, R;
 sc_out<boolS> Q;
 boolS reg;
 SC_HAS_PROCESS(DFF);
 DFF(sc_module_name n){
   SC_METHODS(proc);
   sensitive << CLK.pos() << R.pos();
   reg=0;}
 void proc(){
   START_SYMB
   ifS(R.read()) reg=0;
   elseS reg=1; endS;
   Q.write(reg);
   END_SYMB }};
```

The frequency of the voltage controlled output signal f_{vco} is a function of the input voltages V_{vco}, V_m, V_t. The output signal V_{out} is generated comparing its phase $\Phi_v(t) = \int_0^t 2\pi f_{vco} dt$ with 2π; Comparison of the phase with $2 * \pi$ is implemented as an ifS-elseS statement with the condition $(\Phi_v(t) > 2 * \pi)$. When the phase crosses $2 * \pi$ the output signal value is 1, otherwise 0. The condition $(\Phi_v(t) > 2*\pi)$ is evaluated at each simulated time step t. When it results in uncertain value (both true and false) a BDD with a new internal node representing this condition is returned. Fig. 5 shows a BDD of the VCO

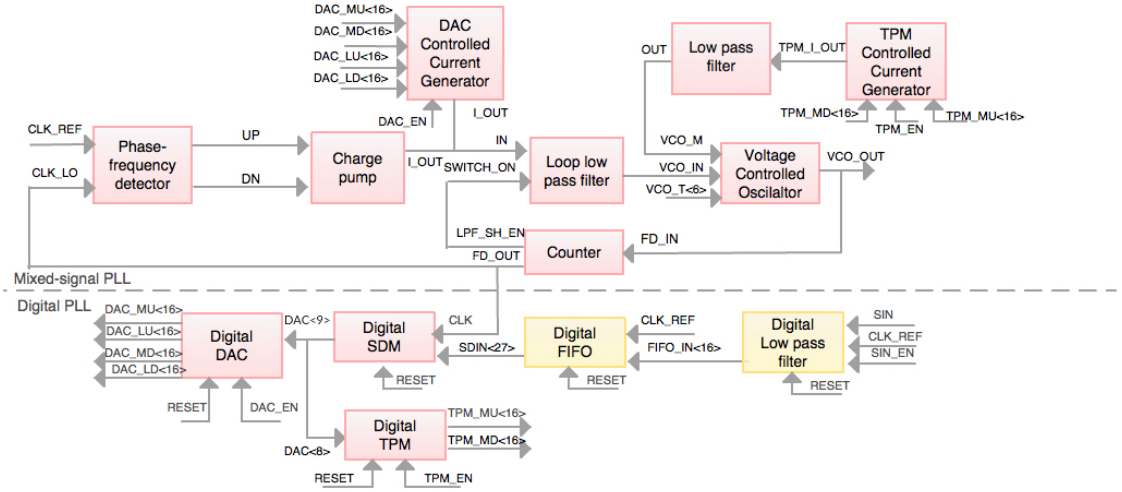

Fig. 3: Block diagram of the PLL circuit.

output signal with 6 leaves. Each leaf represents the condition $(\Phi_v(t) > 2 * \pi)$ at the time step t at which this condition for $\Phi_v(t)$ is uncertain.

In the locked state the PLL should reach the final frequency $f_{final} = N * f_{ref}$; for this circuit $f_{final} = 3.52GHz$ within the tolerance of $\pm 0.001GHz$. Symbolic simulation of the SystemC model of the given circuit is performed considering the uncertainties in the analog components of the circuit listed in Table I and colored red in Fig. 3. These uncertainties propagate through the digital parts of the circuit, also colored red in Fig. 3. The circuit is simulated for $1.5\mu s$ with the sampling frequency $f_s = 20GHz$. Fig. 4 plots lower and upper bounds of the output frequency f_{vco} computed in one symbolic simulation run and compared with 10 runs of corner case simulation.

One numeric block-level simulation run took only 2 seconds. The circuit-level simulation of the same circuit can take 9 hours [17]. Even with the fast block-level simulation, our approach is still competitive considering the fact that numeric simulations require a high number of simulation runs to achieve maybe a sufficient, but still not a comprehensive coverage.

Table II shows the simulation performance including the number of leaves of AADD value of f_{vco} at the end of the simulation and the PLL locking time t_{lock} in the worst case. The locking time is given in the clock cycles of the PLL reference signal with $f_{ref} = 32MHz$. The run times of symbolic simulation includes the computation of lower and upper bounds of the frequency f_{vco} for each time point plotted in Fig. 4. It can be noted that the run time changes almost linear with the level of uncertainty.

V. Conclusion and Future Work

The main contribution of this work is to introduce symbolic simulation for DE processes that are activated by symbolic conditions and its integration in the existing SystemC proof-of-concept simulator. Together with the support for the CT and

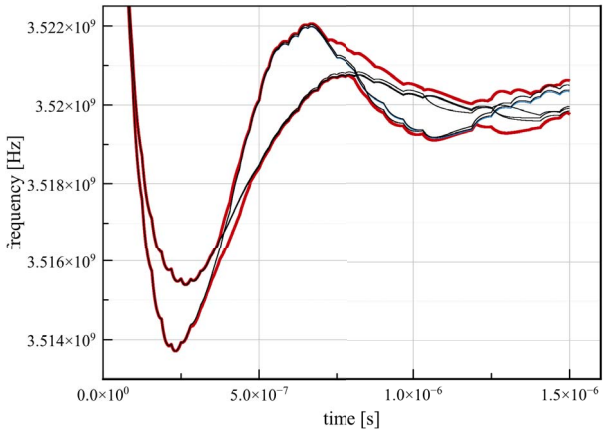

Fig. 4: PLL output frequency with 20 % of the current uncertainties: symbolic simulation (red); corner-case simulation (black)

TABLE I: Parameter uncertainties

Block	Parameter Uncertainty	Symbolic value
Low pass filter connected to TPM cur. generator	$R_1 = 100 \pm 10\%[k\Omega]$ $R_2 = 100 \pm 10\%[k\Omega]$ $C = 0.5 \pm 10\%[nF]$	$R_1 = 100 + \varepsilon_1 * 10[k\Omega]$ $R_2 = 100 + \varepsilon_2 * 10[k\Omega]$ $C = 0.5 + \varepsilon_3 * 0.05[nF]$
Current sources of TPM controlled current generator	$I_{tpm1} = 40 \pm 20\%[\mu A]$ $I_{tpm2} = 10 \pm 20\%[\mu A]$	$I_{tpm1} = 40 + \varepsilon_4 * 8[\mu A]$ $I_{tpm2} = 10 + \varepsilon_5 * 2[\mu A]$
Charge pump current source	$I_{cp} = 40 \pm 20\%[\mu A]$	$I_{cp} = 40 + \varepsilon_6 * 8[\mu A]$

TDF MoC from previous work this permits us the symbolic simulation of AMS systems that also include digital logic, i.e. the phase comparator and frequency divider. The main approach is to overwrite the virtual methods of SystemC signals to consider symbolic events in the event detection mechanism. Note that, despite vast new functionality, no re-compilation of the SystemC library is needed. This allows us to maintain compatibility with the existing SystemC eco-

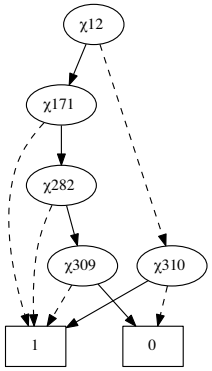

Fig. 5: BDD of VCO output signal with 6 leaves

TABLE II: Simulation performance.

	Run time [s]	Number of Leaves	t_{lock}
10 % of current uncert.	160	105	28
20 % of current uncert.	314	226	29

system while turning the existing SystemC (AMS, TLM) – where needed – into a symbolic simulator.

The approach currently only supports symbolic values, but not symbolic points in time for activation conditions. Support for such process activations is work in progress.

Another line of research deals with the underlying symbolic model. AADD perform excellent for analog parts, but have limitations in the digital part. Those could be overcome by SMT solvers such as [23]–[25] that can be combined with AADD where both approaches can profit from each other.

REFERENCES

[1] MentorGraphics. (2012, October) Improving Analog/Mixed-Signal Verification Productivity. [Online]. Available: https://verificationacademy.com/verification-horizons/october-2012-volume-8-issue-3/improving-analog-mixed-signal-verification-productivity

[2] T. Dang, A. Donzé, and O. Maler, "Verification of Analog and Mixed-Signal Circuits Using Hybrid System Techniques," in *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*, 2004, pp. 21–36. [Online]. Available: https://link.springer.com/content/pdf/10.1007%2Fb102264.pdf

[3] M. Althoff, A. Rajhans, B. H. Krogh, S. Yaldiz, X. Li, and L. Pileggi, "Formal Verification of Phase-Locked Loops Using Reachability Analysis and Continuization," in *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, November 2011, pp. 659–666. [Online]. Available: https://mediatum.ub.tum.de/doc/1287214/87226.pdf

[4] I. Seghaier, H. Aridhi, M. H. Zaki, and S. Tahar, "A Qualitative Simulation Approach for Verifying PLL Locking Property," in *ACM Great Lakes Symposium on VLSI*, 2014, pp. 317–322. [Online]. Available: http://dl.acm.org/citation.cfm?id=2591593

[5] S. Little, D. Walter, C. Myers, R. Thacker, S. Batchu, and T. Yoneda, "Verification of Analog/Mixed-Signal Circuits Using Labeled Hybrid Petri Nets," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 4, pp. 617–630, April 2011. [Online]. Available: https://ieeexplore.ieee.org/document/5737853/

[6] C. Zivkovic, C. Grimm, M. Olbrich, O. Scharf, and E. Barke, "Hierarchical Verification of AMS Systems with Affine Arithmetic Decision Diagrams," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, accepted for publication under DOI 10.1109/TCAD.2018.2864238, 2018.

[7] M. H. Zaki, S. Tahar, and G. Bois, "Formal Verification of Analog and Mixed Signal Designs: Survey and Comparision," in *IEEE Northeast Workshop on Circuits and Systems*, 2006, pp. 281–284.

[8] K. Marquet and M. Moy, "PinaVM: A SystemC Front-End Based on an Executable Intermediate Representation," in *Proceedings of the Tenth ACM International Conference on Embedded Software*, ser. EMSOFT '10. New York, NY, USA: ACM, 2010, pp. 79–88. [Online]. Available: http://doi.acm.org/10.1145/1879021.1879032

[9] K. Marquet, M. Moy, and B. Karkar, "A Theoretical and Experimental Review of SystemC Front-ends," in *Forum on Specification and Design Languages 2010*, 2010. [Online]. Available: https://ieeexplore.ieee.org/document/5775120/

[10] C. Grimm and M. Rathmair, "Dealing with Uncertainties in Analog/Mixed-Signal Systems," in *Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017*, 2017, pp. 35:1–35:6. [Online]. Available: http://doi.acm.org/10.1145/3061639.3072949

[11] C. Zivkovic and C. Grimm, "Symbolic Simulation of SystemC AMS without Yet Another Compiler," in *Forum on Specification and Design Languages (to appear)*, Sep. 2018.

[12] C. Radojicic, C. Grimm, A. Jantsch, and M. Rathmair, "Towards Verification of Uncertain Cyber-Physical Systems," in *Proceedings 3rd International Workshop on Symbolic and Numerical Methods for Reachability Analysis (SNR 2017)*. EPTCS 247, April 2017, pp. 1–17. [Online]. Available: https://arxiv.org/abs/1705.00519

[13] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 263–272. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081750

[14] T. Ball and J. Daniel, "Deconstructing Dynamic Symbolic Execution," in *Proceedings of the 2014 Marktoberdorf Summer School on Dependable Software Systems Engineering*, 2014. [Online]. Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/dse.pdf

[15] *SystemC 2.0.1 Language Reference Manual*, OSI, 2003. [Online]. Available: http://homes.di.unimi.it/~pedersini/AD/SystemC_v201_LRM.pdf

[16] *Standard SystemC AMS extensions 2.0 Language Reference Manual*, OSI, 2016. [Online]. Available: http://www.accellera.org/images/downloads/standards/systemc/SystemC_AMS_2_0_LRM.pdf

[17] Z. Chen, Y. Wang, L. Liao, Y. Zhang, A. Aytac, J. H. Müller, R. Wunderlich, and S. Heinen, "A SystemC Virtual Prototyping Based Methodology for Multi-Standard SoC Functional Verification," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14. New York, NY, USA: ACM, 2014, pp. 59:1–59:6. [Online]. Available: http://doi.acm.org/10.1145/2593069.2593079

[18] C. Zivkovic and C. Grimm, "Verification of Analog/Mixed-Signal Systems with AADD," in *Analog*, Sep. 2018.

[19] C. Radojicic, C. Grimm, F. Schupfer, and M. Rathmair, "Verification of Mixed-Signal Systems with Affine Arithmetic Assertions," *VLSI Design*, vol. 2013, 2013. [Online]. Available: https://www.hindawi.com/journals/vlsi/2013/239064/

[20] J. Stolfi and L. H. de Figueiredo, "An Introduction to Affine Arithmetic," *TEMA Trend. Mat. Apl. Comput.*, vol. 4, pp. 297–312, 2003. [Online]. Available: https://tema.sbmac.org.br/tema/article/view/352

[21] A. Jantsch and I. Sander, "Models of computation and languages for embedded system design," *IEE Proceedings - Computers and Digital Techniques*, vol. 152, no. 2, pp. 114–129, Mar 2005. [Online]. Available: http://dx.doi.org/10.1049/ip-cdt:20045098

[22] E. A. Lee and A. Sangiovanni-Vincentelli, "A Framework for Comparing Models of Computation," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 17, no. 12, pp. 1217–1229, Nov. 2006. [Online]. Available: http://dx.doi.org/10.1109/43.736561

[23] L. de Moura and N. Bjørner, *Z3: An Efficient SMT Solver*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[24] R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich, "The OpenSMT Solver," in *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 150–153. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-12002-2_12

[25] K. Scheibler, F. Neubauer, A. Mahdi, M. Fränzle, T. Teige, T. Bienmüller, D. Fehrer, and B. Becker, "Accurate ICP-based Floating-point Reasoning," in *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '16. Austin, TX: FMCAD Inc, 2016, pp. 177–184. [Online]. Available: http://dl.acm.org/citation.cfm?id=3077629.3077660